D3.2
Device-Level & Continuum-Wide Trust Extensions and Security Controls (First Release)

Project number:	101167904
Project acronym:	**CASTOR**
Project title:	Continuum of Trust: Increased Path Agility and Trustworthy Device and Service Provisioning
Project Start Date:	1st October, 2024
Duration:	36 months
Programme:	HORIZON-CL3-2023-CS-01
Deliverable Type:	Other
Reference Number:	HORIZON-CL3-2021-CS-01-101167904/ D3.2 / v1.0
Workpackage:	WP3
Due Date:	31st March, 2026
Actual Submission Date:	7th April, 2026
Responsible Organisation:	NVIDIA
Editor:	Fabian Schwarz
Dissemination Level:	Public
Revision:	1.0
Abstract:	This deliverable focuses on CASTOR's device-level trust extensions. It details the implementation of the Trust Network Device Extension (TNDE) and its SPDM-based JOIN/ONBOARDING protocols for enrolling platforms and onboarding TNDIs. A three-tiered attestation architecture is introduced, spanning device-level Configuration Integrity Verification, layered platform attestation, and composite path-level attestation with cryptographic ordering guarantees. The deliverable also specifies an FSM-based behavioural trust source, an eBPF-based tracing layer for runtime evidence collection, and an initial key management scheme, completing the first end-to-end instantiation of CASTOR's device-side trust architecture.
Keywords:	Trusted Network Device Extensions, Composite Attestation, Layered Attestation, Tracing, Introspection, Finite State machine Model

Funded by EU's Horizon Europe programme under Grant Agreement number 101167904 (CASTOR) and supported by the European Cybersecurity Competence Centre. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Cybersecurity Competence Centre. Neither the European Union nor the European Cybersecurity Competence Centre can be held responsible for them.
This work has received funding from the Swiss State Secretariat for Education, Research and Innovation (SERI). Funded by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee.

Copyright Notice

Project Funded by the European Commission in the Horizon Europe Programme		
Nature of the deliverable	**OTHER***	
	Dissemination Level	
PU	Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page)	X
SEN	Sensitive, limited under the conditions of the Grant Agreement	
Classified R-UE/ EU-R	EU RESTRICTED under the Commission Decision No2015/ 444.	
Classified C-UE/ EU-C	EU CONFIDENTIAL under the Commission Decision No2015/ 444	
Classified S-UE/ EU-S	EU SECRET under the Commission Decision No2015/ 444	

* R: Document, report (excluding the periodic and final reports)
DEM: Demonstrator, pilot, prototype, plan designs
DEC: Websites, patents filing, press & media actions, videos, etc.
DATA: Data sets, microdata, etc.
DMP: Data management plan
ETHICS: Deliverables related to ethics issues
SECURITY: Deliverables related to security issues
OTHER: Software, technical diagram, algorithms, models, etc.

Editor

Fabian Schwarz (NVIDIA)

Contributors (ordered according to beneficiary numbers)

Nikos Fotos, Sofianna Menesidou, Georgia Basagianni, Stefanos Vasileiadis, Giorgos Fotiadis, Vasilis Kalos, Thanassis Giannetsos (UBITECH)
Fabian Schwarz, Meni Orenbach (NVIDIA)
Yalan Wang, Liqun Chen (SURREY)
Riccardo Orizio, Stelios Basayiannis (COLLINS)
Alexandru Coleș, Ioan Constantin (ORO)
Pablo Martinez, Antonio Skarmeta (UMU)
Jamie Pont, Budi Arief, Theo Dimitrakos (UKENT)

Disclaimer

Executive Summary

CASTOR's goal is to realize trusted path routing in a below-zero-trust model, where routing decisions and traffic engineering explicitly incorporate trust metrics that evolve at runtime rather than relying on static configuration or topology alone. The concepts of the Trust Network Device Extension (TNDE), Trust Network Device Interfaces (TNDIs), and the overall CASTOR architecture were introduced in earlier deliverables, in particular D2.1 [22] and D3.1 [24]. Building on that foundation, this deliverable D3.2 reports on the status of the first implementation of these device-side components and concepts. It details how the initial prototypes of TNDE services, trust sources, and associated attestation and tracing schemes have been realized on concrete router platforms.

The document first revisits and refines the routing-plane threat model for CASTOR's target environments, where virtual routers are deployed as containers or VMs and operate under a strong Dolev–Yao adversary who controls the network and can exploit container-to-host jailbreaks, shared-kernel vulnerabilities, and control-plane message manipulation. This updated threat model sharpens the traceability mandate derived in earlier work: instead of generic telemetry, CASTOR requires specific, low-level evidence about host–container interactions, elevated privilege use, and protocol behavior to support continuous trust assessment. Within this setting, D3.2 emphasizes resilience, continuous attestation, and rapid detection. Compromise cannot be ruled out, and routing decisions must therefore be driven by dynamically enforced Required Trust Levels (RTL) over paths, while individual nodes' Trust Levels (TL) are monitored and updated over time.

Within this threat context, D3.2 details the first concrete implementation of the TNDE as the device-side trusted software stack that manages TNDIs, implements the TNDI Security Protocol (TNDI-SP), and exposes evidence to the CASTOR Service Orchestrator and global trust components such as the Global TAF and Phala/DLT. The TNDE is realized as a set of cooperating user-space services—including the Trust Network Device Security Manager (TN-DSM), a Tracing Hub, multiple Trust Sources, and a Local TAF Agent—connected via host-local APIs. The deliverable describes how these services are instantiated on platforms hosting Cisco XRd vRouter containers and how TNDE startup is orchestrated via manifest files. It also explains how the TN-DSM acts as the device-side endpoint of the SPDM-based JOIN and ONBOARDING flows of TNDI-SP. These protocol flows are specified at message and state-machine level and show how TNDE platforms are enrolled into a CASTOR domain, how TNDIs are on-boarded, and how per-TNDI control and data channels are established for evidence export.

A major part of the deliverable is devoted to the first implementation status of CASTOR's trust sources and their attestation schemes. The Attestation Trust Source is placed within a three-tiered security architecture that spans device-level configuration integrity verification, layered platform attestation, and composite path-level attestation, anchored in hardware roots of trust where available. D3.2 specifies concrete cryptographic and protocol mechanisms for these tiers, including configuration integrity verification (CIV), a layered attestation methodology over multiple privilege levels, and composite attestation schemes for end-to-end paths, for example based on sequential aggregate signatures and ordered multi-signature constructions. The deliverable discusses protocol roles, key material, and message flows. It explains how these mechanisms are integrated with the TNDE and TNDI-SP to provide attestation-backed evidence to the orchestration layer.

On the behavioral side, D3.2 reports on the design and initial implementation of a Finite State Machine

(FSM)-based Trust Source that serves as a lightweight and interpretable source of runtime behavioral evidence for selected router processes. Rather than presenting a finalized solution, the chapter documents the current pipeline and intermediate results. It describes how CASTOR's tracing layer provides syscall-level traces, how feature engineering and entropy-based selection are used to derive a reduced FSM alphabet, and how nominal FSM models are trained and evaluated on CASTOR eBPF trace data. The deliverable demonstrates that FSMs can be constructed from real traces and used to detect deviations, but also highlights open challenges such as process attribution, dataset breadth, and integration of the runtime FSM evaluation into the end-to-end trust assessment workflow.

To support both attestation and FSM-based behavioral monitoring, the deliverable specifies and evaluates the tracing architecture implemented in the first prototype. A concrete eBPF-based Trace Unit is integrated into the TNDE and deployed alongside Cisco XRd vRouter containers, attaching to stable syscall and scheduler tracepoints and to selected kprobes and uprobes in the XRd control-plane stack. The implementation applies cgroup-based filtering to isolate per-TNDI activity and emits CBOR-encoded trace batches over device-local endpoints towards Trust Sources, under configuration of the Tracing Hub. Using a flexible-algorithm routing scenario (FlexAlgo 128) as a case study, D3.2 reports experimental observations on how correlated syscall patterns and control-plane function probes form a characteristic execution pattern for healthy route updates. It shows how these observations can feed both static attestation checks and FSM models. This experimental chapter is explicitly framed as an exploration of trace granularity and feasibility and identifies which hooks and evidence are most useful for CASTOR's trust assessment goals.

Finally, D3.2 presents an initial key management scheme for the TNDE and its TNDIs, cataloging the keys and credentials used in the version 1 proof-of-concept. The deliverable describes the roles and bindings of RoT keys, TNDE platform keys, per-TNDI identity keys, composite attestation keys, and the symmetric session keys for TNDI-SP control and data channels. It explains how these are generated, bound to devices and TNDIs, and used across the different attestation and evidence-export protocols. Together with the protocol, tracing, and FSM components, this key management framework completes the first end-to-end instantiation of CASTOR's device-side trust architecture.

Overall, D3.2 does not introduce entirely new architectural concepts but instead moves CASTOR from the conceptual device-side designs of D3.1 to an initial integrated prototype that spans threat modeling, TNDE and TNDI-SP protocol realization, concrete attestation schemes, FSM-based behavioral monitoring, trace collection mechanisms, and key management on containerized routers. The deliverable provides a snapshot of the current implementation status. It identifies where designs have been turned into detailed protocols and code and where work remains exploratory, and it lays the groundwork for subsequent iterations that will harden these components and integrate them more tightly into CASTOR's end-to-end trusted path routing framework.

Contents

List of Figures

List of Tables

Versioning and contribution history

Version	Date	Author	Notes
v0.1	26.01.2026	Nikos Fotos (UBITECH)	Release first table of contents
v0.2	23.02.2026	Nikos Fotos, Georgia Basagianni, Stefanos Vasileiadis, Thanassis Giannetsos (UBITECH)	First draft of the routing-plane threat model under Dolev–Yao assumptions, container-specific vulnerabilities and jailbreak vectors (Chapter 2)
v0.3	25.02.2026	Fabian Schwarz, Meni Orenbach (NVIDIA)	TNDE services, TN-DSM role and platform-side initialisation, JOIN and ONBOARDING protocol flows via SPDM over TCP (Chapter 3, Sec. 3.1–3.5)
v0.4	28.02.2026	Fabian Schwarz (NVIDIA), Nikos Fotos, Stefanos Vasileiadis (UBITECH)	Tracing Hub, eBPF-based Trace Unit (libbpf syscall tracer), cgroup-based filtering, CBOR-encoded trace batches (Chapter 3, Sec. 3.6–3.7)
v0.5	06.03.2026	Stefanos Vasileiadis, Sofianna Menesidou, Vasilis Kalos, Thanasssis Giannetsos (UBITECH), Yalan Wang, Liqun Chen (SURREY)	CASTOR Attestation Service: tiered security architecture, CIV, layered attestation, composite path-level attestation based on sequential aggregate signatures (Chapter 4, Sec. 4.1–4.5)
v0.6	10.03.2026	Yalan Wang, Liqun Chen (SURREY)	Order-preserving constructions based on matrix-vector relations, misbehaviour tracking, cost analysis (Chapter 4, Sec. 4.6)
v0.65	14.03.2026	Riccardo Orizio, Stelios Basagiannis (COLLINS)	FSM-based attestation source: scope and assumptions, training and runtime pipelines, feature engineering, learnt models on CASTOR eBPF traces (Chapter 5)
v0.7	17.03.2026	Yalan Wang, Liqun Chen (SURREY)	Delegatable Order-Revealing Encryption (DORE): scheme overview and concrete construction (Chapter 6)
v0.75	21.03.2026	Georgia Basagianni, Nikos Fotos, Thanassis Giannetsos (UBITECH), Fabian Schwarz (NVIDIA)	Execution path determinism analysis for traffic engineering: FlexAlgo 128 case study, trace granularity, observable extraction flow (Chapter 7)
v0.8	25.03.2026	Thanassis Giannetsos (UBITECH)	TNDE key management scheme: RoT keys, platform keys, per-TNDI identity keys, composite attestation keys (Chapter 8); Summary and Conclusions (Chapter 9)
v0.85	26.03.2026	Fabian Schwarz (NVIDIA)	Summary and Conclusions (Chapter 9)
v0.9	01.04.2026	Nikos Fotos, Sofianna Menesidou, Thanassis Giannetsos (UBITECH), Fabian Schwarz (NVIDIA), Yalan Wang (SURREY)	Final refinements, cross-chapter consistency, terminology alignment, and internal review
v0.95	06.04.2026	ALL	Final version of CASTOR Trust Extensions vocabulary
v1.0	07.04.2026	Daphne Galani (UBITECH)	Final editorial review and submission of the deliverable

Chapter 1

Introduction

1.1 Scope and Purpose

Deliverable D3.2 constitutes the first release of the CASTOR trust extensions that enable the vision of runtime trust assessment within the context of the traffic engineering process. Building upon the conceptual architecture, component definitions, and functional requirements established in D3.1, the present deliverable provides a comprehensive specification and initial implementation of all the functionalities and services that must be exposed by the Trust Network Device Security Manager (TN-DSM) to protect the entire lifecycle of the infrastructure elements—namely the Trust Network Device Extension (TNDE) and Trust Network Device Interfaces (TNDIs)—**from the initial on-boarding through to the verifiable sharing of all evidence and trustworthiness claims to the Global Trust Assessment Framework (Global TAF) for the CC-wide trust quantification**.

In contrast to the existing landscape in the IETF Trusted Path Routing (TPR) architecture, where all routing elements are verified during the onboarding into the topology, this enrolment-time verification enables the establishment of a baseline of trust, it does not account for a fact that a device's trustworthiness may change over time. If a device becomes misconfigured, compromised, or enters a degraded trust state after initial enrollment, this change should be reflected in the trusted path routing decisions. The TPR model, as currently defined, provides no mechanism to detect or respond to such changes. Extending it with a runtime trust assessment phase raises several open issues that we need to resolve: **CASTOR envisions to overcome this gap by introducing adaptive-to-changes mechanisms for capturing the devices (HW and SW) trust scores, anchored to decentralized roots-of-trust.**

In practice, this is manifested through the definition of a Trusted Computing Base (TCB) anchoring set of customized TEE extensions (deployed in all nodes/elements, thus, constituting a Network of Distributed Roots-of-Trust (RoTs)) equipped with novel attestation and state introspection capabilities that allow the continuous monitoring of a device's configurational and behavioural properties. Such properties are represented through another core innovation of CASTOR by employing state-space abstractions to depict the vital operations (as finite state machine models) of the device per trust/security property of interest. Generation is based on sound learning techniques where accuracy of the state representation is proved to be equivalent to the real operation for the intended property. Examples include configuration integrity verification (runtime integrity of loaded software stack); operational correctness/resilience of safety-critical and sensitive workloads (been executed in TEE instances) enabled through the implementation of novel runtime behavioural attestation services that can identify exactly which portions of the system can be trusted as part of the exchanged security claims; and even system measurements on the underlying OS-/FW/RTOS acting as evidence on the availability of its resources and the instantiated security controls (dictating the level of robustness of this device against runtime attacks).

In this context, the present deliverable puts forth all details and inner-workings of the first version of CASTOR's three-tiered security framework that progressively extends trust from the device level to the

network path level: **Tier 1 establishes a device-level trust anchor through Configuration Integrity Verification (CIV)**; **Tier 2 introduces layered platform attestation across four privilege layers with weakened trust boundaries**; and **Tier 3 extends trust to end-to-end routing paths via composite attestation, incorporating cryptographic enforcement of the correct path order through sequential aggregate signature and ordered multi-signature constructions.** These mechanisms ensure that both individual nodes and entire routing paths can be verified with provable ordering guarantees.

This philosophy, tracing its lineage to early multiprogramming systems like Multics [29], advocates for fragmenting a system into (isolated) layers separated by hardware-enforced boundaries. Ideally, this structure resembles an inverted pyramid or a series of concentric circles with decreasing radii (Figure 4.1): the deepest layers —constituting the TCB are architected to compress the trust boundary to the extreme, including only the necessary code for supporting the business logic of upper layers, so that it can be *static* (i.e., minimize runtime dependencies) and *tighlty verified* while exposing only narrow interfaces to the outside world. As one ascends to the outer layers, software becomes increasingly complex and dynamic requiring an extended set of security safeguards for ensuring not only their boot-up integrity but also their uncompromised execution. This, in turn, requires the proper security abstractions and privilege separation compartmentalizing high-privilege code to a trusted lower layer (Layer-0 depicted by the Blue-boxes in Figure 4.1) bounding its functionality and interfaces to the outer world in order to reduce subtle attacks. This reduction of complexity at the core is a critical security property, ensuring that the most privileged components remain formally verifiable and predictable, even when subjected to adversarial conditions in the outer, less trusted layers.

Complementing this novel layered attestation methodology, D3.2 introduces a separate FSM-based trust source for runtime behavioural monitoring. The FSM pipeline ingests syscall-level traces from CASTOR's eBPF tracing layer, applies entropy-based feature selection to derive a reduced event vocabulary, and trains nominal models that capture expected routing daemon behaviour. At runtime, deviations from these models produce behavioural evidence that feeds into the trust assessment workflow alongside the attestation claims.

Furthermore, D3.2 delivers the first version of the Trace Unit implementation leveraging extended Berkeley Packet Filters (eBPFs), together with the instantiation and analysis of execution traces captured from routing functions. It integrates memory introspection and control-flow graph extraction to strictly verify execution flow. This approach ensures robust state continuity and integrity, achieving strong security guarantees with the efficiency required for decentralized, collaborative environments. The eBPF-based Trace Unit is deployed alongside Cisco XRd vRouter containers, attaching to stable syscall and scheduler tracepoints as well as selected kprobes and uprobes in the XRd control-plane stack. Using a flexible-algorithm routing scenario (FlexAlgo 128) as a case study, the deliverable demonstrates how correlated syscall patterns and control-plane function probes form characteristic execution patterns for healthy route updates, feeding both static attestation checks and FSM-based behavioural models. The deliverable also specifies the initial key management scheme for the TNDE and its TNDIs, completing the first end-to-end instantiation of CASTOR's device-side trust architecture.

1.2 Relation with other WPs and Deliverables

D3.2 is a core outcome of WP3 and closely interacts with several other CASTOR work packages. In particular, it builds upon the high-level CASTOR architecture and use-case analysis presented in D2.1, the conceptual architecture of CASTOR TCB and composite attestation model specification in D3.1, and provides essential input to the Local TAF defined in D4.2 [25] and the establishment of authenticated and encrypted communication channel between TN-DSM with the backend orchestration layer in D5.2. In addition, this document provides all the necessary information in the CASTOR use cases developed in WP6. Figure 1.1 depicts the relation of D3.2 with other WPs and deliverables.

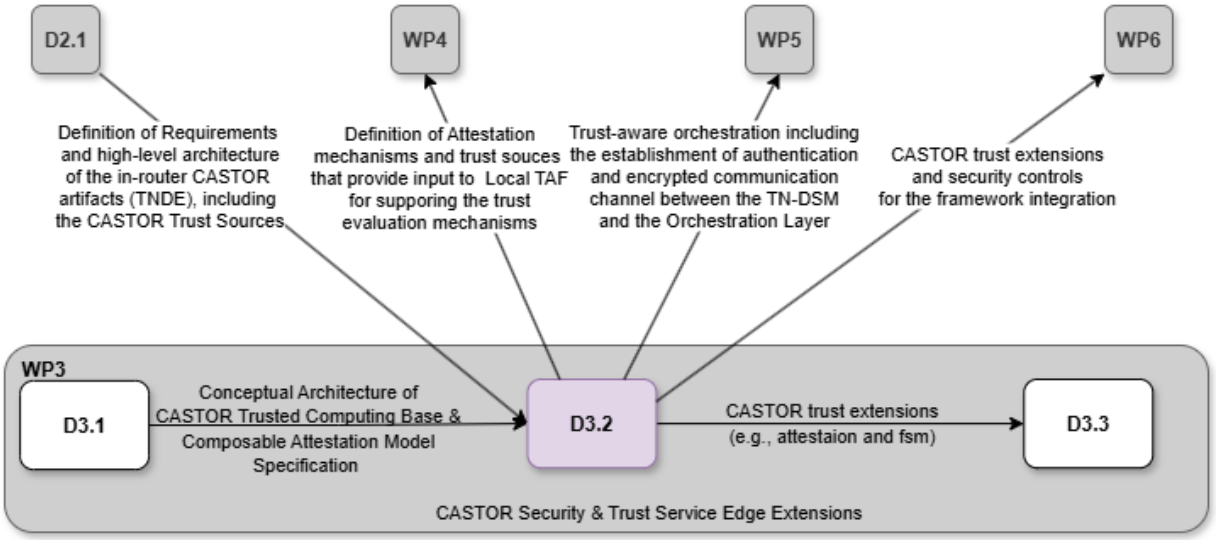

Figure 1.1: Relation of D3.2 with other WPs and Deliverables

1.3 Deliverable Structure

The remainder of this deliverable is organised as follows.

Chapter 2 revisits and refines the routing-plane threat model for CASTOR's target environment, where vRouters are deployed as containers sharing the host kernel. It formalises the Dolev–Yao adversary within this setting, catalogues container-specific vulnerabilities and jailbreak vectors, and derives the traceability mandate that dictates what evidence the CASTOR tracing architecture must continuously collect.

Chapter 3 details the Trust Network Device Extension (TNDE) as the device-side trusted software stack. It describes the TN-DSM and its role in platform-side initialisation, the SPDM-based JOIN and ONBOARDING protocol flows of the TNDI Security protocol (TNDI-SP) the establishment of per-TNDI control and data channels for evidence export, the Tracing Hub responsible for runtime configuration and CBOR-encoded trace dispatch, and the eBPF-based Trace Unit with its cgroup-based per-TNDI filtering.

Chapter 4 presents the CASTOR Attestation Service organised around a three-tiered security architecture: Tier 1 establishes a device-level trust anchor through Configuration Integrity Verification (CIV) with policy-restricted attestation keys; Tier 2 introduces layered platform attestation across four privilege layers with weakened trust assumptions; and Tier 3 extends trust to end-to-end routing paths via composite attestation based on sequential aggregate signatures and ordered multi-signature constructions with provable path-ordering guarantees.

Chapter 5 describes the FSM-based trust source as a complementary mechanism for lightweight runtime behavioural monitoring. It details the pipeline from syscall-level trace ingestion through entropy-based feature selection and nominal model training, and reports on current integration results using CASTOR eBPF trace data.

Chapter 6 introduces Delegatable Order-Revealing Encryption (DORE) scheme, which enables the zero-knowledge sharing of path-centric trust metrics across administrative domains while preserving the confidentiality of the underlying trust scores.

Chapter 7 analyses execution path determinism for traffic engineering through experimental trace analysis. Using a FlexAlgo 128 routing scenario as a case study, it examines how correlated syscall patterns and control-plane function probes form characteristic execution signatures for healthy route updates, and identifies which hooks and evidence are most useful for CASTOR's trust assessment goals.

Chapter 8 specifies the TNDE key management scheme for the TNDE and its TNDIs, listing the Root-of-Trust keys, platform keys, per-TNDI identity keys, composite attestation keys, and symmetric session keys used across the attestation and evidence-export protocols.

Chapter 9 concludes with a summary and outlook.

Chapter 2

Threat Modelling in the CASTOR Routing Plane

2.1 Introduction and Architectural Paradigm

The **CASTOR** project represents a paradigm shift in securing routing paths across next-generation computing continuums, specifically targeting Beyond 5G (B5G) and 6G networks. While traditional infrastructure relies heavily on physical appliances, the first version of **CASTOR** realizes the core concepts established in Chapter 3 by instantiating Trusted Network Domain Interfaces (TNDIs) directly as virtual routers (vRouters). These vRouters can be deployed either as container-based instances running on the host or as hypervisor-isolated Virtual Machines (VMs), with future iterations generalizing this deployment model even further. This flexible, cloud-native approach—utilizing platforms such as Kubernetes or lightweight orchestration frameworks—ensures rapid provisioning, minimal overhead, and high scalability.

However, this architectural agility fundamentally alters the system's attack surface. The security of the routing plane is no longer strictly bound to the protocol specification or the hardware appliance. Instead, it becomes inevitably linked to the underlying container runtime, the host operating system's kernel, and the orchestration control plane. Consequently, the **CASTOR** threat model must pivot from traditional network-centric threat analyses to a hybrid model that accounts for advanced adversarial capabilities in the network and the inherently porous boundaries of OS-level virtualization. The core objective of this threat model is to formally define the Required Trust Level (RTL) for dynamic trust assessment and to dictate the exact systematic evidence that must be continuously collected by the **CASTOR** tracing architecture.

2.2 Adversarial Model: The Dolev-Yao Assumption

To ensure rigorous security bounds, **CASTOR** evaluates the routing and control planes under the formal **Dolev-Yao adversarial model**. In the context of the **CASTOR** routing architecture, this model assumes that the underlying network infrastructure is entirely untrusted and under the full control of the adversary.

Specifically, the Dolev-Yao adversary possesses the following capabilities within our threat landscape:

- **Omnipresence and Interception:** The adversary can overhear, intercept, and record any message traversing the network, including inter-domain BGP advertisements, intra-domain OSPF/IS-IS link-state packets, and SDN controller telemetry.
- **Message Manipulation and Synthesis:** The attacker can arbitrarily modify, drop, delay, replay, or inject newly synthesized messages into the network fabric. They can impersonate any legitimate

vRouter or control plane entity, provided they possess the requisite cryptographic material.

- **Cryptographic Boundaries:** The adversary is computationally bounded. They cannot break secure cryptographic primitives (e.g., they cannot forge valid digital signatures without the private key or decrypt payloads without the symmetric key).

Applying the Dolev-Yao model to a virtualized routing environment implies that traditional, plaintext routing protocols are inherently compromised by default. An adversary can effortlessly inject forged route advertisements, leading to traffic blackholing, prefix hijacking, or deterministic routing loops. Therefore, trust in the **CASTOR** ecosystem cannot be derived from network location or packet delivery mechanisms; it must be continuously mathematically proven through cryptographic attestation and runtime behavioral evidence.

2.3 Container-Specific Vulnerabilities and Jailbreak Vectors

Unlike traditional VMs, which utilize a hypervisor to provide strict hardware-level isolation, containers share the single underlying kernel of the host Operating System. Isolation is enforced logically via Linux namespaces (restricting visibility) and cgroups (restricting resource consumption).

Because vRouters frequently manipulate networking components, such as virtual network interfaces (vNICs), routing tables, and firewall rules, they historically struggle to run as fully unprivileged containers. In standard control-plane deployments, they typically require elevated networking privileges. Even in data-plane scenarios where host-level networking components are bypassed, such as using DPDK with physical NIC pass–through, high-level access is still necessary to interact directly with hardware (e.g., managing hugepages and VFIO devices). While modern isolation techniques can help deprivilege containers by scoping these permissions to dedicated network namespaces, the underlying reliance on elevated host access for advanced packet processing creates a persistent and critical threat vector: **Container-to-Host Jailbreaks**.

If a Dolev-Yao adversary crafts a malicious BGP update that triggers a buffer overflow in the routing daemon (e.g., FRRouting or Quagga) running inside the container, they gain remote code execution (RCE) within the container's namespace. From this beachhead, the adversary can leverage the vRouter's elevated capabilities to execute a container escape.

2.3.1 Mechanisms of Container Escape

In the **CASTOR** threat model, we explicitly identify the following container escape methodologies as high-priority threats:

1. **Exploitation of Shared Kernel Subsystems:** An attacker with code execution inside the vRouter container can interact directly with the host kernel's system call interface. Vulnerabilities in network protocol stacks, eBPF verifiers, or shared filesystem drivers can be exploited to overwrite kernel memory, granting the attacker root privileges on the underlying host node.

2. **Abuse of Elevated Privileges:** While standard network namespace isolation normally prevents a container from tampering with arbitrary host interfaces, vRouters often must run within the host's network namespace to function effectively. In such shared-namespace deployments, an attacker who compromises the container inherits the elevated networking privileges needed to manipulate routing tables and intercept traffic destined for other vRouters. Furthermore, if the container is granted broader system-level access, an attacker can exploit host kernel mechanisms to escape the container entirely. For example, by mounting the host's control group (cgroup) filesystem, the

attacker can configure a malicious `release_agent`. Because this agent is a native kernel feature designed to run cleanup scripts directly on the host whenever a cgroup empties, the attacker can trigger it to execute arbitrary commands outside the container boundary.

3. **Exposed Orchestration Sockets:** While granting access to the host's container runtime socket (e.g., `docker.sock` or `containerd.sock`) is not standard practice for isolated data-plane operations, it is a realistic misconfiguration in hyper-converged edge architectures or unified deployments where management and orchestration agents are co-located within the vRouter environment. If inadvertently mounted, an attacker who compromises the vRouter can interact directly with the host daemon to instantiate highly privileged peer containers, effectively achieving a total host takeover.

Once a jailbreak is successful, the adversary transcends the network plane. They can passively extract cryptographic keys from the host's memory, bypassing Dolev-Yao cryptographic restrictions, and laterally compromise the entire **CASTOR** cluster.

2.4 Comprehensive Threat Catalogues

Based on the intersection of the Dolev-Yao model and our containerized architecture, we categorize the threats into distinct domains. The following tables summarize the critical threats that the **CASTOR** trust assessment framework must mitigate and monitor.

Table 2.1: Threats in the Control Service and Container Infrastructure

ID	Title	Description
TM-C.01	Dolev-Yao SDN Spoofing	Under Dolev-Yao, the adversary synthesizes forged SDN control messages. If the control channel lacks robust cryptographic authentication, the attacker can rewrite vRouter flow tables globally.
TM-C.02	Container Escape via Capabilities	Exploitation of required networking capabilities to bypass namespace isolation, allowing an attacker who has compromised the routing daemon to execute code directly on the host OS.
TM-C.03	Shared Kernel eBPF Exploit	Malicious manipulation of the Extended Berkeley Packet Filter (eBPF) subsystem. An attacker escapes the container by loading malicious eBPF programs via compromised vRouter daemons, achieving ring-0 code execution on the host.
TM-C.04	Orchestrator API Hijacking	Unauthorized access to the container orchestrator (e.g., Kubernetes API server) by stealing service account tokens from a compromised vRouter container, leading to cluster-wide compromise.
TM-C.05	Routing Daemon RCE	Remote Code Execution vulnerabilities in the routing software (e.g., parsing errors in BGP/OSPF engines). This serves as the primary entry point for subsequent container jailbreak attempts.
TM-C.06	Shared Resource Exhaustion	A compromised container aggressively consumes shared host resources (CPU, Memory, IO), resulting in a localized Denial of Service for all adjacent vRouters running on the same hardware.

Table 2.2: Threats in the Network and Routing Layer

ID	Title	Description	Mitigation Strategy
TM-N.01	Malicious Route Injection	The Dolev-Yao adversary injects forged BGP/OSPF updates to alter the routing topology, facilitating traffic interception or blackholing.	Cryptographic route origin validation (RPKI), BGPsec, and continuous runtime evidence tracing.
TM-N.02	Flex-Algo Tampering	Manipulation of Segment Routing Flexible Algorithm advertisements to force traffic engineering paths through adversary-controlled network nodes.	Mutual TLS for control plane, strict signature verification on Segment Routing descriptors.

ID	Title	Description	Mitigation Strategy
TM-N.03	Protocol Replay Attacks	The adversary captures legitimate routing updates or PCE requests and replays them out-of-band to cause route flapping or unauthorized state changes.	Nonces, timestamp validation, and sequence number enforcement in protocol payloads.
TM-N.04	Bidding-Down Attack	The Dolev-Yao attacker actively drops cryptographic negotiation packets, forcing the vRouters to fall back to unencrypted or legacy routing protocols.	Strict configuration validation; disabling fallback to unauthenticated protocols.
TM-N.05	State Disruption via DoS	Injection of crafted, high-volume protocol hello messages (e.g., IS-IS hellos) designed to overwhelm the container's CPU allocation and drop routing adjacencies.	Hardware-level rate limiting, Control Plane Policing (CoPP), and strict cgroup CPU limits.

2.5 Conclusion and Traceability Mandate

Because **CASTOR** assumes a highly hostile Dolev-Yao network environment and a fragile container isolation boundary, absolute prevention of compromise is an unrealistic objective. Instead, this threat model dictates that the **CASTOR** architecture must prioritize **resilience, continuous attestation, and rapid detection**.

The exact threats documented herein –specifically the mechanics of container escapes, elevated privilege abuse, and unauthenticated message synthesis –directly establish the **Traceability Mandate** for the **CASTOR** Tracing Hub. Rather than relying on generic network telemetry, **CASTOR**'s tracing implications are deeply intertwined with its four-layer attestation architecture. To effectively detect container-to-host jailbreaks and Dolev-Yao manipulations, the Tracing Hub must collect highly specific, low-level evidence. This involves monitoring host-level eBPF hooks for anomalous system calls (such as unauthorized `release_agent` cgroup manipulation or unexpected access to orchestration sockets), extracting hardware-anchored cryptographic measurements from the Secure Element, and validating the continuous runtime integrity of the routing daemons against the Policy Authority's reference values.

By mapping these specific threat vectors to verifiable runtime evidence, **CASTOR** establishes a dynamic, mathematically grounded trust ecosystem. It is crucial to distinguish that while the **CASTOR** framework continuously *monitors and evaluates* the individual Trust Level (TL) of each vRouter (or Trusted Network Domain Interface - TNDI), it dynamically *enforces* the Required Trust Level (RTL) of the overall **routing path**. We cannot strictly enforce the security state of an untrusted node under attack; we can only monitor its TL. However, if the Tracing Hub detects that a specific node's TL has degraded—for instance, due to anomalous host-namespace activity indicating a potential container escape—the **CASTOR** Traffic Engineering Policy Engine proactively enforces the path's RTL. It does so by seamlessly rerouting traffic, bypassing the compromised or untrusted node entirely before a localized breach can escalate into a systemic infrastructure failure.

Chapter 3

Trust Network Device Extension

CASTOR's trusted path routing relies on a device-side trusted software stack that can manage TNDIs, implement the TNDI Security Protocol (TNDI-SP), and expose trustworthiness evidence towards the orchestration layer. On each router, this stack consists of the Trust Network Device Extension (TNDE) services together with the Trace Units. In this chapter, we describe the TNDE services, their key management, and their interfaces on a router platform, using one concrete prototype instantiation as a running example rather than as the only valid realization. The TNDE roles and the TNDI-SP control and data channel protocols are, however, designed to be agnostic to the underlying roots of trust and isolation layers and can be realized on different device platforms.

We focus in particular on the TN-DSM and its role as the device-side endpoint of the TNDI-SP control channel, which in version 1 is realized over an SPDM-over-TCP transport [31]. The initial JOIN phase of this control channel attests the TNDE platform and establishes an SPDM Secured Messages [32] session for subsequent TNDI-SP control messages that manage the TNDI lifecycle and onboarding into CASTOR's trusted path routing domain. Building on this, we describe how the TNDE sets up per-TNDI TNDI-SP data channels and uses them to expose runtime traces and trust reports towards the Global TAF and Phala/DLT, thereby enabling the orchestration layer to consume device-side evidence. Finally, we outline the integration of an eBPF-based Trace Unit into the TNDE, which provides syscall-level runtime tracing for TNDIs in version 1 and serves as the basis for the experimental evaluation in the tracing chapter.

3.1 TNDE Services

3.1.1 Overview

In the current CASTOR prototype, the Trust Network Device Extension (TNDE) is realized as a set of co-operating user-space services that together form the device-side trusted software stack alongside the Trace Units. Each TNDE service runs as a separate process, which provides basic compartmentalization of functionality even before any hardware-enforced isolation such as TEEs is applied. Mapping individual services into TEEs or other RoT-backed isolation mechanisms is left to concrete platform instantiations, and CASTOR specifically plans to explore Intel SGX as a RoT/TEE for the TNDE in version 2 of the framework.

The TNDE includes the Trust Network Device Security Manager (TN-DSM), the Tracing Hub, one or more Trust Sources such as the Attestation Trust Source, the Local TAF Agent, and the auxiliary TPL Data Connector used by the Local TAF. Additional Trust Sources such as an FSM-based Trust Source and advanced cryptographic schemes (e.g., DORE) are conceptually explored and defined but not yet integrated into the runtime flows of this version. The TNDE services communicate with each other over long-lived,

host-local TCP connections using HTTP REST APIs in version 1. For version 1, we deliberately assume that this intra-TNDE communication is not subject to active attacks and that non-TNDE components do not connect to these local endpoints, which can be seen as reflecting a deployment where all TNDE services execute inside a single trusted computing boundary, for example a TEE-protected domain like Arm TrustZone, and only the TNDIs and other device software remain outside. In later versions, CASTOR intends to strengthen this model by introducing protected and mutually authenticated channels between TNDE services, for example when TNDE services run in separate TEEs or isolation domains.

For evaluation, we instantiate TNDIs as pre-launched Cisco XRd containers (vRouters) on the host system and run the TNDE services alongside them on the same platform, with the TNDE responsible for discovering these TNDIs and managing their TNDI-SP control and data channels. That is, in version 1 of the CASTOR prototype, the host operating system's process and container isolation acts as the isolation layer between the TNDE and the TNDIs. Alternative instantiations, such as TNDIs as dedicated virtual machines (VMs) with a hypervisor-based isolation layer or physical routers with TNDE running on a management CPU, and combinations where Intel SGX enclaves and a hypervisor jointly provide the RoT and isolation layer for TNDE and TNDIs, are conceptually supported by the TNDI and TNDI-SP concepts but are outside the scope of this version 1 prototype.

3.1.2 TNDE Startup

In version 1, the TNDE startup sequence is intentionally simple and based on static configuration files that describe both the local TNDE endpoints and the available Trace Units on a given platform. Each TNDE service reads a JSON configuration, referred to as the TNDE Manifest, during startup; this manifest enumerates the TNDE sub-components (i.e., services) and assigns each of them host-local TCP endpoints so that components can discover where to bind and how to connect to their peer services. A separate Tracing Infrastructure Profile JSON document, introduced later in the tracing section (Section 3.6.1), describes the Trace Units and their internal endpoints and is consumed primarily by the Tracing Hub and Trust Sources.

In the current prototype, the TNDE follows a fixed startup order so that communication channels are available when needed. First, the Trace Units are launched so that they can accept initial registration and configuration messages from the Tracing Hub. Next, the Tracing Hub starts and connects to the Trace Units using the endpoints from the Tracing Infrastructure Profile, preparing the device-local tracing topology. After that, the Trust Sources and the Local TAF Agent start and establish their host-local TCP connections: the Trust Sources connect to the Trace Units (and, in later versions, may instead connect to the Tracing Hub) so that they can later query and obtain trace batches, and communication channels are set up between the Trust Sources and the Local TAF Agent so that derived evidence can be forwarded to the Local TAF Agent, which will use it to produce local trust reports for each TNDI. Finally, the TN-DSM starts and reads the TNDE Manifest to establish long-lived TCP connections to the Tracing Hub, Local TAF Agent, and other TNDE services; once these connections are in place, downstream services can optionally open reverse or auxiliary connections back to the TN-DSM, for example to push trust reports or trace summaries without polling.

The TNDE Manifest itself is represented as a small JSON document that provides a stable description of the TNDE-internal interfaces. At a minimum, it lists all TNDE services and assigns each a stable name, an optional role tag (for example `"role": "attestation-trust-source"` for the Attestation Trust Source), and host-local TCP endpoints. At startup, each TNDE service parses this manifest, locates its own entry to determine which local address to bind to, and discovers the endpoints of its peers so that it can actively establish the required TCP connections. In future versions, the integrity and authenticity of the manifest should be protected, for example by signing the JSON file. This configuration-driven approach keeps the TNDE topology explicit and auditable, while remaining independent of any particular RoT or isolation-layer instantiation on the device. In practice, the baseline TNDE Manifest is expected to

be provided by the TNDE implementation, while operators may adjust local endpoints as needed for a given deployment, whereas the Tracing Infrastructure Profile primarily reflects the Trace Units supported by a specific device platform and is therefore largely determined by the platform vendor.

3.2 TN-DSM: Role and Platform-Side Initialization

3.2.1 TN-DSM Role and Interfaces

The Trust Network Device Security Manager (TN-DSM) forms the main control interface between a network device and the CASTOR orchestration layer. It enables the CASTOR Service Orchestrator to join the TNDE platform into a CASTOR domain via the JOIN protocol and, once the platform is enrolled, to onboard the device's TNDIs into the trusted path routing topology using the TNDI-SP control channel.

Within the TNDE, the TN-DSM therefore acts both as the attestor of the TNDE platform and as the manager of the device's TNDIs, their lifecycle, and their cryptographic identities, exposing the TNDI-SP control channel endpoint that the orchestrator uses to configure tracing, trust policies, and TNDI-SP data channels. The TN-DSM exposes a network-facing TCP endpoint that serves as the entry point for both the JOIN protocol and all subsequent TNDI-SP control traffic from the Service Orchestrator. The detailed JOIN protocol that operates via the TN-DSM is described in Section 3.3, while the TNDI lifecycle state machine and the TNDI-SP onboarding flows that run over the same TN-DSM endpoint are described in Section 3.4.1 and Section 3.4.3.

3.2.2 TNDE Platform Key Setup

As shown in Figure 3.1, at startup the TN-DSM initializes its local attestation state before accepting any incoming JOIN connections. First, the TN-DSM generates the long-lived TNDE platform attestation key pair $(pk_{\text{plat}}, sk_{\text{plat}})$ inside the TN-DSM and arranges for this key pair to be persisted across reboots in a way that ties its use to the same platform or TEE instance, for example by using a TEE-specific sealing mechanism or platform-provided protected storage. This key pair serves as the TNDE's platform identity key in version 1 and is later used in the TNDE JOIN protocol as the authentication and attestation key of the TNDE platform towards the Service Orchestrator (see Section 3.3).

In version 1 of the CASTOR framework, the TNDE platform key pair is generated locally during TN-DSM initialization and is not yet explicitly bound to a hardware Root of Trust (RoT) key hierarchy such as a TPM. In version 2, CASTOR plans to extend this design by establishing an explicit cryptographic binding between the TNDE platform key and the device's RoT, for example by having a TPM attestation key (AK) certify the TNDE platform key so that the orchestrator can verify that the TNDE identity is anchored in a genuine TPM-based RoT. This may include integrating a one-time TPM credential-activation step for each device into the initial TNDE JOIN procedure, which lets the orchestrator verify that the attestation key originates from a RoT key hierarchy it trusts (for example, a TPM endorsement key). The exact mechanism will be detailed in the version 2 design and is not implemented in the version 1 proof-of-concept.

3.2.3 TNDE Measurement Preparation

After key generation, the TN-DSM continues with the measurement steps of the initialization sequence illustrated in Figure 3.1. In version 1, this preparation is conceptually performed once during TNDE startup: the TN-DSM first derives a platform-level integrity digest from the relevant roots of trust (for example, TPM-based boot measurements or SGX firmware measurements) and records this digest as one

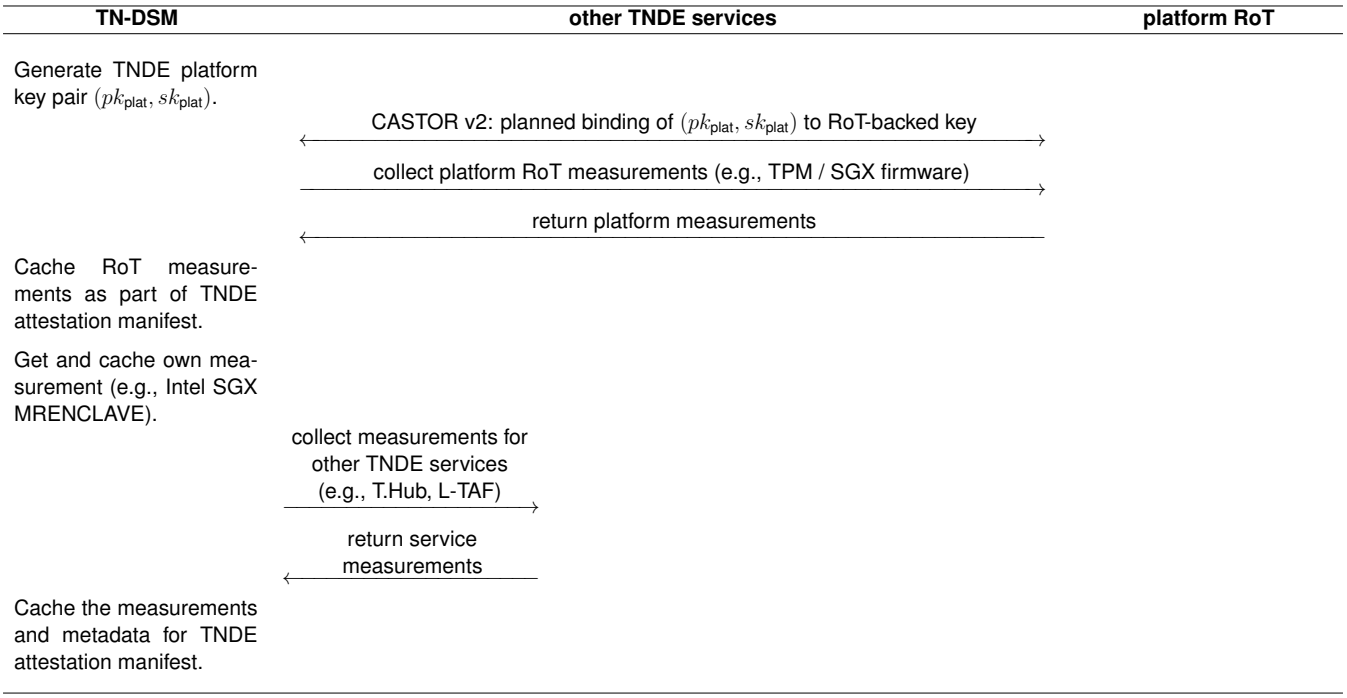

Figure 3.1: TNDE platform key setup and measurement preparation sequence (of version 1).

manifest component, establishing the baseline platform state. Next, the TN-DSM obtains a measurement of its own code, such as an enclave measurement (e.g., MRENCLAVE) in an SGX-based instantiation, and adds this digest as another manifest component. Finally, it collects measurements from other trusted TNDE services, such as additional enclaves or protected processes providing the Tracing Hub and Local TAF functionality, and appends their digests to the manifest as further components.

In version 1, all resulting measurements, together with metadata describing their origin, are cached locally inside the TN-DSM so that, when the CASTOR Service Orchestrator later initiates the JOIN protocol, the TN-DSM can assemble these cached components into a composite attestation manifest and return it for verification without incurring additional measurement overhead during the handshake. For later versions, CASTOR may extend this model so that the TN-DSM can refresh or augment these measurements periodically or on demand; in particular, instead of pre-fetching all measurements at startup, the TN-DSM could collect some or all evidence in response to explicit measurement requests, or periodically re-query platform or service measurements. In the version 1 proof-of-concept implementation, the integration with concrete RoT measurement mechanisms is only partially realized, and some attestation steps are still emulated or stubbed rather than being fully wired to RoT reporting.

3.2.4 TNDI Discovery

Conceptually, the TN-DSM must maintain a view of all TNDIs present on a device in order to manage their lifecycle, keys, and TNDI-SP configuration. In version 1 of the CASTOR framework, this view is derived from a static JSON configuration file that the TN-DSM loads during TNDE startup. This file enumerates the TNDIs known to a given TNDE instance, including a stable identifier for each TNDI (e.g., a UUID or name) and basic descriptive metadata such as type, vendor, product, version, and an optional human-readable description, as illustrated in Listing 3.1.

After loading this discovery file, the TN-DSM initializes, for each listed TNDI, the corresponding internal data structures – in particular, a TNDI-SP lifecycle state machine instance – so that it can manage that TNDI and later expose it to the Service Orchestrator during onboarding. The detailed TNDI-SP state machine and the control messages that operate on these TNDI instances during onboarding, configuration, and runtime operation are introduced in Section 3.4.1 and Section 3.4.3.

```
{
  "version": 1,
  "device_id": "host-router-01",

  "tndis": [
    {
      "tndi_uuid": "tndi-vr1",
      "container_name": "vrouter-1",
      "type": "virtual-router",
      "vendor": "Cisco",
      "product": "XRd",
      "version": "7.9.1",
      "description": "Virtual edge router"
    },
    {
      "tndi_uuid": "tndi-vr2",
      "container_name": "vrouter-2",
      "type": "virtual-router",
      "vendor": "FRRouting",
      "product": "FRR (open source)",
      "version": "10.0.0",
      "description": "Core router instance"
    }
  ]
}
```

Listing 3.1: Example for a static TNDI discovery JSON configuration parsed by the TN-DSM.

For future versions, CASTOR will explore more dynamic models for TNDI discovery and creation. One possible direction is a tighter integration with virtualization and orchestration platforms, for example by deploying TNDIs as dynamically managed virtual routers via Kubernetes or similar systems and letting the TN-DSM support on-demand registration of newly instantiated TNDIs, rather than relying solely on a static discovery JSON.

3.3 JOIN (Platform/TNDE onboarding) Flow

3.3.1 TNDE Platform JOIN via SPDM over TCP

Before any of a device's TNDIs can be onboarded into a CASTOR trust network, the CASTOR Service Orchestrator must first establish trust in the underlying device platform and its TNDE. In the current CASTOR prototype, this is achieved through a JOIN flow in which the Service Orchestrator connects to the TN-DSM over TCP, runs an SPDM [33, 31] handshake to authenticate the TNDE platform key and retrieve an attestation manifest, then establishes an SPDM Secured Messages [32] channel that serves as the transport for all subsequent TNDI-SP control traffic. Conceptually, this JOIN flow forms the first phase of the TNDI-SP control channel between the orchestrator and the TN-DSM: it uses SPDM for capability negotiation and authentication, establishes an SPDM secure session, and then retrieves platform measurements, resulting in an attested, encrypted control channel to the TNDE platform. This ensures that all later TNDI-SP lifecycle and configuration operations are cryptographically bound to both the TNDE platform identity and the attestation evidence collected during the JOIN phase. In the current version 1

prototype, this JOIN flow does not yet authenticate the CASTOR Service Orchestrator towards the TN-DSM. Instead, we assume that access to the TN-DSM's network-facing endpoint is already restricted and authenticated through orthogonal deployment mechanisms, for example by placing a reverse proxy in front of the TN-DSM that authenticates the Service Orchestrator, or by exposing the JOIN endpoint only on a dedicated management network that enforces access control.

The TN-DSM's local measurement preparation is summarized in Figure 3.1, while the full JOIN message flow between the orchestrator and the TN-DSM is shown in Figure 3.2. Once the TN-DSM startup phase (platform measurement and caching, as described in Figure 3.1) is complete, the TN-DSM listens on a TCP server socket for incoming connections from the CASTOR Service Orchestrator. When the orchestrator connects, it initiates SPDM over this TCP channel and runs the standard SPDM base negotiation, including version selection, capability negotiation, and algorithm selection. The orchestrator then retrieves the TNDE certificate chain from the TN-DSM and extracts the TNDE platform public key pk_{plat}, which it checks against its registration store to decide whether this platform is already known or should be treated as a new TNDE JOIN event in version 1. In version 1, the orchestrator only verifies the TNDE platform key and does not yet enforce a verifiable binding between this key and a hardware Root of Trust (see Section 3.2.2). A subsequent SPDM challenge–response step lets the TN-DSM prove possession of the TNDE platform key pair $(pk_{\text{plat}}, sk_{\text{plat}})$.

After successful authentication, the CASTOR Service Orchestrator and the TN-DSM establish an SPDM secure session by performing an authenticated key exchange with ephemeral Diffie–Hellman keys and finishing messages that are MAC-protected under the newly derived symmetric session keys. Once this step completes, SPDM Secured Messages are enabled and protect all subsequent traffic between the orchestrator and the TN-DSM. Within this encrypted context, the orchestrator requests the platform's measurements, and the TN-DSM assembles the previously cached measurements into a composite manifest and returns it as a single SPDM measurement payload protected under the SPDM Secured Messages session keys. These session keys are derived from an ephemeral Diffie–Hellman exchange but are authenticated and bound to the TNDE platform key pair through the SPDM handshake. The CASTOR Service Orchestrator parses the manifest, checks its components against its reference store (for example, enclave code and platform firmware measurements), and, if appraisal succeeds, records internal state that associates pk_{plat} with the validated manifest and an "enrolled" status for this TNDE platform. Subsequent SPDM connections that present the same pk_{plat} are therefore treated as re-attestations rather than new JOIN events: the orchestrator re-authenticates the TNDE via SPDM, reuses the stored registration state, and re-validates the current manifest against the reference values and policy.

Once the TNDE platform is enrolled and an SPDM Secured Messages session is in place, CASTOR uses this channel to exchange TNDI-SP control messages as SPDM Vendor Defined Messages (VDMs). In this second phase of the TNDI-SP control protocol, all per-TNDI lifecycle and configuration traffic remains cryptographically bound to the attested TNDE platform and the established SPDM session. The resulting TNDI-SP control channel between the Service Orchestrator and the TN-DSM, including TNDI discovery, onboarding, and lifecycle transitions, is described in Section 3.4.3.

The orchestrator's reference store for attestation manifests is assumed to be populated with out-of-band information from, e.g., the device or router vendor, the CASTOR service operator that selects known-good TNDE builds, or another trusted party that maintains acceptable measurement sets for a given domain. CASTOR does not prescribe a single source for these reference values, but requires that the orchestrator maintains a consistent reference database that links measurement manifests to policy decisions about enrolling or rejecting TNDE platforms.

As described in Section 3.2.2, the version 1 proof-of-concept uses a locally generated TNDE platform key pair without an enforced hardware RoT binding, so the JOIN decision relies on trust in the TNDE certificate chain and verification of the reported measurements. In version 2, CASTOR plans to introduce an explicit RoT-backed binding for this key, so that during TNDE JOIN, the Service Orchestrator can verify that both the TNDE identity and the platform evidence are anchored in the same RoT key hierarchy.

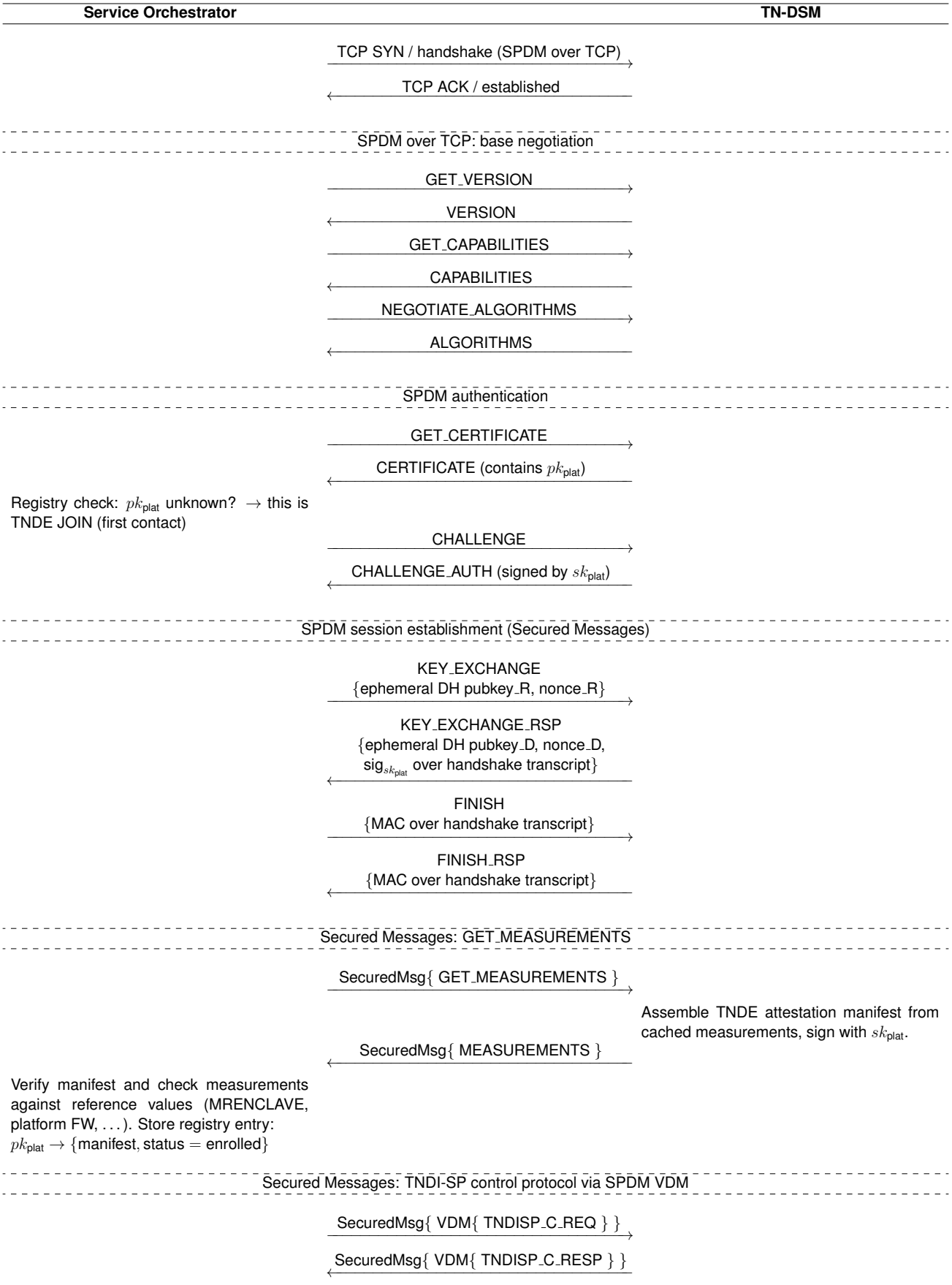

Figure 3.2: TNDE platform JOIN based on SPDM over TCP

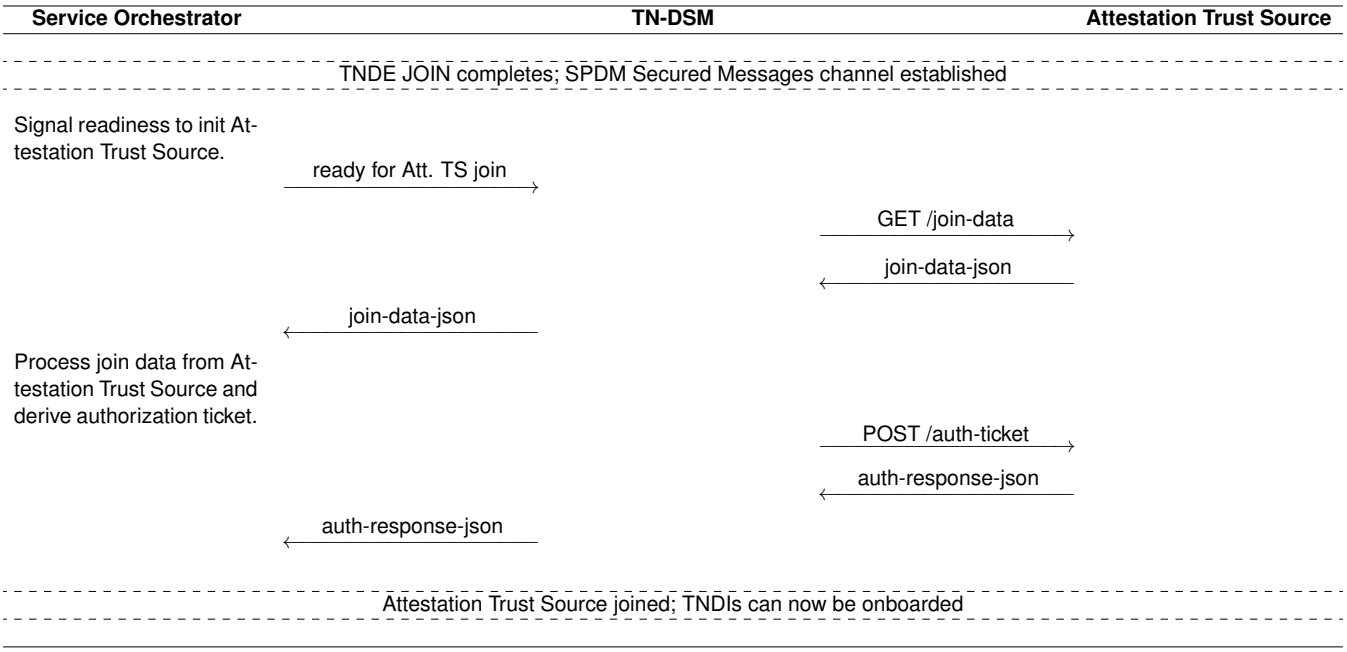

Figure 3.3: Attestation Trust Source Join sequence (between TNDE JOIN and TNDI ONBOARDING)

3.3.2 Attestation Trust Source Join

The Attestation Trust Source serves as a Trust Source for TNDIs by contributing attestation-related evidence to their trust assessment and, in addition, explores defense-in-depth techniques for the TNDE using its Configuration Integrity Verification (CIV) and layered attestation concepts, as described in Chapter 4. This subsection focuses on the latter role, in particular on how CIV and layered attestation are integrated with the TNDE JOIN and TNDI onboarding flows. CIV aims to provide device-level evidence that the TNDE's critical services are running with an authorized configuration and software version, so that only correctly configured services can produce valid attestation evidence. Layered attestation builds on this so that higher-layer trust decisions can rely on the integrity of the underlying CIV-protected services. In order to enable these mechanisms, after the TNDE has been joined into the CASTOR domain via the platform JOIN flow (Section 3.3) but before onboarding specific TNDIs, the Attestation Trust Source must be initialized. The CIV scheme is currently loosely coupled with the TN-DSM by letting the TN-DSM drive the CIV initialization flow (also called "Join" in the context of the Attestation Trust Source) as an intermediate between the Service Orchestrator and the Attestation Trust Source. In version 1, the layered attestation scheme is not implemented yet.

In version 1, as shown in Figure 3.3, the Service Orchestrator informs the TN-DSM through the established SPDM Secured Messages channel that it is ready to initialize the Attestation Trust Source. The TN-DSM then acts as a communication forwarding proxy, interfacing with the Service Orchestrator remotely via SPDM Secured Messages and with the Attestation Trust Source locally via an HTTP REST interface to relay messages between them. Once this Attestation-Trust-Source-level join has completed successfully, the CASTOR framework considers both the TNDE platform and the Attestation Trust Source ready, so that TNDIs can be onboarded and, in later versions, can benefit from the additional layered-attestation guarantees. In version 2 of the CASTOR framework, we plan to integrate both schemes more tightly with the TN-DSM and the TNDE JOIN procedure and consider exploring an extension of the CIV towards the TNDIs, with an integration into the TNDI key hierarchy managed by the TN-DSM.

3.4 TNDI ONBOARDING via TNDI-SP Control Channel

CASTOR's trusted path routing relies on device-side TNDIs (e.g., vRouters) that can be onboarded into, and continuously assessed within, a CASTOR trust network. The TN-DSM manages a per-TNDI lifecycle state machine for each TNDI on the device and exposes the TNDI-SP control channel, which the CASTOR Service Orchestrator uses to discover TNDIs, drive their lifecycle, and configure tracing and trust-policy settings so that the resulting trustworthiness evidence can be consumed by the Global TAF for topology-wide routing decisions.

3.4.1 TNDI Lifecycle State Machine

Each TNDI is associated with its own TNDI-SP lifecycle state machine that tracks its progress from initial discovery through onboarding, activation, and de-onboarding. The TN-DSM instantiates and manages these per-TNDI state machines on the device side as part of the TNDE control plane, while the Service Orchestrator indirectly drives state transitions by issuing TNDI-SP control messages over the TNDI-SP control channel, as detailed in Section 3.4.3. Conceptually, this mirrors the role of the TDI state machine in TDISP [54], but elevates the per-interface lifecycle of a PCIe device to TNDIs as first-class, router-level trust entities that are managed by the TN-DSM and controlled by the Service Orchestrator via TNDI-SP.

During static discovery (Section 3.2.4), the TN-DSM parses the TNDI discovery JSON, creates a state-machine instance for each entry, and initializes that TNDI in the UNLOCKED state as shown in Figure 3.4. In UNLOCKED, the TNDI has been discovered by the TN-DSM but is not yet claimed by the Service Orchestrator; it can still be queried for status, metadata, and static attestation manifests via read-only operations. When the Service Orchestrator decides to use a particular TNDI, it issues a registration request (`REGISTER_TNDI`) that moves the corresponding state machine to MANAGED, where configuration, tracing, trust-policy, and data-channel settings are staged and validated before activation.

Once configuration staging in MANAGED is complete, the Service Orchestrator commits this configuration via `COMMIT_TNDI_POLICY`, transitioning the TNDI to POLICYLOCKED. In POLICYLOCKED, the security-relevant configurations for this TNDI are fixed and only lifecycle operations and read-only queries are accepted. A subsequent start operation (`START_TNDI`) enters RUN, where tracing, evidence generation, and trust assessment for this TNDI are active under the locked configuration, and a corresponding stop operation returns the TNDI to POLICYLOCKED without changing that configuration. A controlled de-onboarding operation (`UNREGISTER_TNDI`) from MANAGED or POLICYLOCKED returns the TNDI to UNLOCKED and discards both staged and committed configuration, as well as any per-TNDI cryptographic state such as its TNDI identity key pair (see Section 3.4.2). Invalid or failing control operations in UNLOCKED, MANAGED, POLICYLOCKED, or RUN, as well as internal failures in the TN-DSM, can lead the TNDI state machine to ERROR, from which a reset operation (`RESET_TNDI`) returns the TNDI to UNLOCKED, ready for a fresh onboarding cycle.

This strict separation between the MANAGED, POLICYLOCKED, and RUN states provides concrete security benefits: all security-relevant configuration and the setup of long-lived keys and credentials are confined to MANAGED and the subsequent commit to POLICYLOCKED, while POLICYLOCKED and RUN operate only on a fixed, committed configuration. This simplifies the trusted code paths in the TN-DSM and makes it easier to reason about which operations are allowed in each state. For any TNDI in POLICYLOCKED or RUN, the security-relevant configurations therefore remain stable until the TNDI is explicitly de-onboarded.

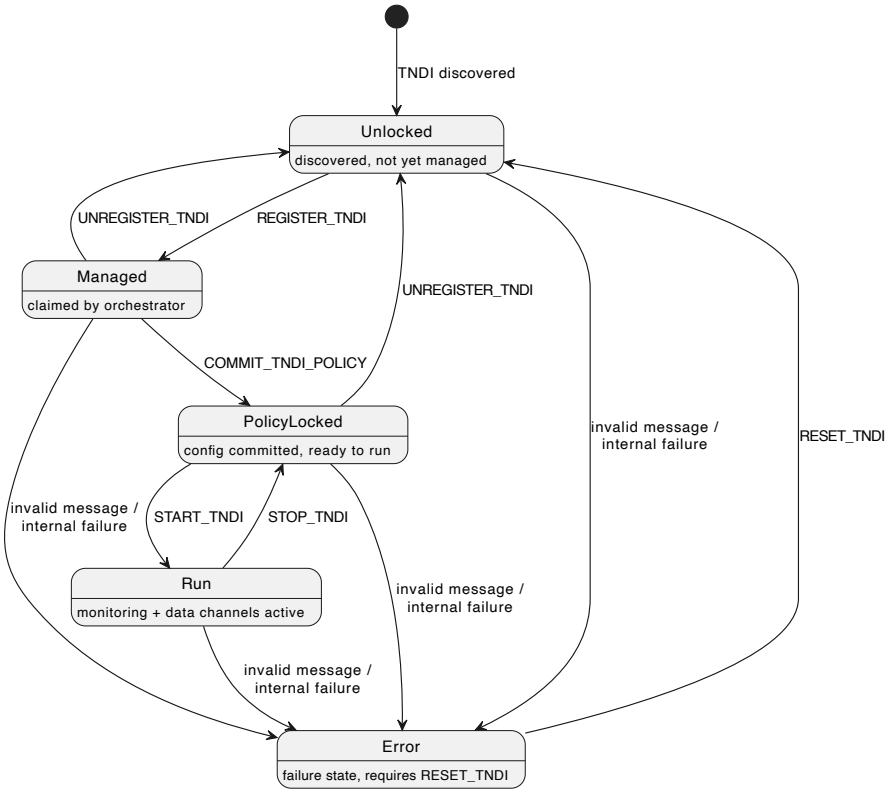

Figure 3.4: Per-TNDI lifecycle state machine managed by the TN-DSM. Transitions between the Unlocked, Managed, PolicyLocked, Run, and Error states are driven by TNDI-SP control messages from the CASTOR Service Orchestrator over the TNDI-SP control channel.

3.4.2 Per-TNDI Identities and Key Hierarchy

In TNDI-SP version 1, CASTOR adopts a deliberately simple per-TNDI key hierarchy to ease implementation while preserving clear trust anchors. For each TNDI managed by a TNDE, the TN-DSM maintains a single per-TNDI key pair $(pk_{\text{tndi}}, sk_{\text{tndi}})$ that represents the long-lived cryptographic identity of that TNDI. The Service Orchestrator learns the public part pk_{tndi} as part of the TNDI onboarding process (see Section 3.4.3), pins it, and distributes it to downstream orchestration services such as the Global TAF and Phala so that they can authenticate connections and evidence associated with that TNDI.

In version 1, the per-TNDI key pair is not yet cryptographically derived from, or explicitly certified under, the TNDE platform key pair $(pk_{\text{plat}}, sk_{\text{plat}})$. Instead, the binding is indirect: the orchestrator first attests and enrolls the TNDE platform via the TNDI-SP JOIN flow under pk_{plat}, and then learns and pins pk_{tndi} over the thereby established attested control channel. This establishes a trust chain in which per-TNDI identities are associated with, and managed under, an already attested TNDE platform, even though version 1 does not yet define an explicit certificate chain or key-derivation relation between the TNDE platform and TNDI identity key pairs.

While version 1 uses a single per-TNDI identity key pair that is only indirectly bound to the TNDE trust chain via the attested TNDI-SP control channel, CASTOR plans to explore a richer per-TNDI key hierarchy in version 2. In such an evolution, each TNDI would obtain a dedicated attestation key pair that acts as the local root of trust for that TNDI and is explicitly certified under the TNDE platform key pair. Additional functional keys such as a separate TLS client key pair or per-TNDI trace-signing keys could then be created as children of this TNDI attestation key. Designing and evaluating such an extended TNDI key hierarchy is planned to be explored as part of version 2 of the CASTOR framework.

3.4.3 TNDI Onboarding Flow (TNDI-SP Control Protocol)

The TNDI onboarding flow describes how the CASTOR Service Orchestrator discovers, enrolls, and activates a pre-deployed TNDI via the TNDI-SP control channel once the TNDE platform has been attested and joined. It assumes that the TN-DSM has already started and discovered locally configured TNDIs from the static discovery JSON, and that the orchestrator has completed the JOIN protocol to establish an SPDM Secured Messages channel for TNDI-SP control messages.

As shown in Figure 3.5, the orchestrator first queries the TN-DSM for the list of available TNDIs and their static properties. Using `GET_TNDI_LIST`, it retrieves the set of TNDIs known to the TN-DSM together with their current lifecycle state; `GET_TNDI_METADATA` then provides descriptive information such as type, vendor, product, and version for a selected TNDI. If needed, the orchestrator can additionally call `GET_TNDI_REPORT` to obtain a static attestation manifest that can characterize the TNDI definition and boot chain, as described in the static TNDI attestation report subsection below.

When the Service Orchestrator decides to onboard a particular TNDI into the CASTOR domain, it issues `REGISTER_TNDI` for that TNDI. This moves the corresponding lifecycle state machine from UNLOCKED to MANAGED (see Figure 3.4) and lets the TN-DSM initialize the per-TNDI state, including the per-TNDI identity key pair $(pk_{\text{tndi}}, sk_{\text{tndi}})$ as described in Section 3.4.2. The TN-DSM returns the public part pk_{tndi} to the orchestrator, which pins it as the long-lived cryptographic identity of that TNDI and distributes it to orchestration-layer services such as the Global TAF and Phala.

In the MANAGED state, the orchestrator incrementally stages all security-relevant configuration for the TNDI under this identity. It uses `SET_TNDI_BASE_CONFIG` to establish basic parameters (e.g., addressing or TNDI capabilities), `SET_TNDI_TRACE_CONFIG` to select and parameterize the Trace Units to be applied to this TNDI, and `SET_TNDI_TRUST_POLICY_REF` to bind the TNDI to a particular trust-policy reference that indicates which trust policy to use and where to obtain it. In version 1, the Local TAF Agent, assisted by the TPL Data Connector, will later use this trust-policy reference to fetch the corresponding trust policy via Phala from the CASTOR DLT, with the TN-DSM helping to authenticate the policy-fetch requests using the TNDI identity key pair $(pk_{\text{tndi}}, sk_{\text{tndi}})$. As part of the same staging phase, the orchestrator may also issue one or more `REQUEST_TNDI_DATA_CHANNEL` commands to request TNDI-SP data channels towards the Global TAF and Phala, as detailed in Section 3.5. Basic structural and consistency checks on these inputs are already applied during the staging calls, so that obviously malformed configurations can be rejected early.

Once configuration staging and validation are complete, the orchestrator calls `COMMIT_TNDI_POLICY` to atomically commit the staged configuration. This transitions the TNDI lifecycle state machine from MANAGED to POLICYLOCKED and prompts the TN-DSM to derive auxiliary credentials associated with the committed configuration, such as the TLS client certificate based on $(pk_{\text{tndi}}, sk_{\text{tndi}})$ that will later authenticate TNDI-SP data channels. As part of this commit, the TN-DSM and the involved TNDE services also validate that the staged base configuration, trace configuration, trust-policy reference, and requested data channels form a consistent and enforceable configuration tuple.

After the configuration has been committed, the orchestrator activates the TNDI by issuing `START_TNDI`, which moves the TNDI from POLICYLOCKED to RUN. In this state, the TN-DSM enables tracing, evidence generation, and trust assessment for the TNDI according to the committed configuration and brings up the requested TNDI-SP data channels using the per-TNDI TLS client identity, as described in Section 3.5. When the TNDI is in RUN, orchestration-layer services such as the Global TAF and Phala can continuously receive TNDI-specific traces, evidence, and trust reports over the authenticated TNDI-SP data channels. From this point on, the orchestrator manages the TNDI's lifecycle with operations such as `STOP_TNDI`, `UNREGISTER_TNDI`, and `RESET_TNDI`, while the invariant holds that the security-relevant configurations remain fixed for any TNDI in POLICYLOCKED or RUN. Throughout all lifecycle states, the orchestrator can query the TNDI's lifecycle state using `GET_TNDI_STATUS` to detect, for example, transitions into an error state.

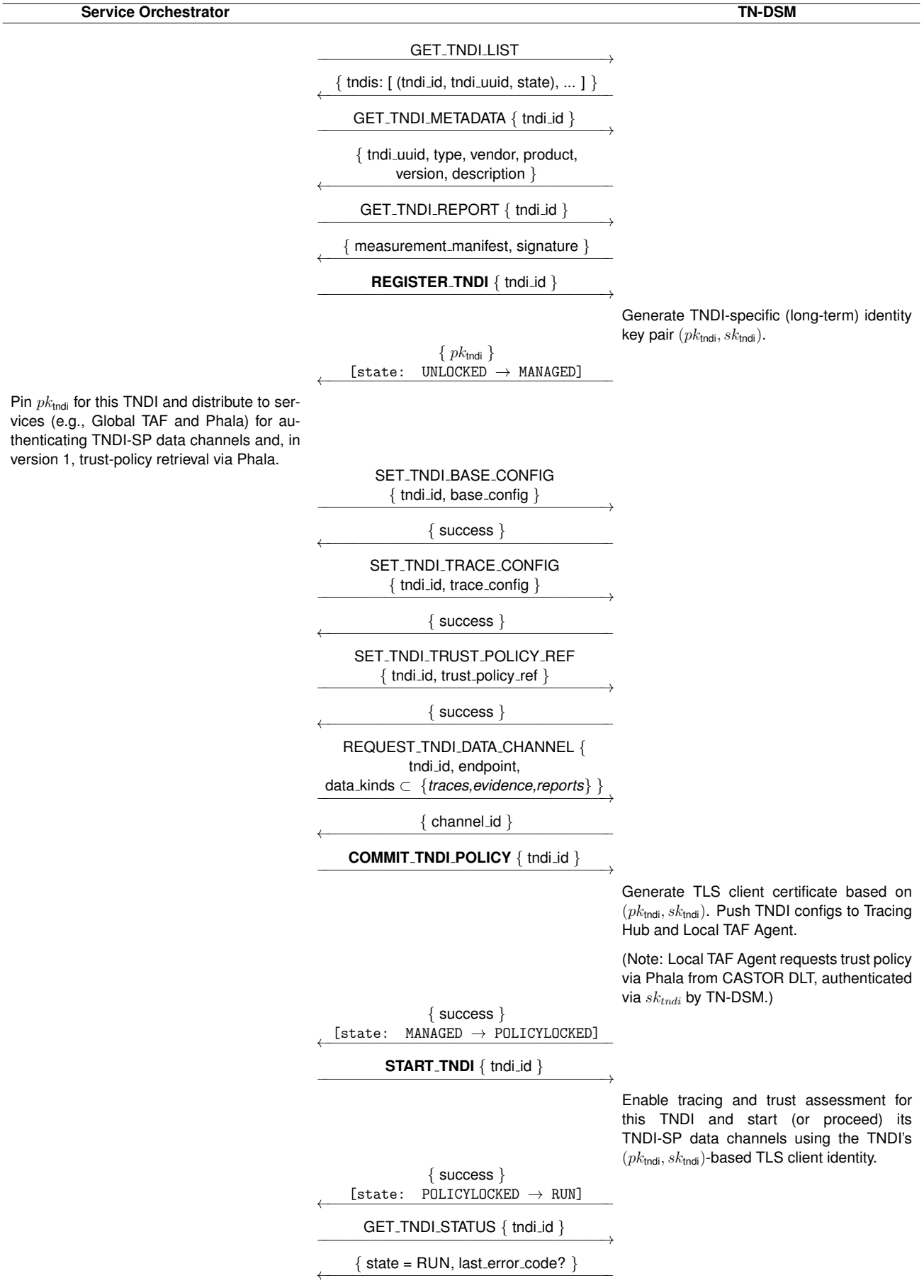

Figure 3.5: TNDI onboarding flow via the TNDI-SP control channel protocol. As a prerequisite, the flow requires that: 1. the TN-DSM discovered the TNDIs, and 2. the Service Orchestrator has performed the JOIN protocol to attest the TNDE and establish a secure channel for the TNDI-SP messages.

3.4.3.1 Static TNDI Attestation Report

In CASTOR v1, the GET_TNDI_REPORT control message is deliberately scoped to static TNDI measurements rather than runtime behavior. Concretely, it returns a manifest that is intended to characterize the TNDI definition and its boot chain (e.g., hashes over the TNDI container image, configuration, and optionally secure-boot or load-time measurements), allowing the orchestrator to appraise whether the instantiated TNDI matches an expected trusted profile before or during onboarding. The precise structure of this manifest may vary by platform and TNDI type, but its content is expected to be stable across normal operation and to change only at well-defined lifecycle events such as software or firmware updates. Dynamic runtime tracing evidence and local trust assessments, which may evolve frequently over time, are exported via TNDI-SP data channels and are therefore not included in GET_TNDI_REPORT in version 1.

3.5 TNDI-SP Data Channel

We now focus on the establishment of the TNDI-SP data channels which are used to share trust-related evidence on each TNDI (e.g., traces, reports) with the orchestration-layer services. In version 1 of the CASTOR framework, the TNDI-SP data channels are implemented based on HTTPS connections, authenticated with TLS client certificates derived from the per-TNDI identity key pairs.

3.5.1 Channel Establishment during TNDI Onboarding

Once the TNDE platform has been joined into the CASTOR domain and a specific TNDI has been registered and configured, the CASTOR Service Orchestrator can request per-TNDI data channels that carry traces, evidence, and trust reports from the device to orchestration-layer services such as the Global TAF and Phala. Figure 3.6 shows the part of the TNDI onboarding flow that covers TNDI key distribution to the orchestration layer and the subsequent establishment of these TNDI-SP data channels.

As part of configuring a TNDI, the orchestrator issues one or more REQUEST_TNDI_DATA_CHANNEL commands to the TN-DSM, specifying for each requested data channel the target endpoint (e.g., Global TAF or Phala, including network address and optional endpoint-authentication metadata) and the kinds of data to be transported (e.g., traces, evidence, and/or reports). The TN-DSM allocates local channel identifiers for the requested endpoints and returns them to the orchestrator, which can use these identifiers for bookkeeping and later status queries.

When the orchestrator commits the staged configuration via COMMIT_TNDI_POLICY, the TN-DSM generates a TLS client certificate whose key pair is the per-TNDI identity key pair $(pk_{\text{tndi}}, sk_{\text{tndi}})$. By this point, the orchestrator has already learned pk_{tndi} during REGISTER_TNDI and has distributed this public key to the Global TAF and Phala so that they can pin it as the cryptographic identity of the TNDI. Using the TLS client certificate based on $(pk_{\text{tndi}}, sk_{\text{tndi}})$ on the device side and the pinned copy of pk_{tndi} on the orchestration side, the orchestration-layer services can later authenticate each TNDI-SP data channel (TLS connection) as belonging to that specific TNDI.

Finally, when the orchestrator transitions the TNDI into the RUN state via START_TNDI, the TN-DSM brings up the configured TNDI-SP data channels by initiating TLS handshakes towards the Global TAF and Phala using the TNDI-specific TLS client certificate. Each orchestration-layer endpoint verifies the TLS client certificate against its pinned copy of pk_{tndi} and, upon successful verification, completes the handshake, thereby establishing one or more authenticated TNDI-SP data channels for that TNDI. Once these TLS connections are in place, the TN-DSM enables the tracing, evidence generation, and trust assessment for the TNDI and can start exposing traces, evidence, and trust reports over the established channels to the orchestration-level services.

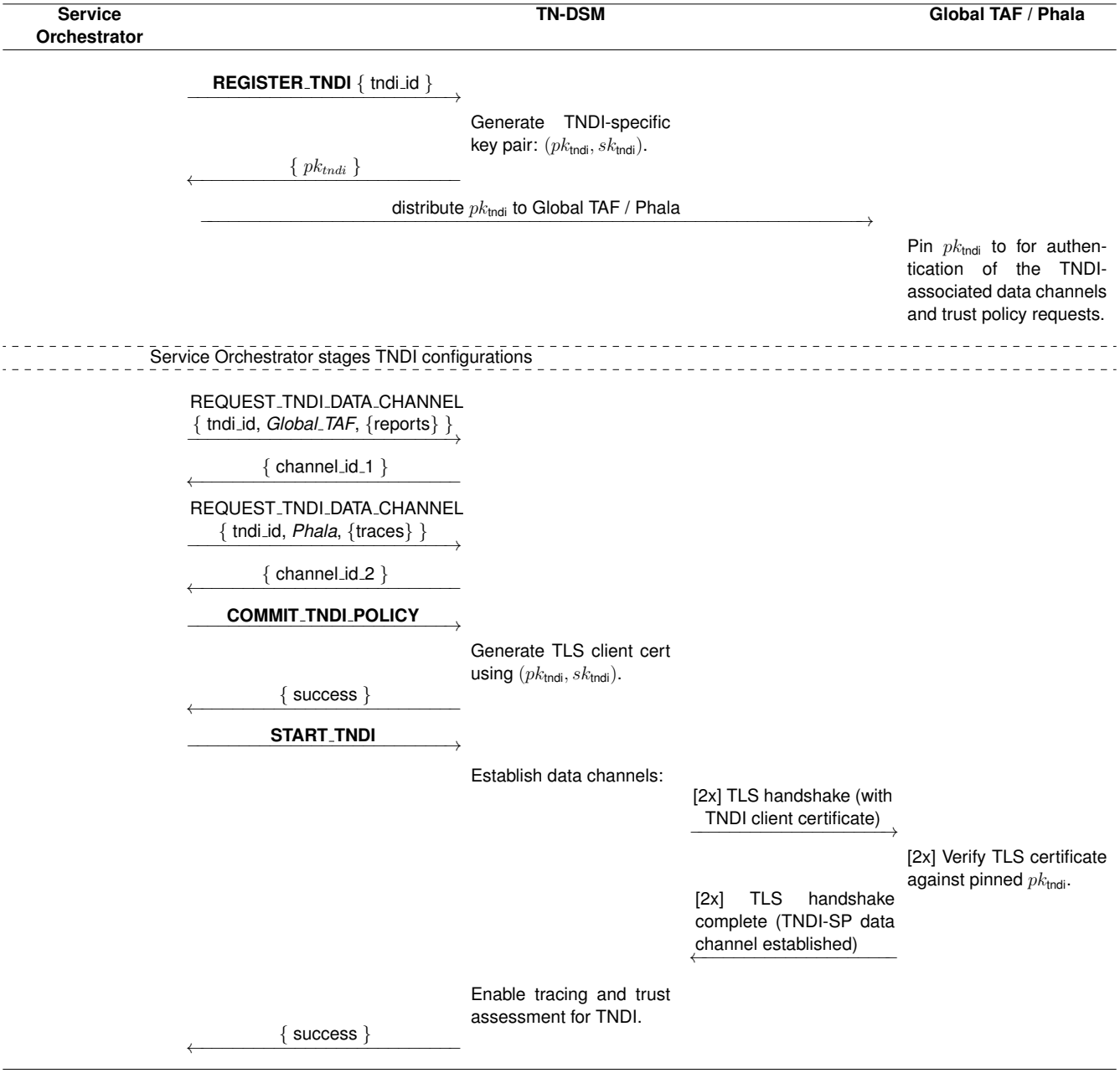

Figure 3.6: Snippet of TNDI onboarding flow highlighting the TNDI key distribution to the orchestration layer and the TNDI-SP data channel establishment to the Global TAF and Phala (CASTOR DLT).

3.5.2 Runtime Evidence and Trace Sharing via TNDI-SP Data Channels

In version 1, TNDI-SP data channels are realized as long-lived HTTPS connections that expose simple REST-style interfaces for pulling traces and pushing trust reports. For trace sharing, Trace Units on the device deliver per-TNDI trace batches to the TN-DSM, which the Phala component then pulls over the TNDI-SP data channel by issuing HTTP requests. Because this channel is protected by TLS and authenticated with the TNDI identity key sk_{tndi}, the resulting TLS session keys – and therefore the sent traces - are indirectly bound into the same trust chain as the attested TNDE platform and the per-TNDI identity. For local trust assessment, the Attestation Trust Source obtains trace batches from the Trace Units (see Section 3.7.7) and processes them to derive evidence, which it exposes to the Local TAF Agent. The Local TAF Agent verifies this evidence and computes a trust report that summarizes the current trust level of the TNDI, then sends this report to the TN-DSM over a device-local channel. In turn, the TN-DSM forwards the trust report to the Global TAF over the per-TNDI TNDI-SP data channel.

3.6 Tracing Hub

In version 1 of the CASTOR framework, the Tracing Hub has a deliberately limited but well-defined role in the TNDE runtime flows. It acts as the central configuration manager for Trace Units on the device: the TN-DSM pushes per-TNDI trace configurations to the Tracing Hub, and the Tracing Hub in turn applies these configurations to the available Trace Units during TNDI onboarding. This lets the CASTOR Service Orchestrator control which tracing backends are active for each TNDI without exposing Trace Unit–specific details into the TNDI-SP control protocol.

The Tracing Hub is not yet involved in the trace transport path or in exposing trace batches to other TNDE components. Instead, Trust Sources obtain trace batches directly from the Trace Units. Later versions of CASTOR are expected to give the Tracing Hub a more active role in trace dissemination and coordination, as outlined in the following subsections.

3.6.1 Static Discovery of Trace Units

In version 1, the TNDE discovers Trace Units using a static Tracing Infrastructure Profile that is represented as a JSON document and shipped with the device. This profile defines the set of Trace Units available on a given platform, including their identifiers, types, default parameters, and internal endpoints, as exemplified in Listing 3.2. The profile is consumed by the Tracing Hub so that it can instantiate and configure Trace Units correctly, and by Trust Sources so that they can locate the Trace Units' export endpoints and obtain trace batches directly.

Logically, the Tracing Hub acts as the registry for Trace Units when applying per-TNDI trace configurations, and it validates all Trace Unit identifiers and configuration parameters against this static profile. In the current prototype, however, the Tracing Hub does not yet expose Trace Units to other TNDE components or act as a discovery proxy. Instead, Trust Sources parse the same Tracing Infrastructure Profile and use the configured export endpoints to open host-local TCP connections to Trace Units and pull trace batches for their analysis. This split model keeps version 1 simple while still allowing a future design in which Trace Units dynamically register with the Tracing Hub and the Tracing Hub becomes the single TNDE interface towards the Trace Units.

The Tracing Infrastructure Profile enumerates all Trace Units supported by a given TNDE platform and captures their key properties. At a minimum, each Trace Unit entry includes a stable identifier, a human-readable name, a tracer type, and basic configuration hints such as default enablement, default batch size, and one or more device-local configuration or export endpoints, as illustrated in Listing 3.2. This format enables platform vendors to declaratively describe both syscall-level Trace Units (such as the eBPF-

```
{
  "version": "1.0",
  "tracing_units": [
    {
      "id": "tu-ebpf-syscalls-all",
      "name": "ebpf-syscalls-all-pids",
      "type": "ebpf",
      "default_enabled": false,
      "default_batch_size": 128,
      "config_endpoint": "tcp://127.0.0.1:8000",
      "export_endpoint": "tcp://127.0.0.1:9000",
      "encoding": "cbor",
      "process_filter": {
        "mode": "all"
      }
    },
    {
      "id": "tu-ebpf-spf-control-plane",
      "name": "ebpf-spf-control-plane",
      "type": "ebpf",
      "default_enabled": true,
      "default_batch_size": 256,
      "config_endpoint": "tcp://127.0.0.1:8001",
      "export_endpoint": "tcp://127.0.0.1:9001",
      "encoding": "cbor",
      "process_filter": {
        "mode": "names",
        "names": [ "isis_uv", "lwm_manager" ]
      }
    }
  ]
}
```

Listing 3.2: Example of a static Tracing Infrastructure Profile for a TNDE platform

based tracer described later in this chapter) and Trace Units that emit higher-level control-plane events, while remaining extensible so that additional tracer types (e.g., VM-introspection-based or hardware-assisted units) can introduce further fields without affecting the Tracing Hub's core discovery and configuration model.

3.6.2 Runtime Configuration of per-TNDI Trace Units

At runtime, the Service Orchestrator configures which Trace Units are active for a specific TNDI by providing a trace configuration that references Trace Units by their identifiers and optionally overrides their default parameters. The TN-DSM receives this configuration over the TNDI-SP control channel, translates it into a device-local runtime configuration for the Tracing Hub, and forwards it together with the TNDI identifier and any platform-specific target information such as the container name. In version 1, the TN-DSM obtains this target information from the static TNDI discovery file that it loads during TNDE startup.

The Tracing Hub resolves each referenced Trace Unit identifier against the static Tracing Infrastructure Profile (Section 3.6.1), applies any per-TNDI overrides (such as enablement or batch size), and instantiates or updates the corresponding Trace Units accordingly. For example, for the eBPF-based Trace Unit described in Section 3.7, the Tracing Hub may enable both the generic syscall and process-lifecycle tracing baseline and the deeper kprobe and uprobe hooks that target routing control-plane logic for one TNDI, while restricting another TNDI to the syscall baseline only. This allows the orchestrator to request richer introspection for selected TNDIs, such as those carrying sensitive traffic or participating in critical parts of the topology, without incurring the cost of deep tracing everywhere. In the current version 1 proof-of-concept implementation, only a subset of these configuration options is exercised, but the overall interface between the TN-DSM and the Tracing Hub is designed to support finer-grained policies in later iterations.

In version 1, the Service Orchestrator relies on a pre-agreed set of Trace Unit identifiers defined by the platform vendor, and the TN-DSM treats these identifiers as opaque strings that it simply forwards to the Tracing Hub. The Tracing Hub validates all referenced Trace Units and parameters against the local Tracing Infrastructure Profile and reports any configuration that cannot be applied back to the TN-DSM. The TN-DSM then returns a corresponding error to the Service Orchestrator via the existing TNDI-SP control responses, consistent with the TNDI onboarding and lifecycle flows. For version 2, CASTOR plans to explore extending this model so that the Service Orchestrator can issue dynamic runtime configuration changes to Trace Units on a per-TNDI basis over the TNDI-SP control channel. This would allow tracing to react to changes in the network topology or trust requirements, rather than relying only on static per-TNDI configurations established during onboarding.

3.6.3 CBOR-based Encoding of Traces

Independent of how traces are transported between components, version 1 of the CASTOR framework supports a common CBOR-based encoding for trace batches. Each batch consists of metadata that identifies the emitting Trace Unit and, where applicable, the associated TNDI, followed by a collection of trace records whose concrete structure depends on the Trace Unit type. This model is intended to remain flexible enough to carry low-level syscall and process-lifecycle events, higher-level control-plane events, and other evidence types that future Trace Units may produce, such as hashes over TNDI runtime configurations.

In the current prototype, Trace Units themselves are responsible for encoding their trace batches in CBOR and exposing them over device-local TCP interfaces. Trust Sources act as clients of these export endpoints and pull batches on demand for further analysis and evidence derivation. A concrete instantiation of this model is given by the eBPF-based Trace Unit described in Section 3.7, which explains how syscall and control-plane tracing events are encoded and exposed in the version 1 proof-of-concept. An example for the CBOR data model of a syscall trace batch, in JSON notation, is shown in Listing 3.3 in the eBPF Trace Unit section.

In version 1, the Tracing Hub is not involved in the trace transport path or in CBOR encoding and only participates in Trace Unit configuration. For version 2, CASTOR plans to explore a more active role for the Tracing Hub in this path, for example by participating in trace signing or acting as an encoding and forwarding point for the trace batches towards Trust Sources for evidence generation and towards the TN-DSM for exposure via TNDI-SP data channels.

3.7 eBPF Trace Unit: libbpf Syscall Tracer

This section describes the concrete implementation of the eBPF-based Trace Unit used in the current CASTOR prototype. Building on the tracing model introduced in the TNDE services and Tracing Hub

sections, it focuses on a single, concrete deployment: an eBPF-based Trace Unit that performs system-level tracing with targeted control-plane introspection, is configured by the Tracing Hub, and exposes trace batches directly to Trust Sources. The following subsections cover the placement of the tracer, the events it collects, the filtering mechanism used to restrict traces to the TNDI workload, the choice of libbpf over alternative frameworks, and the format of the produced trace stream. As part of version 2 of the CASTOR framework, we will investigate further tracing opportunities with the eBPF tracer and look into additional Trace Units (e.g., an out-of-TNDI Trace Unit) as well as tighter integration with the Tracing Hub for trace signing and forwarding. In Chapter 7, we describe experimental observations from applying the eBPF Trace Unit for tracing of in-TNDI route configuration updates.

3.7.1 Placement and scope of the prototype

In the CASTOR version 1 prototype, each TNDI is instantiated as a pre-deployed Cisco XRd vRouter container on the host system, and the TNDE services run as user-space processes on the same host operating system. The host OS process and container isolation therefore forms the isolation layer between the TNDE and the TNDIs, as described in Section 3.1. The eBPF Trace Unit follows this model: it consists of a user-space process running on the host OS alongside the TNDE services and the XRd containers, which loads and attaches eBPF programs to the *host* kernel. From this position, the tracer has visibility into all system calls and process lifecycle events on the host. In order to restrict the tracing to a given TNDI's workload, the eBPF program applies cgroup-based filtering so that only activity belonging to the XRd container and its descendants is emitted. This placement keeps the design simple, because it does not require separate tracer components inside the XRd containers and relies only on standard interfaces of the host kernel. The tradeoff is that the tracer shares the host kernel with both TNDE and TNDIs and is therefore subject to the same security and performance considerations as described for in-host eBPF Trace Units in the conceptual architecture (Deliverable 3.1). In version 1, the Trace Unit interfaces with the Tracing Hub for configuration and exposes per-TNDI trace batches over device-local HTTP/TCP endpoints to Trust Sources, which pull CBOR-encoded trace batches for evidence generation.

For version 2 of the CASTOR framework, we plan to explore an alternative instantiation in which each TNDI runs as a virtual machine (VM) with its own guest Linux kernel and XRd container, and the eBPF Trace Unit executes inside that guest. In that model, the guest kernel would provide the tracing hooks and the Trace Unit would run as part of the TNDI's software stack rather than on the host, while the TNDI-SP control and data channel protocols would remain unchanged.

3.7.2 Tracing primitives and event types

The implementation employs a hybrid tracing architecture that combines a stable system-level baseline with targeted application-level introspection. For the system-level baseline, the eBPF program attaches to three core kernel tracepoints:

- `raw_syscalls:sys_enter` and `raw_syscalls:sys_exit`, which fire on every system call entry and exit, and
- `sched:sched_process_exit`, which fires when a process or thread exits.

Using these tracepoints provides a stable, ABI-oriented interface that is maintained across kernel versions and avoids dependence on internal kernel symbols, which simplifies long-term maintenance when the host OS or the XRd stack is updated.

Tracepoints alone, however, do not provide sufficient semantic detail to monitor networking or control-plane-specific operations. To obtain deeper insight into the routing logic without compromising the

stability of the baseline, the Trace Unit augments this design with selected kernel probes (`kprobes`) and application-level user-space probes (`uprobes`). To identify suitable probe points in the Cisco XRd routing stack, we performed black-box system tracing and correlation of trace events, observing which processes and functions become active during routing updates and control-plane reconfiguration (also see Chapter 7). Based on these observations, we selected a small set of core internal functions responsible for Shortest Path First (SPF) calculations and instrument them using `uprobes`, in particular `igp_lscache_update_link`, which extracts link-state attributes, and `igp_lscache_update_node`, which finalizes next-hop calculations and incorporates custom metrics such as trust quantization of peer nodes. Crucially, tracing `igp_lscache_update_node` allows the framework to natively observe and verify how the router incorporates custom metrics, such as the trust quantization of peer nodes, into its pathing decisions.

To place these user-space executions into their system-level context, the Trace Unit simultaneously deploys `kprobes` targeting a specific subset of system calls, including `recvfrom`, `futex`, `epoll_wait`, `write`, and `sendto`. Empirical testing indicates that this particular sequence forms a characteristic control-plane update pattern, which the Trace Unit can use as a "golden reference graph" for a healthy routing control-plane update. By correlating the `uprobe` events (internal SPF logic) with the `kprobe` events (network and synchronization system calls), the Trace Unit can derive strong evidence of a complete Routing Information Base (RIB) to Forwarding Information Base (FIB) update cycle: a routing update was received, processed by the routing daemon, and its result applied to the forwarding plane.

From these hooks, the Trace Unit emits a fixed set of event types that characterize both individual system or function calls, and the process lifecycle. Every filtered system call, kprobe, and uprobe results in a respective *syscall*, *kprobe*, and *uprobe* event. The first system call observed for a given thread ID is additionally marked as a *new* event, which allows downstream analysis (e.g., by Trust Sources) to identify thread birth and to correlate threads with their parent via *clone* events. A *clone* event is emitted at `sys_enter` of `clone`, `fork`, `vfork`, or `clone3` on the parent side. Emitting at entry rather than exit ensures that the timestamp of the clone event is strictly before any system call made by the child, which helps correct parent–child correlation in the trace. When an `execve` or `execveat` system call succeeds, the Trace Unit emits an *exec* event from the `sys_exit` hook, including the new process name (the *comm* field) as seen by the kernel after the exec. Whenever a process or thread exits, the `sched:sched_process_exit` tracepoint yields an *exit* event.

Each event carries a monotonic kernel timestamp, the process and thread IDs as seen by the kernel (in the host PID namespace), the system call number where applicable, and, in a subset of cases, the process name (*comm*). To limit overhead and buffer pressure, the process name is included only on the first event per thread, on clone and exec events, and on exit; for all other syscall events, the user-space component maintains a small per-TID cache and resolves the name when producing the output stream. In addition, the user-space component takes a *process snapshot* at trace start and at trace end, listing the PIDs, TIDs, command names, and related metadata of processes present in the container at those times. These snapshots allow consumers of the trace to relate the event stream to the actual process set of the TNDI and to detect threads that may have been created or destroyed outside the observed event window.

3.7.3 Cgroup-based filtering and identification of the XRd container

To restrict the trace to the XRd container, and thereby to the TNDI's workload, the eBPF program filters every event by the cgroup of the executing task. The current tracer targets hosts using cgroup v2 (unified hierarchy), as used by typical container runtimes on current Linux distributions. The kernel exposes `bpf_get_current_cgroup_id()`, which returns a stable 64-bit cgroup ID for the current task's cgroup in the default hierarchy. On cgroup v2, that ID matches the 64-bit value obtained from `stat()` on the corresponding directory under `/sys/fs/cgroup` (the inode number of that directory in the cgroup filesystem),

so user space and BPF use the same numeric key without a separate conversion step. The Trace Unit uses this as the primary filter: at startup, the user-space component resolves the container's cgroup path (e.g., by inspecting `/proc/<pid>/cgroup` for a process inside the container) and recursively walks that directory tree under `/sys/fs/cgroup`, recording that identifier for each cgroup directory. Those values populate a BPF hash map (`target_cgroups`). On each tracepoint hit, the eBPF program compares `bpf_get_current_cgroup_id()` to the map: if it matches an allowed entry, the event is emitted; otherwise it is discarded.

The container's cgroup path can be specified either by providing a container identifier (e.g., a Docker container ID or name, which is resolved to a PID and then to a cgroup path) or by providing the cgroup path directly. For Docker-style runtimes, the init process of the container may reside in a *child* cgroup of the main container scope, such as `docker-<id>.scope/init.scope`. The implementation therefore walks up from the init process's cgroup path until it finds a directory whose name matches the Docker scope pattern (`docker-*.scope`) and uses that directory as the root for collecting cgroup IDs across the container subtree, so that the entire subtree is represented in `target_cgroups` rather than only the init process's cgroup; if no such scope is found, the path obtained from `/proc/<pid>/cgroup` is used as the collection root. The three subnamespaces of the Cisco XRd vRouter container (mentioned in Section 3.7.1) correspond to cgroup subtrees under that root; recursive descent adds the cgroup ID of each directory under the resolved scope to the map, so all of them are covered as long as they lie under that root. Support for dynamically created child cgroups, e.g. spawned during XRd vRouter startup, may be explored at a later stage as part of the CASTOR version 2 framework, for example using cgroup or namespace event listeners.

3.7.4 Choice of libbpf and implementation tradeoffs

The Trace Unit is implemented as a C program using libbpf rather than a scripting-based framework such as BCC with Python. The eBPF programs are written in C and compiled ahead of time with Clang targeting the BPF instruction set. The resulting object file is loaded by libbpf, which populates the maps and attaches the programs to the tracepoints and probes. The user-space side is a single-threaded C program that reads from the per-CPU perf event buffer and writes encoded trace records to the output.

This design was chosen for performance and operational reasons. First, pre-compiled BPF avoids runtime compilation, e.g. by an embedded LLVM, which reduces startup time and memory footprint and removes the need for heavy toolchain dependencies on the device. Second, the event-processing loop runs in native code without a scripting runtime or global interpreter lock; in high-load scenarios (e.g., containers with many threads and high syscall rates), a libbpf-based tracer can sustain substantially higher event throughput than a typical BCC/Python-based equivalent. Higher throughput reduces the risk of perf buffer overruns and dropped events, which is important for CASTOR because dropped events can create gaps in the trace and may affect the ability of Trust Sources to generate complete or accurate evidence. The Trust Assessment Framework already relies on subjective logic; in version 2 of the CASTOR framework, we plan to explore modeling such event loss explicitly in this framework, e.g. by increasing an uncertainty score when traces indicate dropped events. Third, a single compiled binary is easier to deploy and to reason about in terms of the trusted computing base than a stack that includes a scripting runtime and dynamic compilation.

The downside is that the tracer is tied to the target kernel at build time (via kernel headers or BTF); changing the set of tracepoints or the filter logic requires recompilation. For the current prototype, where the TNDI environment (host kernel and container layout) is under control of the deployment, this tradeoff is acceptable. Future work may explore a middle ground (e.g., loading different pre-compiled BPF objects based on configuration) or deeper integration with the Tracing Hub's configuration interface so that the Hub can select among a small set of tracer variants without requiring full runtime compilation.

```
"meta": {
  "v": 1,
  "tndi": "tndi-1234",
  "trace_unit_id": "tu-ebpf-syscalls-all"
},
"data": {
  "values": [
    [ "new",     25616532664171, 2959, 3049, "evm_signal", "gsp", "write" ],
    [ "new",     25616532687869, 2959, 2959, "gsp",        "gsp", "read"  ],
    [ "syscall", 25616532695666, 2959, 2959, "gsp",        "gsp", "futex" ],
    [ "syscall", 25616532717023, 2959, 2959, "gsp",        "gsp", "epoll_wait" ]
  ]
}
```

Listing 3.3: Example CBOR data model for a syscall trace batch (JSON notation)

3.7.5 Event format, encoding, and output

Events are passed from the eBPF program to user space via a per-CPU perf event array (ring buffer). The kernel writes fixed-size records into the buffer. The user-space component polls the buffer, decodes each record, resolves the system call number to a symbolic name (using the kernel's syscall table when available or a fallback list), and fills in the process name from the event payload or from the TID cache. The output can be emitted in three formats: plain text (for human inspection and debugging), JSON Lines (one JSON object per line, for downstream tooling and integration), or CBOR (Concise Binary Object Representation), which is compact and in line with the trace encoding approach for CASTOR trace records described in the conceptual architecture (Deliverable 3.1). Each trace record includes the event type (`syscall`, `new`, `clone`, `exec`, `exit`, `kprobe`, or `uprobe`), the timestamp, PID and TID, the process name when available, and the system call name for syscall-related events and the function name for kprobe- or uprobe-related events respectively. If the perf buffer overflows, the kernel invokes a lost-event callback in user space: the implementation can write an explicit error record (e.g., `type: error`, `error: lost_events`, `count: N`) into the stream so that consumers (i.e., the Trust Sources) are aware of gaps and can account for them in evidence generation or reporting.

For illustration, Listing 3.3 shows an example CBOR data model, in JSON notation, for a syscall trace batch produced by the eBPF Trace Unit. The `meta` section identifies the TNDI and the Trace Unit instance, while the `values` array encodes individual events as positional arrays that start with the event type and timestamp and then list PID, TID, process name, executable name, and syscall name.

In version 1, the Trace Unit emits unsigned CBOR batches over device-local HTTP/TCP endpoints towards Trust Sources. When these batches are forwarded over the per-TNDI TNDI-SP data channels, they become indirectly bound to the TNDI through the data channel's TLS session keys and the underlying TNDI identity. Defining explicit per-TNDI trace-signing keys to authenticate trace batches via signatures, beyond this channel-level binding, is planned for exploration in version 2 of the CASTOR framework.

3.7.6 Integration with the Tracing Hub

In the version 1 CASTOR prototype, the eBPF Trace Unit is configured by the Tracing Hub but exposes trace batches directly to Trust Sources rather than streaming them through the Hub. The Tracing Hub discovers the Trace Unit via the static tracing infrastructure profile (Section 3.6.1), establishes a device-local control connection to its configuration endpoint, and applies per-TNDI trace configurations as received from the TN-DSM. These configurations can, for example, enable or disable the generic syscall- and process-lifecycle tracing, select specific kprobe or uprobe sets, or adjust batch sizes for a given TNDI. The Trace Unit, in turn, uses these parameters to adjust its filtering and event selection for the processes that

belong to that TNDI. The resulting CBOR-encoded trace batches are consumed by local Trust Sources and, where configured, are also made available to the TN-DSM so that they can be forwarded over the per-TNDI TNDI-SP data channels towards orchestration-layer components such as Phala and the DLT.

3.7.7 Sharing Traces with the Trust Source

In the current prototype, Trust Sources obtain traces from the eBPF Trace Unit over a device-local TCP interface. Conceptually, the Trace Unit exports CBOR-encoded trace batches and a Trust Source acts as a client of this export endpoint, typically by issuing simple request–response interactions over a host-local TCP connection. The concrete choice of application protocol (e.g., HTTP versus a custom framing) and whether traces are pulled by Trust Sources or pushed by the Trace Unit is kept flexible and may be revisited in version 2 of the CASTOR framework.

Chapter 4

CASTOR Attestation Service

4.1 CASTOR Tiered Security

Security-by-design is not merely a feature, but a fundamental architectural principle that requires systems to be structured as hierarchies of distinct privilege levels rather than as monolithic entities. This paradigm, rooted in early multiprogramming systems such as Multics [30], promotes the decomposition of systems into layers separated by hardware-enforced boundaries. Ideally, such an architecture can be visualised as an inverted pyramid or a set of concentric circles with progressively decreasing radii, where the innermost layers—forming the Trusted Computing Base (TCB) [42]—are deliberately kept minimal, static, and rigorously verified, exposing only narrow interfaces to the outer layers. Reducing complexity at the core is a key security property, as it ensures that the most privileged components remain predictable and amenable to formal verification, even in the presence of adversarial behaviour in the surrounding, less trusted layers.

The CASTOR architecture follows this architectural principle and adopts a **three-tiered security model**, in which the TCB is deliberately designed to be minimal, static, and tightly verified, exposing only narrow interfaces to external components. This design enables the system to progressively extend trust from the device level to the platform software stack—where trust assumptions are weakened—and ultimately to the network path level. In parallel, the CASTOR architecture follows the principle that only a minimal set of components should be inherently trusted, while all other components are continuously verified through attestation mechanisms. Figure 4.1 illustrates the CASTOR tiered security model and the relationships among its different layers. Together, these three tiers enable CASTOR to establish end-to-end trust from the underlying device, to infrastructure element and routing path. The three security tiers are presented from bottom to top and are distinguished by different colours, as follows:

- **Tier 1: Device-level Configuration Integrity Verification (CIV)** — The first tier of the CASTOR tiered security model focuses on device-level configuration integrity verification, implemented through the Configuration Integrity Verification (CIV) mechanism. The goal of this tier, as already introduced in D3.1 [24], is to establish a reliable trust anchor at the device level (e.g., vRouter A), ensuring that the fundamental components of a router or network element operate as expected. More specifically, CIV provides the verifiable evidence on the static properties of deployed TNDI router functions during runtime. Static properties refer to configuration artifacts that describe the expected state of critical components of the routing infrastructure. These properties may include, for example: (a) the hash of the entire virtualized container hosting the routing plane, and (b) the integrity state of routing-related data structures (e.g., NF table, FIB, RIB). To preserve privacy and prevent implementation disclosure attacks, CASTOR does not expose raw attestation evidence. Instead, evidence is signed and verified using policy-restricted attestation keys (key restriction usage policies), ensuring that sensitive configuration details are not revealed to external verifiers. These

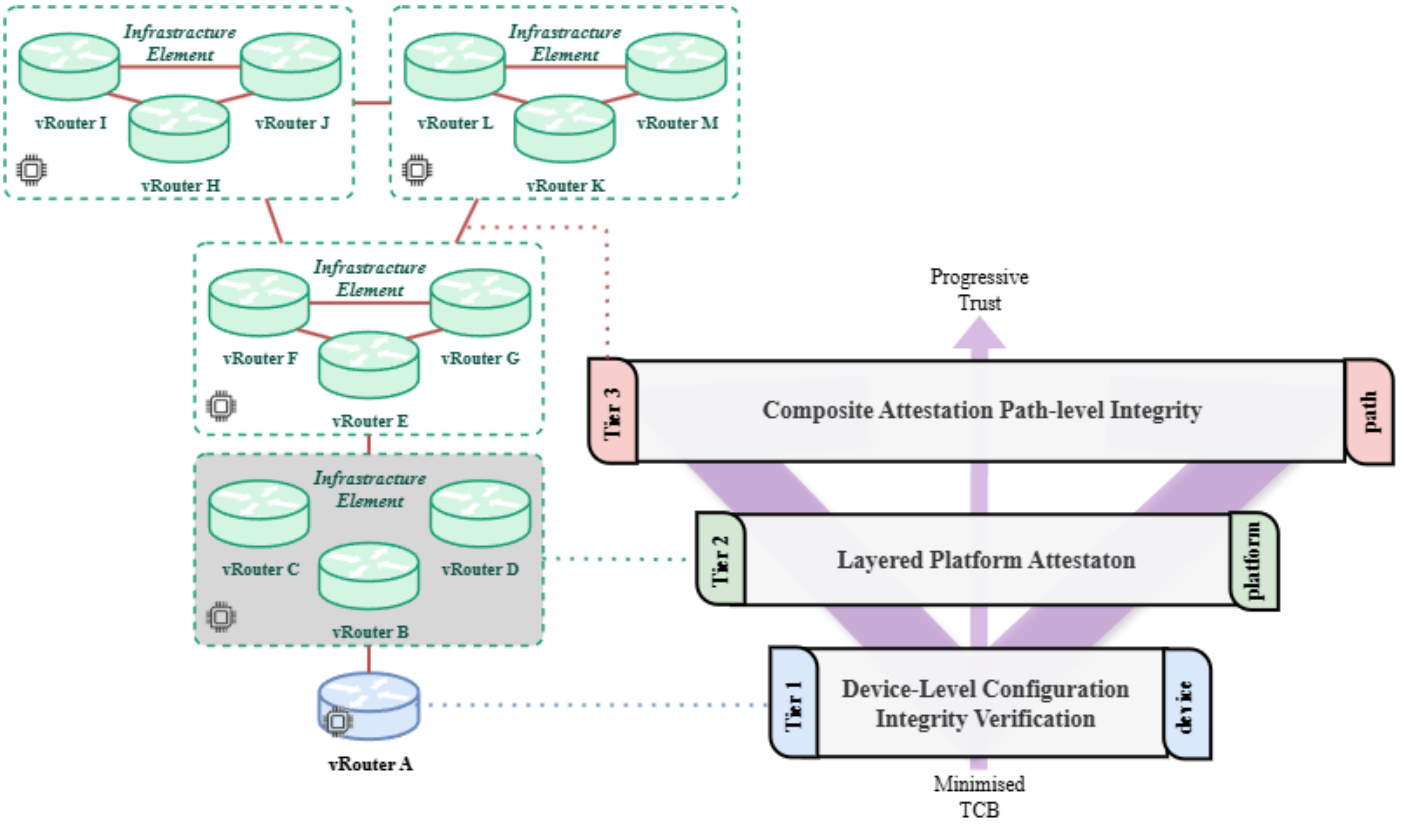

Figure 4.1: CASTOR Tiered Security

policies ensure that attestation keys can only be used under predefined conditions, thus strengthening the security guarantees of the attestation process. Thus, this implicit attestation scheme allows a prover TNDI (e.g., a vRouter) to share evidence to an external verifier (e.g., the CASTOR Service Orchestrator) without disclosing sensitive information pertaining to its platform or router state. On top of that, this scheme allows arbitrary verifiers to assess the integrity of other TNDIs (e.g., peer vRouter) without knowing their internal state, thereby alleviating critical trust assumptions regarding the verifiers' trustworthiness. Section 4.3 documents the detailed attestation protocol with enhanced capabilities on providing additional evidence on the key restriction usage policy that has been enforced.

- **Tier 2: Layered platform attestation with weakened trust assumptions** — The second tier of the CASTOR tiered security model focuses on the infrastructure element comprising some vRouters and the four-layer attestation methodology used in CASTOR, where each layer provides distinct isolation properties. The four layers are: (a) Layer 0: Hardware Root of Trust, (b) Layer 1: Hardware-Isolated Tracing & Attestation, (c) Layer 2: Trust Evaluation Plane, and (d) Layer 3: Runtime TNDI Layer. More specifically, Layer 0: Hardware Root of Trust protects long-term secrets and enforces cryptographic key usage. Layer 1: Hardware-Isolated Tracing & Attestation is the core attestation logic. Layer 2: Trust Evaluation Plane focuses on evaluating the traces and telemetry data from Layer 1. Layer 3: Runtime TNDI Layer is the untrusted vRouter and hence the most dynamic component. CASTOR does not attempt to attest every component of the system stack. Instead, the layered design focuses on verifying the components that act as control points for launching and managing other layers (e.g., Layer 2 and Layer 3). Once these control points are verified, the trust relationship can be extended transitively to the components they manage. The primary motivation behind this design choice is to minimize the Trusted Computing Base (TCB). By limiting the set of components that must be directly attested, the architecture reduces complexity and enhances efficiency while maintaining strong security guarantees. This layered attestation approach also aligns with the requirements defined in CASTOR Deliverable D3.1 [24], which emphasizes the

importance of minimizing trusted components while ensuring the integrity of critical orchestration elements within the network stack. Section 4.4 describes thoroughly this layered platform attestation architecture focusing on the high-level protocol overview and the key technologies used (e.g., trust decomposition, memory introspection and sharing using zero-knowledge proofs).

- **Tier 3: Composite attestation for path-level verification** — The third tier of the CASTOR tiered security model focuses on attestations over complete routing paths instead of routing elements. In a nutshell, tier 3 extends the trust model to the end-to-end routing path across the network infrastructure. In this context, CASTOR's novel composite attestation aggregates attestation evidence from multiple nodes participating in a routing path guaranteeing through cryptographic constructions the enforcement of the correct path order. The goal is to ensure that each node involved in forwarding network traffic operates in a verified and trustworthy configuration. This process enables CASTOR to provide path-level integrity guarantees, where routing decisions can be evaluated not only based on network performance and the verified trust state of the participating nodes, but also on the correct order of the path, allowing for a more efficient and privacy-preserving verification of path integrity (e.g., SSLA violations monitoring at the path level). Through this hierarchical attestation model, CASTOR enables a secure and verifiable routing infrastructure where trust is established progressively—from the device level, to the platform level, and ultimately to the network path level. Sections 4.5 and 4.6 document the two possible variants of crypto-primitives schemes for the composite attestation based on sequential aggregate signature and ordered multi-signature/matrix-vector primitives, so as to allow for the latter cost analysis and identify the most efficient scheme.

4.2 System and Threat Model

4.2.1 System Model

As aforementioned the CASTOR architecture is built upon a **minimal, explicitly delimited computing base** responsible for cryptographic operations, policy evaluation, and the generation and validation of integrity measurements across layers of descending privilege. The schemes described in the following sections implement continuous chained verification: nothing is taken as fully trusted except the hardware Root of Trust (RoT) (e.g., the secure element), and all software outside the TCB (including the vRouter containers) is considered untrusted. The three classes of entities that participate in the protocols are the *CASTOR Service Orchestrator*, the *Verifier* and the *Device* (e.g., Prover).

CASTOR Service Orchestrator: The *CASTOR Service Orchestrator* is responsible for provisioning reference measurements and policies that define acceptable vRouter states (e.g., approved BGP daemon hashes, valid Segment Routing configurations), authenticated under the CASTOR Service Orchestrator's digital signature. The *CASTOR Service Orchestrator* provides the "programmable firmware emulation" capability: policy predicates that dictate the Required Trust Level (RTL) for a routing path are provisioned externally by the *CASTOR Service Orchestrator* and evaluated internally by the host's SE. It has to be noted that, in the context of CASTOR and as already described in D5.1 [23], the Orchestration Layer consists of multiple components. For clarity, these components are abstracted and collectively referred to as the *CASTOR Service Orchestrator*.

Verifier (V): The entity that initiates attestation by issuing a fresh challenge nonce. Within CASTOR, this could be a peer vRouter attempting to establish an inter-domain trust boundary, or the CASTOR Service Orchestrator verifying node integrity before computing a path. Upon receiving evidence, the *Verifier* validates the attestation signature and checks challenge freshness. The Verifier learns *only* that the SE admitted the current vRouter state under *CASTOR Service Orchestrator*-approved policies. We term this property *evidence minimality*.

Device (D): The physical infrastructure node being attested (e.g., the Prover), composed of the following four architectural layers.

- *Layer 0: Hardware Root of Trust* - The hardware root of trust provides isolated cryptographic computation, protected measurement registers with PCR-equivalent semantics, and an internal assertion engine that gates key usage via routing policy evaluation. A long-term endorsement key (EK) establishes the node's cryptographic identity; a distinct attestation key (AK) produces routing evidence. Under the symbolic model, we assume ideal cryptography in the Dolev–Yao sense [34].
- *Layer 1: Hardware-Isolated Tracing & Attestation* - Attestation source and Tracing Hub components harvest deep system telemetry. This layer is the core attestation logic that attests the integrity of the above layers (Layers 2 and 3).
- *Layer 2: Trust Evaluation Plane* - Trust Assessment Framework (TAF) and Finite State Machine (FSM) components evaluate the raw traces and telemetry data produced by Layer 1 and calculate the actual and required trust levels.
- *Layer 3: Runtime TNDI Layer* - The virtualised routers. This layer is explicitly untrusted. Thus, Layer 3 is continuously introspected by the Layers 1 and 2 for its tracing capabilities and its behavioral trust respectively.

4.2.2 Threat Model

This section provides the threat model which is also aligned with the threat modelling in the routing plane described in Chapter 2. The CASTOR Service Orchestrator is assumed to be trusted: it provisions correct reference measurements and policies, and its signing key is not compromised. The possibility of malicious policy provisioning by a compromised CASTOR Service Orchestrator is considered outside the CASTOR scope.

The adversary is assumed to control all software components outside the TCB. It may obtain kernel-level privileges, execute arbitrary code in user space, manipulate application memory and control flow, and influence the timing and ordering of protocol operations. However, the adversary cannot compromise the hardware-isolated tracing and attestation (Layer 1). We assume that the platform enforces runtime kernel integrity protections (e.g., kernel lockdown, signed module loading, or equivalent mechanisms) that prevent any post-boot modification of the attestation's logic code, data, or its interface with the SE.

Furthermore, the adversary has full control over the network in the sense of the Dolev–Yao model [34]: it can intercept, delay, reorder, and replay all communications among the Device, the Verifier, and the CASTOR Service Orchestrator. It may also attempt to substitute measurements, inject falsified policy states, or construct seemingly fresh attestation transcripts by replaying nonces or manipulating state outside the SE.

Despite these capabilities, the adversary is unable to: (i) extract secrets from the SE or tamper with its protected storage; (ii) obtain the endorsement key (EK) or any attestation key (AK); (iii) cause the assertion engine to evaluate a policy as satisfied unless the actual system state complies with the policy conditions; (iv) arbitrarily modify or reset monotonic counters; (v) write to protected measurement registers; or (vi) forge signatures under the AK without access to the corresponding private key. Also, physical attacks on the SE and permanent Denial-of-Service (Dos) conditions are explicitly out of CASTOR scope.

Under these assumptions, any attestation accepted by the Verifier must include a valid signature generated by the AK over a fresh challenge. Crucially, this signature can only be produced if the SE's assertion engine has verified that the above layer's measurements match the Service Orchestrator's reference values and that all relevant policy conditions, including counter-based constraints, are satisfied.

4.3 Device-Level Configuration Integrity Verification

Configuration Integrity Verification (CIV) is an attestation mechanism used to ensure the correctness of the configuration of an attested device. In the context of CASTOR, a novel CIV scheme is proposed. The scheme is agnostic to the underlying root of trust and can be used to attest any router or network element by verifying that its configuration is correct and that it therefore operates as expected. In essence, CASTOR's Device-Level CIV scheme follows a challenge-based protocol. A Verifier (e.g., a peer vRouter or the CASTOR Service Orchestrator) challenges the Prover (i.e., the Device) with a fresh nonce. If the Prover successfully produce a valid signature using its policy-restricted Attestation Key—the Verifier gains assurance that the Prover is operating in a correct configuration state. By generating this valid signature, the Prover provides attestation evidence in a privacy-preserving manner, as it does not disclose the raw attestation evidence, thereby preventing implementation disclosure attacks.

Within this framework, the Device-Level CIV scheme aims to minimise the trust assumptions placed on any device acting as a Verifier, given that CASTOR's attestation mechanism is designed to preserve privacy. Consequently, any Verifier can attest a device while remaining unable to infer meaningful information about the device's internal configuration or operational state. The scheme relies on a policy-restricted Attestation Key that serves as the trust anchor of the attestation process. This enables enhanced authorisation mechanisms for the Prover's Attestation Key, allowing it to be used only if the relevant protection policies—enforced as part of the underlying root of trust—are satisfied. In CASTOR, key usage restriction policies are defined and enforced by the CASTOR Service Orchestrator, which acts as a trusted entity governing the overall process. This ensures that the Prover (e.g., the Device) can only use the Attestation Key to sign challenges when its configuration complies with the predefined policies. By conditioning the Prover's ability to sign a request for attestation evidence on the correctness of its configuration (i.e., configurations describing the deployment environment), the system enables verification of the Prover's compliance through a simple challenge-based protocol that neither requires nor reveals configuration information to the Verifier.

When an action results in a change to the configuration of a device, the previously issued Key Restriction Usage Policy becomes invalid. Such configuration changes introduce a significant attack surface, as a compromised device could retain multiple valid policies, thereby enabling the generation of valid signatures even when the device is operating in a malicious state. To address this issue, the Device-Level CIV scheme supports the creation of time-limited policies, ensuring that a policy is valid only within a specific time window. This approach reduces the risk of outdated policies being exploited after configuration changes. A key innovation of our design is that it enables the Prover to provide attestation evidence not only regarding the correctness of its configuration but also regarding the version of the trusted application that is expected to be running on the device.

4.3.1 High-level Overview of CIV

In this section, we present a high-level overview of the protocol, which comprises three distinct phases: *Join*, *Runtime*, and *Update*. Before delving into the details, we first introduce the notation used throughout to guide the reader.

Notations. We denote key pairs by (sk, pk) and write hashes as $\mathsf{H}(\cdot)$. The Device's SE holds an endorsement key pair $(sk_{\mathsf{EK}}, pk_{\mathsf{EK}})$ and a distinct attestation key pair $(sk_{\mathsf{AK}}, pk_{\mathsf{AK}})$. The verifiable policy enforcement (VPE) key pair is $(sk_{\mathsf{VPE}}, pk_{\mathsf{VPE}})$ and is provisioned by the CASTOR Service Orchestrator, which holds $(sk_{\mathsf{CSO}}, pk_{\mathsf{CSO}})$. The attestation key name commits to the attestation public key and the policy bound to that key as $\mathsf{name}_{\mathsf{AK}} = \mathsf{H}\big(pk_{\mathsf{AK}} \,\|\, \mathrm{repr}(P_{\mathsf{AK}})\big)$. An authorization ticket t is a CASTOR Service Orchestrator signature over the runtime policy digest bound to the attestation key. Nonces are sampled uniformly at random; we use n_1, n_2 for internal randomness and c for a challenge. A session-specific random value is r. We write $\mathsf{Enc}_K(M)$ for symmetric encryption and $\mathsf{Enc}_{pk}(M)$ for public-key encryption. The

VPE key is bound to a join-time policy that validates both secure boot integrity and TSS measurements. The attestation key is bound to a runtime policy that accumulates process measurements, counter predicates, and VPE-signed linkage before being authorized by the CASTOR Service Orchestrator. These policy bindings ensure keys can only be used when their associated security conditions are satisfied.

Join. The *Prover* provisions its attestation key and binds it to an authorization policy inside the *SE*. Let $\mathsf{name}_{\mathsf{AK}} = \mathsf{H}(pk_{\mathsf{AK}} \| \mathrm{repr}(P_{\mathsf{AK}}))$. The *Prover* conveys $\mathsf{name}_{\mathsf{AK}}$ to the *CASTOR Service Orchestrator* (the *CASTOR Service Orchestrator* obtains pk_{EK} from a pre-provisioned registry). The *CASTOR Service Orchestrator* verifies device eligibility, collects reference values, computes the VPE policy digest d_{VPE}—the hash commitment to the join-time policy requiring secure-boot PCR and TSS measurement validation—and generates a fresh $(sk_{\mathsf{VPE}}, pk_{\mathsf{VPE}})$ cryptographically bound to d_{VPE}. The *CASTOR Service Orchestrator* creates a certificate over pk_{AK} signed under sk_{CSO}, binding the attestation public key to the device identity and authorized policies. The *CASTOR Service Orchestrator* then computes the attestation key policy digest d_{AK} and issues an authorization ticket t signed under sk_{CSO}; this ticket signature is the critical authentication mechanism that binds the attestation key to *CASTOR Service Orchestrator*-approved policies. Crucially, the SE can only produce valid attestation signatures if and only if it possesses a ticket created and signed by the legitimate *CASTOR Service Orchestrator*—this ensures only the *CASTOR Service Orchestrator* can authorize attestation operations and prevents any other party (including adversaries with network control) from enabling the attestation key. The *CASTOR Service Orchestrator* samples a random r, derives $K = \mathsf{KDF}(r \| \mathsf{name}_{\mathsf{AK}})$, and transmits an activate-credential package [62]: a public-key encryption of r under pk_{EK} and an AEAD ciphertext of $\langle \mathsf{VPE}, t, c \rangle$ under K with associated data committing to $\mathsf{name}_{\mathsf{AK}}$. The encryption and AEAD provide *confidentiality and integrity for the channel* but do not authenticate the *CASTOR Service Orchestrator*; authentication is achieved through the SE's verification of the ticket signature t under pk_{CSO} before accepting any authorization. The *SE* recovers r, re-derives K, verifies AEAD integrity, imports the *VPE* key, and returns (t, c) to the *Trusted Software Stack (TSS)*. The *TSS* opens an authorization session; the *SE* admits join-time policy predicates (secure boot PCR and TSS measurement) and signs c under sk_{VPE}; the *CASTOR Service Orchestrator* verifies this signature and finalizes the join. The authenticated decryption during the join phase follows the credential activation flow [62]. The *Trusted Software Stack* invokes *ActivateCredential* so that the *SE* performs all cryptographic operations.

Runtime. The *Verifier* samples a fresh challenge n and sends it to the *Prover*. The *TSS* initiates two concurrent authorization sessions with the *SE*: a *VPE* session s_{vpe} and an attestation session s_{ak}. These sessions form a two-layer authorization chain: (i) the VPE session validates the *static* system properties (TSS integrity and secure boot) and produces a signature that authorizes the AK session; (ii) the AK session then validates the *dynamic* properties (safety-critical process measurements, counter thresholds, CASTOR Service Orchestrator ticket) and authorizes attestation key usage. The VPE session must complete successfully before the AK session can be authorized. The *SE* returns session handles and nonces $(h_{\mathsf{vpe}}, \nu_{\mathsf{vpe}})$ and $(h_{\mathsf{ak}}, \nu_{\mathsf{ak}})$.

- **VPE session construction**. The *TSS* instructs the *SE* to evaluate a join-consistency policy that validates the TCB through dual-layer verification: it applies (i) $\mathrm{PolicyOR}(m^{\downarrow}_{\mathsf{TSS}})$ using the user-space TSS measurement as observed by the kernel-resident mediator, validating it against the expected reference measurement for the TSS user-space component, and (ii) $\mathrm{PolicyPCR}(\mathrm{PCR}_{\mathsf{SB}})$ to bind to secure boot, ensuring the integrity of the kernel-resident TSS component. Any alteration to the lower-ring TSS would produce a different secure boot hash, and if it doesn't match the expected value, the *VPE* key remains locked and unable to sign. This dual validation ensures both user-space TSS integrity (via measurement comparison) and lower-ring TSS integrity (via secure boot validation). Upon successful policy evaluation, the *SE* signs $\mathsf{H}(\nu_{\mathsf{ak}})$ under sk_{VPE}, producing σ_{VPE}. This signature links s_{vpe} to s_{ak} and attests to the complete TSS integrity validation.
- **AK session construction**. The *TSS* extracts the current measurements for all safety-critical processes to be attested and applies $\mathrm{PolicyOR}(\mathcal{R})$ inside s_{ak}. It then applies $\mathrm{PolicyMC}(i, \cdot)$

to assert the required monotonic counter predicate. Next, it applies $\mathrm{PolicySigned}(pk_{\mathsf{VPE}}, \sigma_{\mathsf{VPE}})$, which verifies the *VPE* signature on $\mathsf{H}(\nu_{\mathsf{ak}})$ bound to the session handle h_{ak}, establishing that the VPE-validated TSS integrity claim authorizes this specific AK session. Finally, the *TSS* applies $\mathrm{PolicyAuthorize}(pk_{\mathsf{UA}}, t)$, which validates the authorization ticket, confirming that all policy assertions—measurements, counters, and VPE linkage—satisfy the CASTOR Service Orchestrator-approved authorization criteria.

- **Evidence production**. With s_{ak} authorized, the *SE* is permitted to use sk_{AK}. The *SE* returns $e = \mathsf{Sign}_{sk_{\mathsf{AK}}}(\mathsf{H}(n))$. The *Verifier* checks $\mathsf{Vrfy}_{pk_{\mathsf{AK}}}(e, \mathsf{H}(n)) = 1$ and freshness of n.

Update. When safety-critical applications are updated (adding functionality or patching vulnerabilities), the extracted measurements and the required counter predicate change. The *CASTOR Service Orchestrator* computes the new pre-authorization digest $d_{\mathsf{AK}}^{\mathsf{pre}}$ using the updated reference set $\mathcal{R}'$, the required monotonic counter value $v{+}1$, and the same pk_{VPE} input to $\mathrm{PolicySigned}$ as before. It then issues a fresh authorization ticket t' under sk_{CSO} for this digest and samples a fresh challenge c'. Using the activate-credential channel [62], *CASTOR Service Orchestrator* transmits t' and c' to the *SE* by sending a public-key encryption of r' under pk_{EK} together with an AEAD ciphertext of $\langle t', c' \rangle$ under $K' = \mathsf{KDF}(r' \| \mathsf{name}_{\mathsf{AK}})$ with associated data committing to $\mathsf{name}_{\mathsf{AK}}$. The *TSS* invokes *ActivateCredential*; upon successful decryption and integrity verification, it then *verifies the CASTOR Service Orchestrator ticket* under pk_{CSO}, *extracts current measurements* and validates $R' = m$ (no attack). *Only then* does it increment the monotonic counter. After this validated increment, the *TSS* triggers the runtime authorization sequence against t' and the updated predicates, culminating in a signature over $\mathsf{H}(c')$ returned to the *CASTOR Service Orchestrator*. In the update phase, the CASTOR Service Orchestrator issues new manifests and reference hashes signed under sk_{CSO}. Acceptance transitions are modeled as SE state updates guarded by the signature verification and policy predicates, after which the monotonic counter is advanced and new policy digests may be computed and rebound to the attestation key if needed. We model PolicyAuthorize(pk_{CSO}, t) command for a secure element. This operation resets the current policy digest to a value that depends only on the authorizer's public key. The ticket t is verified against the pre-authorization digest and the current session state; once validated, the digest collapses to $d_{\mathsf{AK}}^{\mathsf{final}} = \mathsf{H}(\mathrm{cc}_{\mathsf{PolicyAuthorize}} \| \mathsf{H}(pk_{\mathsf{CSO}}))$. This keeps the runtime binding minimal and aligns with our goal that evidence discloses nothing beyond freshness and possession of the authorized key.

4.3.2 Building Blocks

This section provides all the necessary building blocks for out scheme (e.g., cryptography, keys and secrecy, policies and binding, context, measurement and counters, session binding and network security, join/activate credential, evidence and verification and the target properties). As already mentioned, protocols are modelled in the Dolev–Yao setting [34] with perfect cryptography.

Cryptography (functions and equations):

- **Hash** as a public function $h(\cdot)$, used for measurements and key names.
- **Public-key encryption/signatures** with standard inverses:
  $\mathsf{adec}(\mathsf{aenc}(m, pk(sk)), sk) = m$, $\mathsf{verify}(\mathsf{sign}(m, sk), m, pk(sk)) = \mathsf{ok}$.
- **Symmetric encryption/MAC** (*AEAD*) modeled as separate primitives: $\mathsf{sdec}(\mathsf{senc}(m, k), k) = m$ and MACs checked by requiring the premise $\mathsf{mac}(x, k) = \mathsf{mac}(x, k)$. We treat **AEAD integrity** by jointly requiring decryption and matching MAC; associated data can be concatenated into x.
- **KDF** as an uninterpreted function $\mathsf{kdf}(r, \mathsf{name}_{\mathsf{AK}})$; security is captured by **freshness** and **non-reversibility** (no equations beyond syntactic equality).

Keys and secrecy:

- *EK*: sk_{EK} remains inside the *SE*. The *CASTOR Service Orchestrator* learns pk_{EK} via a **device registry**; it is not exchanged in protocol messages, limiting **identity exposure**.
- *AK*: pk_{AK} is public; **evidence verification** requires it. Evidence is restricted to $\mathsf{sign}(h(n), sk_{\mathsf{AK}})$.
- *VPE*: a second signing key used during join and runtime to authenticate challenges and provide TCB linkage; it is imported via the **activate-credential flow** and bound to **join-time policy** requiring dual-layer TSS validation (user-space measurement match and lower-ring secure boot integrity).

Policies and binding:

- **Policy terms** are abstract constructors (e.g., $\mathsf{pol\_pcr}(\cdot), \mathsf{pol\_mc}(\cdot), \mathsf{pol\_signed}(pk_{\mathsf{CSO}}), \mathsf{pol\_or}(\cdot)$).
- **Binding** is modeled with a digest $h(\mathrm{id}_k, \mathrm{repr}(P))$; rules that use a key require matching recomputation from the current context facts.
- **Ticket** t is a *CASTOR Service Orchestrator* signature over the runtime policy digest; *SE* side requires a $\mathsf{VerifiedTicket}(t)$ premise before enabling *AK*.
- **Cryptographic verification** ensures $\mathsf{verify}(t, \langle \mathsf{name}_{\mathsf{AK}}, R, v\rangle, pk_{\mathsf{CSO}}) = 1$.
- **Reference validation** enforces **attack detection**: ticket references R must match current measurements m for authorization to proceed. When no attacks on applications have occurred, $R = m$; when attacks are present, $R \neq m$ and authorization fails, ensuring **integrity**.
- This validation prevents acceptance of **compromised or stale state** by requiring **exact correspondence** between expected references and runtime measurements.

Context, measurement, and counters:

- Measurements are derived by a $\mathsf{Measure}(m)$ event; acceptance requires a $\mathsf{RefOK}(m)$ fact obtained from CASTOR Service Orchestrator manifests.
- PolicyOR combines multiple measurements by serializing them one after another: $\mathcal{R} = \mathrm{ser}(m_1)\|\mathrm{ser}(m_2)\| \ldots \|\mathrm{ser}(m_k)$.
- Reference values R in update tickets must equal the runtime measurements m when no attacks on applications have occurred; validation enforces $R = m$.
- Monotonic counters are facts $\mathsf{MC}(\mathrm{nameAK}, v)$; updates replace $\mathsf{MC}(\mathrm{nameAK}, v)$ with $\mathsf{MC}(\mathrm{nameAK}, v{+}1)$ where the increment is bound to specific attestation keys. No rule decrements counters. Counters initialize to zero.
- Secure boot/configuration is captured by a $\mathsf{PCR\_SB}(\cdot)$ fact established at initialization.

Session binding and network security:

- Session handles are explicitly modeled using cryptographic commitments: $\mathsf{session\_handle}(\mathrm{seed}, \mathrm{type})$.
- VPE signatures are bound to specific AK session handles through $\mathsf{SessionBinding}$ facts, preventing cross-session attacks.
- The network model includes adversary capabilities: message interception, replay, injection, and delay attacks.

- Session verification ensures VPE signatures can only authorize their intended AK sessions.

Join/activate-credential:

- *CASTOR Service Orchestrator* samples fresh r, sends $\mathsf{aenc}(r, pk_{\mathsf{EK}})$ and an AEAD ciphertext $\mathsf{senc}(\mathrm{payload}, K)$ with associated data committing to $\mathsf{name}_{\mathsf{AK}}$, where $K = \mathsf{kdf}(r, \mathsf{name}_{\mathsf{AK}})$.
- *SE* rule derives r via adec, recomputes K, verifies AEAD integrity, and imports *VPE* and t. This ensures *binding to* $\mathsf{name}_{\mathsf{AK}}$ without revealing *EK*.

Evidence and verification:

- **Runtime evidence** is $\mathsf{sign}(h(n), sk_{\mathsf{AK}})$ with a prior $\mathsf{Fr}(n)$ by the *Verifier*. No counters or measurements appear in evidence; they gate the SignAK rule as premises.

Target properties (trace properties first):

- **Freshness (no-replay).** If a Verifier accepts evidence on nonce n, then there exist events $\mathsf{VerifierChallenge}(n)$ and $\mathsf{SignAKEv}(\mathsf{name}_{\mathsf{AK}}, h(n))$ with the expected ordering, and n was generated fresh by the Verifier (cf. *freshness_no_replay*).
- **Policy admission.** Every SignAKEv event is preceded by satisfied policy premises: validated ticket (*RuntimeTicketVerified*/VerifiedTicket), current counter state $\mathsf{MC}(\mathsf{name}_{\mathsf{AK}}, v)$ meeting the threshold after validation, reference equality $R{=}m$ (*MeasurementValidated*), and complete TSS integrity validation via VPE (VPEPolicyOK with user-space and lower-ring checks) together with session verification (*SessionVerified*).
- **Anti-rollback.** Counters are strictly monotonic per attestation key (cf. *counter_strict_monotonic*, *no_rollback*); authorization increments occur only after successful measurement validation and ticket verification (cf. *update_requires_increment*).
- **VPE policy binding and TSS dual-layer integrity.** VPE policy binding is unique and requires both user-space TSS measurement validation and lower-ring secure boot integrity (cf. *vpe_unique_binding*, *tss_dual_validation_required*, *vpe_requires_tss_integrity*).
- **Session binding integrity.** VPE signatures authorize only their intended AK sessions; AK authorization requires prior session verification (cf. *session_binding_integrity*, *vpe_session_authorization*, *ak_requires_session_verification*).
- **Onboarding authenticity.** Valid AK signatures imply successful onboarding by the legitimate UA; the SE can only produce attestation signatures after verifying the UA ticket signature, ensuring only UA-authorized devices can attest.

4.3.3 Device-Level CIV Algorithm

The device-level CIV uses the entities described in the System Model (see section 4.2). Having described all the entities and components that participate in the Device-Level CIV scheme, we will now provide the three distinct phases in detail.

4.3.3.1 Phase 1: Join

During the Join phase, a device wants to join the network. To do so, the *CASTOR Service Orchestrator* sets up the key restriction usage policy (authorization policy) and creates the Verifiable Policy Enforcer (VPE) key. More specifically, the *CASTOR Service Orchestrator* acting as a Trusted Entity, is responsible for issuing and validating the key restriction usage policies for each device that wishes to participate to the network and for creating unique key pairs for every VPE. Then the *Prover* (e.g., the device) provisions its Attestation Key (*AK*) and binds it to the authorization policy issued by the *CASTOR Service Orchestrator* inside the CASTOR TCB. We specify concrete algorithms to pin down the join flow.

- **KDF**: HKDF-SHA256 [7] with domain separation. Let $K = \text{HKDF-Expand}(\texttt{salt} = \texttt{0x00},\ \text{IKM} = r,\ \text{info} = \texttt{"join-v1"} \| \text{name}_{\text{AK}},\ L = 32)$.
- **AEAD**: AES-256-GCM [28] with associated data $\text{ad} = \texttt{"join-v1"} \| \text{name}_{\text{AK}}$.
- **PKE**: RSA-OAEP [39] for transporting r under pk_{EK}.

The logical/API and network messages (see Figure 4.2) are as follows:

1. TSS $\rightarrow$ SE (API): CreateAK&Bind(P_{AK}). SE returns $(pk_{\text{AK}}, \text{name}_{\text{AK}})$.
2. TSS $\rightarrow$ CSO: JoinReq(name_{AK}).
3. CSO (local): Lookup pk_{EK} in device registry; compute $d_{\text{VPE}}, d_{\text{AK}}$; issue ticket t under sk_{UA}; generate $(sk_{\text{VPE}}, pk_{\text{VPE}})$ bound to d_{VPE}; sample r; derive K.
4. CSO $\rightarrow$ TSS: JoinResp(C_2, C_1)
   - $C_2 = \text{RSA-OAEP\_Enc}(r, pk_{\text{EK}})$
   - $C_1 = \text{AES-GCM\_Enc}_K(\langle \text{VPE}, t, c \rangle; \text{ad})$
5. TSS $\rightarrow$ SE (API): ActivateCredential(C_2, C_1).
6. SE (local): $r = \text{RSA-OAEP\_Dec}(C_2, sk_{\text{EK}})$; $K = \text{HKDF-Expand}(r, \texttt{"join-v1"} \| \text{name}_{\text{AK}})$; decrypt C_1 under K and verify AEAD tag using ad=$\texttt{"join-v1"} \| \text{name}_{\text{AK}}$; import VPE; output (t, c).
7. SE $\rightarrow$ TSS (API): (t, c).
8. TSS $\leftrightarrow$ SE (API): Start policy session; SE authorizes VPE under join-time policy (PCR_{SB} and OR over TSS measurement); SE signs c under sk_{VPE}.
9. TSS $\rightarrow$ CSO: $(\sigma_{\text{VPE}} = \text{Sign}_{sk_{\text{VPE}}}(c),\ pk_{\text{VPE}})$.
10. CSO (local): Verify σ_{VPE} on c under pk_{VPE}; finalize join.

Acceptance and error handling: SE aborts if RSA OAEP decryption fails, if K is not derivable (e.g., wrong name_{AK}), or if AEAD tag verification fails. Authorization of VPE fails if PCR_{SB} or TSS measurement do not satisfy the join-time policy. CSO rejects if signature on c under pk_{VPE} is invalid.

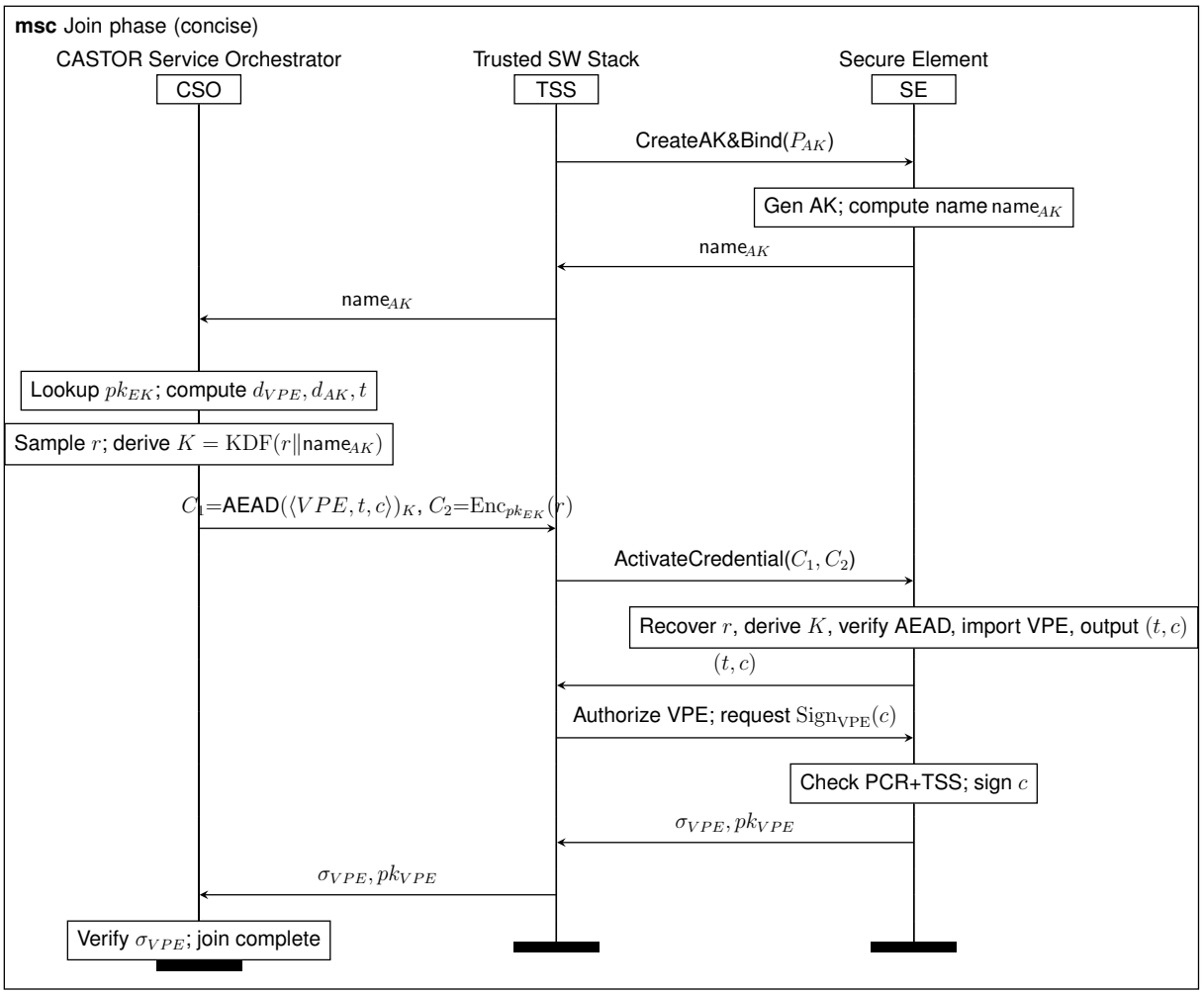

Figure 4.2: Join: NameAK → CSO; CSO packages (C1,C2) via ActivateCredential; SE imports VPE and signs c; CSO verifies.

4.3.3.2 Phase 2: Runtime

During the Runtime phase, a Verifier, for instance a vRouter or the CASTOR Service Orchestrator wishes to attest a device (e.g., another vRouter) that is participating in the network, and initiates a challenge response attestation protocol. Through this protocol the Prover creates evidence for the Verifier that he is indeed in a correct configuration but without disclosing any information regarding its state. We model two SE authorization sessions: s_{vpe} and s_{ak}, with handles $h_{\text{vpe}}, h_{\text{ak}}$ and nonces $\nu_{\text{vpe}}, \nu_{\text{ak}}$. This phase messages (see Figure 4.3) are as follows:

1. Verifier → Prover: Challenge(n).
2. TSS → SE (API): StartAuthSession() → $(h_{\text{vpe}}, \nu_{\text{vpe}})$ and $(h_{\text{ak}}, \nu_{\text{ak}})$.
3. TSS → SE (API, s_{vpe}): PolicyOR($m^{\downarrow}_{\text{TSS}}$); PolicyPCR($\text{PCR}_{\text{SB}}$); then SignWithVPE(H($\nu_{\text{ak}}$)) → σ_{VPE}.
4. TSS → SE (API, s_{ak}): PolicyOR($\mathcal{R}$); PolicyMC(i, ·); PolicySigned(pk_{VPE}, σ_{VPE}) verifying H(ν_{ak}) for handle h_{ak}; PolicyAuthorize(pk_{UA}, t) ⇒ digest $= d^{\text{final}}_{\text{AK}}$.
5. TSS → SE (API): SignAK(H(n)) → e.
6. Prover → Verifier: Evidence(e). Verifier checks $\text{Vrfy}_{pk_{\text{AK}}}(e, \text{H}(n)) = 1$ and freshness of n.

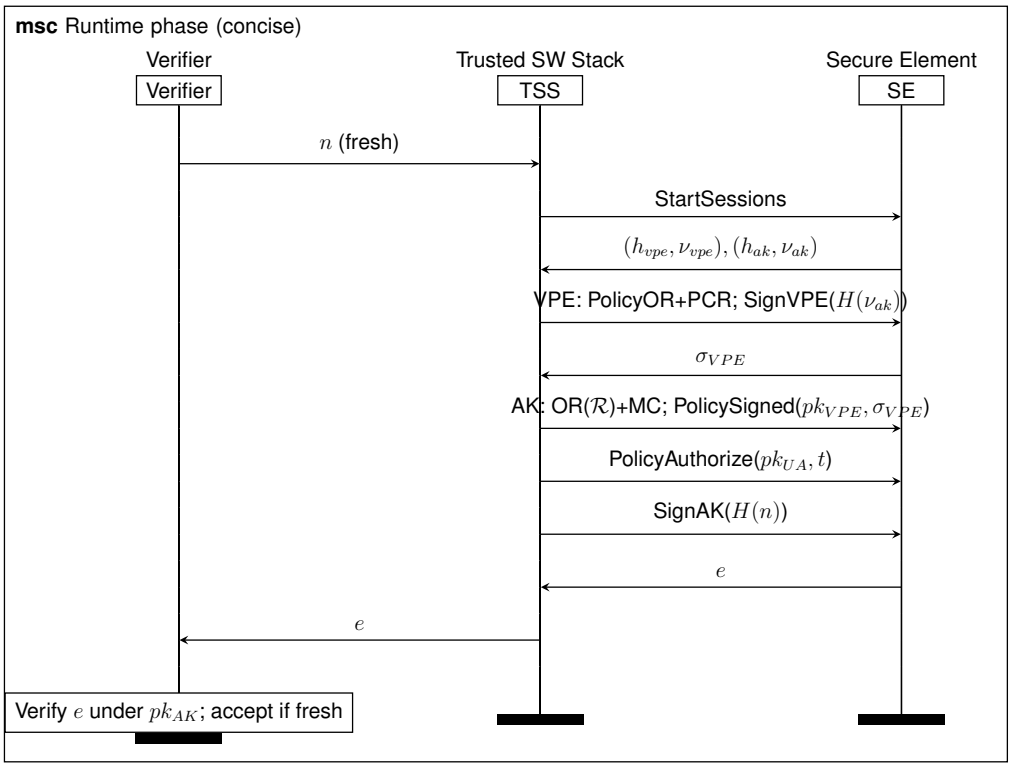

Figure 4.3: Runtime: Verifier sends fresh challenge; TSS builds VPE and AK sessions, applies PolicyAuthorize(pk_{UA}, t), and SE signs $H(n)$.

4.3.3.3 Phase 3: Update

During the Update phase, safety-critical applications are updated (adding functionality or patching vulnerabilities), the extracted measurements and the required counter predicate change. Thus, the CASTOR Service Orchestrator provisions signed policy tickets encoding acceptance configurations (e.g., measurements and version thresholds). We reuse the credential activation channel to refresh tickets and convey the CASTOR Service Orchestrator's challenge. This phase messages (see Figure 4.4) are as follows:

1. CSO (local): Recompute $d_{\mathsf{AK}}^{\mathsf{pre}}$ using $(\mathcal{R}', \mathrm{PolicyMC}(i,\, v{+}1),\, pk_{\mathsf{VPE}})$; issue t' under sk_{UA}; sample r', derive $K' = \mathsf{KDF}(r' \| \mathsf{name}_{\mathsf{AK}})$; sample fresh c'.
2. CSO $\rightarrow$ TSS: UpdateResp(C_2', C_1')
   - $C_2' = \mathsf{RSA\text{-}OAEP\_Enc}(r', pk_{\mathsf{EK}})$ [39]
   - $C_1' = \mathsf{AES\text{-}GCM\_Enc}_{K'}(\langle t', c' \rangle; \mathsf{ad})$ [28] with $\mathsf{ad} = \texttt{"join-v1"} \| \mathsf{name}_{\mathsf{AK}}$.
3. TSS $\rightarrow$ SE (API): ActivateCredential(C_2', C_1').
4. SE (local): $r' = \mathsf{RSA\text{-}OAEP\_Dec}(C_2', sk_{\mathsf{EK}})$; derive $K' = \mathsf{HKDF\text{-}Expand}(r', \texttt{"join-v1"} \| \mathsf{name}_{\mathsf{AK}})$ [7]; verify AEAD tag with ad; decrypt C_1' to (t', c').
5. SE/TSS (API): Verify t' under pk_{UA}; extract measurements; validate $R' = m$; then increment $\mathsf{MC}(i, v) \rightarrow \mathsf{MC}(i, v{+}1)$.
6. TSS $\rightarrow$ SE (API): Execute the runtime authorization sequence using PolicyOR($\mathcal{R}'$); PolicyMC($i, v{+}1$); PolicySigned($pk_{\mathsf{VPE}}, \cdot$); PolicyAuthorize(pk_{UA}, t').; then request SignAK($\mathsf{H}(c')$) $\rightarrow e'$.
7. Prover $\rightarrow$ CSO: Evidence(e'). CSO verifies $\mathrm{Vrfy}_{pk_{\mathsf{AK}}}(e', \mathsf{H}(c')) = 1$.

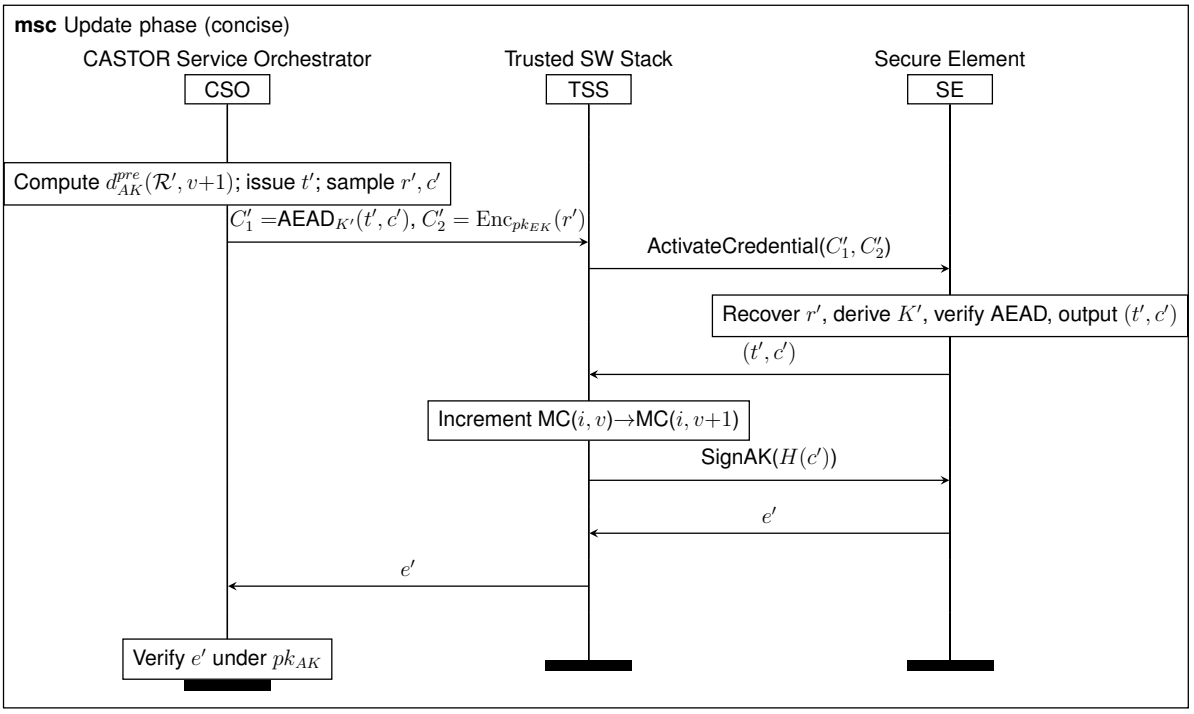

Figure 4.4: Update: CSO refreshes authorization (ticket t') via ActivateCredential; TSS verifies t' and $R' = m$, increments counter, re-authorizes AK, and signs $H(c')$.

4.4 Layered Attestation

Within the CASTOR project, securing the routing plane across B5G/6G computing continuum requires treating infrastructure nodes as **hierarchies of privilege**. A minimal hardware layer provides strong isolation and forms the root of trust, while successive layers introduce additional functionality but progressively weaker isolation guarantees—particularly in virtualised and containerised environments. Each layer therefore offers distinct isolation properties, and a sound security architecture within CASTOR must enable every layer to **attest the integrity of the one above it**, constructing an unbroken chain of trust from the hardware root up to the virtualised routing functions.

At the top of this hierarchy lies the Runtime layer, which represents the largest and most rapidly evolving attack surface. This layer includes Trusted Network Domain Interfaces (TNDIs) instantiated as containerised vRouters, together with their routing daemons (e.g., BGP, OSPF), supporting libraries, and container configurations. Because the runtime environment is highly dynamic and directly exposed to **Dolev–Yao network adversaries** as well as potential container-escape attempts, the layers beneath it must continuously introspect and re-validate its integrity to maintain trust across the system.

Remote attestation is the standard mechanism for realising such chains. A verifier (e.g., a peer vRouter) challenges a node and expects cryptographic evidence, anchored in restricted signing keys and protected registers, that the routing node is in a known, acceptable state complying with its Required Trust Level (RTL). In practice, however, existing attestation schemes fall short of the layered ideal required for lightweight, container-based routing. Most designs either (i) bind trust rigidly to a single hardware root, forcing a full network reconvergence when any routing software component changes, or (ii) require the verifier to maintain an omniscient model of every possible vRouter state. Neither approach scales to heterogeneous, continuously evolving B5G/6G topologies.

The root cause is architectural. Ideally, custom attestation logic would reside at the lowest hardware level—in *programmable firmware*—so that the mechanism responsible for measuring the vulnerable vRouter runtime is itself outside the reach of container-escape adversaries. While architectures such as RISC-V approach this ideal, the commodity servers hosting CASTOR's virtualised infrastructure (x86

and AArch64) do not expose an equivalent programmable firmware layer. This creates a *portability gap* for dynamic routing infrastructures.

To address the portability gap of commodity hardware while providing rigorous security guarantees, CASTOR implements a *four-layer attestation architecture*. This architecture establishes a continuous trust chain leveraging CASTOR's core evaluation components:

1. **Layer 0: Hardware Root of Trust**—The absolute trust anchor of the physical host, instantiated via a discrete TPM or a Trusted Execution Environment (TEE) such as Intel SGX. This layer protects long-term secrets and enforces cryptographic key usage.
2. **Layer 1: Hardware-Isolated Tracing & Attestation**—Comprising the **Attestation source** and the **Tracing Hub**. This layer leverages eBPF or direct kernel extensions to harvest deep system telemetry. Crucially, the core attestation logic operates with hardware-level isolation (e.g., rooted in the TPM secure boot chain or running within an SGX enclave), allowing it to securely attest the integrity of the layers above it.
3. **Layer 2: Trust Evaluation Plane**—Comprising the **Trust Assessment Framework (TAF)** and the **Finite State Machine (FSM)**. These components are responsible for evaluating the raw traces and telemetry harvested by Layer 1, calculating the required and actual trust levels, and performing stateful integrity checks. Layer 1 is explicitly responsible for continuously attesting the integrity of Layer 2.
4. **Layer 3: Runtime TNDI Layer**—The containerised vRouters. This layer is explicitly untrusted due to its network exposure. It is continuously introspected by the Layer 1 tracing capabilities, and its behavioral trust is evaluated by Layer 2.

By structuring the node in this manner, the trust boundary remains narrow while preserving strong guarantees across the software stack. The most dynamic component—the routing runtime—resides at the top of the hierarchy as containerised vRouters. These dynamic routing containers are constrained by the stateful evaluations of the TAF and FSM (Layer 2), which enforce policy-aware monitoring and validation of routing behaviour. In turn, the integrity of these control components is protected by the hardware-isolated Attestation Agent (Layer 1) and ultimately anchored in the physical Root of Trust (Layer 0).

In this layered design, each layer attests the one above it, forming a continuous chain of trust. The Secure Element (SE) validates the secure-boot measurements of the host kernel mediator; the mediator gates controlled access to the SE for the CASTOR Tracing Hub; and the Tracing Hub measures the vRouter container before submitting attestation evidence under SE-enforced policies. A Policy Authority—implemented by the CASTOR Traffic Engineering Policy Engine—provisions the SE with cryptographically signed reference measurements and authorisation tickets. The SE releases the attestation key only when the integrity of the full four-layer chain is satisfied. As a result, the trust boundary remains tightly scoped, and the behaviour of the dynamic routing containers is constrained by cryptographically verifiable policies rather than assumptions that cannot be validated at runtime.

4.4.1 Assets and Security Properties

The **primary asset** within this architecture is the integrity of the routing execution state (the Layer 3 TNDIs), the stateful evaluation mechanisms (the Layer 2 TAF and FSM), and the policy state within the Layer 0 Secure Element governing network participation. We target the following security properties:

Integrity. Any accepted routing attestation corresponds to a state reachable only through authenticated CASTOR policies.

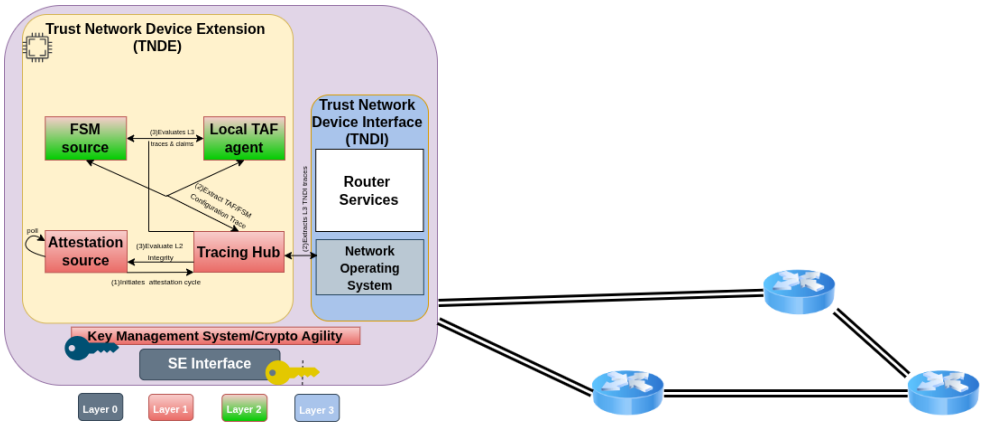

Figure 4.5: Castor's Layered Attestation Architecture

Freshness. Each attestation binds a verifier-chosen nonce; replayed or stale network evidence is cryptographically rejected.

State Integrity. The Finite State Machine (FSM) and Trust Assessment Framework (TAF) operating in Layer 2 cannot be tampered with to report a false "trusted" state for a compromised vRouter. Their execution integrity is strictly bound to reference hashes that are continuously verified by the hardware-isolated Attestation Agent in Layer 1.

Policy enforcement. Successful use of the Attestation Key implies the bound Required Trust Level (RTL) policy evaluated to true within the trusted stack.

4.4.2 High-Level Overview of Layered Attestation

To realise the four-layer attestation architecture within the CASTOR computing continuum, the system implements a phased cryptographic protocol. This protocol establishes an immutable trust foundation during node provisioning and dynamically bridges it to the rapidly updating vRouter container landscape during runtime. The protocol operates over a Dolev-Yao hostile network, relying strictly on cryptographic enforcement. The lifecycle is divided into three distinct phases: Onboarding, Policy Provisioning, and Runtime Chained Attestation.

4.4.2.1 Phase 1: Node Onboarding and Static Trust Foundation

The lifecycle begins when a new physical routing node attempts to join the CASTOR infrastructure, triggering the implicit platform JOIN phase (as detailed in Section 3.3). During this initial phase, the node registers *only* its hardware root (Layer 0) and the hardware-isolated tracing and attestation components (Layer 1).

Rather than relying on a traditional, direct Endorsement Key exchange, the onboarding is brokered through TN-DSM. The CASTOR Service Orchestrator establishes a connection to the TN-DSM and executes the SPDM handshake. During the `CHALLENGE_AUTH` round, the TN-DSM proves possession of the **TNDE platform key**, which assumes the role of the primary attestation key for the node.

When the Orchestrator encounters this previously unknown SPDM certificate and public key, it recognizes a new onboarding event. Following successful authentication and the establishment of an encrypted SPDM secure session, the Orchestrator issues a `GET_MEASUREMENTS` request. Within this secure context, the TN-DSM returns an initial composite manifest. Crucially, at this stage, this manifest contains the static reference values for Layer 1 (the expected cryptographic hashes of the Attestation source and the core tracing capabilities).

By creating a new registry entry that binds the TN-DSM's public key to these Layer 1 static reference values, the Orchestrator establishes a rigid, unbreakable baseline. This foundational trust is explicitly anchored *before* the Trust Evaluation Plane (Layer 2) and the dynamic Runtime TNDIs (Layer 3) are even spun up on the node. Subsequent re-attestations will reuse this registered SPDM key to securely validate the upper layers.

4.4.2.2 Phase 2: Policy Digesting and Dynamic Ticket Provisioning

With the static foundation established, the Orchestrator must bridge this immutable base to the dynamic operational layers (Layers 2 and 3). The Orchestrator translates the static reference values of Layer 1 into an **access policy digest**. This digest cryptographically binds an intermediate *Foundation Key*, which is encrypted using the node's EK_{pub} and provisioned to the Layer 0 Secure Element.

Next, the Orchestrator calculates an **Authorisation Ticket**. To securely bind the integrity of the evaluation plane (Layer 2), this ticket explicitly contains the reference hash measurements for the TAF and FSM binaries. The ticket dictates a strict rule: the Attestation source (Layer 1) must use these reference hashes to continuously evaluate the integrity of Layer 2. The Orchestrator transmits this Authorisation Ticket to the device, empowering Layer 1 to securely validate the trust evaluation components before they are permitted to assess the active TNDIs.

4.4.2.3 Phase 3: Runtime Chained Attestation

During network operation, a Verifier triggers a runtime attestation event by sending a fresh cryptographic challenge nonce to the node. The chained attestation sequence is orchestrated seamlessly across the four layers:

1. **Attesting the Foundation:** The Secure Element (Layer 0) evaluates its internal state against the access policy digest. If the Layer 1 Attestation source and Tracing Hub remain uncompromised, the SE unlocks the Foundation Key, mathematically proving the static foundation.
2. **Dynamic Measurement Extraction:** The Layer 1 Tracing Hub leverages its deep eBPF/kernel extensions to extract real-time measurements for *both* the Trust Evaluation Plane (Layer 2) and the Runtime TNDIs (Layer 3).
3. **Layer 2 Integrity Evaluation:** The Attestation source (Layer 1) evaluates the extracted measurements of the TAF and FSM against the reference hashes stored in the Authorisation Ticket. This satisfies the *State Integrity* property, proving the evaluators have not been tampered with.
4. **Layer 3 Trust Evaluation:** Concurrently, the newly validated TAF and FSM (Layer 2) evaluate the behavioral trust state of the active TNDIs (Layer 3), relying on the granular telemetry harvested by Layer 1.
5. **Challenge Response:** If Layer 1 successfully validates Layer 2, and Layer 2 confirms the required trust state of Layer 3, the Authorisation Ticket's conditions are met. The SE unlocks the primary Attestation Key (AK), which signs the Verifier's challenge nonce alongside the aggregated, multi-layered evidence.

This protocol ensures that the final signature delivered to the Verifier inherently proves the integrity of the entire stack. A compromise at any level—whether a tampered FSM or a container-escaped vRouter—breaks the chain, rendering the node cryptographically incapable of proving its trust level to the CASTOR network.

4.5 Path-oriented Composite Attestation based on Ordered multi-Signature/Sequential Aggregate Schemes

As already described, Trusted Path Routing (TPR) in the Compute Continuum requires trust decisions to be made over complete routing paths rather than individual routing elements. While CASTOR enables runtime trust assessment at the device level through a hardware-anchored TCB, routing decisions inherently involve multiple routers that may belong to different administrative domains. As a result, CASTOR's trust architecture must support the **composition of device-level attestation evidence into path-level trust assertions**, while **preserving the ordering of routing elements along the path**, since trust evaluation may depend on hop sequence, minimum trust levels, or cumulative trust properties. In real-world scenario, some routers may not be trusted. In such cases, to ensure the trusted routing path, any untrusted routers must be revoked. Therefore, from a cryptographic perspective, revocation mechanisms are also necessary.

As analysed in D3.1 [24], achieving path-oriented composite attestation requires several fundamental requirements to be satisfied in order to ensure both security and practicality. These requirements are summarised below:

1. There should be multiple signers and verifiers, while the provers are instantiated as vRouters.
2. Multiple proofs generated by different routers along an enforced path should be collected by the CASTOR Service Orchestrator, enabling the evaluation of both the number and order of routers in the path.
3. Revocation of compromised or rogue routers must be supported by the global TAF during evidence collection.

A sequential aggregate signature/ordered multi-signature can be used to meet these requirements. A sequential aggregate signature involves multiple signers producing signatures on **different messages** in sequence. In contrast, an ordered multi-signature involves multiple signers generating signatures on **the same message**, which are then aggregated in a specific order into a single signature.

System Model: To illustrate this scenario and solution, as shown in Figure 4.6, consider four TNDIs (which can be instantiated as routers in CASTOR) as an example. The workflow proceeds as follows: TNDI_1 generates the first signature Σ_1, representing its proof of trust, and sends it to TNDI_2, who keeps running the algorithm AggSign to generate an aggregated signature Σ_2, which is an aggregation of Σ_1 and TNDI_2's own proof of trust. Following the same process, Σ_2 is sent to TNDI_3 to generate the aggregated signature Σ_3, which is then forwarded to TNDI_4 to produce the final aggregated signature Σ_4. This final aggregated signature Σ_4 is verified by the CASTOR Service Orchestrator using the *AggVerify* algorithm.

If the verification is successful, this means the aggregated signature is valid and the order of all signatures aggregated is correct. If the verification fails, this indicates that at least one router in the path is compromised. In this case, the CASTOR Service Orchestrator, acting as the verifier, executes the *Revoke* algorithm to identify all compromised routers along the path.

In this scenario, the definition of correct ordering and some assumptions are given as follows:

Definition of correct ordering. All TNDIs are arranged in a order from 1 to N. The order is known by the Service Orchestrator, which means the order and values of all TNDIs' public keys are known by the the Service Orchestrator.

Assumptions. In this scenario, the Service Orchestrator is trusted. TNDIs are not trusted, which means a TNDI can be compromised to try to forge a signature or pretend to form a fake order. One attack that two TNDIs, who are neighbours, try to collude to form a wrong order, is out of the scope of these two schemes.

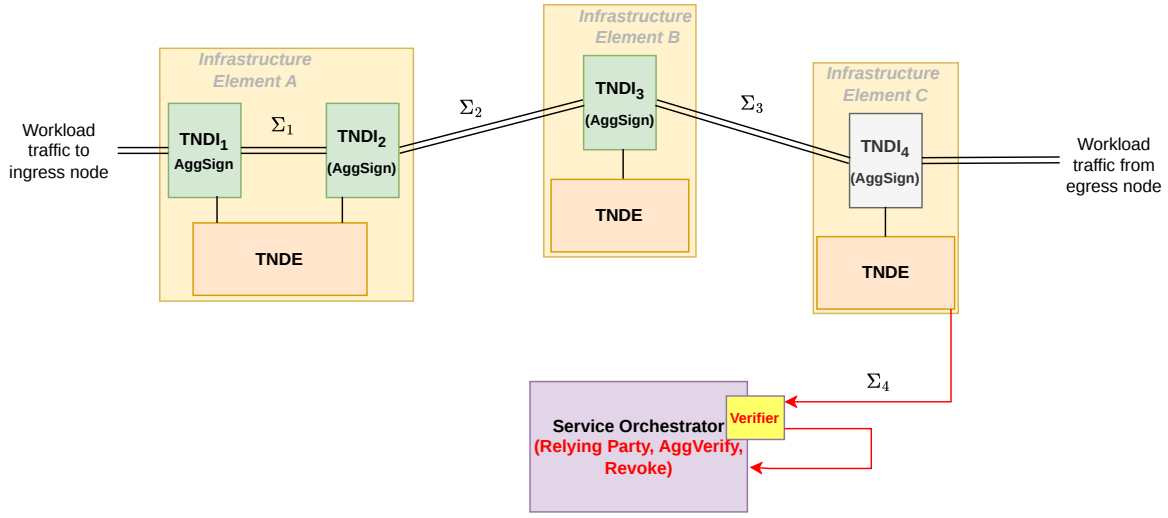

Figure 4.6: Workflow of a sequential aggregate signature in CASTOR, taking 4 TNDIs as an example

Based on the above description, we require a cryptographic primitive that enables the CASTOR Service Orchestrator to verify both the trust and the ordering of all routers in a path. Can such a primitive achieve efficient proof generation and revocation simultaneously?

To answer this question, we propose two solutions for realizing path-oriented composite attestation in CASTOR. The first is a novel sequential aggregate signature scheme with revocation based on the PS signature scheme [55] and the second is a new ordered multi-signature scheme based on BLS signature scheme [15]. The PS-based version preserves the order by including the previous signature in the current message to be signed, which is publicly verifiable. The BLS-based version preserve the order by aggregating public keys. However, to be against the rogue public key attacks, random values are added as weights for different public keys during public keys aggregation. These two schemes will be described in detail in the following sections.

4.5.1 Related work in Trusted Path Routing

Sequential aggregate signatures. The notion of aggregate signatures (AS) was introduced by Boneh *et al.* [13]. An AS scheme allows an aggregator to compress an arbitrary number of individual signatures into a a single short aggregate signature. The validity of all individual signatures can then be verified by checking the aggregate signature. AS schemes are particularly useful in applications where many signatures must be stored, transferred, or verified together, such as blockchain and e-voting systems. In 2004, Lysyanskaya *et al.* [50] proposed an important variant of aggregate signatures, called *sequential aggregate signatures* (SAS). In an SAS scheme, signatures are aggregated sequentially: each signer takes a pre-existing aggregate signature as input and produces a new aggregate signature by incorporating its own signature. Unlike traditional AS schemes, aggregation cannot be performed publicly by any party; it must be carried out by the participating signers themselves. However, the construction in [50] **does not preserve the order the signers**, meaning different signers can swap their signing order without detection. SAS schemes are well suited for applications such as certificate chains and routing protocols. Following [50], SAS schemes have been extensively studied [8, 11, 11, 18, 37, 49, 51]. Existing constructions are typically based on trapdoor permutations (TDPs) [8, 11, 51] or bilinear pairings [8, 37, 49, 51]. Since achieving verifier-local revocation is particularly challenging in TDP-based SAS schemes, we focus on pairing-based constructions in CASTOR. The BLS signature can serve as a building block for aggregate signatures. However, when signatures are generated over different messages (i.e., each signature uses a different generator), the number of pairing operations required during verification grows linearly with the number of signers [61], leading to suboptimal performance. The state-of-the-art SAS scheme is

based on the Schnorr signature [27]. The original Schnorr signature scheme consists of two elements: a challenge and a response (a randomized value derived from the secret key). In [27], a variant is used in which the signature consists of a commitment and a challenge. All commitments can be aggregated to produce a shorter sequential aggregate signature. However, since responses cannot be aggregated, the overall signature size still grows linearly with the number of signers. In terms of performance, verification avoids pairings but requires a linear number of exponentiations and hash computations. To date, most existing SAS schemes do not adequately address revocation.

Ordered multi-signatures. Both SAS and ordered multi-signature schemes can be used to realize path-oriented composite attestation. It is important to highlight their key difference: in SAS schemes, signatures from different signers are generated over different messages, whereas in ordered multi-signature schemes, all signers sign the same message. This distinction influences the choice of underlying signature primitives and leads to differences in performance and signature size. The first ordered multi-signature was proposed by Boldyreva *et al.* [11], based on Type-I pairings. Subsequent work further explored ordered multi-signatures [65]. However, these constructions typically require a number of pairing operations during verification that is linear in the number of signers. More broadly, multi-signature schemes have also been extensively studied [12, 52, 61]. These works are generally based on either BLS or Schnorr signatures. In BLS-based multi-signature [61], ordering and revocation are not supported. In schnorr-based schemes [52, 6], although constant-size signatures can be achieved, ordering and revocation are not preserved.

Based on the literature review, *there is currently no efficient order-preserving aggregate signature/multi-signature that simultaneously achieves constant signature size and efficient verification*. Supporting revocation is an additional but essential requirement, making the problem even more challenging. One possible direction is to use BLS or Schnorr signatures as a building block to construct ordered multi-signature schemes with revocation. Alternatively, other signature schemes may be considered. For instance, [27] is a promising candidate, as it supports aggregation and can be extended to enable revocation, which however will introduce extra assumption and storage overhead.

4.5.2 Preliminaries & Building Blocks

Because we focus on using BLS or PS signatures as building blocks for composite attestation, this section introduces these two signatures schemes.

Bilinear pairing Throughout the cryptographic schemes, let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_\mathrm{T}$ be groups of prime order q. Let g_1 and g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. The bilinear pairing function e maps elements from the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ to a target group $\mathbb{G}_\mathrm{T}$, i.e., $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_\mathrm{T}$. A concrete instantiation of the pairing groups is described in Section 4.5.4.

BLS signature [15]. The BLS signature scheme is defined as the set of the usual four algorithms: $\mathsf{BLS} = (\mathsf{BLS.Setup}, \mathsf{BLS.Keygen}, \mathsf{BLS.Sign}, \mathsf{BLS.Verify})$

- $\mathsf{BLS.Setup}(1^\lambda) \rightarrow \mathsf{pp}$: This algorithm takes the security parameter λ as input. It selects a cryptographic hash function $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{G}_1$. The public parameters are defined as the tuple $\mathsf{pp} = (p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{H})$.
- $\mathsf{BLS.Keygen}(\mathsf{pp}) \rightarrow (pk, sk)$: This algorithm takes the public parameter pp as input. It selects a random value sk as secret key. The public key is computed as $pk = g_2^{sk}$.
- $\mathsf{BLS.Sign}(sk, m) \rightarrow \sigma$: This algorithm takes the secret key sk and a message m as inputs, and computes the signature $\sigma = \mathsf{H}(m)^{sk}$.
- $\mathsf{BLS.Verify}(pk, m, \sigma) \rightarrow 0/1$: This algorithm takes the public key pk, the message m and signature σ as inputs. It checks whether the following equation holds: $e(\sigma, g_2) = e(\mathsf{H}(m), pk)$. If the equation holds, the signature is accepted; otherwise, it is rejected.

PS-variant signature [56]. We introduce the multi-message PS signature scheme, denoted as PS $=$ (PS.Setup, PS.Keygen, PS.Sign, PS.Verify) and described as follows:

- PS.Setup$(1^\lambda) \rightarrow$ pp: This algorithm takes the security parameter λ as input and outputs the public parameters pp $= (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$. In the following, we denote $\mathbb{G}_1^* = \mathbb{G}_1 \backslash \{1_{\mathbb{G}_1}\}$.
- PS.Keygen(pp) $\rightarrow (pk, sk)$: This algorithm selects $\tilde{g} \xleftarrow{\$} \mathbb{G}_2$ and $(x, y_1, ..., y_L, y_{L+1}) \xleftarrow{\$} \mathbb{Z}_p^{L+2}$. It then computes $(\tilde{X}, \tilde{Y}_1, ..., \tilde{Y}_L, \tilde{Y}_{L+1}) \leftarrow (\tilde{g}^x, \tilde{g}^{y_1}, ..., \tilde{g}^{y_L}, \tilde{g}^{y_{L+1}})$ and sets sk as $(x, y_1, ..., y_L, y_{L+1})$ and pk as $(\tilde{g}, \tilde{X}, \tilde{Y}_1, ..., \tilde{Y}_L, \tilde{Y}_{L+1})$.
- PS.Sign$(sk, (m_1, m_2, ..., m_L)) \rightarrow \sigma$: This algorithm selects two random numbers $h \xleftarrow{\$} \mathbb{G}_1^*$, $m' \xleftarrow{\$} \mathbb{Z}_p$ and outputs $\sigma = (m', \sigma_1, \sigma_2) = (m', h, h^{(x+\sum_{j=1}^{j=L} m_j y_j + y_{L+1} m')})$.
- PS.Verify$(pk, (m_1, ..., m_L), \sigma) \rightarrow 0/1$: This algorithm parses σ as (m', σ_1, σ_2) and checks whether $\sigma_1 \neq 1_{\mathbb{G}_1}$ and $e(\sigma_1, \tilde{X} \cdot \prod \tilde{Y}_j^{m_j} \cdot \tilde{Y}_{L+1}^{m'}) = e(\sigma_2, \tilde{g})$ are both hold. If so, it outputs 1; otherwise, it outputs 0.

4.5.3 A Provably Secure PS-based Sequential Aggregate Signature Scheme

Design rationale. As discussed in section 4.5.1, designing an efficient order-preserving aggregate signature scheme for composite attestation in CASTOR is highly challenging. To date, there is no order-preserving aggregate or multi-signature scheme that achieves constant signature size. We now break down the main technical challenges from a cryptographic perspective.

First, some sequential aggregate signature schemes [50] and ordered multi-signature schemes [61] do not preserve the order of signers, meaning that signers can arbitrarily swap their signing order. This issue arises because these schemes aggregate signatures by combining the current signer's contribution with previous public keys or signature elements in a commutative manner. To preserve ordering, the aggregation process must explicitly incorporate the intermediate aggregate (i.e., the partial signature generated so far) into the next signer's computation. Second, to support verifier-local revocation—i.e., enabling the verifier to identify all compromised signers solely through aggregate verification—the verifier must be able to access each signer's contribution. However, minimizing communication overhead requires the aggregate signature to remain as compact as possible. In other words, there is an inherent tension between achieving revocation (which requires retaining per-signer information) and maintaining a constant-size aggregate signature. *Therefore, the goal is to simultaneously achieve order preservation and revocation, while optimizing performance in terms of signature size and verification efficiency.*

Based on the requirements and observations, we choose the PS signature as the building block for a new order-preserving aggregate signature scheme. Although PS signatures have previously been used to construct sequential aggregate signature [55], the scheme in [55] does not preserve the order of signers. In our scheme, we overcome this limitation and additionally enable verifier-local revocation.

In this section, we first present an overview of the scheme, followed by a detailed description. The notation used throughout the scheme is summarized in Table 4.1. *Note that the key pairs used in this PS-based sequential aggregate signature scheme are assumed to be pre-enrolled, as described earlier.*

Entities. There are two types of entities, i.e., n signers and a verifier:

- Signer. A signer i generates an aggregated signature based on the $(i-1)$-th signer's aggregate signature, along with the list of previous public keys and messages.
- Verifier. The verifier checks the validity of the aggregated signature and, if verification fails, identifies all compromised signers.

Symbol	Description
λ	Security parameter
p	A prime order
$\mathbb{G}_1$	A cyclic group of order q
$\mathbb{G}_2$	A cyclic group of order q
$g_1,\ g_2$	Generators of $\mathbb{G}_1$ and $\mathbb{G}_2$
$\mathbb{G}_T$	A multiplicative cyclic group of order q
e	Type III pairing defined by $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$
H	A hash function defined as $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$
H_0	A hash function defined as $\mathsf{H}_0 : \{0,1\}^* \rightarrow \mathbb{Z}_q$
H_1	A hash function defined as $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{G}_1$
H_2	A keyed hash function: $\mathsf{H}_2 : \{0,1\}^* \times \{0,1\}^t \rightarrow \{0,1\}^\lambda$
$\leftarrow_\$$	Indicates a uniform random sampling
pp	Public parameters
i	The i-th signer, $i \in [n]$
sk_i	Secret key for the signer i
pk_i	Public key for the signer i
m_i	A message for the signer i to be signed
$\mathbf{PK}_{i-1}$	A list of public keys for previous $i-1$ signers
Σ_{i-1}	An aggregated signature for previous $i-1$ signers
$\mathbf{PK}_n$	A list of public keys for n signers
$\mathbf{M}_n$	A list of messages for previous n signers
Σ_n	An aggregated signature for n signers
I	A revocation list

Table 4.1: Notation Summary for composite attestation

In CASTOR, n signers correspond to n routers (TNDIs) along a path, while the verifier corresponds to the CASTOR Service Orchestrator.

Syntax. A sequential aggregate signature scheme with revocation consists of five algorithms, denoted as $\mathsf{SAS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{AggSign}, \mathsf{Aggverify}, \mathsf{Revoke})$, defined as follows:

- $\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{pp}$: This algorithm takes the security parameter λ as input and outputs the public parameters pp, which are implicitly provided to all subsequent algorithms.
- $\mathsf{KeyGen}(i) \rightarrow (\mathsf{sk}_i, \mathsf{pk}_i)$: This algorithm is executed by a signer i by taking the index i as input and outputs a secret and public key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$.
- $\mathsf{AggSign}(\mathsf{sk}_i, m_i, \Sigma_{i-1}) \rightarrow \Sigma_i$: This algorithm is executed by the i-th signer. It takes as input the secret key sk_i, message m_i and the current aggregate Σ_{i-1}, and outputs an updated aggregate signature Σ_i.
- $\mathsf{AggVerify}(\Sigma_n, \mathbf{PK}_n, \mathbf{M}_n) \rightarrow \mathsf{halt}/0/1$: This algorithm is executed by the verifier. It first checks whether any public key appears more than once in $\mathbf{PK}_n$; if so, it halts. Otherwise, it verifies the aggregate signature Σ_n. If verification succeeds, it outputs 1 meaning that the aggregated signature is correct and the order is preserved; otherwise, it outputs 0 and proceeds by executing the Revoke routine.
- $\mathsf{Revoke}(\Sigma_n, \mathbf{PK}_n, \mathbf{M}_n) \rightarrow I$: This algorithm identifies all compromised signers and outputs their indices in the set I.

Security model. A sequential aggregate signature scheme with revocation should satisfy correctness and unforgeability:

Experiment $\mathrm{Exp}^{EU-CMA}_{\mathsf{SAS},\mathcal{A}}(\lambda)$

- pp $\leftarrow$Setup(λ).
- $(L^*, m_i^*, \sigma^*) \leftarrow A^{\mathsf{KeyGen}(i),\mathsf{AggSign}(m_i,pk_i,\cdot)}(\mathsf{pp})$.
- Return 1 iff
  - ✓ AggVerify$(\Sigma^*, \mathbf{PK}^*, \mathbf{M}^*) = 1$, where $\mathsf{pk}_{i^*} \in \mathbf{PK}^*$;
  - ✓ $\exists \mathsf{pk}_{i^*}$ and m^* have not been queried before;
  - ✓ The corresponding i^* is not in revocation list I.

Figure 4.7: EU-CMA game for an aggregatable signature.

- **Correctness**. A valid aggregate signature must be accepted by AggVerify. If compromised signers are involved, they must be correctly identified by Revoke.
- **Unforgeability**. An adversary without access to valid secret keys should not be able to produce a valid aggregate signature that passes AggVerify without being detected.

Before formally defining these security properties, we describe the threat model, particularly the capabilities of an adversary $\mathcal{A}$.

Threat model. An adversary $\mathcal{A}$ may perform the following:

- **Queries**: The adversary $\mathcal{A}$ can query key generation oracle and aggregated signing oracle to obtain valid keys and aggregate signatures, except for the target signer it aims to forge.

Definition 1. (**Correctness**) A sequential aggregatable signature scheme SA is correct and the claimed order is preserved, if given the public parameters pp $\leftarrow$ Setup(λ), then for any signing key pair $(sk_i, pk_i) \leftarrow$ KeyGen(i) and any sequential aggregate signature $\Sigma_i \leftarrow$ AggSign$(sk_i, m_i, \Sigma_{i-1})$, for $i = 1, \ldots, n$, it holds:

$$\mathsf{AggVerify}(\Sigma_n, \mathbf{PK}_n, \mathbf{M}_n) = 1 \qquad \text{and} \qquad \mathsf{Revoke}(\Sigma_n, \mathbf{PK}_n, \mathbf{M}_n) \in I$$

Unforgeability. The unforgeability of SAS is defined via an experiment $\mathrm{Exp}^{\text{EU-CMA}}_{\mathsf{SAS},\mathcal{A}}$, played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$, as shown in Figure 4.7. Here, $\mathcal{M}$ denotes the message space from which $\mathcal{A}$ queries the aggregated signing oracle AggSign. Given public parameters, the key generation oracle KeyGen and the aggregated signing oracle AggSign are defined as follows:

- KeyGen$(i) \rightarrow (\mathsf{sk}_i, \mathsf{pk}_i)$: When $\mathcal{A}$ queries the keyGen oracle for signer i, the oracle returns the corresponding secret and public key pair, $(\mathsf{sk}_i, \mathsf{pk}_i)$.
- AggSign$(m_i, \mathsf{pk}_i, \cdot) \rightarrow \Sigma_i$: When an adversary $\mathcal{A}$ submits a query consisting of a message m_i and a public key pk_i, the oracle returns an aggregated signature Σ_i.

We denote the advantage of $\mathcal{A}$ as $\mathsf{adv}^{\text{EU-CMA}}_{\mathsf{SAS},\mathcal{A}}(\lambda)$, which is defined by the probability that $\mathcal{A}$ wins the unforgeability game.

Definition 2. (**Unforgeability**) A sequential aggregatable signature scheme with revocation SAS is said to be unforgeable if for any polynomial time adversary $\mathcal{A}$, the following equation holds:

$$\mathsf{adv}^{\text{EU-CMA}}_{\mathsf{SAS},\mathcal{A}}(\lambda) = \Pr\left[\mathrm{Exp}^{\text{EU-CMA}}_{\mathsf{SAS},\mathcal{A}}(\lambda) = 1\right] \leq negl(\lambda)$$

The CASTOR proposed sequential aggregate signature

- Setup$(1^\lambda) \rightarrow$ pp: This algorithm takes the security parameter λ as input. It instantiates a Type-III bilinear pairing with parameters $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$, and selects two hash functions $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$, $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{G}_1$. It outputs the public parameters pp $= (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, \mathsf{H}, \mathsf{H}_1)$, which are implicitly provided to all subsequent algorithms.
- KeyGen$(i) \rightarrow (\mathsf{sk}_i, \mathsf{pk}_i)$: This algorithm is run by a signer i by choosing $\mathsf{sk}_i = (x_i, y_{1i}, y_{2i}) \leftarrow_\$ \mathbb{Z}_p$ and computing $\mathsf{pk}_i = (X_i = g_2^{x_i}, Y_{1i} = g_2^{y_{1i}}, Y_{2i} = g_2^{y_{2i}})$.
- AggSign$(\mathsf{sk}_i, m_i, \Sigma_{i-1}) \rightarrow \Sigma_i$: This algorithm is run by i-th signer to generate a sequential aggregate signature Σ_i, m_i is the message to be signed, Σ_{i-1} is a so-far aggregated signature generated by the $(i-1)$-th signer.
  - ✓ When $i = 1$, which means this algorithm is run by the first signer. Then $\Sigma_0 = \{\emptyset\}$ and $S_0 = 0$ as inputs. Further, the first signer uses H_1 with the time-stamp t to generate two generators: $\mathsf{H}_1(t_1||0) \rightarrow h_1$, $\mathsf{H}_1(t_1||1) \rightarrow h_2$. Then generates the signature $\Sigma_1 = \sigma_1 = (t, h_1, h_2, s_1 = h_1^{x_1 + y_{11} m_1} \cdot h_2^{y_{21}})$.
  - ✓ When $i \neq 1$, i parses $\Sigma_{i-1} = (h_1, h_2, s_1, ..., s_{i-1})$ and computes as follows:
    - $\bar{m}_i = \mathsf{H}(m_i||s_{i-1})$
    - $\Sigma_i = (t, h_1, h_2, s_1, ..., s_{i-1}, s_i = h_1^{x_i + y_{1i}\bar{m}_i} h_2^{y_{2i}})$
- AggVerify$(\Sigma_n, \mathbf{PK}_n, \mathbf{M}_n) \rightarrow halt/0/1$: This algorithm is run by the verifier by taking $\Sigma_n = (t, h_1, h_2, s_1, ..., s_n)$, $\mathbf{PK}_n = (\mathsf{pk}_1, ..., \mathsf{pk}_n)$ and $\mathbf{M}_n = (m_1, ..., m_n)$ as inputs. The verifier firstly checks any key appears twice in $|\mathbf{PK}_n|$. If yes, halt. Otherwise, the verifier checks the following equations:

$$h_1 \stackrel{?}{=} \mathsf{H}_1(t||0),\ h_2 \stackrel{?}{=} \mathsf{H}_1(t||2) \tag{4.1}$$

$$e(\prod_{i=1}^{n} s_i, g_2) \stackrel{?}{=} e(h_1, \prod_{i=1}^{n} X_i Y_{1i}^{\bar{m}_i}) e(h_2, \prod_{i=1}^{n} Y_{2i}),\ \text{where } \bar{m}_i = \mathsf{H}(m_i||s_{i-1}) \tag{4.2}$$

  If all equations hold, outputs "1" for accepting the sequential aggregate signature. Otherwise, the verifier runs the algorithm Revoke.
- Revoke$(\Sigma_n, \mathbf{PK}_n, \mathbf{M}_n) \rightarrow I$: This algorithm is run by the verifier, who works as follows:
  - ✓ For $i = 1$ to n, checks

$$e(s_i, g_2) \stackrel{?}{=} e(h_1, X_i \cdot Y_{1i}^{\bar{m}_i}) e(h_2, Y_{2i}) \tag{4.3}$$

    If the above equation does not hold, store i in I.

Order preservation. The order-preserving property of the PS_SAS scheme is ensured by requiring each node N_i to generate a signature over the concatenation of its message m_i and the previous signature s_{i-1}, i.e., $\bar{m}_i = \mathsf{H}(m_i \| s_{i-1})$, thereby cryptographically binding the two. Since the Verifier (Orchestrator) knows the order of the nodes, it can correctly reconstruct the sequence of messages. Any malicious modification of the order of signatures will be detected by the verifier, since it will break the cryptographic binding between each message m_i and the previous signature s_{i-1}.

4.5.4 Instantiation and cost analysis

Having described the PS-based multi-message signature scheme and our proposed PS-based sequential aggregated signature scheme in an abstract setting, we now turn to the concrete realization of the two schemes. More precisely, the present subsection serves as an implementation roadmap, aiming to **translate the abstract operations into explicit algorithmic building blocks and specifications**. We

start by identifying the pairing and elliptic curve instantiation, and proceed with the description of the necessary algorithmic components required for the practical implementation of the PS-based multi-message signature and PS-based sequential aggregated signature scheme. We further provide an initial analysis of the computational cost of the two schemes, identify the computationally intensive operations, and conduct a cost analysis with respect to the size requirements of key parameters of the two schemes.

4.5.4.1 Optimal ate pairing on BLS12-381

BLS12-381 curve. The BLS12-381 curve, discovered by Bowe [16], is a prominent and widely used elliptic curve for pairing-based cryptosystems and is also recommended in the IETF draft [58]. BLS12-381 belongs to the BLS12 family of pairing-friendly elliptic curves with embedding degree 12, introduced by Barreto, Lynn and Scott (BLS) in [5]. The BLS12 family supports a parametric description where the curve parameters are represented via polynomials $t(x), r(x), p(x) \in \mathbb{Q}[x]$, defined as:

$$t(x) = x + 1, \quad r(x) = x^4 - x^2 + 1, \quad p(x) = \frac{1}{3}(x-1)^2(x^4 - x^2 + 1) + x$$

which correspond to the trace of Frobenius, the prime factor of the curve order and the underlying base field. Members of the BLS12 family are generated by evaluating the polynomials $t(x)$, $r(x)$ and $p(x)$ at some *seed* $z \in \mathbb{Z}$, such that $z \equiv 1 \bmod 3$. In particular, the BLS12-381 curve is obtained by evaluating the above polynomials at the seed $z = -0\text{xd}201000000010000$; this yields a base field prime $p = p(z)$ of length 381 bits and a prime $r = r(z)$ of length 255 bits. The equation of BLS12-381 is $E_1/\mathbb{F}_p : y^2 = x^3 + 4$ and it has a composite order $\#E_1(\mathbb{F}_p) = h_1 \cdot r$, where h_1 is a *cofactor* of length 126 bits. The seed z has a low Hamming weight (only 6), which enables a fast implementation of the optimal ate pairing, since it reduces the number of addition steps in the Miller loop. In addition, the low Hamming weight of the seed improves the performance of the final exponentiation by reducing the cost of exponentiations by z.

Optimal ate pairing on BLS12-381. The optimal ate pairing, introduced by Vercauteren [63], and particularly its instantiation with BLS12-381, currently represents the best choice for achieving an ideal balance between performance and security. To maximize efficiency, the formula for computing the optimal ate pairing has to be tailored to the characteristics of the underlying curve. We refer to [63] for the generic definition of the optimal ate pairing and concentrate here primarily on the instantiation over the BLS12-381 curve. The optimal ate pairing is a map $e : \mathbb{G}_2 \times \mathbb{G}_1 \to \mathbb{G}_\mathsf{T}$, where the *pairing groups* $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_\mathsf{T}$ are cyclic, of the same prime order r. More concretely, the *source groups* are defined as $\mathbb{G}_1 = E_1(\mathbb{F}_p)[r] \cap \ker(\pi_p - [p])$ and $\mathbb{G}_2 = E_2(\mathbb{F}_{p^2})[r] \cap \ker(\pi_p - [1])$, where $E/\mathbb{F}_{p^2}$ is the degree-6 twist of E_1 with equation $E_2/\mathbb{F}_{p^2} : y^2 = x^3 + 4(i+1)$ and order $\#E_2(\mathbb{F}_{p^2}) = h_2 \cdot r$, assuming the quadratic extension field $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/\langle i^2 + 1\rangle$. In addition, the map $\pi_p : E_1 \to E_1$ is the Frobenius endomorphism, defined as $(x, y) \mapsto (x^p, y^p)$. The *target group* $\mathbb{G}_\mathsf{T}$ contains all r-th roots of unity μ_r in $\mathbb{F}_{p^{12}}$, where 12 corresponds to the embedding degree of BLS12-381. Given two generators $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$, the optimal ate pairing on the BLS12-381 curve is defined as the map:

$$(Q, P) \mapsto f^{(p^{12}-1)/r}$$

and operates in two steps, namely the Miller loop and the final exponentiation.

Miller loop. The Miller function $f \in \mathbb{F}_{p^{12}}$ is computed via the Miller loop, which is an iterative procedure, with $\lfloor \log_2(|z|) \rfloor = 63$ iterations composed of a *doubling step* and an *addition step*. Each iteration executes the doubling step, which includes the computation of the line function $\ell_{T,T}(P)$, i.e., the tangent line passing through T and evaluated at P, the computation of the point $[2]T$ (point doubling), and an update on the Miller function f (see Algorithm 1, lines 3-5). The addition step is executed for every "1" bit in the binary expansion of the seed $|z|$. It computes the line function $\ell_{T,Q}(P)$ passing

Algorithm 1: Miller loop.

Input: $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$ and $|z| = (b_n, b_{n-1}, \ldots, b_1, b_0)_2$ with $\lfloor \log_2(|z|) \rfloor = n$.
Output: Miller function $f = f_{z,Q}(P)$.

1 $f \leftarrow 1, T \leftarrow Q$
2 **for** $i \leftarrow \lfloor \log_2(|z|) \rfloor - 1$ **down to** 0 **do**
3 $\quad \ell \leftarrow \ell_{T,T}(P), T \leftarrow [2]T$ /* Line computation/evaluation and point doubling */
4 $\quad \upsilon \leftarrow \upsilon_{[2]T}(P)$ /* Vertical line */
5 $\quad f \leftarrow f^2 \cdot \ell / \upsilon$ /* Update Miller function */
6 $\quad$ **if** $b_i = 1$ **then**
7 $\quad\quad \ell \leftarrow \ell_{T,Q}(P), T \leftarrow T + Q$ /* Line computation/evaluation and point addition */
8 $\quad\quad \upsilon \leftarrow \upsilon_{T+Q}(P)$ /* Vertical line */
9 $\quad\quad f \leftarrow f \cdot \ell / \upsilon$ /* Update Miller function */
10 **if** $z < 0$ **then**
11 $\quad f \leftarrow f^{-1}$
12 **return** f

through T and Q and evaluated at P, as well as the sum $T + Q$ (point addition), before updating the Miller function f (see Algorithm 1, lines 6-9). The use of the degree-6 twist $E_2/\mathbb{F}_{p^2}$ allows to further optimize the computation of the Miller function f. In particular, since the embedding degree of `BLS12-381` is even and a degree-6 twist is used, denominator elimination applies and, therefore, the vertical lines υ are excluded from the computation [63]. In addition, since the seed z is negative, the Miller loop updates the Miller function $f = 1/f$, where the inversion can be efficiently computed via a conjugation. A high-level description of the Miller loop is presented in Algorithm 1.

Final exponentiation. This is the process of raising the Miller function f to the exponent $(p^{12} - 1)/r$. The usual strategy is to split the process in two parts, namely an *easy part* and a *hard part*, which suggests the following decomposition:

$$\frac{p^{12} - 1}{r} = \underbrace{(p^6 - 1) \cdot (p^2 + 1)}_{\text{easy part}} \cdot \underbrace{(p^4 - p^2 + 1)/r}_{\text{hard part}}$$

The easy part can be efficiently computed, while the hard part is significantly more computation intensive. The currently most efficient method for computing the hard part in the case of `BLS12-381` is due to Hayashida, Hayasaka, and Teruya [45, Theorem 4]. In particular, the hard part can be equivalently transformed to the exponent:

$$3 \cdot \frac{p^4 - p^2 - 1}{r} = (z - 1)^2 \cdot (z + p) \cdot (z^2 + p^2 - 1) + 3$$

Numerous cryptographic libraries exist, which include an optimized implementation of the optimal ate pairing on the `BLS12-381` curve. We focus on the `blst` library (version 0.3.13), which is a well-known BLS signature library, written in C and Assembly for x64 and ARM64. `blst` is used by Ethereum for the implementation of polynomial commitments and other components, and its source code is available at: `https://github.com/supranational/blst`. Furthermore, based on the evaluation in [44], the optimal ate pairing on `BLS12-381` offers a security level of approximately 126 bits, when taking into account the improved Special Tower Number Field Sieve (STNFS) attacks of Kim and Barbulescu [46], targeting the Discrete Logarithm Problem (DLP) in extension field of composite degree.

4.5.4.2 Multi-message PS signature scheme with `BLS12-381`

As described earlier in Subsection 4.5.2, the PS multi-message signature scheme is composed of four functions PS $=$ (Setup, KeyGen, Sign, Verify). In Algorithm 2 we present the concrete algorithmic speci-

Algorithm 2: PS-based multi-message signature scheme: PS $=$ (Setup, KeyGen, Sign, Verify).

function Setup(1^λ)**:**
- $P \leftarrow_\$ \mathbb{G}_1$
- $Q \leftarrow_\$ \mathbb{G}_2$
- $\mathrm{pp} \leftarrow (p, r, h_1, h_2, P, Q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{T}, e)$
- **return** (pp)

function Sign$(\mathrm{sk}, (\mathrm{m}_1, \ldots, \mathrm{m}_\ell))$**:**
- $R \leftarrow_\$ \mathbb{G}_1$
- $\mathrm{m}' \leftarrow_\$ \mathbb{F}_r^*$
- $\sigma_1 \leftarrow R$, $q \leftarrow s$
- **for** $i = 1, \ldots, \ell$ **do**
  - $q \leftarrow q + q_i \cdot \mathrm{m}_i$
- $q \leftarrow q + q_{\ell+1} \cdot \mathrm{m}'$
- $\sigma_2 \leftarrow [q]R$
- $\sigma \leftarrow (\mathrm{m}', \sigma_1, \sigma_2)$
- **return** (σ)

function KeyGen(pp)**:**
- $(s, q_1, \ldots, q_\ell, q_{\ell+1}) \leftarrow_\$ (\mathbb{F}_r^*)^{\ell+2}$
- $S \leftarrow [s]Q$
- **for** $i = 1, \ldots, \ell + 1$ **do**
  - $Q_i \leftarrow [q_i]Q$
- $\mathrm{sk} \leftarrow (s, q_1, \ldots, q_\ell, q_{\ell+1})$
- $\mathrm{pk} \leftarrow (S, Q_1, \ldots, Q_\ell, Q_{\ell+1})$
- **return** $(\mathrm{sk}, \mathrm{pk})$

function Verify$(\mathrm{pk}, (\mathrm{m}_1, \ldots, \mathrm{m}_\ell), \sigma)$**:**
- $R \leftarrow S$
- **for** $i = 1, \ldots, \ell$ **do**
  - $R \leftarrow R + [\mathrm{m}_i]Q_i$
- $R \leftarrow R + [\mathrm{m}']Q_{\ell+1}$
- $e_1 \leftarrow e(\sigma_1, R)$, $e_2 \leftarrow e(\sigma_2, Q)$
- **if** $e_1 = e_2$ **then**
  - **return** Accept
- **else**
  - **return** Reject

fication for each function, using elliptic curve notation and assuming that the scheme is instantiated with the BLS12-381 curve and the optimal ate pairing.

Setup. The Setup phase generates the public parameters pp. In particular, it randomly chooses the generators P and Q of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. Then the public parameter set is a tuple $\mathrm{pp} = (p, r, h_1, h_2, P, Q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{T}, e)$, where p defines the BLS12-381 base field, $h_1 \cdot r = \#E_1(\mathbb{F}_p)$, $h_2 \cdot r = \#E_2(\mathbb{F}_{p^2})$ and $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_\mathsf{T}$ are the three pairing groups for the optimal ate pairing e. Recall that in the BLS12-381 setting, p is a prime of size 381 bits and r is a prime of size 255 bits.

KeyGen. This function creates the signing key pair $(\mathrm{sk}, \mathrm{pk})$. The secret signing key is composed of $(\ell + 2)$ scalars, each chosen randomly from the scalar field $\mathbb{F}_r^*$, where ℓ is the number of messages to be signed. More precisely, $\mathrm{sk} = (s, q_1, \ldots, q_\ell, q_{\ell+1})$, hence the size of sk is $32(\ell + 2)$ bytes. The corresponding public key is defined as a tuple of $(\ell + 2)$ points in $\mathbb{G}_2$:

$$\mathrm{pk} = (S, Q_1, \ldots, Q_\ell, Q_{\ell+1}) = ([s]Q, [q_1]Q, \ldots, [q_\ell]Q, [q_{\ell+1}]Q)$$

The computational cost of the KeyGen function is dominated by the $(\ell + 2)$ scalar multiplications in $\mathbb{G}_2$, which are expensive as they are executed over $\mathbb{F}_{p^2}$. In order to minimize the size of the public key, we can assume that only the x-coordinate of each component is stored in pk (compressed representation). In this case, since each point is defined over $\mathbb{F}_{p^2}$, the size of each component in pk is 96 bytes (one element in $\mathbb{F}_{p^2}$ is represented as two elements in $\mathbb{F}_p$), which results in a public key of total size $96(\ell + 2)$ bytes.

Sign. The PS multi-message signature scheme assumes that a single user signs ℓ different messages $(\mathrm{m}_1, \ldots, \mathrm{m}_\ell)$ using its secret key sk. According to the description of the Sign function in Algorithm 2, the signature generation requires $(\ell + 1)$ multiplications and $(\ell + 1)$ additions in the scalar field $\mathbb{F}_r$, while the most computational-heavy part is the scalar multiplication in $\mathbb{G}_1$ for computing the point $\sigma_2 = [q]R$. The signature $\sigma = (\mathrm{m}', \sigma_1, \sigma_2) \in \mathbb{F}_r^* \times \mathbb{G}_1 \times \mathbb{G}_1$ has constant size, which does not depend on the number of messages ℓ. It contains a scalar field element (32 bytes) and two points σ_1 and σ_2 in $\mathbb{G}_1$. Assuming

the compressed representation of points in $\mathbb{G}_1$ (i.e., one $\mathbb{F}_p$ element is stored, corresponding to the x-coordinate of each point), the size of the signature σ is 128 bytes.

Verify. The verification of the signature σ is the most computationally-intensive function of the scheme. According to the description of the Verify function (Algorithm 2), verification requires $(\ell + 1)$ scalar multiplications and $(\ell + 1)$ point additions in $\mathbb{G}_2$, hence these operations are executed over $\mathbb{F}_{p^2}$. In addition, it executes two pairing computations, in order to verify the equality $e(\sigma_1, R) = e(\sigma_2, Q)$. This can be optimized by considering an alternative verification equation. In particular, setting:

$$e_1 = e(\sigma_1, R) = f_1^{(p^{12}-1)/r} \quad \text{and} \quad e_2 = e(\sigma_2, Q) = f_2^{(p^{12}-1)/r}$$

we obtain the following equation:

$$e_1 = e_2 \Rightarrow e(\sigma_1, R) = e(\sigma_2, Q) \Rightarrow f_1^{(p^{12}-1)/r} = f_2^{(p^{12}-1)/r} \Rightarrow (f_1 \cdot f_2^{-1})^{(p^{12}-1)/r} = 1.$$

This verification equation computes two Miller functions and only one final exponentiation instead of two. Considering that this process executes arithmetic operations in the extension field $\mathbb{F}_{p^{12}}$, saving one final exponentiation in the Verify function offers a significant gain in the overall performance of the function.

Remark. The instantiation of the PS multi-signature scheme assumes that public keys are in $\mathbb{G}_2$ and signatures belong in $\mathbb{G}_1$. This instantiation aims at minimizing the length of the signature, as it is defined over $\mathbb{F}_p$ rather than $\mathbb{F}_{p^2}$. This instantiation also allows a more efficient singing process, since the scalar multiplication in the Sign algorithm (line 13) is carried out in $\mathbb{G}_1$. On the other hand, this instantiation results in a less efficient verification, since the $(\ell + 1)$ scalar multiplications in the Verify algorithm are executed over $\mathbb{G}_2$. However, the instantiation of the PS multi-signature scheme can be reversed, i.e., public keys can be chosen from $\mathbb{G}_1$, in which case signatures lie in $\mathbb{G}_2$. This choice minimizes the size of public keys and improves the execution time of signature verification, at the cost of larger signatures and computationally heavier signing (one scalar multiplication in $\mathbb{G}_2$). In Table 4.2 we present the expected size of key material and signatures for the PS multi-signature scheme, considering two modes of operations; targeting minimal signature size (mode 1) and targeting minimal public key size (mode 2).

Table 4.2: Expected size of secret key sk, public key pk and signature σ of the PS multi signature scheme for the two modes of operation, assuming ℓ messages $(\mathrm{m}_1, \ldots, \mathrm{m}_\ell)$.

Mode	optimization	size of sk	size of pk	size of σ
1	Minimal-signature-size	$32(\ell + 2)$	$96(\ell + 2)$	128
2	Minimal-public-key-size	$32(\ell + 2)$	$48(\ell + 2)$	224

In our scenario, signing is performed at each vRouter, which, while not computationally constrained, they are restricted on the bandwidth they can allocate to the control plane (so as to not affect the communication of service-related traffic). Consequently, the preferred mode of operation is the one optimized for signing (i.e., mode 1: minimal signature size). In addition, verification is performed by the Orchestrator, which is assumed to be a powerful computing entity and can tolerate higher computational costs.

Table 4.3: Notable ECC and field operations in the PS multi-signature scheme per function, for mode 1 and ℓ messages $(\mathrm{m}_1, \ldots, \mathrm{m}_\ell)$.

Function	mult. in $\mathbb{F}_r^*$	scalar mult. in $\mathbb{G}_1$	scalar mult. in $\mathbb{G}_2$	point add. in $\mathbb{G}_1$	point add. in $\mathbb{G}_2$	Miller loop	final exp.
KeyGen	-	-	$\ell + 2$	-	-	-	-
Sign	$\ell + 1$	1	-	-	-	-	-
Verify	-	-	$\ell + 1$	-	$\ell + 1$	2	1

Algorithm 3: PS-based sequential aggregate signature scheme:
PS_SAS $=$ (Setup, KeyGen, AggSign, AggVerify, Revoke).

function Setup(1^λ)**:**
- $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$
- $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$
- $P \leftarrow_\$ \mathbb{G}_1$
- $Q \leftarrow_\$ \mathbb{G}_2$
- $\mathrm{pp} \leftarrow (p, r, h_1, h_2, P, Q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{T}, e, \mathsf{H}, \mathsf{H}_1)$
- **return** (pp)

function KeyGen(pp, i)**:**
- $(s_i, q_{1i}, q_{2i}) \leftarrow_\$ \mathbb{F}_r^* \times \mathbb{F}_r^* \times \mathbb{F}_r^*$
- $P_i \leftarrow [s_i]Q$
- $Q_{1i} \leftarrow [q_{1i}]Q$
- $Q_{2i} \leftarrow [q_{2i}]Q$
- $\mathrm{sk}_i \leftarrow (s_i, q_{1i}, q_{2i})$
- $\mathrm{pk}_i \leftarrow (P_i, Q_{1i}, Q_{2i})$
- **return** $(\mathrm{sk}_i, \mathrm{pk}_i)$

function AggSign$(\mathrm{sk}_i, \mathrm{m}_i, \Sigma_{i-1})$**:**
- $\Sigma_0 \leftarrow \{\}$, $S_0 \leftarrow \infty_{\mathbb{G}_1}$
- **if** $i = 1$ **then**
  - $T_1 \leftarrow \mathsf{H}_1(t||0)$
  - $T_2 \leftarrow \mathsf{H}_1(t||1)$
  - $\Sigma_{i-1} \leftarrow \mathsf{append}(\Sigma_{i-1}, \{t, T_1, T_2\})$
- $\mathrm{m}'_i \leftarrow \mathsf{H}(\mathrm{m}_i||S_{i-1})$
- $R_1 \leftarrow [s_i + q_{1i} \cdot \mathrm{m}'_i]T_1$
- $R_2 \leftarrow [q_{2i}]T_2$
- $S_i \leftarrow R_1 + R_2$
- $\Sigma_i \leftarrow \mathsf{append}(\Sigma_{i-1}, S_i)$
- **return** (σ)

function AggVerify$(\Sigma_n, \mathrm{PK}_n, \mathrm{M}_n)$**:**
- $\mathrm{res} \leftarrow \mathsf{check\_uniqueness\_pk}(\mathrm{PK}_n)$
- $T'_1 \leftarrow \mathsf{H}_1(t||0)$, $T'_2 \leftarrow \mathsf{H}_1(t||1)$
- **if** $(T'_1 \neq T_1)$ *or* $(T'_2 \neq T_2)$ **then**
  - **return** Halt
- $S \leftarrow \infty_{\mathbb{G}_1}$, $Q_1 \leftarrow \infty_{\mathbb{G}_2}$, $Q_2 \leftarrow \infty_{\mathbb{G}_2}$
- **for** $i = 1, \ldots, n$ **do**
  - $\mathrm{m}'_i \leftarrow \mathsf{H}(\mathrm{m}_i||S_{i-1})$
  - $S \leftarrow S + S_i$
  - $Q_1 \leftarrow Q_1 + P_i + [\mathrm{m}'_i]Q_{1i}$
  - $Q_2 \leftarrow Q_2 + Q_{2i}$
- $e \leftarrow e(S, Q)$, $e_1 \leftarrow e(T'_1, Q_1)$, $e_2 \leftarrow e(T'_2, Q_2)$
- $e' \leftarrow e_1 \cdot e_2$
- **if** $e = e'$ **then**
  - **return** Accept
- **else**
  - $I \leftarrow \mathsf{Revoke}(\Sigma_n, \mathrm{PK}_n, \mathrm{M}_n)$
  - **return** I

function Revoke$(\Sigma_n, \mathrm{PK}_n, \mathrm{M}_n)$**:**
- **for** $i = 1, \ldots, n$ **do**
  - $\mathrm{m}'_i \leftarrow \mathsf{H}(\mathrm{m}_i||S_{i-1})$
  - $Q_1 \leftarrow S_i + [\mathrm{m}'_i]Q_{1i}$
  - $e \leftarrow e(S_i, Q)$, $e_1 \leftarrow e(T_1, Q_1)$, $e_2 \leftarrow e(T_2, Q_{2i})$
  - $e' \leftarrow e_1 \cdot e_2$
  - **if** $e \neq e'$ **then**
    - $I \leftarrow \mathsf{append}(I, i)$
- **return** (I)

optimizat	mult. in	size of pk	size of σ

Table 4.3 outlines the most costly operations per function, for the PS-based multi-signature scheme, considering the mode of operation 1, and assuming that the protocol considers ℓ messages to be signed by one public key.

4.5.4.3 Sequential aggregate signature scheme with BLS12-381

Recall from the description in Section 4.5.3, that our proposed PS-based sequential aggregate signature scheme consists of five algorithms PS_SAS $=$ (Setup, KeyGen, AggSign, AggVerify, Revoke). In Algorithm 3 we present the PS_SAS scheme in the elliptic curve setting, assuming the instantiation with the BLS12-381 curve, where each function is described in more detail below.

Setup. The Setup function of the PS_SAS scheme (Algorithm 3) is similar to the Setup function of the PS-based multi-message signature scheme described in Algorithm 2. It produces the same set of public parameters pp with the addition of two hash functions $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ and $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$. More precisely, the set of public parameters is defined as $\mathrm{pp} = (p, r, h_1, h_2, P, Q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{T}, e, \mathsf{H}, \mathsf{H}_1)$. The hash function H is used to map a string of arbitrary size to a digest of fixed length 256 bits. The resulting

bitstring is then converted to a random element in the scalar field $\mathbb{F}_r^*$. In this case, H can be instantiated with any standardized cryptographic hash function, such as SHA-256. The function H_1 is used to map a bitstring of arbitrary size to random point in the source group $\mathbb{G}_1$. This is a more computationally intensive hash function and requires special treatment. In particular, there is no standard instantiation for the hash-to-curve function H_1, as the optimal choice depends on the underlying elliptic curve and its characteristics. However, the most efficient hash-to-curve functions are based on the SwiftEC construction [26]. We discuss the instantiation of H_1 in our PS_SAS scheme in the next paragraph.

KeyGen. For each user i, the KeyGen function generates a pair of signing keys $(\mathrm{sk}_i, \mathrm{pk}_i)$. The private signing key consists of three random scalars in $\mathbb{F}_r^*$, hence the size of each sk_i is 96 bytes. The public key pk_i consists of three elliptic curve points in $\mathbb{G}_2$ and hence, with coordinates in $\mathbb{F}_{p^2}$. Assuming compressed point representation where only the x-coordinate is stored for each point, we conclude that the size of the public key is 288 bytes. Note that the cost of the KeyGen function is dominated by the three scalar multiplications in $\mathbb{G}_2$ for computing the points P_1, Q_{1i} and Q_{2i}.

AggSign. This function is used by each user i to create an aggregated signature Σ_i, by appending its signature component S_i to the previous aggregated signature Σ_{i-1}. Assuming n users, the aggregation process starts by setting $\Sigma_0 = \{\}$ (empty list) and $S_0 = \infty_{\mathbb{G}_1}$ (identity element in $\mathbb{G}_1$, i.e., point at infinity), and proceeds differently depending on whether the first user is considered ($i = 1$) or a subsequent user ($i > 1$). In particular, when $i = 1$, the first user executes two times the hash-to-curve operation to generate two random points $T_1 = \mathsf{H}_1(t||0) \in \mathbb{G}_1$ and $T_2 = \mathsf{H}_1(t||1) \in \mathbb{G}_1$, on input a timestamp t of size 64 bits. Then it uses the secret key sk_1 to generate its signature component $S_1 \in \mathbb{G}_1$, defined as:

$$S_1 = [s_1 + q_{11} \cdot \mathrm{m}'_1]T_1 + [q_{21}]T_2, \quad \text{where} \quad \mathrm{m}'_1 = H(\mathrm{m}_1||S_0)$$

on the message m_1. The first aggregated signature corresponds to the tuple $\Sigma_1 = (t, T_1, T_2, S_1)$, containing the timestamp, the two random (digest) points T_1, T_2 and the signature component S_1 of the first user. For each subsequent user, i.e., for every $i > 1$, the AggSign process is slightly simpler, since they do not need to reconstruct the random points T_1, T_2; these are obtained from Σ_{i-1}. Each user $i > 1$ creates only its signature component S_i on the message m_i and appends it to the previous aggregated signature Σ_{i-1}. Based on this description, the aggregate signature Σ_n obtained by the nth user is a tuple $\Sigma_n = (t, T_1, T_2, S_1, S_2, \ldots, S_n)$. It contains $(n + 2)$ points in $\mathbb{G}_1$ plus the timestamp t, hence the size of Σ_n is $48(n + 2) + 8$ bytes. For each user $i > 1$, the cost for computing the signature component S_i is dominated by the two scalar multiplications $R_1 = [s_i + q_{1i} \cdot \mathrm{m}'_i]T_1$ and $R_2 = [q_{2i}]T_2$ in $\mathbb{G}_1$. It also requires one multiplication and one addition in the scalar field $\mathbb{F}_r$, as well as a point addition $S_i = R_1 + R_2$ in $\mathbb{G}_1$. However, the first user has the additional cost of computing the two points T_1 and T_2, which requires two executions of the hash-to-curve function H_1 (see Algorithm 3 for more details).

AggVerify. The function AggVerify in Algorithm 3 describes the process of verifying the aggregate signature Σ_n. The verifier first checks whether the list of public keys PK_n contains any duplicate public keys. This is described by the abstraction check_uniqueness_pk, which has negligible cost. If no duplicates exist, the verification proceeds to the next steps. In particular, the verification checks whether the points T_1, T_2 are correctly constructed, with respect to the timestamp t. This requires the reconstruction of the points $T'_1 = \mathsf{H}_1(t||0)$ and $T'_2 = \mathsf{H}_1(t||1)$ and hence two executions of the hash-to-curve operation in $\mathbb{G}_1$. If both points are correct, the verification proceeds to the following steps, otherwise it returns Halt and stops the process. The verification of the aggregated signature Σ_n is accomplished via the pairing equation:

$$e(S, Q) = e(T'_1, Q_1) \cdot e(T'_2, Q_2) \Rightarrow e\left(\sum_{i=1}^{n} S_i, Q\right) = e\left(T'_1, \sum_{i=1}^{n}(P_i + [\mathrm{m}'_i]Q_{1i})\right) \cdot e\left(T'_2, \sum_{i=1}^{n} Q_{2i}\right)$$

The above verification equation requires $(n-1)$ point additions in $\mathbb{G}_1$ for constructing the point S, $(n-1)$ point additions in $\mathbb{G}_2$ for constructing the point Q_2, as well as $(2n-1)$ point additions in $\mathbb{G}_2$ and n scalar multiplications in $\mathbb{G}_2$ for constructing the point Q_1. The above pairing equation can be simplified and an equivalent verification equation can be used, saving two final exponentiations. More concretely we have:

$$e(S,Q) = e(T_1', Q_1) \cdot e(T_2', Q_2) \Rightarrow f^{(p^{12}-1)/r} = f_1^{(p^{12}-1)/r} \cdot f_1^{(p^{12}-1)/r} \Rightarrow (f_1 \cdot f_2 \cdot f^{-1})^{(p^{12}-1)/r} = 1$$

The new verification equation requires the computation of 3 Miller functions f, f_1 and f_2, 1 inversion in $\mathbb{F}_{p^{12}}$, which is free because it can be performed via $\mathbb{F}_{p^{12}}$ conjugation, 2 multiplications over $\mathbb{F}_{p^{12}}$ and 1 final exponentiation. This is significantly cheaper than computing the entire three pairings. If the above equation is satisfied, the aggregated signature is accepted, otherwise, the Revoke function is executed.

Revoke. This function sequentially verifies each signature component S_i, in order to identify the rogue signatures in the list Σ_n. For each rogue S_i, the user's index i is stored in the list of revoked signatures I. The function executes in total n scalar multiplications in $\mathbb{G}_2$ and n point additions in $\mathbb{G}_2$ for computing all pairing input points. The verification equation requires $3n$ Miller function computations $2n$ multiplications in $\mathbb{F}_{p^{12}}$ and n final exponentiations, since for each $i = 1, \ldots, n$ the following equation is verified:

$$e(S_i, Q) = e(T_1, Q_1) \cdot e(T_2', Q_{2i}) \Rightarrow (f_1 \cdot f_2 \cdot f^{-1})^{(p^{12}-1)/r} = 1$$

The cost of the Revoke function is added to the cost of AggVerify in the case where the aggregated signature Σ_n is not correct. The function is described in Algorithm 3.

Table 4.4: Expected size of secret key sk_i, public key pk_i, signature component S_i and aggregate signature Σ_n of the PS_SAS scheme for the two modes of operation, assuming n users.

Mode	optimization	size of sk	size of pk	size of Σ_1	size of S_i $(i>1)$	size of Σ_n
1	Minimal-signature-size	96	288	152	48	$48(n+2)+8$
2	Minimal-public-key-size	96	144	296	96	$96(n+2)+8$

Table 4.4 shows the expected size of the signing key pair $(\mathrm{sk}_i, \mathrm{pk}_i)$ for each user $i \in \{1, \ldots, n\}$, the signature of the first signer Σ_1, the signature component S_i, for each user $i > 1$, and the final aggregate signature Σ_n. Similarly to the case of the PS-based multi-signature scheme, the proposed PS_SAS can operate in two modes; mode 1, which offers shorter signatures and larger public keys, and mode 2, which offers larger signatures and shorter public keys. In mode 1, the two scalar multiplications and the point addition are executed over $\mathbb{G}_1$ and hence over $\mathbb{F}_p$ instead of $\mathbb{F}_{p^2}$. Since the signing process is executed by the vRouters which are constrained by bandwidth limitations, we conclude that mode 1 is the suitable choice for the instantiation of our novel PS_SAS scheme.

Table 4.5: Notable ECC and field operations in the PS_SAS scheme for mode 1, per function and assuming n users.

Function	mult. in $\mathbb{F}_r^*$	scalar mult. in $\mathbb{G}_1$	scalar mult. in $\mathbb{G}_2$	point add. in $\mathbb{G}_1$	point add. in $\mathbb{G}_2$	hash to $\mathbb{G}_1$	mult. in $\mathbb{F}_{p^{12}}$	Miller loop	final exp.
KeyGen	-	-	3	-	-	-	-	-	-
AggSign $(i=1)$	1	2	-	1	-	2	-	-	-
AggSign $(i>1)$	1	2	-	1	-	-	-	-	-
AggVerify	-	-	n	$n-1$	$2n-1$	2	2	3	1
Revoke	-	-	n	-	n	-	$2n$	$3n$	n

Table 4.5 summarizes the key ECC and finite field arithmetic operations performed in each function in

the PS_SAS scheme, assuming n signers. Note that signing cost differs only for the first signer, who is required to perform two hash-to-curve operations. In addition, the table shows that the verification cost increases significantly when at least one signature is invalid, mainly because the number of pairing computations increase, since the Revoke function verifies each signature individually, in order to identify the invalid ones in the list of n signature components S_i.

4.5.4.4 Hashing to pairing source groups

As described in Algorithm 3 , our proposed PS_SAS scheme requires hashing bitstrings of arbitrary length to the elliptic curve subgroup $\mathbb{G}_1$. This construction should respect the usual security requirements of an ideal collision-resistant hash function, and thus behave as closely as possible to a conventional random oracle. This property is formally defined as *indifferentiability from a random oracle*, according to [17]. Usually, the hashing to $\mathbb{G}_1$ (also the hashing to $\mathbb{G}_2$) is constructed as the composition of four functions, in particular $\mathsf{H}_1 : [h_1] \circ H \circ \eta \circ \chi$, where each of these functions is defined as follows:

- $\chi : \{0,1\}^* \to \{0,1\}^\ell$. This is a message expansion function which maps a bitstring of arbitrary size to a bitstring of fixed length ℓ.
- $\eta : \{0,1\}^\ell \to \mathbb{F}_p^n$. This is a mapping that converts an ℓ-bit string into n field elements in $\mathbb{F}_p$.
- $H : \mathbb{F}_p^n \to E_1(\mathbb{F}_p)$. This is an admissible encoding to the group of points of an elliptic curve. It maps n random field elements to a single point in the elliptic curve group $E_1(\mathbb{F}_p)$.
- $[h_1] : E_1(\mathbb{F}_p) \to \mathbb{G}_1$. This function executes a scalar multiplication of a point in $E_1(\mathbb{F}_p)$ by the cofactor h_1. This operation guarantees that the resulting point has the desired prime order r and is therefore known as *cofactor clearing*.

Different methods exist for defining the hashing to pairing groups, where the main point of difference lies in the construction of the encoding function H. Wahby and Boneh [64] introduced an approach for a hash-to-curve function, which is also recommended in RFC 9380 [36]. However, the most efficient method in the literature for hashing to pairing groups, including the case of `BLS12-381`, is the SwiftEC approach [26], with [4] reporting speedups up to $\times 2$ compared to Wahby and Boneh approach. Therefore, in our implementation we will consider the SwiftEC approach to instantiate the function $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$.

SwiftEC. We give a brief description of the SwiftEC approach and concentrate in the case of hashing arbitrary-length bitstrings to points in the elliptic curve subgroup $\mathbb{G}_1$. We note that the description can be trivially extended to the case of hashing to $\mathbb{G}_2$ as well. As explained earlier, the main difference between SwiftEC and other techniques is the definition of the encoding function H. SwiftEC requires two pseudorandom field elements and one extra random bit as input. These random values are derived in the following way. First, a function $\chi : \{0,1\}^* \to \{0,1\}^{1025}$ is required which maps a bitstring of arbitrary length to a bitstring of length 1025 bits. Note that χ can be instantiated as any message expansion function, such as `expand_message_xmd` in RFC 9380 [36]. Then the function $\eta : \{0,1\}^{1025} \to \mathbb{F}_p \times \mathbb{F}_p \times \{0,1\}$ is used to map this 1025-bit string into two field elements $u, t \in \mathbb{F}_p$ and a random bit $s \in \{0,1\}$. For instance, η can be instantiated as the function `hash_to_field`, defined in RFC 9380 [36].

The SwiftEC admissible encoding to the elliptic curve group $E_1(\mathbb{F}_p)$ is defined as the function

$$H : \mathbb{F}_p \times \mathbb{F}_p \times \{0,1\} \to E_1(\mathbb{F}_p).$$

Given the two random field elements $u, t \in \mathbb{F}_p$ and $s \in \{0,1\}$, the function computes a point $Q = (x_Q, y_Q)$ (in affine coordinates) on the curve $E_1/\mathbb{F}_p : y^2 = g(x) = x^3 + b$, where $b = 4$ in `BLS12-381`. The method

constructs three candidate x_Q-coordinates $x_1, x_2, x_3 \in \mathbb{F}_p$ via the formulas:

$$x_{1,2} = \frac{u\big(\tau(u^3+b-t^2) \mp (u^3+b+t^2)\big)}{2(u^3+b+t^2)} \quad \text{and} \quad x_3 = \frac{3u^3t^2 - (u^3+b+t^2)^2}{3u^2t^2}$$

where $\tau = \sqrt{-3} \in \mathbb{F}_p$ and computes the evaluations $v_1 = g(x_1)$, $v_2 = g(x_2)$ and $v_3 = g(x_3)$. SwiftEC takes $x_Q = x_1$ as valid coordinate and $v_Q = v_1$ as valid evaluation, meaning that v_1 is a square in $\mathbb{F}_p$. Then executes two square root tests for the evaluations v_2 and v_3. If some evaluation is a perfect square, x_Q and v_Q are updated. The method outputs the affine point $Q = (x_Q, y_Q)$, where $y_Q = (-1)^s\sqrt{v_Q}$ and s is the random bit given as input, which is used to randomly select the positive or negative square root.

An optimized and constant-time implementation of the SwiftEC method can be found in the RELIC library [3]. The most expensive operations in the SwiftEC approach are the two square root tests for v_2 and v_3 and the square root extraction to obtain the y_Q coordinate. The quadratic residuosity tests usually correspond to the computation of the Legendre symbol. An optimized square root test is described in the `blst_fp_is_square` function of the `blst` library, while the square root extraction can be implemented following the `blst_fp_sqrt` function[1].

Cofactor clearing. Another dominant operation in the hash-to-curve process is the cofactor clearing. Given a point $Q \in E_1(\mathbb{F}_p)$ as the output of the function H, this step maps Q to the prime-order subgroup $\mathbb{G}_1$ via the scalar multiplication $P = [h_1]Q$, with the cofactor $h_1 \mid \#E_1(\mathbb{F}_p)$. Recall that h_1 is a 126-bit scalar and hence this scalar multiplication is relatively expensive. To avoid costly scalar multiplications by the full cofactor, we adopt the optimization trick of Wahby and Boneh [64] for $\mathbb{G}_1$. More concretely, $h_1 = (z-1)^2/3$, where z is the seed, used to derive the `BLS12-381` curve. Since h_1 is a perfect square and $\sqrt{h_1}$ divides $p-1$, the trick of Wahby and Boneh simplifies cofactor clearing in $\mathbb{G}_1$ to a smaller scalar multiplication $P = [z-1]Q$, where $z-1$ has size 63 bits. Methods for optimizing the cofactor clearing in $\mathbb{G}_2$ are also discussed in [19] and are tailored to the `BLS12-381` case.

4.5.4.5 Additional components

We briefly discuss certain additional components that are critical to the efficiency and security of the pairing-based constructions described in this section. First, we focus on the scalar multiplication and present the state-of-the-art methods for implementing this operation, especially when the underlying protocol is instantiated using the `BLS12-381` curve. In addition, we discuss the concept of subgroup membership tests, which serve as the primary countermeasure against small subgroup attacks; a well-known vulnerability in elliptic curve cryptosystems in cases where the chosen elliptic curve has composite order that contains a small cofactor.

Scalar multiplication. Scalar multiplication is a core arithmetic operation in elliptic curve cryptography in general, hence also in the PS-based multi-signature and PS_SAS schemes. All functions in both schemes require at least one scalar multiplication, either in $\mathbb{G}_1$, or in $\mathbb{G}_2$. Therefore, performing this elliptic curve operation efficiently is crucial for the overall performance of the scheme. Several methods have been proposed for optimizing this subroutine and here, we concentrate on the special case of pairing-friendly elliptic curves. Depending on the length of the scalar, the implementation of the scalar multiplication in pairing-based schemes is categorized into the following cases: 1. For full-length 255-bit scalars (e.g., private keys) in $\mathbb{F}_r^*$, the general strategy is to deploy endomorphism-based decompositions to split the large 255-bit scalar into smaller sub-scalars. In particular, these methods include the Gallant-Lambert-Vanstone (GLV) [41] and Galbraith-Lin-Scott (GLS) [40] decompositions for $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively; 2. For small and fixed scalars, such as the seed z, the classical "double-and-add" technique

[1] See the `blst` Github repo: https://github.com/supranational/blst.

is followed, where such scalar multiplications appear mainly in the subgroup membership tests (see next paragraph). We briefly describe the GLV and GLS methods, focusing on the BLS12-381 case:

GLV decomposition. This method is used for scalar multiplications in $\mathbb{G}_1$. It splits a 255-bit scalar $\ell \in \mathbb{F}_r^*$ into two 128-bit sub-scalars ℓ_0 and ℓ_1 using the endomorphism $\phi(x, y) = (\zeta x, y)$, where ζ is a primitive third root of unity in $\mathbb{F}_p$. Then for any point $P \in \mathbb{G}_1$, the scalar multiplication is computed equivalently as $[\ell]P = [\ell_0]P + [\ell_1]\phi(P)$.

GLS decomposition. This method is used for scalar multiplications in $\mathbb{G}_2$. It uses an efficient endomorphism $\psi = \rho \circ \pi_p \circ \rho$, where $\rho : E_2(\mathbb{F}_{p^2}) \to E_1(\mathbb{F}_{p^{12}})$ is the twisting isomorphism, to decompose a 255-bit scalar $\ell \in \mathbb{F}_r^*$ into four 64-bit sub-scalars $\ell_0, \ell_1, \ell_2, \ell_3$. Then for any point $Q \in \mathbb{G}_2$, the scalar multiplication is computed equivalently as $[\ell]Q = \sum_{i=0}^{3}[\ell_i]\psi^i(Q)$.

Subgroup membership tests. Recall the BLS12-381 curve has order that contains a non-trivial cofactor h_1. The same is true for the degree six twist E_2 of E_1 as well, since $\#E_2(\mathbb{F}_{p^2}) = h_2 \cdot r$. This makes the proposed PS_SAS scheme susceptible to small-subgroup attacks, both in $\mathbb{G}_1$ and $\mathbb{G}_2$. Such an attack can be mounted by forcing a participant in a protocol to execute a scalar multiplication with a secret scalar, on a non-prime order point, aiming at obtaining information about the secret scalar. Such attacks are mitigated by verifying that core elements in a protocol (public keys and signatures), belong to the correct subgroup, i.e., they have the correct prime order. This is done by executing a scalar multiplication with the prime r and verifying that the resulting point is equal to the identity element in $\mathbb{G}_1$, or $\mathbb{G}_2$. However, this scalar multiplication is quite costly, since r is a 255-bit prime. In addition, considering the description of our proposed PS_SAS scheme in Algorithm 3, we see that the verification of the aggregated signature should include subgroup membership tests for the public keys $\mathrm{pk}_1, \ldots, \mathrm{pk}_n$ and for the aggregated signature $\Sigma_n = (t, T_1, T_2, S_1, \ldots, S_n)$, Hence, a total of n subgroup membership tests in $\mathbb{G}_2$ and $(n + 2)$ subgroup membership tests in $\mathbb{G}_1$ have to be executed.

The endomorphism-based tests proposed by Scott [60] are currently the most efficient methods to speedup subgroup membership tests both in $\mathbb{G}_1$ and $\mathbb{G}_2$, as they reduce the size of the scalar multiplication. More concretely, we distinguish the following two cases, depending on the underlying pairing-group:

Subgroup tests in $\mathbb{G}_1$. Given a point $P \in \mathbb{G}_1$, the test verifies whether the equation $\phi(P) = [-z^2]P$ is satisfied, where ϕ is the endomorphism $\phi(x, y) = (\zeta x, y)$, and ζ is a (precomputed) primitive third root of unity in $\mathbb{F}_p$. This test requires one $\mathbb{F}_p$ multiplication for computing the point $\phi(P)$ and one scalar multiplication for computing $[-z^2]P$, i.e., a multiplication with a scalar of size 126 bits.

Subgroup tests in $\mathbb{G}_2$. Given a point $Q \in \mathbb{G}_2$, the test verifies where the equation $\psi(Q) = [z]Q$ is satisfied, where $\psi = \rho \circ \pi_p \circ \rho$ is an endomorphism and $\rho : E_2(\mathbb{F}_{p^2}) \to E_1(\mathbb{F}_{p^{12}})$ is the twisting isomorphism. This approach significantly improves the subgroup test in $\mathbb{G}_2$ as it requires a scalar multiplication in $\mathbb{G}_2$ by the seed z (i.e., 63 bits), as well as the application of the Frbenius endomorphism π_p and the twisting isomorphism ρ, which are efficienctly computed.

4.6 Ordered-Preserving Constructions based on Matrix-Vector Relations

In addition to the PS_SAS scheme presented in Section 4.5.3, we introduce an alternative variant for path-oriented composite attestation based on BLS signatures, in which key constructions are represented as matrices. As this second scheme is currently at an early stage, we plan to evaluate and compare both schemes in terms of efficiency in the upcoming WP3 deliverables.

4.6.1 Building Blocks

This section provides the core building blocks for the second variant of composite attestation. The rest of the building blocks are already defined in section 4.5.2.

Upper Triangular Matrices. For a field $\mathbb{F}$ we define $H_{\mathbb{F}}$ to be the set of matrices in $\mathbb{F}^{2\times 2}$, for which the element in position $(1,0)$ is 0. More specifically, we define

$$T_{\mathbb{F}} = \left\{ \begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \mid (a,b,c) \in \mathbb{F} \right\}$$

Note that $T_{\mathbb{F}}$ is closed under matrix multiplication and addition as well as scalar multiplication. We will use $T_{\mathbb{Z}_q} \subset \mathbb{Z}_q^{2\times 2}$ in this work, where q is the order of a cyclic group $\mathbb{G}$.

For a group $\mathbb{G}$, we will also define the subset $\bar{T}_{\mathbb{G}}$ of $\mathbb{G}^{2\times 2}$ for which the element in position $(1,0)$ is the identity element of $\mathbb{G}$. More specifically, we define

$$\bar{T}_{\mathbb{G}} = \left\{ \begin{pmatrix} h_1 & h_2 \\ 1_{\mathbb{G}} & h_3 \end{pmatrix} \mid (h_1,h_2,h_3) \in \mathbb{G} \right\}$$

Matrix Exponent. For a matrix $S = \begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \in T_{\mathbb{Z}_q}$ and a group point $g \in \mathbb{G}$, we define the matrix g^S as:

$$g^S = \begin{pmatrix} g^a & g^b \\ 1_{\mathbb{G}} & g^c \end{pmatrix} \in \bar{T}_{\mathbb{G}}$$

In this work and following the notation summarized in Table 4.1, we will consider Type III bilinear maps of the form $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_{\mathsf{T}}$, meaning that $\mathbb{G}_1 \neq \mathbb{G}_2$ and there exists no efficient isomorphism $\tau : \mathbb{G}_2 \to \mathbb{G}_1$.

4.6.2 Construction

At a high level, each device's (e.g., vRouter's) secret key will be replaced by a 2×2 random matrix in $T_{\mathbb{Z}_q}$. For a secret key $S \in T_{\mathbb{Z}_q}$, the corresponding public key is defined as $PK = g^S \in \bar{T}_{\mathbb{G}_2}$, where $g \in \mathbb{G}_2$. For each ordered collection of nodes (notated by N_i) the trusted authority (TA), i.e. the CASTOR Service Orchestrator, will calculate a unique public key in $\mathbb{G}_2^{2\times 2}$. That public key will uniquely identify the order of the nodes except with negligible probability.

Let a set of network elements (e.g., nodes) $N = (N_0, N_1, ..., N_m)$ (denoted as a path in the following). Each $N_i \in N$, $i = 0, ..., m$ will collaborate with the TA to jointly calculate it's secret key $S_i \in T_{\mathbb{Z}_q}$ and its public key $PK_i = g^{S_i} \in \bar{T}_{\mathbb{G}_2}$. The TA will then calculate and publish the unique public key of the path as $PK_{path} = g^S$, for which it holds that $S = \prod_{i=0}^{m} S_i$.

A signature on message msg from the set of nodes N will be denoted as $\sigma_{path} = \mathsf{H}_1(msg)^{\prod_{j=0}^{m} S_j}$, where $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$. Let σ_{i-1} be the signature that has been calculated from the nodes $(N_0, ..., N_{i-1})$. It holds that $\sigma_{i-1} = \mathsf{H}_1(msg)^{\prod_{j=0}^{i-1} S_j}$. Then, node N_i, given the message msg, σ_{i-1} and its secret key $S_i = \begin{pmatrix} a_i & sk_i \\ 0 & b_i \end{pmatrix}$, will calculate σ_i, before forwarding it to the next node. Security of our scheme is based on the fact that with high probability, random matrices are non-cumulative. Following, we will describe each step in more detail.

Key Gen. The key generation procedure of our system is divided into two faces; first the key generation of each node N_i, which is calculated jointly between the TA and the node, and second, the generation of the path's public key, which is calculated and published by the TA. Following we describe each phase separately.

Algorithm 4: Path public key calculation

1: **Input:** $\{PK_i\}_{i=0,...,m}$
2: Let $\{PK_i = \begin{pmatrix} g^{a_i} & pk_i \\ 1_{\mathbb{G}_2} & g^{b_i} \end{pmatrix}\}_{i=0,1,...,m}$
3: Set $PK_{\text{path}} \leftarrow \begin{pmatrix} g^{a_0} & pk_0 \\ 1_{\mathbb{G}_2} & g^{b_0} \end{pmatrix}$
4: **for** $i = 1$ to m **do**
5: Let $h_1, h_2, h_3 \in \mathbb{G}_2$ so that $PK_{\text{path}} = \begin{pmatrix} h_1 & h_2 \\ 1_{\mathbb{G}_2} & h_3 \end{pmatrix}$
6: Set $PK_{\text{path}} \leftarrow \begin{pmatrix} h_1^{a_i} & h_2^{b_i} \cdot pk_i^{a_{i-1}} \\ 1_{\mathbb{G}_2} & h_3^{b_i} \end{pmatrix}$
7: **end for**
8: **return** PK_{path}

A. Node key generation. Let $g \in \mathbb{G}_2$ be a generator of $\mathbb{G}_2$. Each node N_i samples a random key $sk_i \xleftarrow{R} \mathbb{Z}_p$ and calculates the public part as $pk_i = g^{sk_i} \in \mathbb{G}_2$. Each node will then submit to the TA the public key pk_i, together with a proof of possession of sk_i, i.e., $\pi_i = PoP\{sk_i \,|\, pk_i = g^{sk_i}\}(n_i)$, where n_i a random nonce provided by the TA. On its turn, the TA, after receiving and validating the (pk_i, π_i) message from the node $N_i \in N$, will generate two random scalars $a_i, b_i \xleftarrow{R} \mathbb{Z}_p$ and set:

$$PK_i = \begin{pmatrix} g^{a_i} & pk_i = g^{sk_i} \\ 1_{\mathbb{G}_2} & g^{b_i} \end{pmatrix}$$

Finally, the TA will (securely) return the a_i and b_i values to N_i.

B. Path public key calculation. After each node in $N_0, N_1, ..., N_m$ has successfully executed the node key generation procedure defined above, the TA will calculate the public key of the path using Algorithm 4. It is easy to verify that the derived public key PK_{path} lies in in $\bar{T}_{\mathbb{G}_2}$ and that $PK_{path} = g^{\prod_{j=0}^{m} S_j}$ (see Lemma 1, in Section 4.6.3).

Sign. Following, we will describe the Sign procedure of our scheme. Our construction will use the matrix version of BLS signatures, i.e., the final signature of the path for a message msg will be of the form $\sigma_{path} = \mathsf{H}_1(msg)^S \in \bar{T}_{\mathbb{G}_1}$ where $S = \prod_{j=0}^{m} S_i$. Next, we will start by describing the signature generation of the first node N_0 and then of each node N_i for $i = 1, ..., m$.

Sign of N_0. The node N_0, on input the message msg and it's secret key $S_0 = \begin{pmatrix} a_0 & sk_0 \\ 0 & b_0 \end{pmatrix}$, will generate the signature:

$$\sigma_0 = \mathsf{H}_1(msg)^{S_0} = \begin{pmatrix} \mathsf{H}_1(msg)^{a_0} & \mathsf{H}_1(msg)^{sk_0} \\ 1_{\mathbb{G}_1} & \mathsf{H}_1(msg)^{b_0} \end{pmatrix}$$

Sign of N_i. On input the message msg, it's key $S_i = \begin{pmatrix} a_i & sk_i \\ 0 & b_i \end{pmatrix}$ and the signature σ_{i-1} calculated thus far by nodes $N_0, N_1, ..., N_{i-1}$, node N_i will calculate the signature σ_i as follows:

1. If σ_{i-1} is not in $\bar{T}_{\mathbb{G}_1}$, return false
2. Let $t_1, t_2, t_3 \in \mathbb{G}_1$ so that $\sigma_{i-1} = \begin{pmatrix} t_1 & t_2 \\ 1_{\mathbb{G}_1} & t_3 \end{pmatrix}$

3. Set $\sigma_i = \begin{pmatrix} t_1^{a_i} & t_2^{b_i} t_1^{sk_i} \\ 1_{\mathbb{G}_1} & t_3^{b_i} \end{pmatrix}$

The final σ_{path} signature of the path will be the signature returned by the last node N_m. As we will demonstrate with Lemma 2 in Section 4.6.3, it holds that $\sigma_{path} = \mathsf{H}_1(msg)^{\prod_{j=0}^{m} S_i}$.

Verify. Given the public key of the path PK_{path} and the signature of the path σ_{path} on the message msg, the verification procedure will validate the signature, using an adjusted version of the BLS signature verification algorithm. More specifically, the verification procedure is involves the following 5 steps:

1. If σ_{path} is not in $\bar{T}_{\mathbb{G}_1}$, return false.
2. If PK_{path} is not in $\bar{T}_{\mathbb{G}_2}$, return false.
3. Let $t_{path}^{(1)}, t_{path}^{(2)}, t_{path}^{(3)} \in \mathbb{G}_1$ so that $\sigma_{path} = \begin{pmatrix} t_{path}^{(1)} & t_{path}^{(2)} \\ 1_{\mathbb{G}_1} & t_{path}^{(3)} \end{pmatrix}$.
4. Let $h_{path}^{(1)}, h_{path}^{(2)}, h_{path}^{(3)} \in \mathbb{G}_2$ so that $PK_{path} = \begin{pmatrix} h_{path}^{(1)} & h_{path}^{(2)} \\ 1_{\mathbb{G}_2} & h_{path}^{(3)} \end{pmatrix}$.
5. For, $l = 1, 2, 3$, if $e(t_{path}^{(l)}, g) \neq e(\mathsf{H}_1(msg), h_{path}^{(l)})$ return false, otherwise, return true.

4.6.3 Security

We will start by showcasing the correctness of our scheme before discussing unforgeability.

4.6.3.1 Correctness

Correctness will be argued for each phase of our construction separately, i.e., for **KeyGen**, **Sign** and **Verify**. Let nodes $N_0, N_1, ..., N_m$ having completed the node key generation procedure as described in Section 4.6.2, resulting to each node having a matrix secret key $\{S_i\}_{i=0,1,..,m}$. Correctness of KeyGen will amount to showing that for the PK_{path} returned by the path public key calculation procedure (Algorithm 4), it holds that $PK_{path} = g^{\prod_{j=0}^{m} S_i}$. For the correctness of the Sign process, we will showcase that for a signature σ_{path} returned by the path $N_0, N_1, ..., N_m$, it holds $\sigma_{path} = \mathsf{H}_1(msg)^{\prod_{j=0}^{m} S_i}$. Finally, we will show that for honest nodes $N_0, N_1, ..., N_m$, collaborating to sign a message msg, the Verify algorithm described in Section 4.6.2 will always return true.

Starting with KeyGen correctness, we will prove the following Lemma.

Lemma 1. Let $S_i = \begin{pmatrix} a_i & sk_i \\ 0 & b_i \end{pmatrix}$ the secret key of node N_i, corresponding to its public key $PK_i = \begin{pmatrix} g^{a_i} & pk_i = g^{sk_i} \\ 1_{\mathbb{G}_2} & g^{b_i} \end{pmatrix}$. Then, at iteration i of Algorithm 4, for the PK_{path} calculated at step (6) it holds that $PK_{path} = g^{\prod_{j=0}^{i} S_j}$.

Proof. We will prove the Lemma by induction. Let PK_{path}^{i} to be the value of PK_{path} calculated at step (6) of Algorithm 4 in iteration i. It holds that $PK_{path}^{0} = \begin{pmatrix} g^{a_0} & pk_0 \\ 1_{\mathbb{G}_2} & g^{b_0} \end{pmatrix}$. Assume that $PK_{path}^{i} = g^{\prod_{j=0}^{i} S_j}$. Note that, $T_{\mathbb{Z}_q}$ is closed under matrix multiplication. So there are, $a', c', b' \in \mathbb{Z}_p$ so that $\prod_{j=0}^{i} S_j = \begin{pmatrix} a' & c' \\ 0 & b' \end{pmatrix}$.

Then, using Algorithm 4, at iteration $i+1$, by assumption, PK^i_{path} (i.e., PK_{path} as calculated at step (5) in the previous iteration i) will be

$$PK^i_{path} = \begin{pmatrix} h_1 & h_2 \\ 1_{\mathbb{G}_2} & h_3 \end{pmatrix} = g^{\prod_{j=0}^{i} S_j} = \begin{pmatrix} g^{a'} & g^{c'} \\ 1_{\mathbb{G}_2} & g^{b'} \end{pmatrix}$$

Then, in step (6) of Algorithm 4 (where PK_{path} is calculated at iteration $i+1$), it will hold that

$$PK^{i+1}_{path} = \begin{pmatrix} h_1^{a_{i+1}} & h_2^{b_{i+1}} pk_{i+1}^{a_i} \\ 1_{\mathbb{G}_2} & h_3^{b_{i+1}} \end{pmatrix} = \begin{pmatrix} g^{a'a_{i+1}} & g^{c'b_{i+1}} pk_{i+1}^{a_i} \\ 1_{\mathbb{G}_2} & g^{b'b_{i+1}} \end{pmatrix}$$

Note that, $pk_{i+1} = g^{sk_{i+1}}$ meaning that $g^{c'b_{i+1}} pk_{i+1}^{a_i} = g^{c'b_{i+1} + sk_{i+1}a_i}$. Given that

$$\left(\prod_{j=0}^{i} S_j\right) S_{i+1} = \begin{pmatrix} a' & c' \\ 0 & b' \end{pmatrix} \begin{pmatrix} a_{i+1} & sk_{i+1} \\ 0 & b_{i+1} \end{pmatrix} = \begin{pmatrix} a'a_{i+1} & a'sk_{i+1} + c'b_{i+1} \\ 0 & b'b_{i+1} \end{pmatrix}$$

we get that $PK^{i+1}_{path} = g^{\prod_{j=0}^{i+1} S_j}$, witch concludes the proof. □

Following, we will showcase Sign correctness by proving the following Lemma.

Lemma 2. Let the secret key of node N_i be $S_i = \begin{pmatrix} a_i & sk_i \\ 0 & b_i \end{pmatrix}$. For a message msg, let σ_{path} be the signature of the path $N_0, N_1, ..., N_m$, calculated by the above procedure. Then it holds that $\sigma_{path} = \mathsf{H}_1(msg)^S$, where $S = \prod_{j=0}^{m} S_j$.

Proof. We prove that, for $i = 0, 1, ..., m$ it holds that $\sigma_i = \mathsf{H}_1(msg)^{\prod_{j=0}^{i} S_j}$ by induction. By construction, for $i = 0$, we have $\sigma_0 = \mathsf{H}_1(msg)^{S_0}$. We now assume that the statement holds for some $i \in 1, ..., m-1$, i.e., we assume that $\sigma_i = \mathsf{H}_1(msg)^{\prod_{j=0}^{i} S_j}$. Since $T_{\mathbb{Z}_q}$ is closed under matrix multiplication, there exist $a', c', b' \in \mathbb{Z}_p$ so that $\prod_{j=0}^{i} S_i = \begin{pmatrix} a' & c' \\ 0 & b' \end{pmatrix}$ and $t_1, t_2, t_3 \in \mathbb{G}_1$, so that,

$$\sigma_i = \begin{pmatrix} t_1 & t_2 \\ 1_{\mathbb{G}_1} & t_3 \end{pmatrix} = \mathsf{H}_1(msg)^{\prod_{j=0}^{i} S_j} = \begin{pmatrix} \mathsf{H}_1(msg)^{a'} & \mathsf{H}_1(msg)^{c'} \\ 1_{\mathbb{G}_1} & \mathsf{H}_1(msg)^{b'} \end{pmatrix}$$

Then, by the Sign procedure described in Section 4.6.2, for the node N_{i+1}, the following hold:

$$\sigma_{i+1} = \begin{pmatrix} t_1^{a_{i+1}} & t_2^{b_{i+1}} t_1^{sk_{i+1}} \\ 1_{\mathbb{G}_1} & t_3^{b_{i+1}} \end{pmatrix}, \quad t_1^{a_{i+1}} = \mathsf{H}_1(msg)^{a'a_{i+1}}, \quad t_3^{b_{i+1}} = \mathsf{H}_1(msg)^{b'b_{i+1}}$$

as well as

$$t_2^{b_{i+1}} t_1^{sk_{i+1}} = \mathsf{H}_1(msg)^{c'b_{i+1} + a'sk_{i+1}}$$

Then we obtain the following relation:

$$\left(\prod_{j=0}^{i} S_i\right) S_{i+1} = \begin{pmatrix} a' & c' \\ 0 & b' \end{pmatrix} \begin{pmatrix} a_{i+1} & sk_{i+1} \\ 0 & b_{i+1} \end{pmatrix} = \begin{pmatrix} a'a_{i+1} & a'sk_{i+1} + c'b_{i+1} \\ 0 & b'b_{i+1} \end{pmatrix}$$

As a result, $\sigma_{i+1} = \mathsf{H}_1(msg)^{\prod_{j=0}^{i+1} S_j}$, concluding the proof. □

Given Lemma 1 and Lemma 2, we can showcase that the Verify algorithm described in Section 4.6.2 will always succeed to verify honestly generated signatures. More specifically, since $T_{\mathbb{Z}_q}$ is close under matrix multiplication, there exist $a, c, b \in \mathbb{Z}_q$ so that $\prod_{j-0}^{m} S_j = \begin{pmatrix} a & c \\ 0 & b \end{pmatrix}$. From Lemma 1 we have that

$$PK_{path} = g^{\prod_{j-0}^{m} S_j} = \begin{pmatrix} g^a & g^c \\ 1_{\mathbb{G}_2} & g^b \end{pmatrix}$$

while from Lemma 2 we have:

$$\sigma_{path} = \mathsf{H}_1(msg)^{\prod_{j-0}^{m} S_j} = \begin{pmatrix} \mathsf{H}_1(msg)^a & \mathsf{H}_1(msg)^c \\ 1_{\mathbb{G}_1} & \mathsf{H}_1(msg)^b \end{pmatrix}$$

As a result, for $\{t_{path}^{(l)}\}_{l=1,2,3}$, $\{h_{path}^{(l)}\}_{l=1,2,3}$ as set at steps (3) and (4) of Verify and for $v_1 = a, v_2 = c$ and $v_3 = b$, it will hold that

$$e(t_{path}^{(l)}, g) = e(\mathsf{H}_1(msg)^{v_l}, g) = e(\mathsf{H}_1(msg), g^{v_l}) = e(\mathsf{H}_1(msg), h_{path}^{(l)})$$

meaning that Verify will return true.

4.6.3.2 Unforgeability (sketch)

We sketch here the proof that an adversary cannot create valid signatures for a re-arrangement of the defined order of the nodes, thereby showing that the proposed scheme is order-preserving with respect to signatures. We start by discussing the following intermediate Lemma. In particular, we will show that the secret key of the path, i.e., the value sk_{path} in $\begin{pmatrix} a & sk_{path} \\ 0 & b \end{pmatrix} = \prod_{j \in I} S_i$ can be expressed as a linear combination of the secret keys sk_i of each node, with coefficients that are random and independent from each of those secret keys. To not overload the notation we will apply the following convention;

For a field $\mathbb{F}$ and an arbitrary sequence $\{a_i\} \in \mathbb{F}$
we denote $\prod_{j=m_1}^{m_2} a_j = 1$ if $m_2 < m_1$

We proceed with the Lemma statement.

Lemma 3. Let $\{S_i = \begin{pmatrix} a_i & c_i \\ 0 & b_i \end{pmatrix}\}_{i=I} \in T_{\mathbb{Z}_q}$ for $I = \phi(\{0, 1, 2, ..., m\})$ where ϕ any shuffling. Let $\begin{pmatrix} a & c \\ 0 & b \end{pmatrix} = \prod_{i \in I} S_i$ for $a, b, c \in \mathbb{Z}_q$. Then, $c = \sum_{i=0}^{m} f_i(\{a_j\}_{j \in I}, \{b_j\}_{j \in I}) c_{\phi(i)}$, where f_i has the following form,

$$f_i(\{X_j\}_{j=0,...,m}, \{Y_j\}_{j=0,...,m}) = \prod_{j=0}^{i-1} X_j \prod_{j=i+1}^{m} Y_j$$

Proof. Note that since $T_{\mathbb{Z}_q}$ is close under matrix multiplication, as a result $S \in T_{\mathbb{Z}_q}$, so there exists $a, b, c \in \mathbb{Z}_q$ so that $S = \begin{pmatrix} a & c \\ 0 & b \end{pmatrix}$. We will prove the Lemma by induction on the number m of matrices on the product

Hypothesis: Let $I = \{i_0, i_1, ..., i_m\}$. For every $n \leq m$, let $\begin{pmatrix} a^n & c^n \\ 0 & b^n \end{pmatrix} = \prod_{l=i_0, i_1, ..., i_{n-1}, i_n} S_l$, then $c^n = \sum_{k=0}^{n} f_k(\{a_l\}_{l=i_0,...,i_n}, \{b_l\}_{l=i_0,...,i_n}) c_{\phi(k)}$, where f_k as in the Lemma statement.

The case $n = 0$: Note that $\begin{pmatrix} a^0 & c^0 \\ 0 & b^0 \end{pmatrix} = S_{i_0} \implies c^0 = c_{i_0} = c_{\phi(0)}$. Also $f_0(X_0, Y_0) = \prod_{j=0}^{-1} X_j \prod_{j=1}^{0} Y_j = 1$ by the convention discussed before the Lemma, which shows that the hypothesis holds for $n = 0$.

The case $n > 0$: Let the hypothesis hold for some n. In particular, let:

$$\begin{pmatrix} a^{n+1} & c^{n+1} \\ 0 & b^{n+1} \end{pmatrix} = \prod_{l=i_0,...,i_{n+1}} S_n = \begin{pmatrix} a^n & c^n \\ 0 & b^n \end{pmatrix} \begin{pmatrix} a_{i_{n+1}} & c_{i_{n+1}} \\ 0 & b_{i_{n+1}} \end{pmatrix}$$

which implies that $c^{n+1} = a^n c_{i_{n+1}} + b_{i_{n+1}} c^n$. From the induction hypothesis, we can conclude that c^{n+1} is a linear combination of all c_j, for $j = i_0, i_1, ..., i_{n+1}$, i.e., that $c^{n+1} = \sum_{k=0}^{n+1} r_i c_{i_k} = \sum_{k=0}^{n+1} r_i c_{\phi(k)}$.

We will show know that the r_i have the correct form, given by f_i. Starting with the coefficient r_{n+1} of $c_{i_{n+1}}$, from the above we have that $r_{n+1} = a^n$. From the triangular matrix multiplication properties, it holds that $a^n = \prod_{l=i_0,...,i_n} a_l$. We can see that

$$f_{n+1}(\{a_l\}_{l=i_0,...,i_{n+1}}, \{b_l\}_{l=i_0,...,i_{n+1}}) = \prod_{j=0}^{n} a_{i_j} \prod_{j=n+2}^{n+1} b_{i_j} = \prod_{l=i_0,...,i_n} a_l = a^n$$

i.e., r_{n+1} has the correct form.

For r_k *with* $k = 0, ..., n$: From the induction hypothesis, we know that $c^n = \sum_{k=0}^{n} r'_k c_{\phi(k)}$, meaning that $r_k = b_{i_{n+1}} r'_k$. Again from the induction hypothesis, we get that $r'_k = f_k(\{a_l\}_{l=i_0,...,i_n}, \{b_l\}_{l=i_0,...,i_n})$. We can conclude that,

$$r_k = b_{i_{n+1}} r'_k = (\prod_{j=0}^{k-1} a_{i_j} \prod_{j=k+1}^{n} b_{i_j}) b_{i_{n+1}} = \prod_{j=0}^{k-1} a_{i_j} \prod_{j=k+1}^{n+1} b_{i_j} = f_k(\{a_l\}_{l=i_0,...,i_{n+1}}, \{b_l\}_{l=i_0,...,i_{n+1}})$$

for f_k as described in the Lemma statement, which concludes the proof. □

Using the above Lemma, we will show that the adversary, controlling a number of nodes in the path, cannot change the order that those nodes sign in the path and generate a valid signature, under the path's public key PK_{path} (other forgeries, will reduce to the unforgeability properties of BLS signatures).

Let the indexes $I = \{i_0, i_1, ..., i_m\} = \phi(\{1, 2, ..., m\})$ be the original (correct) ordering of the nodes $\{N_0, N_1, ..., N_m\}$, where ϕ a shuffling (for simplicity we can assume that ϕ the identity function). Let also PK_{path} the public key corresponding to that ordering, as calculated by the TA. Let also ϕ_{adv} a shuffling, different from the identity function and $I' = \phi_{adv}(I) = \{i'_0, i'_1, .., i'_n\}$, the ordering forced by the adversary (note that $I \neq I'$).

Note that, if $S_i = \begin{pmatrix} a_i & sk_i \\ 0 & b_i \end{pmatrix}$, from Lemma 1, $PK_{path} = g^{\prod_{i \in I} S_i}$. By assuming that the adversary is not able to forge a BLS signature, the re-ordering of the nodes by the adversary, should result to the same PK_{path} (which verifies the path signature), meaning that $g^{\prod_{i \in I} S_i} = g^{\prod_{i' \in I'} S_{i'}} \implies \prod_{i \in I} S_i = \prod_{i' \in I'} S_{i'}$. Let $a, sk_{path}, b \in \mathbb{Z}_q$ and $a', sk'_{path}, b' \in \mathbb{Z}_q$ so that

$$prod_{i \in I} S_i = \begin{pmatrix} a & sk_{path} \\ 0 & b \end{pmatrix} = \prod_{i' \in I'} S_{i'} = \begin{pmatrix} a' & sk'_{path} \\ 0 & b' \end{pmatrix}$$

From Lemma 3 we get that

$$sk_{path} = \sum_{i=0}^{m} r_i sk_{\phi(i)} = sk'_{path} = \sum_{i=0}^{m} r'_i sk_{\phi_{adv}(i)}$$

where $r_i = f_i(\{a_j\}_{j\in I}, \{b_j\}_{j\in I})$ and $r'_i = f_i(\{a_j\}_{j\in I'}, \{b_j\}_{j\in I'})$, for f_i as defined in Lemma 3.Then we can conclude that:0

$$\sum_{i=0}^{m}(r_{\phi^{-1}(i)} - r'_{\phi_{adv}^{-1}(i)})sk_i = 0 \tag{4.4}$$

Note that r_i (correspondingly r'_i) is dependent on i and on the order I (correspondingly I') of matrices on the product $\prod_{i\in I} S_i$ (or $\prod_{i\in I'} S_i$). Additionally, each r_i and r'_i depend only on the values $\{a_i\}_{i=0,...,m}$ and $\{b_i\}_{i=0,...,m}$, which are chosen by the TA to be random and independent from each sk_i. We can conclude then that also the r_i and r'_i are random and independent from each sk_i.

From the construction of f_i we can see that for $i \neq i' \implies f_i \neq f_{i'}$. As a result, for random and independent $\{a_i, b_i\}_{i=0,...,m}$, we get that $r_i = f_i(\{a_j\}_{j\in I}, \{b_j\}_{j\in I}) \neq f_{i'}(\{a_j\}_{j\in I'}, \{b_j\}_{j\in I'}) = r'_{i'}$, except with negligible probability. From equation 4.4, since not all $(r_{\phi^{-1}(i)} - r'_{\phi_{adv}^{-1}(i)})$ are 0 (unless $\phi = \phi_{adv}$), we can conclude that the adversary will be able to find a linear combination of the adversarially controlled secret keys, with coefficients that are random and independent from each sk_i, which can happen only with negligible probability.

4.6.4 Misbehaviour tracking & Node Revocation (Work-In-Progress)

In this section we will describe a misbehavior tracking algorithm with which the TA will be able to trace which nodes did not perform the Sign operation correctly. The tracking procedure will require some additions to the Sign operation described in Section 4.6.2, which we analyse below. We note here that the tracking procedure works in a worst case bases. More specifically, in certain cases, the TA will not be able to distinguish, which one of two consequent nodes in the path are adversarial. In that case, our tracking mechanism will label both as misbehaving. A discussion on ways to weaken this assumption will be included in future deliverables.

The key idea in the misbehavior tracing procedure is that each node will keep some state for each signature generation procedure, possibly for a pre-defined time window, for which tracking of misbehaving nodes will be possible (outside that window, all nodes will be revoked and the key generation of Section 4.6.2 will run again). If a signature from the path is invalid, the Verifier will request from the TA the tracing of the nodes that did not follow the Sign protocol correctly, by submitting the invalid signature, together with some extra, constant-size, information provided by the last node in the path. The TA will then engage to an interactive protocol with each node in the path $N_0, N_1, ..., N_m$ (starting from N_m and working towards N_0), until it finds all the misbehaving nodes.

An additional requirement is that each signature is bound to a unique session identifier $ssid$. Such a session id can be generated by the first node using public information (for example $Verifier\_ID||timestamp$), provided by the Verifier as a random nonce etc. In the following paragraphs, we describe the updates to the Sign and Verify operations from Section 4.6.2, as well as the procedure followed by the TA to trace misbehaving nodes.

Sign with tracking. In Sign, the first node on the path, namely N_0, after calculating σ_0, it will also calculate $H_0 = \mathsf{H}_2(msg, \sigma_0, ssid;\, a_0||b_0)$, i.,e., the hash of msg, σ_0 and $ssid$, keyed with $a_0||b_0$ (see Table 4.1 for the definition of the keyed hash function H_2). Then, together with σ_0, it will also provide H_0 to the next node in the path (i.e., to N_1).

For each $i > 0$, each node N_i receives the signature σ_{i-1} (i.e., the signature generated by the first $i-1$ nodes of the path) and the hash H_{i-1}. Then during the Sign process, it will first calculate σ_i following the procedure from Section 4.6.2 and then is computes the hash $H_i = \mathsf{H}_2(\sigma_i, H_{i-1}, ssid;\, a_i||b_i)$. Finally, the last node in the path N_m will send to the Verifier the signature $\sigma_{path} = \sigma_m$ and the hash $H_{path} = H_m = \mathsf{H}_2(\sigma_m, H_{m-1}, ssid;\, a_m||b_m)$.

Algorithm 5: Node Validation

function NodeValidate$(s_i, \sigma_i, \sigma_{i-1}, H_{i-1}, ssid)$**:**
  if $e(s_i, g) \neq e(\mathsf{H}_1(msg), pk_i)$ **then**
    return `NodeKeyError`
  else if $H_i \neq \mathsf{H}_2(\sigma_i, H_{i-1}, ssid;\ a_i||b_i)$ **then**
    return `HashError`
  else
    Using σ_{i-1} and s_i, calculate the expected $\sigma_i^{expected}$
    if $\sigma_i \neq \sigma_i^{expected}$ **then**
      return `SignatureError`
  return `true`

Verify with tracking. Upon receiving the path signature σ_{path} and the digest H_{path}, the Verifier will attempt to verify the signature using the procedure described in Section 4.6.2, with no modifications. If the verification fails, the Verifier will send the path signature σ_{path}, the message msg and the hash H_{path} to the TA, in order for the latter to identify the faulty signatures that resulted in an invalid aggregate signature σ_{path} (tracking of misbehaving nodes).

TA tracking of nodes. We will first showcase that the TA can use the partial signature σ_{i-1} over the message msg and the value $s_i = \mathsf{H}_1(msg)^{sk_i}$ to recover the next signature σ_i. Note that from the proof of Lemma 2, we have $\sigma_{i-1} = H_C(msg)^{\prod_{j=0}^{i-1} S_j}$. Let $t_1, t_2, t_3 \in \mathbb{G}_1$ as in step (2) of Sign of N_i in Section 4.6.2, so that $\sigma_{i-1} = \begin{pmatrix} t_1 & t_2 \\ 1_{\mathbb{G}_1} & t_3 \end{pmatrix}$. From the above we can conclude that $t_1 = \mathsf{H}_1(msg)^{\prod_{j=0}^{i-1} a_j}$. Recall (step(3) of Sign of N_i from Section 4.6.2) that $\sigma_i = \begin{pmatrix} t_1^{a_i} & t_2^{b_2} t_1^{sk_i} \\ 1_{\mathbb{G}_1} & t_3^{b_i} \end{pmatrix}$. As a result, by knowing $s_i = \mathsf{H}_1(msg)^{sk_i}$, the TA can calculate

$$t_1^{sk_i} = s_i^{\prod_{j=0}^{i-1} a_j} = (\mathsf{H}_1(msg)^{sk_i})^{\prod_{j=0}^{i-1} a_j}$$

since all a_i are known to the TA.

Algorithm 5 describes the NodeValidate procedure, executed by the TA. Using NodeValidate, the TA will check if node N_i (for $i > 0$) has generated their outputs correctly.

Following, we will define a node consistency check algorithm NodeConsistency in Algorithm 6. The TA will use the NodeConsistency to check if the claimed inputs of node N_i are consistent with the claimed outputs of node N_{i-1} (for $i > 0$).

Using the above algorithms and depending on the returned error, the TA will take a decision on which node should be labelled as misbehaving. For this purpose, the TA will initiate an empty list $M = ()$. It will the first ping each node starting with N_0. In the following we describe the TA's procedure for each node.

Node N_0. The TA will request the value $s_0 = \mathsf{H}_1(msg)^{sk_0}$, σ_0 and H_0 and will execute the following verification tests:

$$e(s_0, g) \stackrel{?}{=} e(\mathsf{H}_1(msg), pk_0), \quad \sigma_0 \stackrel{?}{=} \begin{pmatrix} \mathsf{H}_1(msg)^{a_0} & s_0 \\ 0 & \mathsf{H}_1(msg)^{b_0} \end{pmatrix}, \quad H_0 \stackrel{?}{=} \mathsf{H}_2(\sigma_0, ssid;\ a_0||b_0)$$

If any of the checks fail, it will add N_0 to the list M.

Node N_i, $i > 0$. For each node N_i, with $i > 0$, the TA will work as follows. First note that, the TA will have ping node N_{i-1} to get $\sigma_{i-1}, H_{i-1}, H_{i-2}$. It will now ping N_i to get s_i, σ_i, H_i and the inputs

Algorithm 6: Node Consistency

```
function NodeConsistency(s_i, σ_i, σ_{i-1}, H_{i-1}, ssid):
    Define event A: σ_{i-1} = σ̄_{i-1}
    Define event B: H_{i-1} = H̄_{i-1}
    Define event C: H̄_{i-1} = H(σ̄_{i-1}, H_{i-2}, ssid; a_{i-1}||b_{i-1})
    if A & ¬B then
        return InconsistentHash
    else if ¬A & B then
        return InconsistentSignature
    else if ¬A & ¬B then
        if C then
            return ConsistentHashInconsistentSignature
        else
            return InconsistentHashAndSignature
    return true
```

that N_i claims to have received from N_{i-1}, i.e., $\bar{\sigma}_{i-1}, \bar{H}_{i-1}$. The TA will then run the algorithm NodeValidate$(\sigma_i, H_i, a_i, b_i, ssid)$ (Algorithm 5). If the algorithm does not return `true`, it will add N_i to the list M. It will then run NodeConsistency$(\sigma_{i-1}, \bar{\sigma}_{i-1}, H_{i-1}, \bar{H}_{i-1}, H_{i-2})$ and distinguish the following two cases:

1. If NodeConsistency returns `ConsistentHashInconsistentSignature` the TA will add N_{i-1} in M (if not already added).
2. In any other case, the TA will add both N_{i-1} and N_i to M (if not already added).

We now discuss case (1) above. If NodeConsistency returns `ConsistentHashInconsistentSignature` after pinging node N_i, it means that N_i and N_{i-1} agree to the hash value H_{i-1}, while the signature value σ_{i-1} that node N_{i-1} claims to have forwarded to N_i is different from the one that node N_i claims to have received (i.e., $\bar{\sigma}_{i-1}$). However it holds that $H_{i-1} = \mathsf{H}_2(\bar{\sigma}_{i-1}, H_{i-2}, ssid; a_{i-1}||b_{i-1})$ (for which both nodes agree), with $\sigma_{i-1} \neq \bar{\sigma}_{i-1}$ and since only the node N_{i-1} could have calculated that hash value, it means that, during that calculation it used $\bar{\sigma}_{i-1}$ and not the claimed σ_{i-1}, resulting to its labelling as misbehaving.

In any other case a inconsistency is discovered (case (2) above) the TA is not able to reliable decide which of the two nodes (N_{i-1} or N_i) is responsible. As a result, it will apply the conservative approach and both nodes will be added to the list M. In the next section, we will describe how the presence of a trusted execution layer in each node can provide more information to the TA, as to make a more fine grained decision.

4.6.4.1 Trusted commitment

In this section, we will assume the presence of a trusted execution layer in each node with the following minimum assumption; After node N_i calculates the signature σ_i, it will forward it, together with H_{i-1} and $ssid$ to the trusted layer, which on its turn, use it to calculate the hash $H_i = \mathsf{H}_2(\sigma_i, H_{i-1}, ssid; a_i||b_i)$ and forward σ_i, H_i to N_{i+1}.

The above will guarantee that whatever signature the node computes, it will be the one hashed and send to the next node. Using that guarantee, the TA will be able to make more informed decisions on which nodes should be labelled as misbehaving after running the NodeConsistency procedure (Algorithm 6). More specifically, we can now distinguish the following cases:

1. If NodeConsistency returns `InconsistentHash` (i.e., $\sigma_{i-1} = \bar{\sigma}_{i-1}$ and $H_{i-1} \neq \bar{H}_{i-1}$) the TA will calculate $H_{correct} = \mathsf{H}_2(\sigma_{i-1}, H_{i-2}, ssid;\ a_{i-1}||b_{i-1})$ and distinguish the following cases;
   (a) If $H_{i-1} = H_{correct}$, then add N_i to M.
   (b) If $\bar{H}_{i-1} = H_{correct}$, then add N_{i-1} to M.
   (c) Else, add both N_i and N_{i-1} to M.
2. If NodeConsistency returns `InconsistentSignature` (i.e., $\sigma_{i-1} \neq \bar{\sigma}_{i-1}$ and $H_{i-1} = \bar{H}_{i-1}$) the TA will distinguish the following cases;
   (a) If $H_{i-1} = \mathsf{H}_2(\sigma_{i-1}, H_{i-2}, ssid;\ a_{i-1}||b_{i-1})$, add N_i to M.
   (b) If $H_{i-1} = \mathsf{H}_2(\bar{\sigma}_{i-1}, H_{i-2}, ssid;\ a_{i-1}||b_{i-1})$, add N_{i-1} to M (note that in that case, the function NodeValidate$(\sigma_{i-1}, H_{i-1}, a_{i-1}, b_{i-1}, ssid)$ should return `HashError`).
   (c) Else add both N_i and N_{i-1} to M.
3. If NodeConsistency returns `ConsistentHashInconsistentSignature`, the TA will add N_{i-1} in M.
4. If NodeConsistency returns `InconsistentHashAndSignature`, the TA will add both N_i and N_{i-1} to the list M.

4.6.5 Discussion and high-level cost analysis

The novel, BLS-like order-preserving aggregate signature construction, introduced in this section, serves as an alternative to the PS_SAS scheme presented in Section 4.5.3, offering different trade-offs in terms of both performance and communication overhead (signature size). In particular, a preliminary cost analysis shows that the BLS-like scheme offers a constant-size aggregate signature at the cost of slower signing, whereas the PS_SAS scheme enables faster signing but produces a larger aggregate signature whose size grows linearly with the number of nodes. Although a detailed comparison between the two schemes will be provided in the next WP3 deliverables, along with their optimized implementation and extensive security analysis, in this subsection we present an initial, high-level assessment on their respective pros and cons. In addition, since the BLS-like scheme is based on pairings, we assume the same instantiation as in the PS_SAS scheme, namely using the `BLS12-381` elliptic curve. This implies that the same building blocks as those described in Subsection 4.5.4 can be considered for the implementation of this new BLS-like order-preserving aggregate signature scheme, namely the optimal ate pairing on `BLS12-381`, the SwiftEC hash-to-curve operation, as well as the scalar multiplication and subgroup membership tests in $\mathbb{G}_1$ and $\mathbb{G}_2$.

In our scenario, the signing operation constitutes the main performance-critical task, as it is executed by the vRouters. Therefore, we provide an initial, high-level assessment of its computational cost. Verification on the other hand, is considered less performance-critical, as the verifier (Orchestrator) is assumed to have sufficient computational resources. Recall from Subsection 4.6.2 that the generation of the signature σ_i is slightly different when $i = 0$, compared to the generic case $i > 0$. Recall also that the same is true for the PS_SAS scheme as well (see Subsection 4.5.3). Therefore, we treat the two cases separately and attempt a high-level comparison between the two schemes, considering only the elliptic curve operations which constitute the main bulk of the computational cost in the signature generation process in both schemes. We distinguish the following two cases:

Signature σ_0. The signature σ_0 is computed as:

$$\sigma_0 = \begin{pmatrix} \mathsf{H}_1(msg)^{a_0} & \mathsf{H}_1(msg)^{sk_0} \\ 1_{\mathbb{G}_1} & \mathsf{H}_1(msg)^{b_0} \end{pmatrix}$$

which requires 3 exponentiations and 1 execution of the function H_1. In the elliptic curve setting, this translates to 3 scalar multiplications in $\mathbb{G}_1$ and 1 execution of the hash-to-curve function (SwiftEC). In addition, assuming that the compressed representation of elliptic curve points is used, the signature σ_0 is 144 bytes long.

Signature σ_i, for $i > 0$. For each node N_i, with $i > 0$, the (partial) aggregate signature σ_i is defined as:

$$\sigma_i = \begin{pmatrix} t_1^{a_i} & t_2^{b_i} t_1^{sk_i} \\ 1_{\mathbb{G}_1} & t_3^{b_i} \end{pmatrix}, \quad \text{where} \quad t_1, t_2, t_3 \in \mathbb{G}_1 \quad \text{and} \quad a_i, b_i, sk_i \in \mathbb{Z}_q^*$$

The computation of σ_i requires 4 exponentiations and 1 multiplication, which translates to 4 scalar multiplications and 1 point addition in $\mathbb{G}_1$ in the elliptic curve setting. The size of σ_i is the same as σ_0, namely 144 byte long.

Table 4.6: Comparison of signature generation in PS_SAS and BLS-like schemes (core ECC operations).

Function	Scheme	scalar mult. in $\mathbb{G}_1$	scalar mult. in $\mathbb{G}_2$	point add. in $\mathbb{G}_1$	point add. in $\mathbb{G}_2$	hash to $\mathbb{G}_1$
Sign (1st node)	PS_SAS	2	-	1	-	2
	BLS-like	3	-	-	-	1
Sign (remaining nodes)	PS_SAS	2	-	1	-	-
	BLS-like	4	-	1	-	-

Table 4.6 summarizes the core elliptic curve operations required for both order-preserving aggregate signature solutions, considering the signature generation process for the first node in the path, and for the remaining nodes. For the first node, the two schemes are comparable: the BLS-like scheme requires 1 additional scalar multiplication in $\mathbb{G}_1$ compared to PS_SAS, whereas PS_SAS requires 1 additional hash-to-curve operation in $\mathbb{G}_1$ (SwiftEC). Therefore, both schemes are expected to have comparable performance for signature generation at the first node in the path. For the remaining nodes, i.e., the generic case, the BLS-like scheme requires twice as many scalar multiplications in $\mathbb{G}_1$ as PS_SAS, which is expected to have a noticeable impact on its performance, potentially making the BLS-like scheme approximately two times slower in signature generation. Despite this impact on performance, the BLS-like scheme is not prohibitively costly, and remains an efficient enough solution to be considered for integration in vRouters.

A major advantage of the BLS-like approach over the PS_SAS scheme is the size of the aggregate signature. The former offers constant-size signatures, independent of the number of nodes in the path; in particular, the final (aggregate) signature σ_{path} is 144 bytes long. On the other hand, as shown in Table 4.4, the aggregate signature in the PS_SAS scheme depends on the number of nodes in the path and has length $48(n + 2) + 8$ bytes (i.e., linear size in the number of nodes). Recall that the size of the aggregate signature is critical, since the vRouters are limited in bandwidth available to the control plane and thus, smaller signatures are preferable.

We emphasize that the comparison presented here offers a high-level view of the performance-critical operations and resource requirements (signature size) in the two schemes, providing a first hint on their computational cost. A detailed comparison, including performance benchmarks, will be presented in the next deliverables.

Chapter 5

CASTOR Finite State Machine-based attestation source

5.1 Overview

This section describes the Finite State Machine (FSM)-based approach for runtime behavioural attestation of selected target processes running on XRd routers. The objective is to model expected process behaviours and detect potential deviations at runtime to be leveraged by the Trust Assessment Framework (TAF) as a source of validation to calculate a trust score of the router itself. The decision to train and deploy FSMs as attestation sources comes from the need of having lightweight and explainable integrity verification models that would have the least computational impact on the router itself, considering both its constrained resources as well as their runtime operational demand. The FSM approach models a process as a sequence of observable states derived from runtime features, provided by the tracing layer in the form of system calls. Once the model has been trained and deployed, a live comparison between the learnt nominal model and the runtime collected traces can be performed, enabling the detection of runtime behavioural discrepancies which will be shared to the TAF for the final trust evaluation.

5.1.1 Scope of FSM and assumptions

The FSM-based attestation focuses on learning specific key target processes of relevance for the customer deployed on the XRd routers, for example, modification of routing tables, changing the status of specific interfaces, reconfiguration of the flex-algo behaviour. The goal is to identify key behavioural patterns from the sequences of traces that can represent the process under analysis and allow to capture relevant deviations that could lead to disruptions of the operational duties of the router, rather than capture the full semantics of the process under analysis. As briefly mentioned previously, the FSMs have been selected as the model of use due to their limited real-time evaluation impact, both from a time and computational perspective. As extensively presented in Section 3.7, the traces needed for the FSM training and runtime evaluation are collected and provided by the CASTOR tracing layer. The information collected by the tracing layer influence the training process and therefore a mutual decision about what should be traced is agreed. The decision on what should be traced is also influenced by the customer needs and their threat models, discussed in Section 2.

To be effective, the FSMs require: (i) sufficient and relevant traces capturing the full nominal behaviour of the process under analysis, (ii) the traces capture relevant and rich runtime information, (iii) minor execution timing variations can be abstracted during the training phase of the models, and (iv) a sufficient amount of traces and set of traces of different executions of the same process are provided.

5.1.2 FSMs integration in CASTOR

The FSM-based attestation source is integrated within CASTOR as a behavioural evidence component positioned between the runtime tracing layer and the Trust Assessment Framework (TAF). Its role is to consume trace-derived observations from the tracing layer, evaluate them against a learnt behavioural model, and provide attestation-relevant evidence to the TAF to support router trust assessment. The FSM component operates in two distinct phases: a training phase, in which a nominal behavioural model is constructed, and a runtime attestation phase, in which live observations are evaluated against this model. During the training phase, traces collected by the CASTOR tracing layer are processed offline to learn the expected behaviour of a target process or activity. This phase includes preprocessing of raw traces (e.g. timestamp alignment, process and thread filtering, and sequence extraction), followed by feature engineering to identify the most informative observable events. Based on this analysis, a reduced set of features is selected, which defines the FSM alphabet, i.e., the set of event symbols used by the FSM model. The traces are then transformed into sequences expressed in this FSM alphabet and used to construct and optimise the FSM, resulting in a compact representation of nominal behaviour. The overall training-phase workflow is illustrated in Figure 5.1.

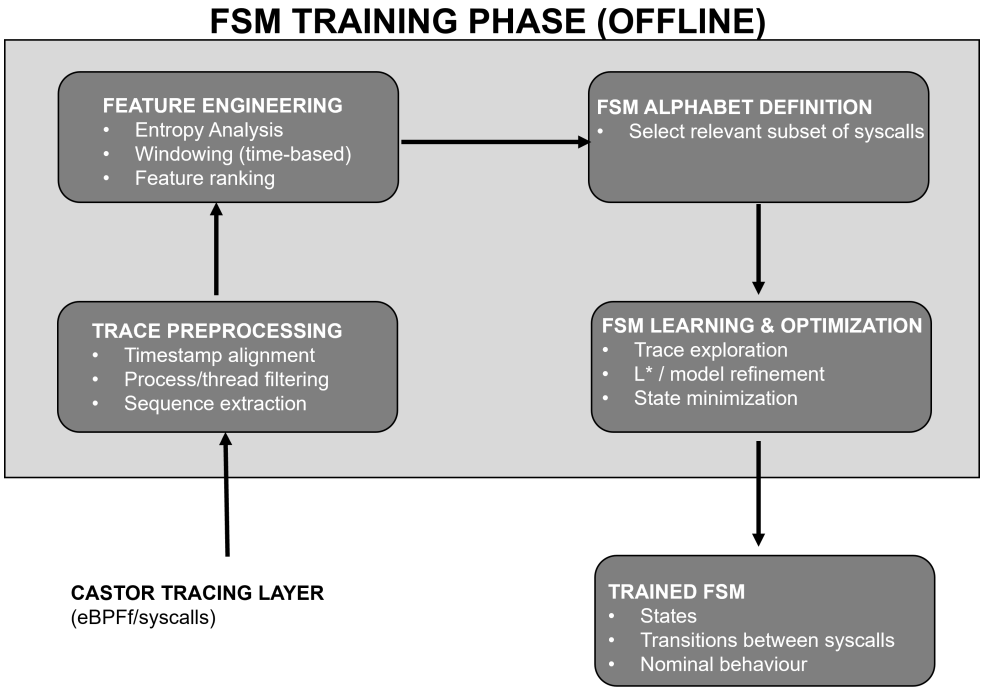

Figure 5.1: FSM training-phase pipeline, from CASTOR trace collection to nominal model generation

During the runtime attestation phase, the trained FSM model is used to evaluate live behaviour. The tracing layer provides a stream of runtime observations, which are pre-processed and mapped onto the FSM alphabet. This mapping step effectively projects the richer trace stream onto the subset of events selected during training, ensuring compatibility with the FSM model while limiting the amount of data required for evaluation. The resulting event sequence in the FSM alphabet is then processed by the FSM, which performs state transitions and checks conformance against the learnt model. Deviations from expected behaviour can be detected as invalid or unexpected transitions, and the point of deviation can be identified. The corresponding runtime workflow is illustrated in Figure 5.2.

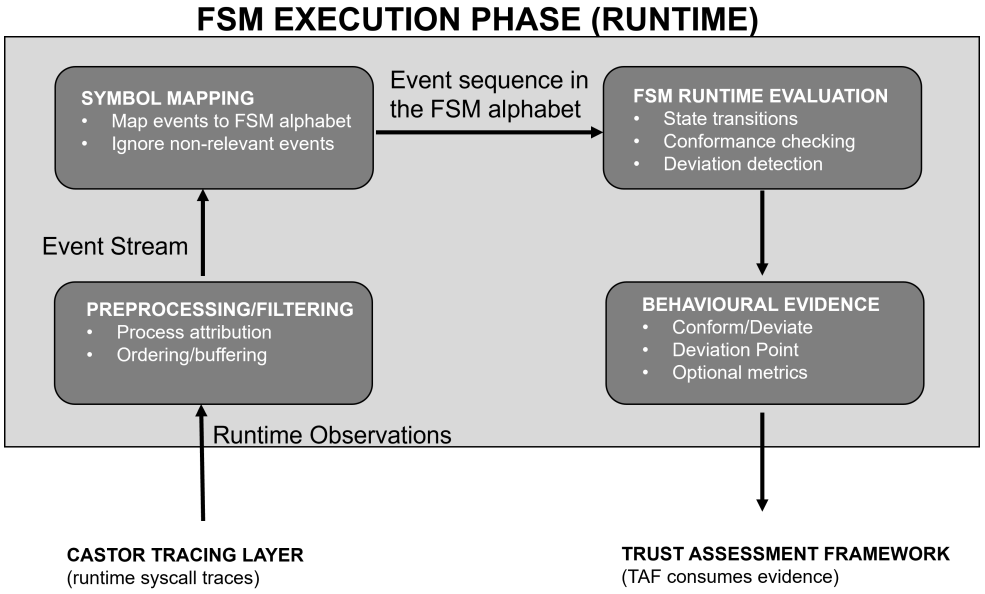

Figure 5.2: FSM runtime attestation pipeline, from live trace observations to behavioural evidence for TAF

The output of the runtime phase consists of behavioural evidence, such as conformity indicators or deviation signals, which is forwarded to the TAF. The separation between training and runtime phases implies different data requirements. The training phase benefits from richer and more diverse trace datasets in order to capture the variability of nominal behaviour and support effective feature selection. In contrast, the runtime phase is expected to operate on a reduced observation set defined by the FSM alphabet, enabling efficient evaluation with limited computational overhead. This integration relies on several assumptions: (i) the behaviour observed at runtime corresponds to the class of behaviour used during training; (ii) runtime observations can be reliably attributed to the target process or activity; (iii) the selected FSM alphabet remains representative of the relevant behavioural patterns; and (iv) significant changes in software or configuration that alter process behaviour are addressed through model update or retraining. Under these conditions, the FSM-based attestation source can provide meaningful behavioural evidence to support trust assessment in CASTOR.

5.2 FSMs working pipeline

This section describes the FSM training pipeline used to construct a nominal behavioural model from traces collected by the CASTOR tracing layer. In line with the integration described in Section 5.1.2, the training phase is performed offline and relies on richer trace datasets than those expected to be used during runtime attestation. Its purpose is to identify the most informative observable events, derive the FSM alphabet, and generate a compact model of nominal behaviour for the target process or activity. The pipeline begins with the ingestion and preprocessing of raw traces provided by the tracing layer, followed by feature engineering and FSM construction. The resulting model is then intended to be used during the runtime attestation phase, where live observations are mapped onto the FSM alphabet and evaluated for conformance against the learnt nominal model.

5.2.1 Feature engineering

After the initial parsing of the raw traces shared by the tracing layer, the first important step of the training process is feature engineering, during which the most informative observable events are identified and selected for subsequent model construction. In this context, the primary candidate features are the

system calls present in the trace dataset and, in this first iteration, these are represented by their names. The feature engineering process can in principle be extended in future iterations to include additional runtime information collected by the tracing layer. Feature selection is performed through an entropy-based analysis in which the candidate features are ranked according to their relevance in the observed execution traces. To improve robustness, the entropy calculation is performed on subsets of the original dataset rather than on the full trace set as a single block. In particular, the dataset is partitioned using time-based windows and, where useful, length-based windows. The dimensions of these subsets vary depending on the size and structure of the original dataset.

For each subset, an entropy value is calculated and used to rank the candidate features. The final set of selected features is derived from the most informative subsets and defines the observation space used for FSM construction, i.e., the FSM alphabet. While both time-based and length-based windowing can in principle contribute to this analysis, the traces analysed so far are relatively compact and therefore do not provide enough depth for length-based analysis to be consistently significant. For this reason, time-based windowing is currently considered the more meaningful basis for feature selection.

An important outcome of this phase is that it separates the richer set of candidate observations available during training from the reduced set of event symbols ultimately retained by the model. This transformation is illustrated in Figure 5.3, where raw trace data is progressively reduced through feature selection to define the FSM alphabet and the corresponding input representation used for model construction.

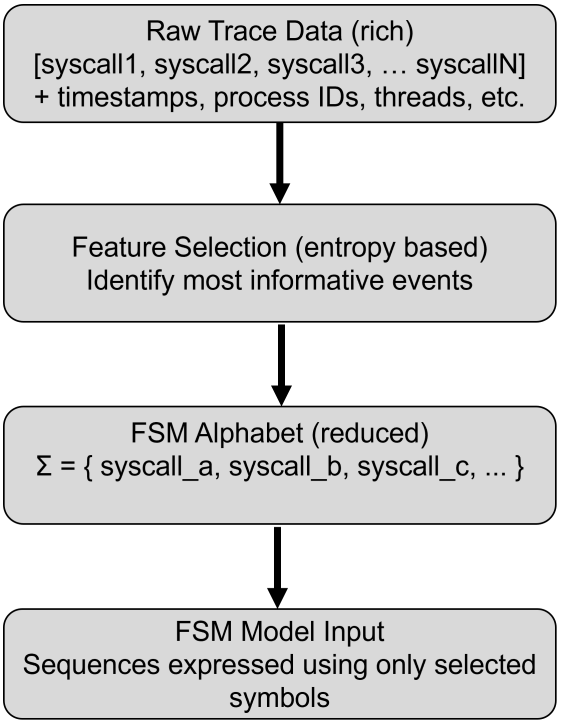

Figure 5.3: Transformation from raw trace data to FSM input through feature selection and FSM alphabet definition

This distinction is important for runtime attestation, where observations are expected to be mapped onto the FSM alphabet rather than evaluated in their full raw form. We seek to iteratively improve the model quality through feature engineering and FSM training to identify a minimal but expressive set of features that can later be used to reduce tracing and evaluation overhead at runtime.

5.2.2 Traces analysis and model progressive generation

Once the features have been identified for the training, the training process will begin by parsing the traces received by the tracing layer and restructuring them in appropriate data structures. The first step is to build the necessary information for the FSM, including the machine alphabet, representing the transactions between the internal states. Through the exploration of the traces, as suggested in [43], the FSM will be built and then optimised based on Angluin's L* algorithm [2] and as previously described in CASTOR deliverable 2.1 [22].

5.3 Current integration and results

This section presents the current status of the FSM integration and the preliminary results obtained using traces collected by the CASTOR tracing layer. At this stage, the objective is to validate the end-to-end FSM pipeline covering trace ingestion, preprocessing, feature engineering, and model learning using real trace data from an XRd router environment. The results should therefore be interpreted as a proof-of-feasibility of the approach rather than as a complete operational attestation of specific router functions.

5.3.1 Integration of current CASTOR eBPF traces

In this first iteration, a dataset of traces was collected by the CASTOR tracing layer while performing a network interface state change on the XRd router. The tracing layer captured runtime system calls across the entire router operating environment, resulting in a dataset composed of more than 50,000 system call events spanning over 300 processes.

Among these, a subset of 16 processes was identified as newly started after activation of the tracing layer. This identification was made possible through the snapshot capability provided by the tracing layer, which records the state of running processes at both the start and termination of tracing. By comparing these snapshots, it is possible to distinguish between pre-existing processes and those initiated during the observation window.

At the current stage of CASTOR integration, it is not yet possible to unambiguously attribute the observed interface state change to a specific process within the trace dataset. For this reason, the analysis presented in this section does not claim to model the exact control process responsible for the interface reconfiguration. Instead, one of the newly identified processes was selected as a representative example to validate the FSM pipeline.

The traces were parsed and pre-processed to enable FSM learning. This included:

- sorting events based on timestamps,
- aligning timestamps from different sources (kernel time vs Unix epoch),
- filtering events based on process and thread identifiers,
- reconstructing execution sequences for the selected process.
- The need for timestamp alignment arises from differences between runtime trace timestamps and snapshot timestamps. Ensuring a consistent temporal ordering is essential for constructing valid event sequences and for supporting future analyses where process activity spans both snapshot and runtime observations.

Overall, this integration demonstrates that CASTOR eBPF traces can be successfully ingested, normalised, and structured into a form suitable for FSM-based modelling.

5.3.2 Learnt models

The selected process generated more than 500 system call events, corresponding to 43 distinct system call types. Following the feature engineering phase, this set was reduced to 13 system calls identified as the most informative based on entropy analysis. After FSM training and optimisation, the effective feature set was further reduced to 6 system calls, indicating that several initially selected features were redundant in the context of the learnt model. The resulting FSM model, shown in Figure 5.4, represents a compact abstraction of the observed behavioural patterns of the selected process. This result demonstrates that the FSM pipeline is capable of extracting structured behavioural representations from CASTOR trace data.

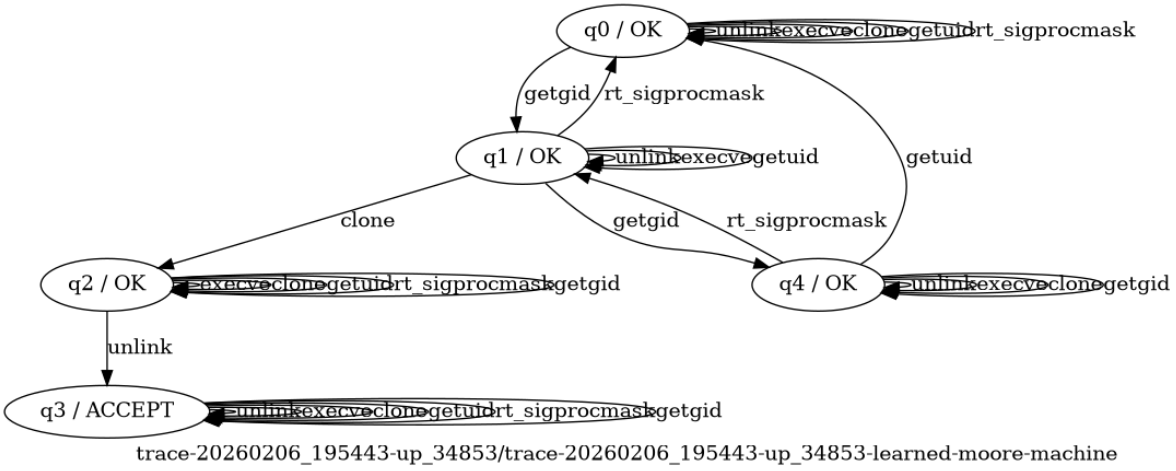

Figure 5.4: Learnt FSM model based on the top ranked system calls

A further observation is that the quality and interpretability of the learnt model depend strongly on the selected feature set. In some configurations, the optimisation process can lead to overly simplified models dominated by a single system call. An example of this behaviour is shown in Figure 5.5, where the *exit_group* system call effectively overshadows other transitions. While this produces a valid FSM, it provides limited insight into the broader behavioural structure of the process. This highlights the importance of careful feature selection in order to obtain meaningful attestation models.

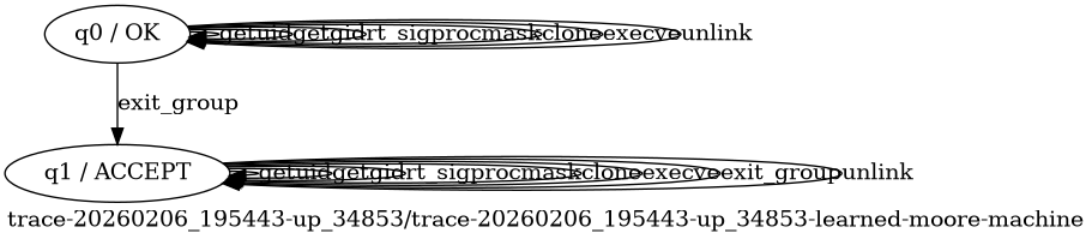

Figure 5.5: Overshadowed FSM model

To explore this further, additional models were derived from the same trace dataset using different subsets of features. These models, shown in Figure 5.6, capture different aspects of the process behaviour, such as communication patterns (e.g. socket-related activity) and memory usage. This demonstrates that the FSM approach is flexible and can be tailored to focus on specific behavioural dimensions depending on the requirements of the use case.

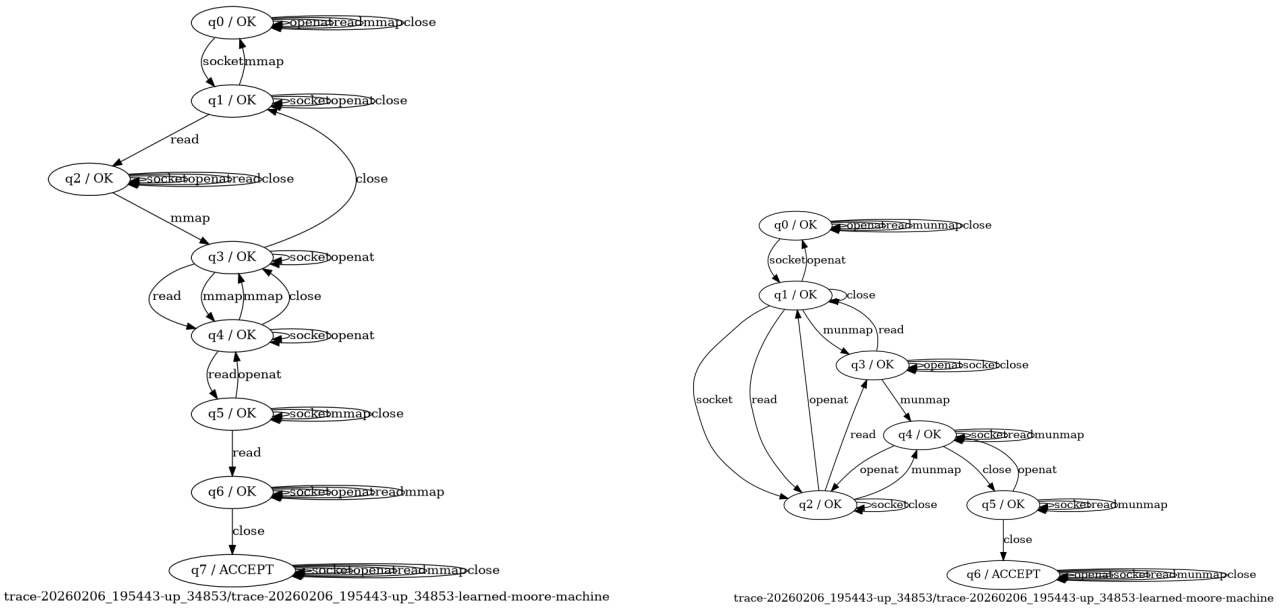

Figure 5.6: Learnt models based on socket communication and memory usage

Taken together, these results support the following conclusions:

- CASTOR eBPF trace data is suitable for FSM-based behavioural modelling after appropriate pre-processing.
- entropy-based feature selection enables reduction of the observation space while preserving relevant behavioural structure.
- the FSM training process can produce compact and interpretable models from real trace data.
- the choice of features has a significant impact on the usefulness of the resulting attestation model.

At this stage, the results should be interpreted as validation of the FSM training pipeline and its compatibility with CASTOR trace data. Further work is required to link the models to specific router control processes and to integrate the runtime attestation phase within the broader CASTOR framework.

5.4 Summary of FSM-based attestation approach

This chapter introduced a Finite State Machine (FSM)-based approach for behavioural attestation within the CASTOR framework. The FSM component is designed as a lightweight and interpretable source of behavioural evidence, leveraging runtime traces collected by the CASTOR tracing layer to assess whether observed process behaviour conforms to a learnt nominal model.

The approach is structured into two distinct phases. In the training phase, trace data is analysed offline to derive a compact FSM model, including the identification of a reduced observation space (FSM alphabet) through feature selection. In the runtime attestation phase, live observations are mapped onto this alphabet and evaluated against the model to detect deviations from expected behaviour. The resulting behavioural evidence is provided to the Trust Assessment Framework (TAF) as one input to overall trust evaluation.

Preliminary results demonstrate that CASTOR eBPF trace data can be successfully processed to produce meaningful FSM models, validating the feasibility of the approach. However, the current implementation remains at an early stage, with ongoing work required to improve process attribution, expand training datasets, and integrate runtime attestation within the broader CASTOR architecture.

5.5 Next steps and future work

The results presented in this chapter demonstrate the feasibility of constructing FSM-based behavioural models from CASTOR trace data and using them as a potential source of attestation evidence. The current implementation represents an initial stage of development, and several steps are required to progress towards a fully integrated and operational attestation capability within CASTOR.

- Integration of the runtime attestation phase. While the training pipeline has been validated offline, further work is required to deploy the trained FSM models in a runtime setting and connect them to the live trace stream produced by the CASTOR tracing layer. This includes defining the mechanisms for mapping runtime observations onto the FSM alphabet, and producing structured behavioural evidence for consumption by the TAF.
- Improving how the learnt models relate to the operational scenario being observed. In this work, traces were collected while bringing a router interface up and down, and the FSM analysis was performed on representative processes from that dataset. Future work should aim to better relate the learnt models to the behaviour associated with this type of operation, in order to improve the interpretability of the resulting attestation evidence.
- Expansion and diversification of training datasets. The robustness of the FSM models depends on the ability to capture the variability of nominal behaviour. This requires collecting traces across multiple executions, configurations, and operating conditions, in order to reduce the risk of false positives during runtime attestation.
- Optimisation of feature selection and tracing configuration. The current entropy-based feature engineering approach provides a solid foundation for identifying relevant system calls. Future work will build on this by using the results of feature selection to guide which system calls are collected by the tracing layer, enabling more efficient data collection while maintaining the ability to detect meaningful behavioural deviations.
- The FSM-based attestation source must be integrated with the broader CASTOR Trust Assessment Framework. This includes defining the format and semantics of the behavioural evidence produced by the FSM, and how it is combined with other trust signals to contribute to the overall trust score of network elements.

These steps will enable the transition from a proof-of-feasibility implementation to a practical and deployable behavioural attestation mechanism within CASTOR.

Chapter 6

Delegatable Order Revealing Encryption

In cross-domain environments, there is a need to exchange trust-related state information without revealing the exact trust level of individual routers or specific paths. Preserving this confidentiality is critical for maintaining the security of both individual routing elements and end-to-end paths. For instance, leakage of such information may lead to implementation disclosure attacks, particularly through profiling and fingerprinting techniques. Traditional approaches rely on blockchain-based mechanisms combined with a trust exposure layer that abstracts queried information (e.g., through filtered queries). However, such exposure layers introduce additional computational and communication overhead. CASTOR aims to address this challenge by investigating cryptographic constructions that enable efficient and privacy-preserving trust evaluation. In particular, CASTOR operates in multi-domain settings where routing elements should not disclose raw trust values or supporting evidence to external domains. At the same time, routing decisions require the ability to compare trust properties across domains—for example, to verify that all elements along a path satisfy a given trust threshold. This creates the need for mechanisms that support ordering-based comparisons of trust values without revealing their actual values, motivating the use of Order-Revealing Encryption (ORE).

As illustrated in Figure 6.1, Network Controller A has full visibility over its domain, including nodes C, D, E, F, and G, where F, G, E, and C form a routing path. Based on this view, Controller A determines the minimum trust level along the selected path. This value is then propagated to the border node C. Inter-domain communication between node C and node S in the neighboring domain is facilitated by the Border Gateway Protocol (BGP). In this setting, the actual value of the minimum trust level must remain confidential. To achieve this, the network controller encrypts the computed trust value before transmitting it to node C. Upon receiving this encrypted value from domain A and the encrypted minimum acceptable trust threshold from domain B, node S performs a comparison to determine whether the trust level of the incoming path satisfies the required threshold. If the condition holds, the route is accepted; otherwise, it is rejected. This scenario highlights two key cryptographic requirements: (1) the ability to compare trust values across domains, and (2) the preservation of trust value confidentiality during the comparison process.

What kind of cryptographic primitive can satisfy these above mentioned two requirements?

To address these requirements, Order-Revealing Encryption (ORE) can be employed, as it enables comparisons over encrypted values while preserving their confidentiality. However, standard ORE schemes are typically defined in a single-user setting. To support the multi-domain scenario considered in CASTOR, we adopt Delegatable Order-Revealing Encryption (DORE), which extends ORE to a multi-user setting. In the following, we review the related work, present the building blocks underlying DORE, provide an overview of the DORE scheme, and finally describe its concrete construction.

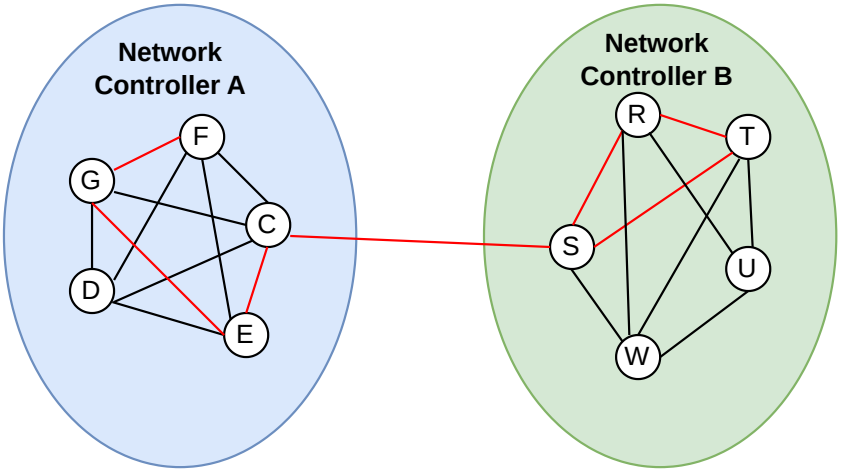

Figure 6.1: CASTOR scenario

6.1 Related work of order-revealing encryption

An Order-Preserving Encryption (OPE) scheme has the property that ciphertexts preserve the numerical order of the plaintexts. The notion of OPE was first introduced in 2005 by Agrawal *et al.* [1]. Boldyreva *et al.* [9] later defined the security model, and proposed an efficient OPE scheme using a tool called hyper geometric distribution. However, this scheme was shown to be insecure by [10], as it reveals plaintext information. Subsequently, Popa *et al.* [57] constructed the first OPE scheme with ideal security, i.e., indistinguishability against order-chosen plaintext attack (IND-OCPA), by leveraging a B-tree data structure. A fundamental limitation, however, is that efficient and non-interactive OPE schemes with strong security guarantees do not exist, which inherently restricts their applicability.

Order-Revealing Encryption (ORE) can be viewed as the generalization of OPE that addresses this limitation. In an ORE scheme, the order of plaintexts is determined by evaluating specific functions on the corresponding ciphertexts. The concept of ORE was first proposed by Boneh *et al.* [14], who also provided a concrete construction of ORE, achieving ideal security based on a powerful but relatively immature cryptographic primitive, namely multilinear maps. Due to the heavy computation burden of current multilinear maps, Chenette *et al.* [35] introduced an efficient ORE scheme at the expense of leaking the index of the first different digit bit with respect to the plaintexts. To better balance the efficiency and security, Lewi and Wu [47] proposed a generalization of the ORE scheme, which leaks the index of the first different block of the plaintexts. Using bilinear maps, Cash *et al.* [20] constructed a more secure ORE scheme, which only leaks the "equality pattern" of the first different digit bit. Furthermore, Cash *et al.* [21] proposed a new notion called parameter-hiding ORE and presented a construction based on asymmetric bilinear map. The parameter-hiding ORE provides a strong security in the case that the database entries are drawn identically and independently from a distribution of known shape, but for which the mean and variance are not known. However, all of the above ORE schemes are all based on single-user setting, which means different ciphertexts are encrypted using the same encryption key, which limits the application of ORE schemes in multi-user environments. In 2019, Zhao *et al.* [48] extended the single-user setting to multi-user setting by using PKI-based approach. In their scheme, except ciphertexts, any two target entities A and B, A should generate a token, which is a commitment of A's secret key sk_A and B's public key pk_B. For the entity B, the token generation is the same as that for A. A tester can get the order by comparing ciphertexts and tokens. However, Park *et al.* [53] showed that the tokens in [48] can be forgeable, leading to security vulnerabilities. Also, they addressed this security issue by incorporating BLS/Schnorr signatures to generate tokens. This improved scheme is suitable for CASTOR, as it is supports the multi-user setting while providing stronger security guarantees.

6.2 Preliminaries & building blocks

As mentioned in the previous section 6.1, we will use the BLS/Schnorr signature schemes to generate tokens. In this section, we introduce the Schnorr signature scheme, which serves as a core building block in the DORE scheme. Bilinear pairings and BLS signatures are already defined in section 4.5.2

6.2.1 Schnorr signature

We denote the Schnorr signature as Sch $= (\mathsf{Sch.Setup}, \mathsf{Sch.Keygen}, \mathsf{Sch.Sign}, \mathsf{Sch.Verify})$

- $\mathsf{Sch.Setup}(1^\lambda) \rightarrow \mathsf{pp}$: This algorithm takes the security parameter λ as input. Chooses a secure hash function $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{Z}_p$. Then the public parameter is $\mathsf{pp} = (p, g, \mathbb{G}, \mathsf{H})$.
- $\mathsf{Sch.Keygen}(\mathsf{pp}) \rightarrow (pk, sk)$: This algorithm takes the public parameter pp as input. Chooses a a random value as secret key sk, Then the public key is $pk = g_2^{sk}$.
- $\mathsf{Sch.Sign}(sk, m) \rightarrow \sigma$: This algorithm takes the secret key and a message as inputs. Computes the signature as follows:
  - ✓ Choose a random value r and computes $R = g^r$.
  - ✓ Computes the challenge $c = \mathsf{H}(R, pk, m)$.
  - ✓ Computes $s = r - c \cdot sk$.

  Then the signature is $\sigma = (c, s)$.
- $\mathsf{Sch.Verify}(pk, m, \sigma) \rightarrow 0/1$: This algorithm takes the public key, the message and signature as inputs. Computes $R' = g^s pk^c$ and checks whether the equation $c \stackrel{?}{=} \mathsf{H}(R', pk, m)$ is satisfied or not. If yes, it accepts the signature, otherwise the signature is rejected.

6.3 Overview of the scheme

The notation used throughout the scheme is summarized in Table 6.1.

Entities. In a DORE scheme, there are two types of entities, i.e., multiple encryptors and a tester.

- **Encryptor**. An encryptor computes a ciphertext of the trust level and a token to be used for comparison. It can be instantiated by an orchestrator.
- **Tester**. A tester can compare two cipherxts to get the order of associated plaintexts. It can be any entity except encryptors.

Conceptual Protocol Overview. As stated earlier, multiple encryptors can exist and a single tester. Here, we assume two encryptors, A and B, and a tester as an example, in order to simplify the description of the DORE scheme. As shown in Figure 6.2, Encryptors A and B need to encrypt their plaintexts, to obtain the corresponding ciphertexts, ct_A and ct_B. Then they both send their ciphertexts to the third party, i.e., the tester. Finally, the tester can get the order of their plaintexts by comparing the two ciphertexts ct_A and ct_B. In CASTOR, the encryptor A is instantiated by the network controller A, the encryptor B is instantiated by the network controller B. The plaintext for A is the lowest trust level in domain A, the plaintext for B is the minimum trust level in domain B that is acceptable. Further, the communication

Symbol	Description
λ	Security parameter
n	The length of a message
p	A prime order
$\mathbb{G}_1$	A cyclic group of order p over an elliptic curve
$\mathbb{G}_2$	A cyclic group of order p over an elliptic curve
$g_1,\ g_2$	Generators of $\mathbb{G}_1$ and $\mathbb{G}_2$
$\mathbb{G}_T$	A multiplicative cyclic group of order p
e	Type III pairing defined by $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$
H	A hash function defined as $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{G}_1$
H_1	A hash function defined as $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_p$
$\leftarrow_{\$}$	Indicates a uniform random sampling
pp	Public parameters
pk	Public key for the encryptor
vk	Verification key for the encryptor
sk	Secret key for the encryptor
m	A message to be encrypted
ct	ciphertext associated with the message m
$\mathsf{tok}_{u_{pk} \rightarrow v_{sk}}$	A token generated by the encryptor v's secret key on u's public key

Table 6.1: Notation Summary for DORE

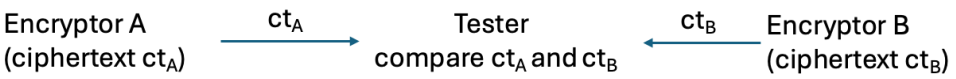

Figure 6.2: Conceptual overview of a DORE scheme

between A and B is realized by BGP nodes C and S. Then we let S act as the tester. In the following, we will describe details of the scheme and how it is applied to CASTOR scenario.

Syntax. A DORE scheme consists of two main building blocks: a signature scheme denoted by $\mathsf{Sig} = (\mathsf{Sig.Setup}, \mathsf{Sig.KeyGen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$, and a Delegatable Equality-Revealing Encoding (DERE) scheme $\mathsf{DERE} = (\mathsf{DERE.Setup}, \mathsf{DERE.KeyGen}, \mathsf{DERE.Enc}, \mathsf{DERE.Token}, \mathsf{DERE.Test}, \mathsf{DERE.Verify})$. We note that DERE.KeyGen and DERE.Enc are run by a single entity; therefore, we do not denote their inputs with u or v. In contrast, DERE.Token, DERE.Test and DERE.Verify require inputs from different entities, so we denote their inputs with u or v. This convention is also used in the DORE scheme description. In the following, we first describe Sig, followed by DERE and finally DORE:

- $\mathsf{Sig.Setup}(\lambda) \rightarrow \mathsf{pp}_{sig}$: This algorithm takes the security parameter λ as input and returns the public parameter pp_{sig}.
- $\mathsf{Sig.KeyGen}(pp_{sig}) \rightarrow (vk_{sig}, sk_{sig})$: It takes a public parameter as input and outputs a pair of keys consisting of a signing key sk_{sig} and a verification key vk_{sig}.
- $\mathsf{Sig.Sign}(pp_{sig}, sk_{sig}, m) \rightarrow \sigma$: It takes public parameter pp_{sig}, signing key sk_{sig} and message as input, and outputs the signature σ.
- $\mathsf{Sig.Verify}(pp_{sig}, vk_{sig}, m, \sigma) \rightarrow 0/1$: It takes public parameter pp_{sig}, the verifying key vk_{sig}, message m, and signature σ as input. If the signature is valid, it returns 1; otherwise, it returns 0.

In DERE, a signature scheme is used to ensure the unforgeability of tokens.

- $\mathsf{DERE.Setup}(1^\lambda) \rightarrow \mathsf{pp}$: This algorithm takes the security parameter λ as input and outputs the public parameter pp by running the algorithm Sig.Setup.
- $\mathsf{DERE.KeyGen}(\mathsf{pp}) \rightarrow (\mathsf{pk}, \mathsf{vk}, \mathsf{sk})$: This algorithm is run by an encryptor by taking pp as input and outputting $(\mathsf{pk}, \mathsf{vk}, \mathsf{sk})$ by running the algorithm Sig.KeyGen, where pk is the public key, vk is the verification key and sk is the secret key.
- $\mathsf{DERE.Enc}(\mathsf{pp}, m, \mathsf{sk}) \rightarrow \mathsf{ct}$: This encryption algorithm is run by the an encryptor by taking $(\mathsf{pp}, m, \mathsf{sk})$ as inputs and outputting the ciphertext ct.
- $\mathsf{DERE.Token}(\mathsf{pp}, \mathsf{pk}_u, \mathsf{sk}_v) \rightarrow \mathsf{tok}_{(u \rightarrow v)}$: This algorithm is run by an encryptor v, who uses his/her sk_v to sign the other entity u's public key pk_u using Sig.Sign to generate the token.
- $\mathsf{DERE.Test}(\mathsf{pp}, \mathsf{ct}_u, \mathsf{ct}_v, \mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}, \mathsf{tok}_{(v \rightarrow u)}) \rightarrow 0/1$: This algorithm is run by the tester, who takes $(\mathsf{pp}, \mathsf{ct}_u, \mathsf{ct}_v, \mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}, \mathsf{tok}_{(v_{pk} \rightarrow u_{sk})})$ as inputs and outputs the comparison result. "1" means two plaintexts associated with the ciphertexts are equal, otherwise, "0" means different.
- $\mathsf{DERE.Verify}(\mathsf{pp}, \mathsf{vk}_v, \mathsf{vk}_u, \mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}, \mathsf{tok}_{(v_{pk} \rightarrow u_{sk})} \rightarrow 0/1$: This algorithm is run by the tester, who takes $(\mathsf{pp}, \mathsf{vk}_v, \mathsf{vk}_u, \mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}, \mathsf{tok}_{(v_{pk} \rightarrow u_{sk})})$ as inputs to run Sig.Verify and outputs "1" for accepting the tokens, otherwise "0" for rejecting the tokens.

Based on the DERE scheme, a DORE scheme is denoted as $\mathsf{DORE} = (\mathsf{DORE.Setup}, \mathsf{DORE.KeyGen}, \mathsf{DORE.Token}, \mathsf{DORE.Verify}, \mathsf{DORE.Enc}, \mathsf{DORE.Test})$. All algorithms are described as follows:

- $\mathsf{DORE.Setup}(1^\lambda) \rightarrow \mathsf{pp}$: With the inputs, run the $\mathsf{DERE.Setup}$ algorithm and output pp.
- $\mathsf{DORE.KeyGen}(\mathsf{pp}) \rightarrow (\mathsf{pk}, \mathsf{vk}, \mathsf{sk})$: With the inputs, run $\mathsf{DERE.KeyGen}(\mathsf{pp})$ algorithm and output $(\mathsf{pk}, \mathsf{vk}, \mathsf{sk})$.

- DORE.Token$(\mathsf{pp}, \mathsf{pk}_u, \mathsf{sk}_v) \rightarrow \mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}$: With the inputs, run DERE.Token algorithm and output $\mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}$.
- DORE.Verify$(\mathsf{pp}, \mathsf{vk}_u, \mathsf{vk}_v, \mathsf{tok}_{(v_{pk} \rightarrow u_{sk})}, \mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}) \rightarrow 0/1$: With the inputs, run DERE.Verify algorithm and output decision bit $0/1$.
- DORE.Enc$(\mathsf{pp}, m, \mathsf{sk}) \rightarrow \mathsf{ct}$: It takes a message $m \in \{0,1\}^*$ and sk as input and outputs the corresponding ciphertext by running the algorithm DERE.Enc.
- DORE.Test$(\mathsf{pp}, \mathsf{ct}_u, \mathsf{ct}_v, \mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}, \mathsf{tok}_{(v_{pk} \rightarrow u_{sk})}) \rightarrow \mathsf{res}$: It takes a pair of ciphertexts $\mathsf{ct}_u, \mathsf{ct}_v$ and tokens $\mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}, \mathsf{tok}_{(v_{pk} \rightarrow u_{sk})}$ as input and outputs the result.

Security model. As described in D3.1 [24], the security of a DORE scheme is captured by three properties: *correctness*, *data privacy* and *token unforgeability*.

Correctness. The correctness of a DORE scheme ensures that the test algorithm accurately gets the order of two plaintexts by comparing two corresponding ciphertexts provided by two mutually authenticated users. Let (m_u, m_v) be a pair of messages, $(\mathsf{pk}_u, \mathsf{vk}_u, \mathsf{sk}_u)$ and $(\mathsf{pk}_v, \mathsf{vk}_v, \mathsf{sk}_v)$ be a pair of keys generated by the DORE.KeyGen algorithm, and ct_u, ct_v be a ciphertext of m_u, m_v with key $\mathsf{sk}_u, \mathsf{sk}_v$ respectively. Then we say a DORE scheme is correct if for any pair of messages m_u, m_v and keys $(\mathsf{pk}_u, \mathsf{vk}_u, \mathsf{sk}_u)$ and $(\mathsf{pk}_v, \mathsf{vk}_v, \mathsf{sk}_v)$, the following holds:

- DORE.Verify$(\mathsf{pp}, \mathsf{vk}_u, \mathsf{vk}_v, \mathsf{tok}_{(v \rightarrow u)}, \mathsf{tok}_{(u \rightarrow v)}) = 1$
- DORE.Test$(\mathsf{pp}, \mathsf{ct}_u, \mathsf{ct}_v, \mathsf{tok}_{(v_{pk} \rightarrow u_{sk})}, \mathsf{tok}_{(u_{pk} \rightarrow v_{sk})}) = \mathsf{res}$
  - ✓ If $m_u > m_v$, then $\mathsf{res} = 1$
  - ✓ If $m_u < m_v$, then $\mathsf{res} = -1$
  - ✓ Otherwise, $\mathsf{res} = 0$

Data Privacy. The data privacy of a DORE scheme ensures that the ciphertexts ct generated by the DORE.Enc algorithm do not leak information except the order. Further, DORE provides data privacy if the DORE.Enc algorithm satisfies indistinguishabity under an ordered chosen plaintext attack (IND-OCPA). The IND-OCPA security definition is parameterized by a leakage function $\mathcal{L}$, which exactly specifies what is leaked by an ORE scheme.

Definition 3. (**Security of DORE with Leakage**). Fix a security parameter $\lambda \in \mathbb{N}$, let $\mathcal{A} = (\mathcal{A}_1, ..., \mathcal{A}_q)$ be an adversary for some $q \in \mathbb{N}$. Let $\mathcal{S} = (\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_q)$ be a simulator, and let $\mathcal{L}(\cdot)$ be a leakage function. Two experiments $\mathrm{REAL}_{\mathcal{A}}(\lambda)$ and $\mathrm{SIM}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$ associated with the real world and simulated world, respectively, are defined as follows:

$\mathrm{REAL}_{\mathcal{A}}(\lambda)$:

1. $\mathsf{pp} \leftarrow$ DORE.Setup(1^λ)
2. $(\mathsf{pk}, \mathsf{vk}, \mathsf{sk}) \leftarrow$ DORE.KeyGen $\left(1^\lambda\right)$
3. $(m_1, \mathrm{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1\left(1^\lambda\right)$
4. $\mathsf{ct}_1 \leftarrow$ DORE.Enc $(\mathsf{pp}, m_1, \mathsf{sk}_1)$
5. for $2 \leq i \leq q$:

(a) $(m_i, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i(\mathsf{st}_{\mathcal{A}}, c_1, \ldots, c_{i-1})$

(b) $\mathsf{ct}_i \leftarrow \mathsf{DORE.Encrypt}(\mathsf{pp}, m_i, \mathsf{sk}_i)$

6. output $(\mathsf{ct}_1, \ldots, \mathsf{ct}_q)$ and $\mathsf{st}_{\mathcal{A}}$

$\mathrm{SIM}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$:

1. $\mathrm{st}_{\mathcal{S}} \leftarrow \mathcal{S}_0\left(1^{\lambda}\right)$
2. $(m_1, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1\left(1^{\lambda}\right)$
3. $(\mathsf{ct}_1, \mathrm{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_1\left(\mathrm{st}_{\mathcal{S}}, \mathcal{L}\left(m_1\right)\right)$
4. for $2 \leq i \leq q$:

(a) $(m_i, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i(\mathsf{st}_{\mathcal{A}}, \mathsf{ct}_1, \ldots, \mathsf{ct}_{i-1})$

(b) $(\mathsf{ct}_i, \mathrm{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_i\left(\mathrm{st}_{\mathcal{S}}, \mathcal{L}\left(m_1, \ldots, m_i\right)\right)$

5. output $(\mathsf{ct}_1, \ldots, \mathsf{ct}_q)$ and $\mathsf{st}_{\mathcal{A}}$

We say that Π is a secure DORE scheme with leakage function $\mathcal{L}(\cdot)$ if for all polynomial-size adversaries $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_q)$ where $q = \mathrm{poly}(\lambda)$, there exists a polynomial-size simulator $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ such that the outputs of the two distributions $\mathrm{REAL}_{\mathcal{A}}(\lambda)$ and $\mathrm{SIM}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$ are computationally indistinguishable.

Definition 4. (**IND-OCPA Security**). Let $\mathcal{L}$ be the following leakage function:

$$\mathcal{L}\left(m_1, \ldots, m_t\right) = \left\{\mathbf{1}\left(m_i < m_j\right) : 1 \leq i < j \leq t\right\}.$$

If an ORE scheme is secure with leakage $\mathcal{L}$, then it is IND-OCPA secure.

Token unforgeability. Token unforgeability means that an adversary $\mathcal{A}$ without access to a valid secret identity key isk and signing key sk, cannot generate a token that passes DORE.Verify algorithm. The definition of token unforgeability is shown in Figure 6.3. The oracles $\mathsf{DORE.KeyGen}(\mathsf{pp})$, $\mathsf{DORE.Token}(pk, \cdot)$ and $\mathsf{Corr}(\mathsf{pk})$ involved in this game are defined as follows:

- $\mathsf{DORE.KeyGen}(\mathsf{pp}) \rightarrow ((\mathsf{pk}, \mathsf{sk}), (\mathsf{isk}, \mathsf{ipk}))$: Given the system public parameter pp, when the adversary $\mathcal{A}$ makes queries about a user's key, this oracle returns the secret key pairs $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{DORE.Token}(\mathsf{pk}_u, \mathsf{pk}_v, \cdot) \rightarrow (\mathsf{tok}_{u_{pk} \rightarrow v_{sk}}, \mathsf{tok}_{v_{pk} \rightarrow u_{sk}})$: Given two public keys $\mathsf{pk}_u, \mathsf{pk}_v$, when the adversary makes queries about two user's authorization tokes, this oracle returns two associated authorization tokens $(\mathsf{tok}_{u_{pk} \rightarrow v_{sk}}, \mathsf{tok}_{v_{pk} \rightarrow u_{sk}})$.
- **Corr**$(\mathsf{pk}) \rightarrow \mathsf{sk}$: When the adversary makes queries using a user's public key pk, this oracle returns the corresponding secret key sk.

We denote the advantage of $\mathcal{A}$ as $\mathsf{adv}^{Unforge}_{\textbf{Token},\mathcal{A}}(\lambda)$, which is defined by the probability that $\mathcal{A}$ wins the token unforgeability game.

Definition 5. (**Token Unforgeability**) A DORE scheme is said to provide token unforgeabiliry if for any polynomial time adversary $\mathcal{A}$, the following equation holds:

$$\mathsf{adv}^{Unforge}_{\textbf{Token},\mathcal{A}}(\lambda) = \Pr\left[\mathrm{Exp}^{\mathsf{Unforge}}_{\textbf{Token},\mathcal{A}}(\lambda) = 1\right] \leq negl(\lambda)$$

Experiment $\mathrm{Exp}^{Unforge}_{\mathsf{token},\mathcal{A}}(\lambda)$

- $\mathsf{pp} \leftarrow \mathsf{DORE.Setup}(1^\lambda)$
- $(\mathsf{tok}^*_{u_{pk}\to v_{sk}}, \mathsf{tok}^*_{v_{pk}\to u_{sk}}) \leftarrow A^{\mathsf{DORE.KeyGen(pp)},\mathsf{DORE.TGen}(\mathsf{pk}_u,\mathsf{pk}_v,\cdot),\mathsf{Corr(pk)}}(\mathsf{pk}_u, \mathsf{pk}_v)$.
- Return 1 iff $\mathsf{DORE.Verify}(\mathsf{pk}^*_u, \mathsf{pk}^*_v, \mathsf{tok}^*_{u_{pk}\to v_{sk}}, \mathsf{tok}^*_{v_{pk}\to u_{sk}}) = 1 \wedge (\mathsf{tok}^*_{u_{pk}\to v_{sk}}, \mathsf{tok}^*_{v_{pk}\to u_{sk}})$ and $(\mathsf{pk}^*_u, \mathsf{pk}^*_v)$ have not been queried before.

Figure 6.3: Token unforgeability game.

6.4 The concrete DORE scheme

In this section, we following the example mentioned before. The scheme can also be applied to multiple domains and network controllers. In this concrete scheme, we first introduce the DERE scheme followed by the DORE scheme.

- $\mathsf{DERE.Setup}(1^\lambda) \to \mathsf{pp}$: This algorithm takes the security parameter λ as input and outputs the public parameter $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, g_1, g_2, \mathbb{G}_T, e, \mathsf{H}, \mathsf{H}_1)$, additionally, sets $\mathsf{pp}_{sig} = (p, \mathbb{G}_1, g_1, \mathsf{H}_1)$, where $(p, \mathbb{G}_1, \mathbb{G}_2, g_1, g_2, \mathbb{G}_T, e)$ are parameters for type III pairings. H and H_1 are two hash functions defined as: H: $\{0,1\}^* \to \mathbb{G}_1$, H_1:$\{0,1\}^* \to \mathbb{Z}_p$.
- $\mathsf{DERE.KeyGen(pp)} \to (\mathsf{pk}, \mathsf{vk}, \mathsf{sk})$: This algorithm is run by an encryptor.
  - ✓ $\mathsf{DERE.KeyGen(pp)} \to (\mathsf{pk}_A, \mathsf{vk}_A, \mathsf{sk}_A)$: if this algorithm is run by the network controller A, randomly chooses $a, x \leftarrow_\$ \mathbb{Z}_p$. Then the secret and public keys are $\mathsf{sk}_A = (a, x)$, $\mathsf{pk}_A = g_2^a$, $\mathsf{vk}_A = g_1^x$. Moreover, the signature signing key is $\mathsf{sk}_{sig,A} = x$, verification key is $\mathsf{vk}_{sig,A} = \mathsf{vk}_A$.
  - ✓ $\mathsf{DERE.KeyGen(pp)} \to (\mathsf{pk}_B, \mathsf{vk}_B, \mathsf{sk}_B)$: if this algorithm is run by the network controller B, randomly chooses $b, y \leftarrow_\$ \mathbb{Z}_p$. Then the secret and public keys are $\mathsf{sk}_B = (b, y)$, $\mathsf{pk}_B = g_2^b$, $\mathsf{vk}_B = g_1^y$. Moreover, the signature signing key is $\mathsf{sk}_{sig,B} = y$, verification key is $\mathsf{vk}_{sig,B} = \mathsf{vk}_B$.
- $\mathsf{DERE.Enc}(\mathsf{pp}, m, \mathsf{sk}) \to \mathsf{ct}$: This encryption algorithm is run by the an encryptor.
  - ✓ $\mathsf{DERE.Enc}(\mathsf{pp}, m_A, \mathsf{sk}_A) \to \mathsf{ct}_A$: if the encryptor is the network controller A, the message m_A is the lowest trust level of the routing path in the domain. Then randomly picks $r_A \leftarrow_\$ \mathbb{Z}_p$ and computes c^0_A and c^1_A as below:
    $$c^0_A = (g_1^{r_A x}\mathsf{H}(m_A))^a,\ c^1 = g_1^{r_A}$$
    Finally, it outputs the ciphertext $\mathsf{ct}_A = (c^0_A, c^1_A)$.
  - ✓ $\mathsf{DERE.Enc}(\mathsf{pp}, m_B, \mathsf{sk}_B) \to \mathsf{ct}_B$: if the encryptor is the network controller B, note that the message m_B is the minimum trust level. Then randomly picks $r_B \leftarrow_\$ \mathbb{Z}_p$ and computes c^0_B and c^1_B as below:
    $$c^0_B = (g_1^{r_B y}\mathsf{H}(m_B))^b,\ c^1 = g_1^{r_B}$$
    Finally, it outputs the ciphertext $\mathsf{ct}_B = (c^0_B, c^1_B)$.
- $\mathsf{DERE.Token}(\mathsf{pp}, \mathsf{pk}_u, \mathsf{sk}_v) \to \mathsf{tok}_{u_{pk}\to v_{sk}}$: This algorithm is run by an encryptor, who uses his/her sk_v to sign the other entity u's public key pk_u to generate the token.
  - ✓ $\mathsf{DERE.Token}(\mathsf{pp}, \mathsf{pk}_B, \mathsf{sk}_A) \to \mathsf{tok}_{B_{pk}\to A_{sk}}$: Concretely, mapping to CASTOR scenario, if this algorithm is run by the network controller A, then the following steps are executed. First, it computes the token $\mathsf{tok}_{B_{pk}\to A_{sk}} = (t^0_{B_{pk}\to A_{sk}}, t^1_{B_{pk}\to A_{sk}}, \sigma_A)$ as follows:
    $$t^0_{B_{pk}\to A_{sk}} = \mathsf{pk}_B,\ t^1_{B_{pk}\to A_{sk}} = \mathsf{pk}_B^{ax},\ \sigma_A = \mathsf{Sig.Sign}(\mathsf{pp}_{sig}, \mathsf{sk}_{sig,A}, (t^0_{B_{pk}\to A_{sk}}, t^1_{B_{pk}\to A_{sk}}))$$

sig can be instantiated by a BLS signature [15] or Schnorr signature [59]. Here we take the Schnorr signature as an example. Then, the signature σ_A is computed as follows:

* Chooses randomly $r_x \leftarrow_{\$} \mathbb{Z}_p$ and computes $C_{vk,A} = g_1^{r_x}$.
* Computes a challenge $c_A = \mathsf{H}_1(C_{vk,A}, \mathsf{vk}_{sig,A}, t^0_{B\to A}, t^1_{B\to A})$
* $s_A = r_x - c_A \cdot x$

Then the signature $\sigma_A = (c_A, s_A)$.

✓ DERE.Token$(\mathsf{pp}, \mathsf{pk}_A, \mathsf{sk}_B) \to \mathsf{tok}_{A_{pk}\to B_{sk}}$: If this algorithm is run by the network controller B the following are executed. First, it computes the token $\mathsf{tok}_{A_{pk}\to B_{sk}} = (t^0_{A_{pk}\to B_{sk}}, t^1_{A_{pk}\to B_{sk}}, \sigma_B)$ as follows:

$$t^0_{A_{pk}\to B_{sk}} = \mathsf{pk}_A,\ t^1_{A_{pk}\to B_{sk}} = \mathsf{pk}_A^{by},\ \sigma_B = \mathsf{Sig.Sign}(\mathsf{pp}_{sig}, \mathsf{sk}_{sig,B}, (t^0_{A_{pk}\to B_{sk}}, t^1_{A_{pk}\to B_{sk}}))$$

The Schnorr signature σ_A is computed as follows:

* Chooses randomly $r_y \leftarrow_{\$} \mathbb{Z}_p$ and computes $C_{vk,B} = g_1^{r_y}$.
* Computes a challenge $c_A = \mathsf{H}_1(C_{vk,B}, \mathsf{vk}_{sig,B}, t^0_{A_{pk}\to B_{sk}}, t^1_{A_{pk}\to B_{sk}})$
* $s_B = r_y - c_B \cdot y$

Then the signature $\sigma_B = (c_B, s_B)$.

- DERE.Test$(\mathsf{pp}, \mathsf{ct}_A, \mathsf{ct}_B, \mathsf{tok}_{B_{pk}\to A_{sk}}, \mathsf{tok}_{A_{pk}\to B_{sk}}) \to 0/1$: This algorithm is run by the tester, which computes the following values:

$$d_0 = \frac{e\left(c^0_A, t^0_{(B_{pk}\to A_{sk})}\right)}{e\left(c^1_A, t^1_{(B_{pk}\to A_{sk})}\right)} = e(\mathsf{H}(m_A), g_2^{ab}),\ d_1 = \frac{e\left(c^0_B, t^0_{(A_{pk}\to B_{sk})}\right)}{e\left(c^1_B, t^1_{(A_{pk}\to B_{sk})}\right)} = e(\mathsf{H}(m_B), g_2^{ab}).$$

Finally, it compares d_0 and d_1. If $d_0 = d_1$, returns 1, otherwise 0.

- DERE.Verify$(\mathsf{pp}, \mathsf{vk}_A, \mathsf{vk}_B, \mathsf{tok}_{(B_{pk}\to A_{sk})}, \mathsf{tok}_{(A_{pk}\to B_{sk})}) \to 0/1$: This algorithm is run by the tester. Then the tester runs verification algorithms as follows:

✓ Sig.Verify$(\mathsf{pp}_{sig}, \mathsf{vk}_A, (t^0_{B_{pk}\to A_{sk}}, t^1_{B_{pk}\to A_{sk}}), \sigma_A) \to res_A$: Computes $C'_{vk,A} = g_1^{s_A} \cdot (vk_{sig,A})^{C_A}$ and checks $c'_A \stackrel{?}{=} \mathsf{H}_1(C'_{vk,A}, VK_{sig,A}, t^0_{B_{pk}\to A_{sk}}, t^1_{B_{pk}\to A_{sk}})$, if yes, $res_A = 1$, otherwise $res_A = 0$.

✓ Sig.Verify$(\mathsf{pp}_{sig}, \mathsf{vk}_B, (t^0_{A_{pk}\to B_{sk}}, t^1_{A_{pk}\to B_{sk}}), \sigma_B) \to res_B$: Computes $C'_{vk,B} = g_1^{s_B} \cdot (vk_{sig,B})^{C_B}$ and checks $c'_B \stackrel{?}{=} \mathsf{H}_1(C'_{vk,B}, VK_{sig,B}, t^0_{A_{pk}\to B_{sk}}, t^1_{A_{pk}\to B_{sk}})$, if yes, $res_B = 1$, otherwise $res_B = 0$.

If $res_A = res_B = 1$, it outputs 1, otherwise outputs 0.

Based on the DERE scheme, we construct the concrete DORE scheme as follows:

- DORE.Setup$(1^\lambda) \to \mathsf{pp}$: With the inputs, run the DERE.Setup algorithm and output pp.
- DORE.KeyGen$(\mathsf{pp}) \to (\mathsf{pk}, \mathsf{vk}, \mathsf{sk})$: With the inputs, run DERE.KeyGen(pp) algorithm and output $(\mathsf{pk}, \mathsf{vk}, \mathsf{sk})$. Instantiated by the network controllers A and B to generate two pairs of keys, i.e., $(\mathsf{pk}_A, \mathsf{vk}_A, \mathsf{sk}_A)$ and $(\mathsf{pk}_B, \mathsf{vk}_B, \mathsf{sk}_B)$.
- DORE.Token$(\mathsf{pp}, \mathsf{pk}_u, \mathsf{sk}_v) \to \mathsf{tok}_{(u_{pk}\to v_{sk})}$: With the inputs, run DERE.Token algorithm and output $\mathsf{tok}_{(u_{pk}\to v_{sk})}$. Instantiated by network controllers A and B to generate two tokens, i.e., $\mathsf{tok}_{A_{pk}\to B_{sk}}$ and $\mathsf{tok}_{B_{pk}\to A_{sk}}$.
- DORE.Verify$(\mathsf{pp}, \mathsf{vk}_u, \mathsf{vk}_v, \mathsf{tok}_{(v_{pk}\to u_{sk})}, \mathsf{tok}_{(u_{pk}\to v_{sk})}) \to 0/1$: With the inputs, run DERE.Verify algorithm and output decision bit $0/1$. Instantiated by network controllers A and B, to generate two tokens $\mathsf{tok}_{(B_{pk}\to A_{sk})}$ and $\mathsf{tok}_{(A_{pk}\to B_{sk})}$.

- DORE.Enc(pp, m, sk) $\rightarrow$ ct: It takes a message $m \in \{0,1\}^*$ and sk as input. It encrypts message bit encodings $\epsilon(m_i, ID)$, which is defined as $\epsilon(m_i, ID) = (i, m_1 m_2 \ldots m_i \| 0^{n-i}, ID)$ for $ID \in \{0,1,2\}$, as follows:

  ✓ If $m_i = 0$, it computes ciphertexts ct$[i] = (\text{ct}[i,0], \text{ct}[i,1])$ as follows:

  $$\text{ct}[i,0] = \text{DERE.Enc}(\text{pp}, \epsilon(m_i, 0), \text{sk}), \quad \text{ct}[i,1] = \text{DERE.Enc}(\text{pp}, \epsilon(m_i, 1), \text{sk}).$$

  ✓ Else if $m_i = 1$, it computes ciphertexts ct$[i] = (\text{ct}[i,0], \text{ct}[i,1])$ as follows:

  $$\text{ct}[i,0] = \text{DERE.Enc}(\text{pp}, \epsilon(m_i, 1), \text{sk}), \quad \text{ct}[i,1] = \text{DERE.Enc}(\text{pp}, \epsilon(m_i, 2), \text{sk}).$$

  Finally, this algorithm returns ct $= (\text{ct}[1], \ldots, \text{ct}[n])$. Then this algorithm is instantiated by network controllers A and B to get ciphertexts ct_A and ct_B.

- DORE.Test(pp, ct_A, ct_B, $\text{tok}_{(A_{pk} \rightarrow B_{sk})}$, $\text{tok}_{(B_{pk} \rightarrow A_{sk})}$) $\rightarrow$ res: It takes a pair of ciphertexts ct_A, ct_B and tokens $\text{tok}_{(A_{pk} \rightarrow B_{sk})}, \text{tok}_{(B_{pk} \rightarrow A_{sk})}$ as input. Test algorithm runs DERE.Test iteratively. In particular, for $i = 1$ to n, the algorithm executes the following:

  1. If $i = n + 1$, then return 0.
  2. Else Compute res_A^i and res_B^i as follows:
     ✓ DERE.Test$(\text{ct}_A[i,0], \text{ct}_B[i,1]) \rightarrow \text{res}_u^i$.
     ✓ DERE.Test$(\text{ct}_A[i,1], \text{ct}_B[i,0]) \rightarrow \text{res}_B^i$.

  (a) If $\text{res}_A^i = 1$, then returns 1, which means $m_A[i] > m_B[i]$.

  (b) Else if $\text{res}_B^i = 1$, then return -1, which means $m_A[i] < m_B[i]$.

  (c) Else $i \leftarrow i + 1$.

When applying this concrete DORE scheme into CASTOR project, we need to consider the communication process between BGP nodes C and B. To ensure the authentication of information transmitted between BGP nodes C and B, a signature algorithm (BLS or Schnorr signature) is supposed to be used. Here, we take the Schnorr signature as an example. Following the syntax of Schnorr signature and the DORE, we describe the communication process between nodes C and B as follows, also shown in Figure 6.4:

Token from node C to node S:

- Sig.KeyGen$(\text{pp}_{sig}) \rightarrow (pk_C, sk_C)$: The node C runs this algorithm to generate a secret and public key pair, i.e. $sk_C \leftarrow_\$ \mathbb{Z}_p$ and $pk_C = g^{sk_C}$.

- Sig.Sign$(sk_C, \text{tok}_{B_{pk} \rightarrow A_{sk}}) \rightarrow \sigma_C$: The node C runs this algorithm. Firstly, computes $R_C = g^{r_C}$, where $r_C \leftarrow_\$ \mathbb{Z}_p$; Then computes the challenge $c_C = \text{H}_1(R_C, pk_C, \text{tok}_{B_{pk} \rightarrow A_{sk}})$,; Then the signature $\sigma_C = (c_C, s_C)$.

- Sig.Verify$(pk_C, \sigma_C, \text{tok}_{B_{pk} \rightarrow A_{sk}}) \rightarrow 0/1$: The node S runs this algorithm. Firstly computes $R'_C = g^{s_C} pk_C^{c_C}$. Then checks whether the equation holds or not $c_C \stackrel{?}{=} \text{H}_1(R'_C, pk_C, \text{tok}_{B_{pk} \rightarrow A_{sk}})$. If yes, outputs "1" for accepting the token information. Otherwise, output "0" for rejecting.

Token from node S to node C:

- Sig.KeyGen$(\text{pp}_{sig}) \rightarrow (pk_S, sk_S)$: The node C runs this algorithm to generate a secret and public key pair, i.e. $sk_S \leftarrow_\$ \mathbb{Z}_p$ and $pk_S = g^{sk_S}$.

- Sig.Sign$(sk_S, \mathsf{tok}_{A_{pk}\to B_{sk}}) \to \sigma_C$: The node S runs this algorithm. Firstly, computes $R_S = g^{r_S}$, where $r_S \leftarrow_{\$} \mathbb{Z}_p$; Then computes the challenge $c_S = \mathsf{H}_1(R_C, pk_C, \mathsf{tok}_{A_{pk}\to B_{sk}})$,; Then the signature $\sigma_S = (c_S, s_S)$.
- Sig.Verify$(pk_S, \sigma_S, \mathsf{tok}_{A_{pk}\to B_{sk}}) \to 0/1$: The node S runs this algorithm. Firstly computes $R'_S = g^{s_S} pk_S^{c_S}$. Then checks whether the equation holds or not $c_S \stackrel{?}{=} \mathsf{H}_1(R'_S, pk_S, \mathsf{tok}_{A_{pk}\to B_{sk}})$. If yes, outputs "1" for accepting the token information. Otherwise, output "0" for rejecting.

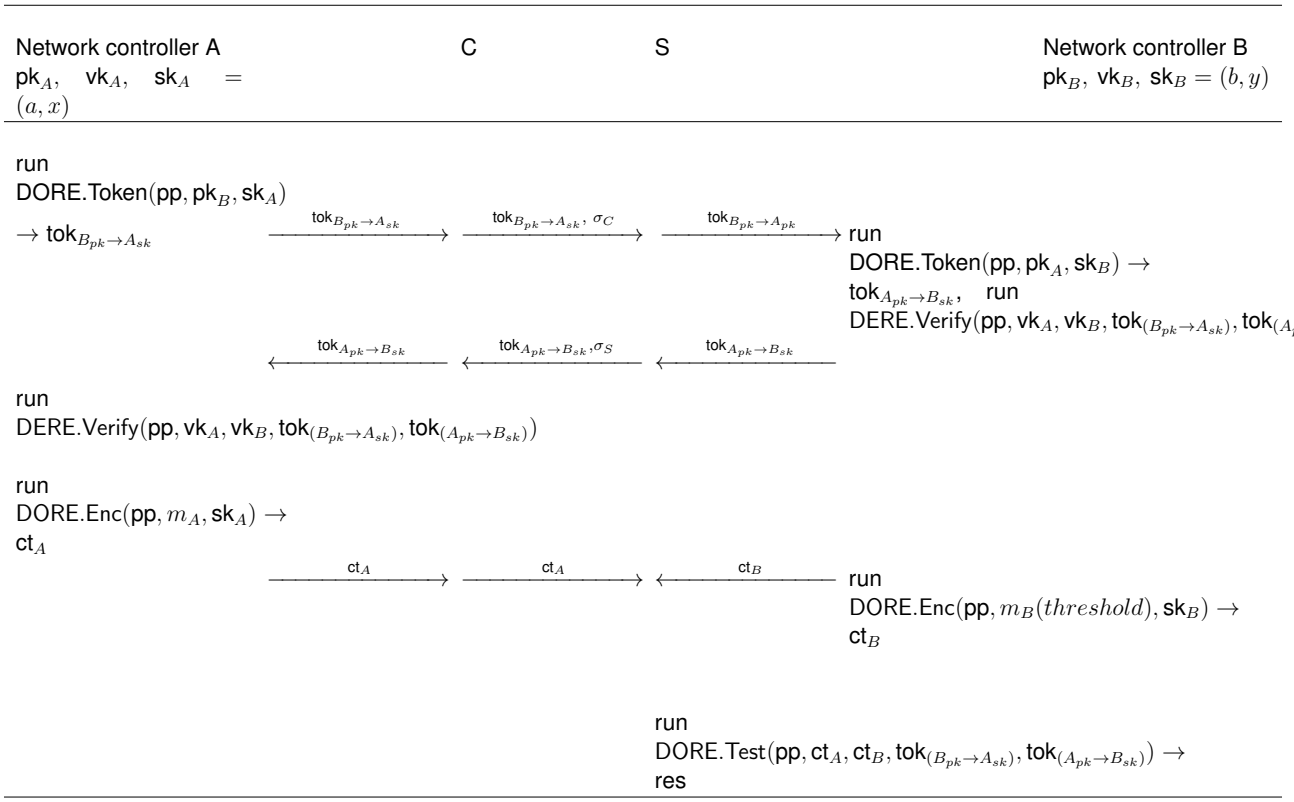

Figure 6.4: Workflow of the concrete DORE scheme used in CASTOR

Chapter 7

Analysing Execution Path Determinism for Traffic Engineering

Having thoroughly described in Chapter 3 the Trust Network Device Extension (TNDE) and the CASTOR security umbrella, and in Chapters 4 and 5 the CASTOR trust sources, the present chapter focuses on the deployed tracing units that enable the later trust characterization of the target vRouter. More specifically, it examines the granularity of system evidence, both for configuration (attestation source) and runtime behaviour (FSM source).

Specifically, this chapter investigates the core hypothesis that highly detailed, non-deterministic execution paths, which are strictly dependent on volatile routing inputs, can be systematically abstracted into a deterministic formal model. To achieve this high-degree, coverage-based verification, the tracing methodology explicitly identify the precise entry and exit boundaries within the execution flow. As subsequent sections will demonstrate, internal operations that dictate the integrity of routing data structures are inherently non-deterministic. Consequently, the assumption that the routing integrity can be guaranteed by simplifying the tracing into a static configuration verification is fundamentally insufficient. Instead, dynamic, non-intrusive tracing is required to extract the necessary granular evidence, bridging the gap between raw, variable system behaviour and a verifiable, deterministic Finite State Machine (FSM) abstraction.

This chapter presents the first instantiation and experimental evaluation of monitoring hooks and introspection probes within the context of a routing function. As part of the experimental setup, the current tracing is limited to the control-plane implementation of the xRD routing. In particular, it explores how updates can be performed through the FlexAlgo routing structure in the context of Segment Routing (SR). The ultimate goal is to identify the boundaries of tracing required to enable runtime behavioral monitoring based on FSM models.

7.1 Granularity of Traces and the Attestation Hypothesis

With the core elements of the CASTOR architecture in place—specifically the TNDE, the FSM, and the primary attestation source—this chapter turns to examining the level of detail and the nature of the traces needed to effectively support these trust mechanisms. The primary goal is to determine whether and how granular system evidence can be extracted and utilized by the attestation source and the FSM to unlock the verification phase of the routing plane within containerized routing element. To this end, the tracing methodology is divided into two distinct work scopes:

1. **Container-to-Host Monitoring:** Capturing the boundary interactions and system calls as the virtual router interfaces with the underlying infrastructure.

2. **Container-Internal Flow Monitoring:** Observing the isolated execution logic protocol inner workings while updating routing policies that trigger the Shortest Path First (SPF) calculation, occurring entirely within the user-space of the container.

Across these scopes, the CASTOR framework must capture two fundamental categories of evidence:

- **Static Evidence:** The resting state of the network intelligence, specifically the content of the Routing Information Base (RIB) – the central database responsible for path calculation and policy application – and the Forwarding Information Base (FIB) – the optimized execution table containing the actual instructions for packet switching.
- **Volatile Evidence:** The dynamic execution flow and ephemeral state transitions that occur strictly during an active administrative policy update.

This dichotomy of evidence introduces a two-fold architectural question: (i) *First, what specific traces can be reliably extracted in a highly virtualized environment characterized by continuous context switching between the containerized virtual router and the host operating system?* (ii) *Second, what technologies allow us to introspect these environments, and are they appropriate for the specific type of evidence required?*

To answer these questions, we must first establish the expected lifecycle of a routing policy update. Regardless of specific vendor architectures, a configuration change follows a predictable sequence: initial administrative intent (committed via an orchestrator, PCE, or CLI) is ingested by the management plane, which then dispatches internal notifications to the PCEP agents and routing protocol engines operating within the instantiated vRouters. Upon notification, these engines execute the necessary mathematical calculations, such as a Shortest Path First (SPF) algorithm, against the current Link State Database, evaluating one-hop neighbors and path profiles with labels of the Flexible Algorithm, to determine the new valid next hop in the path. Finally, this calculated intelligence is simultaneously programmed into the router's local forwarding data structures and broadcast to network neighbors for global synchronization. Mapping this theoretical lifecycle to concrete, observable system events forms the basis of our verification strategy.

To establish a robust mechanism for control plane verification within containerized environments, this chapter investigates the necessary scope of system introspection. The central architectural question driving this analysis is whether a single trace unit, designed exclusively to extract static evidence from routing data structures, is sufficient to guarantee integrity, or if a secondary trace unit is required to capture the volatile evidence of the active execution flow. To evaluate this, we define the core hypothesis of this investigation: **The integrity of the state of the router, established by tracing the static routing data structures observed at the host level, serves as the singular and sufficient metric for control plane verification**. This hypothesis posits that any modification to the routing configuration must manifest as an observable change in the static memory mappings that matches the authorized administrative intent. By monitoring this static boundary between the container and the host, we define the trust bounds of the system model, building upon the integrity measurements of the TNDE.

The analysis reveals that while host-level kernel-extensions are required to directly introspect the static memory mappings of data structures (the resting state), they provide only a snapshot in time. Static evidence alone cannot verify the causal lineage of a routing update, nor can it capture the ephemeral protocol mathematics that justify a state change. Furthermore, to accommodate larger traffic bounds, environments frequently enable DPDK optimizations. This introduces a kernel-bypass architecture that completely hides the active forwarding state from standard static host checks.

Crucially, however, the underlying control plane process, specifically the sequence of systemcalls utilized to instantiate the policy update, remains unchanged regardless of these data plane optimizations.

Consequently, the architectural reality invalidates the core hypothesis that a single trace unit for static evidence is sufficient for control plane verification.The outcome of this chapter demonstrates that we must introduce an additional trace unit, leveraging eBPF, to monitor the volatile execution flow and capture the dynamic events occurring precisely when a policy update is instantiated.

7.2 Deployment Architecture

The deployment strategy is bifurcated to address both the theoretical tracing capabilities and the operational performance of the framework. This differentiation ensures that we can successfully validate the depth of the execution flow that can be monitored across both the container and the host. Furthermore, it demonstrates that these extraction techniques remain highly applicable to production-equivalent scenarios, specifically when dealing with commercial, closed-source virtual routers deployed in complex network topologies.

- **Control Plane Proof of Concept:** To facilitate a thorough investigation into the reachability of system traces, the architecture resides on a single virtual machine host. In this configuration, each Cisco IOS XRd router operates as a containerized instance. This centralized approach enables the eBPF trace unit to monitor concurrent interactions between multiple routing processes and the shared kernel of the host. This environment is specifically optimized to determine the precise level of tracing that can be achieved, identifying the exact system events that must be monitored to extract the necessary observations that will feed the trust sources, most notable the FSM.
- **Data Plane Use Cases:** The topology transitions to a multi virtual machine structure for the evaluation of CASTOR use cases under realistic conditions. In this environment, each virtual router is isolated within its own dedicated virtual machine. Furthermore, the high speed data plane utilizing DPDK is positioned in a separate virtual machine. This architectural separation mirrors the deployment models followed in realistic, production-grade environments to ensure Service Level Agreements (SLA) compliance. By isolating the data plane from control plane interference, we prevent CPU cache misses and resource contention when handling large data packets.

7.3 System Model and Trace Unit Instantiation

The CASTOR framework incorporates an architecture comprising two distinct trace units designed to capture both the static and volatile properties of the system. The first trace unit utilizes eBPF for deep observation of volatile system events, specifically, systemcalls at the shared boundary (Kprobes) and user-space probes (Uprobes) executing within the containerized Cisco IOS XRd router. Because eBPF is inherently event-driven, it cannot be used to introspect the static integrity of the container itself or its internal data structures. Therefore, to address static properties, the second trace unit employs host-level kernel extensions that directly introspect the memory mappings of these various data structures.

By strictly separating the execution environment of the host from the containerized XRd router, this study employs the eBPF trace unit to examine the instantiation of a Flexible Algorithm update. Specifically, the focus is on Algorithm 128, a user-defined Segment Routing topology identifier utilized to compute forwarding paths based on custom administrative constraints (such as link exclusion). This tracing mechanism allows for the verification of routing integrity from the outside looking in, without relying on the internal telemetry of the router itself.

7.3.1 The Flexible Algorithm 128 Instantiation and Logical Data Structures

The dynamic modification of the vanilla **Flexible Algorithm 128** serves as the primary mechanism to trigger trace analysis. Within the context of **Segment Routing Traffic Engineering**, this algorithm allows network operators to define specific routing behaviors, such as explicitly including or excluding certain links based on custom network properties rather than standard shortest-path metrics.

Specifically, Flexible Algorithm 128 is pre-programmed with a strict logical constraint: it must exclude any network links marked with ”Affinity Bit 0”, which we administratively define as a low-trust identifier. To trigger a state transition and observe the system’s reaction, an administrative update is committed that deliberately assigns this low-trust bit to an active, currently in-use network interface. Because the algorithm’s rules explicitly prohibit traffic from traversing any link marked with Affinity Bit 0, the routing engine must respond immediately to the configuration change.

Upon the commit of this administrative policy, the **IS-IS (Intermediate System to Intermediate System)** process performs the following actions:

1. **Topology Pruning (FSM Input Trigger):** The affected interface is removed from the logical topology database constraint set for Algorithm 128. This marks the exact entry point into the Calculation (C) state.
2. **Shortest Path Calculation (Graph Reconstruction):** The mathematical engine executes a constrained **Shortest Path First** calculation using Dijkstra’s algorithm to determine a new valid next hop interface that avoids the untrusted link. When SR-TE is elevated to its CASTOR equivalent, this mathematical engine will interact with the backend CASTOR multi-objective optimization engine.
3. **Instruction Dispatch (FSM Output Realization):** The resulting intelligence is finalized. This marks the exit of the Calculation (C) state; the new next hop and its associated transport labels are synchronized from the RIB and programmed directly into the FIB.

As illustrated in the flow diagram in Figure 7.1, this conceptual deployment model of the XRd router within the host infrastructure maps the exact trajectory of a configuration policy update, such as the instantiation of Flexible Algorithm 128.

To clarify the trace unit’s methodology and the router’s architecture, the diagram employs a specific color-coding schema. The fundamental state flow is represented in black, while the specific tracing mechanisms (the technical mechanics of which are analysed comprehensively in Section 7.7.4) are distinguished by colour:

- **Black** denotes the sequential internal states (A through D) representing the state transition model process flow, during updating a routing policy.
- **Red** highlights the kernel-level kprobes attached to critical system calls (e.g., read, futex, sendto). These capture the entire execution flow across the user-space/kernel boundary, specifically monitoring the protocol engine’s communication phases.
- **Blue** identifies the user-space uprobes (igpls_update_link and igpls_update_node) deployed to establish the causal chain, successfully capturing the precise entry and exit boundaries of the Logic state (State C) calculation.
- **Yellow** represents the target rcmd_spf mathematical function within the calculation engine. Because this user-space function is heavily optimized and stripped from the production binary, it cannot be directly hooked, visually demonstrating the necessity of the bounded Blue uprobe strategy.

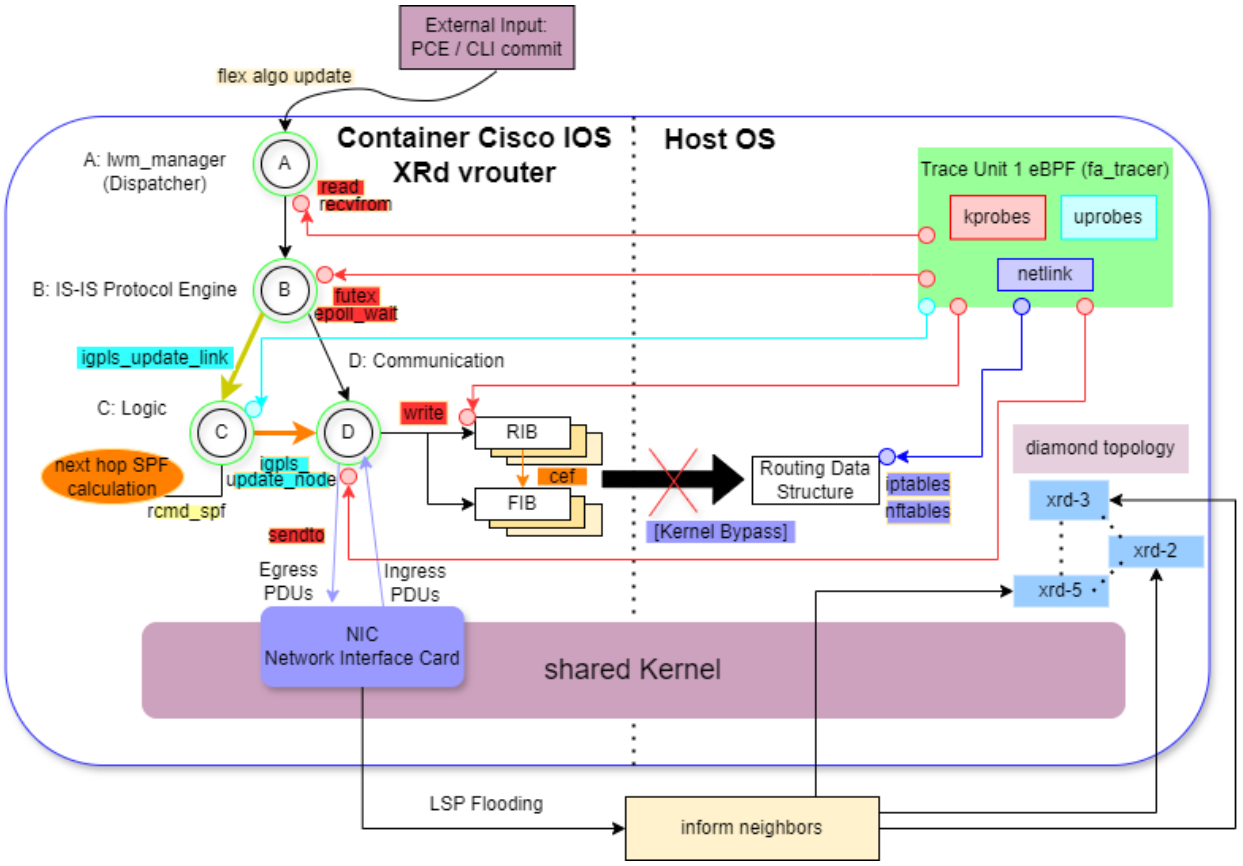

Figure 7.1: Conceptual deployment model and internal flow diagram of a Flexible Algorithm configuration policy update within the Cisco IOS XRd router

- **Purple** denotes the netlink tracer, originally designed to monitor standard host-level iptables and nftables modifications. However, as illustrated by the Kernel Bypass, the XRd protocol engine transmits Egress and Ingress PDUs directly to the Network Interface Card (NIC) to flood the diamond topology. This hardware-direct path bypasses the standard Host OS routing data structures entirely, rendering the netlink tracer unused for this specific protocol traffic.

Crucially, the captured state-level transitions depicted here establish the baseline for both deployment architectures. While the fundamental logical states of the routing process remain structurally consistent, the subsequent analysis will specifically evaluate whether the exact sequence of underlying system calls extracted in the second setup, where the XRd instance is isolated within its own dedicated Virtual Machine for high-speed data plane use cases, strictly mirrors the traces observed in the single-host control plane Proof of Concept.

7.4 Trust Bounds and Observable Extraction

The primary objective is the establishment of a mechanism for control plane verification. This is achieved by validating the integrity of routing data structures through evidence collected from the host environment. By observing the system externally, the framework can verify the internal operations of the router without relying on the internal reporting of the potentially compromised container.

This verification methodology relies on two fundamental principles: introspection and observable extraction analysis. Introspection provides the capability to observe the operational state oof the container without relying on internal application interfaces. Observable Extraction Analysis refers to the deterministic procedure of capturing kernel-level evidence, such as system calls and user-space probes, to verify

the internal logic of the router.

7.4.1 Experimental Setup and Assumptions

The experimental setup is anchored on Intel Trust Domain Extensions (TDX) to provide hardware isolation for the execution of CASTOR components. Within this architecture, we do not actively verify the host operating system, nor do we monitor events that affect the systematic behavior of the host itself, as such vectors fall outside our defined threat model. Consequently, rather than attempting to establish the baseline security of the host environment, the scope of observation is deliberately constrained to facilitate highly targeted experimental validation. This allows the framework to focus exclusively on monitoring the internal execution flows and events that can maliciously alter the containerized routing data structures.

By residing within this isolated host infrastructure, the eBPF trace unit is established as a trusted part of our Layer 1 (e.g., Hardware-isolated Tracing & Attestation). This external perspective leverages the host's ultimate over the shared kernel, enabling the framework to observe the virtual router from an independent vantage point and ensuring that the extracted evidence is an authentic representation of the internal protocol state.

7.4.2 Methodology and Tracer Implementation

Systemcall	Associated Process	Functional Role in Routing Update
read	isis_uv	Reads the internal notification and configuration update dispatched by the lwm_manager, transitioning the intent into the protocol engine.
recvfrom	lwm_manager / isis_uv	For lwm_manager, it pulls the configuration update directly from the System Database (SysDB). For isis_uv, it captures ingress LSPs from the network interface.
futex	lwm_manager	Manages thread synchronization, ensuring that configuration updates are processed without race conditions.
sendto	isis_uv	Executes outbound communication, multicasting topologyupdates to network neighbors and transmitting Netlink messages to request host kernel updates.
write	isis_uv	Commits calculated routing entries to internal infrastructure handles, finalizing the intelligence for data plane use.
epoll_wait	isis_uv	Monitors multiple file descriptors simultaneously, allowing the routing engine to remain in a highly responsive wait state with minimal CPU overhead.

Table 7.1: Monitored Systemcalls and their Functional Roles

The development of the tracer followed a rigorous system call analysis phase. Using the **syscount bpfcc** tool on the container bash, the Cisco ISIS process was monitored during administrative commit operations. This analysis identified a deterministic pattern of systemcalls, specifically **sendto**, **recvfrom**, **write**, **read**, **futex** and **epoll wait**, which characterize a routing policy update and are presented in the collective Table 7.1.

The **fa_tracer** was implemented in C, leveraging **libbpf** to load hooks directly into the host kernel. By targeting the **isis_uv** thread (the I/O engine of the IS-IS process), the tracer captures the exact moment

the routing logic interacts with the underlying operating system. Specifically, it observes how the process utilizes Netlink sockets to transmit structural network configuration messages and state changes, and standard file descriptors to manage active network I/O, internal IPC and event monitoring.

The experimental methodology employs a sequence of two distinct scenarios:

1. **Process Mapping:** A single virtual router instance is deployed initially to analyze thread identifications and specific process names. This stage allows for the precise isolation of the components responsible for the calculation logic of the routing protocol, as well as its communication with internal management components (such as the System Database via lwm_manager) and its attempted external discovery via the Network Interface Card.

2. **Topology Analysis:** A complete diamond topology is deployed to observe the causal chain of events during a configuration update across the network. This multi-node configuration is essential to verify how a local administrative change propagates to neighboring systems and how those neighbors respond to the update.

7.5 Data and Process Structures of the Routing Plane

The integrity of the routing plane is maintained through a sequence of interactions between intellectual databases and operational forwarding tables. By defining these structures, we establish the context required to interpret the trace evidence collected by the **fa_tracer**.

7.5.1 Logical Data Structures: RIB and FIB

The routing plane relies on two primary data structures to manage the lifecycle of a prefix. These structures represent the distinction between the calculation of a path and its actual implementation in the data plane.

- **RIB (Routing Information Base):** The RIB serves as the master decision making database. It is maintained by the central RIB daemon process (e.g., ipv4_rib). This daemon acts as the ultimate arbiter, consolidating network reachability information (prefixes) learned from various routing domains and administrative inputs. These sources include static routing configurations, directly connected interfaces, and dynamic interior gateway protocols such as **IS-IS**. The RIB evaluates all available paths to a destination, resolves recursive routes, and applies administrative constraints, such as the specific link exclusions defined in Flexible Algorithm 128.

  Example Context: Upon the commit of an administrative policy that flags a specific network interface with a **LOW_TRUST** affinity tag (using Affinity Bit 0), the RIB re-evaluates its topology. It initially identifies the prefix 1.10.1.2 is logically reachable via interface *Gi0/0/0/3*. However, because the Algorithm 128 constraint explicitly dictates the exclusion of any links marked with this LOW_TRUST bit, the RIB prunes this specific adjacency from its candidate set and selects a secondary path to maintain policy compliance.

- **FIB (Forwarding Information Base):** The FIB is the optimized execution table derived from the RIB, forming the operational core of the **Cisco Express Forwarding (CEF)** architecture. While the RIB remains in the control plane to process complex protocol logic, CEF pre-compiles this intelligence and pushes the active routes down into the FIB, which resides closer to the data plane's memory execution space. This advanced switching topology ensures that the forwarding engine (such as a DPDK polling thread) can switch packets at line rate without triggering CPU interrupts

to consult the control plane. Consequently, the FIB contains only the hardware-ready instructions required for packet switching. It maps destination prefixes to specific egress interfaces and transport labels, ensuring that the forwarding engine can process without performing complex protocol lookups.

Example Context: Once the ISIS completes the constrained shortest path calculation, CEF synchronizes the result from RIB down to the FIB. The FIB is updated to swap the next hop for prefix 1.10.1.2 to the newly verified secure interface *Gi0/0/0/1*. The table now holds the immediate, cache-optimized instruction to encapsulate the packet with transport label 16202 and egress the system via the trusted interface, bypassing any further protocol lookups.

7.5.2 Process Structures and System Call Mechanics

The synchronization of data between the RIB and FIB, as well as the communication with network neighbors, is facilitated by specific kernel-level processes. The **fa_tracer** monitors these processes by hooking into the fundamental system calls they utilize to move data across the user-space and kernel boundary.

- **lwm manager (Messaging and Synchronization):** This process operates as the internal messaging bus for the virtual router. It is responsible for informing the routing engine when a configuration change has been committed to the system database.
  - ✓ **futex (Fast User-space Mutex):** The *lwm manager* utilizes this systmcall to manage thread synchronization. In a multi-threaded routing environment, futex ensures that the configuration update is processes without race conditions, allowing threads to wait for state changes efficiently.
  - ✓ **read:** This call is employed to ingest the binary configuration data from the system database, transitioning the administrative intent from a static file or memory segment into the active protocol logic.
- **isis_uv (Communication and Forwarding Programming):** This process acts as the interface between the **ISIS** protocol and the external environment. It is the primary engine for topology discovery and the synchronization of the **FIB**.
  - ✓ **sendto:** This systemcall is critical for outbound network communication. The tracer observes sendto when *isis_uv* multicasts LSPs (Link State PDUs) to neighbors or transmits Netlink messages to the host kernel to request a FIB update.
  - ✓ **recvfrom:** This systemcall represents the ingress of external intelligence. It is the mechanism by which the router receives Link State PDUs from its neighbors, including topology updates and sequence number packets (CSNPs/PSNPs).

    Scientific Significance: In the context of the experimental setup, recvfrom is the starting point of the update chain. When xrd-1 receives a topology update from a neighbor, recvfrom captures the raw binary data. If the tracer detects a recvfrom is used to receive Netlink acknowledgments from the kernel, confirming that a requested route was successfully written to the FIB.
  - ✓ **write:** This call is used to commit calculated routing entries to the internal infrastructure handles. It represents the handover point where the calculated intelligence is finalized for data plane use.
  - ✓ **epoll_wait:** High-performance routing engines utilize epoll_wait to monitor multiple file descriptors simultaneously. This call allows *isis_uv* to remain in a wait state until a network event, such as a neighbor hello or a configuration update, requires immediate processing, ensuring the system remains responsive with minimal CPU overhead.

- **libcrmd.so (The Calculation Engine):** This shared library implements the Route Convergence Monitoring and Diagnostics mechanism and houses the core mathematical logic for the Shortest Path First (SPF) calculation. However, direct introspection of this library presents significant tracing challenges. Because Cisco wants to optimize these production binaries and protect its property, it heavily strips and unrolls the compiled code, hiding the core spf and Dijkstra functions from the static symbol table. Furthermore, compiler-level Tail Call Optimizations (TCO) eliminate standard function return instructions, rendering traditional duration-based exit hooks (uretprobes) entirely ineffective. This architectural opacity prevents direct measurement of the algorithm's duration, requiring the trace unit to pivot to a causal-chain methodology.
- **libigpls.so (The Link State and Calculation Engine):** To overcome the tracing limitations discovered in the calculation engine, the framework monitors this closed-source shared library. It acts as the structural brain of the IS-IS process, responsible for managing the Link State Database (LSDB) and orchestrating the data transformation prior to route injection. By leveraging the dynamic symbol table, the tracer places user-space entry hooks (uprobes) on explicit boundary functions to bound the calculation phase:
  - ✓ **Uprobe (Entry Calculation State - igp_lscache_update_link):** This probe acts as the precise Input Trigger. It captures the exact microsecond the router begins ingesting the topology change, such as the application of the LOW_TRUST affinity, and updates the specific link attributes within the Link State cache.
  - ✓ **Uprobe (Exit Calculation State - igp_lscache_update_node):** This subsequent probe captures the ongoing Data Transformation phase. It triggers when the routing engine updates the status of the affected network entities (the local node and its neighbor) connected to the modified link, providing verifiable evidence that the router is actively transforming its internal topological data structures in real-time in response to the trigger.

The successful extraction of these bounded traces represents the first critical milestone in converting these isolated state transitions into a comprehensive formal Finite State Machine (FSM). By isolating the exact entry and exit points for the SPF calculation, the framework proves that internal execution states can be reliably tracked and logically verified. While the tracing methodology will continue to explore deeper system interactions to map additional transitions, establishing this precise causal boundary officially unlocks the FSM capability, providing the structural foundation required for automated control plane verification.

7.6 Experimental Environments and Baseline Analysis

The empirical validation of the core hypothesis and the resultant control plane tracing were conducted across two distinct experimental scenarios. This phased approach allowed for the isolation of specific systemcall patterns generated by the Cisco IOS XRd control plane before observing the complete lifecycle of a Flexible Algorithm update within a production-equivalent topology.

7.6.1 Scenario 1: Single Instance Baseline and System-call Isolation

In the initial scenario, a single Cisco IOS XRd container (xrd-1) was deployed on the host operating system. The primary objective of this configuration was to perform a clean analysis of the dispatch and protocol processes without external network interference. By utilizing an isolated node, the experimental setup achieved the following:

- **Noise Reduction:** In a shared-kernel tracing architecture, monitoring multiple virtual routers simultaneously generates overlapping systemcalls that complicate the attribution of specific events to specific containers. Additionally, dynamic routing protocols like IS-IS are inherently noisy; in a live topology, they continuously broadcast Hello packets and flood Link State PDUs (LSPs) to maintain neighbor adjacencies. By deploying a single node with zero active network links, the setup completely silenced this continuous background protocol chatter and eliminated cross-container interference. This established a sterile monitoring window where any captured kernel interaction was definitively caused by the internal processing of the configuration update, rather than a reaction to external network stimulation.
- **Pattern Identification:** The trace unit successfully isolated the exact, sequential systemcall fingerprint triggered exclusively by a user *commit* operation in the local CLI.

However, this scenario also highlighted a critical dormant protocol state. Because the virtual router possessed no physical or virtual links to other neighbors (Links: 0/0), the IS-IS algorithm correctly calculated zero paths. While the trace unit captured the administrative intent (State A) and the initialization of the protocol engine (State B), the flow halted. The router attempted to discover neighbors via IS-IS Hello packets, but it could not progress to the logic and calculation phase (State C). Consequently, no routing data structures were updated, providing an ideal baseline for secure isolation but necessitating a more complex topology for full verification.

7.6.2 Scenario 2: Four Node Diamond Topology and Complete Flow Realization

To observe the complete handoff from administrative intent to data plane realization, a four-router diamond topology was implemented. This scenario provides the necessary network density to force the routing processes through the complete causal flow: from discovery, to calculation, and finally to internal structure programming.

The topology is structured as follows:

- **Nodes:** The core infrastructure comprises four Cisco IOS-XRd containerized instances (designated as xrd-1, xrd-2, xrd-3, and xrd-5). Arranged in a diamond configuration, these nodes emulate a representative slice of a Segment Routing network, providing the minimum viable complexity required to force multi-path routing calculations and simulate realistic network convergence.
- **Ingress Source node (xrd-1):** The point of configuration where Flexible Algorithm policies are committed and where the primary eBPF trace unit hooks are deployed.
- **Egress Destination node (xrd-2):** The destination node for traffic traversing the Algorithm 128 slice.
- **Transit Path:** The source node xrd-1 is connected to intermediate nodes xrd-3 and xrd-5, which are subsequently connected to xrd-2.

In this scenario, an active configuration update is performed on xrd-1. Unlike the single instance baseline, the trace unit captures the full multi stage execution flow. The protocol engine successfully receives neighbor confirmations, executes the mathematical calculation, and programs the forwarding structures, allowing the Trust Assessment Framework to record a complete observable extraction.

7.7 Mapping the Observable Extraction Flow

The transition from a single-node baseline to the multi node diamond topology allowed for a comprehensive mapping of systemcalls, thread behaviors, and internal process handoffs. By correlating eBPF trace unit hooks with the structural states of the virtual router, the complete causal lineage of a routing update was identified.

7.7.1 The Administrative Trigger

To validate the trace unit without resorting to destructive physical link failures, which generate excessive background noise and network-wide state churn, Segment Routing Traffic Engineering (SR-TE) was utilized. SR-TE allows for the dynamic, logical manipulation of network paths by altering specific interface attributes while the physical links remain active. This approach provided a highly deterministic, cleanly isolated configuration event to trigger the system hooks:

- **The Baseline:** The environment was configured with a Segment Routing Global Block (SRGB), which defines the base range of labels for the network, starting at 16000. Under normal conditions, traffic followed the default, unconstrained SPF calculation (Algorithm 0). The destination node (xrd-2) was assigned a standard node index of 102. Consequently, the routing protocol calculated a baseline transport label of 16102 (Base 16000 + Index 102) to route traffic over the default primary interface.
- **The Update Trigger:** The *GigabitEthernet0/0/0/3* interface (acting as the primary transit link facing intermediate node xrd-5) was administratively tagged with a specific link affinity group: LOW_TRUST (Affinity Bit 0).
- **The Execution:** Flexible Algorithm 128 was pre-configured to strictly avoid links marked with this low-trust affinity. Upon committing this policy update, the trace unit observed the routing engine immediately prune the affected GigabitEthernet0/0/0/3 link from its Dijkstra calculation. To maintain reachability, the protocol dynamically shifted traffic to a secure parallel path (GigabitEthernet0/0/0/1). For this specific constrained topology, the destination was assigned a distinct node index of 202, yielding a completely new, observable transport label of 16202 (Base 16000 + Index 202) mapped into the forwarding structures.

7.7.2 Scenario 1 Observation: The Dormant Protocol Flow

In the isolated single node baseline, the trace unit captures the initial administrative intent but reveals a definitive halt in the execution flow. Upon committing the low trust affinity, the lwm_manager successfully executes read and futex systemcalls, notifying the protocol engine of the configuration change (States A and B).

However, the flow becomes dormant at the network communication phase. Because the single-node baseline possesses no physical or virtual links to neighboring routers, it cannot establish adjacency. The trace unit observes the isis_uv process repeatedly executing a continuous stream of send to systemcalls. These isolated outbound transmissions represent the router futilely multicasting periodic IS-IS Hello Protocol Data Units (PDUs) out of its interfaces, attempting active neighbor discovery.

Because no neighbors exist to respond, the router cannot build a Link State Database. Consequently, the routing process cannot satisfy the essential topological conditions – specifically, the existence of active, verified transit links – required to progress to the logic and calculation phase (State C). Without a populated network graph, no Shortest Path First calculation is triggered, and the trace unit records zero

subsequent write systemcalls to the Routing Information Base. This effectively maps a dormant protocol state, proving the trace unit can successfully identify when an administrative intent is formally ingested but logically fails to achieve data plane realization.

7.7.3 Scenario 2 Observation: Complete State Transitions and Trace Capture

In the four node topology, the observable extraction by the host trace unit aligns directly with the complete functional flow of the routing architecture. The Trust Assessment Framework monitors the following sequential states to verify integrity:

- **State A (The Dispatcher):** The flow initiates at the management phase. Upon the user commit, the trace unit captures the lwm_manager reading the new policy from the system database. The extraction of read and futex systemcalls confirms the ingestion of the administrative trigger.
- **State B (Protocol Engine Initialization):** The lwm_manager notifies the routing processes. The eBPF unit observes the **isis** process waking up via epoll_wait and preparing to handle internal configuration change.
- **State C (Logic and Calculation):** This state represents the mathematical core of the protocol and serves as a primary verification point to prevent unauthorized route injections. Initially, the trace unit targeted the librcmd.so calculation library to extract the Shortest Path First (SPF) execution. However, because these production binaries are heavily stripped to protect intellectual property, the core mathematical functions are hidden from the symbol table, rendering direct function hooks ineffective. To overcome this architectural opacity, the trace unit employs Uprobes directly on the libigpls.so shared library to capture the precise entry and exit boundaries of the calculation state. The extraction of the igp_lscache_update_link execution marks the exact entry point where the router begins processing the new constraints. Crucially, empirical observation reveals that this link-update function is invoked multiple times prior to a single node update. This iterative execution suggests that the routing engine evaluates topological data on a per-path-profile basis, specifically processing individual Flexible Algorithm parameters such as affinity bits and transport labels. Consequently, this creates a distinct, observable computational footprint; monitoring this frequency potentially will allow the framework to trace and verify that these specific constraints have not been maliciously bypassed or altered during the evaluation. The subsequent igp_lscache_update_node execution captures the exit of the logic phase as the data structures are finalized. This bounded sequence proves that a legitimate SPF calculation for the next hop occurred in response to the trigger.
- **State D (Communication and Realization):** Following the mathematical calculation, the **isis** process diverges into two parallel operations.
  1. **Network Communication:** The trace unit captures sendto systemcalls as the router floods the new link state information to its neighbors (xrd-3, xrd-5) via the network interface card (NIC).
  2. **Structure Update:** Simultaneously, the trace unit captures write systemcalls as the protocol engine updates the internal Routing Information Base and subsequently the Forwarding Information Base with the new 16202 transport label.

7.7.4 Tracing Properties: System calls and User-Space Probes

To provide a complete security guarantee and prevent sophisticated evasion techniques, the eBPF trace unit employs different type of properties monitoring strategy based on the depth of functionality and

locality. This is strictly necessary because observing system calls alone provides an external view of process behavior, recording what data is being transmitted or written, but leaves the internal mathematical decision-making completely opaque. Without this internal insight, a compromised routing daemon could bypass the actual protocol algorithms entirely and use a legitimate-looking systemcall to inject a fabricated route.

- **Kernel-Level Systemcall Hooking (The Communication Phase):** The trace unit places **kprobes** on sendto, recvfrom, write, read, futex and epoll_wait system calls specifically assigned to the iss_uv thread. This enables the host to verify that the router is actively attempting to communicate with its network neighbors. Crucially, within this optimized architecture, the virtual router does not rely on the standard host kernel networking stack to route this communication; rather, system calls such as sendto and recvfrom represent the protocol engine interacting directly with the Network Interface Card (NIC) to exchange raw Link State PDUs on the wire. However, while these system calls prove that a message was successfully formatted and transmitted to the physical interface, they are inherently blind to the underlying mathematical algorithms generating that payload.
- **User-Space Probes (The Logic Phase):** To mitigate a scenario where a compromised process transmits a valid-looking Netlink systemcall to update the FIB without actually running the SPF algorithm, the trace unit activates **uprobes** configured to trigger precisely when the calculation logic is invoked within the shared routing libraries.
  - ✓ **Reverse Engineering and Stripped Binaries:** The discovery of the optimal tracing boundaries is currently a semi-automated process requiring manual reconnaissance of the virtual router's memory map. Initial static analysis attempts using standard ldd and nm utilities targeted a broad spectrum of Cisco shared libraries to determine what is invoked during a route calculation, including Inter-Process Communication (IPC) handlers, Object Request Architecture (ORA) boundaries (e.g., libisis_ip_ora.so, librib_ora.so), and core mathematical engines. This analysis revealed that the IS-IS process is heavily stripped. To protect its property, Cisco removes the static symbol tables (.symtab), hiding the internal spf and dijkstra function names. To overcome this, the framework utilized objdump -T to extract the Dynamic Symbol Table (.dynsym), revealing the explicitly exported boundary functions within the Link State (libigpls.so) and Traffic Enginnering (libigpte.so) libraries.
  - ✓ **Tail Call Optimization (TCO) and Dynamic Hooking:** During dynamic profiling, it was discovered that uretprobes (return hooks) failed to capture the exit of the calculation functions. This is a hallmark of Tail Call Optimization (TCO), where the compiler optimizes away return instructions, causing the eBPF return probes to be bypassed. Consequently, the trace unit's methodology was adapted to use a Causal Chain of standard entry uprobes. By hooking the exact start of the database update (igp_lscache_update_link) and the subsequent internal node update (igp_lscache_update_node), the tracer successfully establishes the boundaries of the data transformation phase without relying on fragile return instructions.
  - ✓ **Observation of Dijkstra Profiling:** During the execution of the bounded uprobes, the tracer captures the profiling of the data transformation not merely as a microsecond timer, but as an abstract proof of constraint satisfaction. This execution footprint proves that the algorithm actively evaluated the network constraints before calculating the new next hop. Specifically, it verifies that the mathematical engine processed parameters such as interface trust levels (e.g., explicitly excluding the interface tagged as LOW_TRUST) within libigpls.so. By proving these constraints were mathematically satisfied to determine the safe next hop, the framework guaranteed the logical integrity of the subsequent write systemcall as the IS-IS process updated the RIB.
- **Correctness Verification and State Synchronization:** By synthesizing the extracted evidence from these distinct memory addresses, the framework allows the trust sources (the FSM and the

core attestation engine) to verify that the routing update is shared and populated properly across the system architecture. Correctness is established by ensuring the causal sequence remains intact: the mathematical calculation must first update the RIB (which evaluated all potential interfaces for a node) and subsequently synchronize, through the Cisco Express Forwarding (CEF), the optimal path down to the FIB (which dictates the active interface used for data plane switching). The trust sources can use this causal evidence to attest that the observed interface swap in the FIB, and the resulting transition from transport label 16102 to 16202, was a legitimate computational response to the LOW_TRUST event, rather than an unauthorized path injection.

7.7.5 The Shared Kernel Bypass and Final Attestation

In standard deployments of Virtual Network Functions (VNFs), containerized routers exhibiting ”vanilla” behaviour do not maintain strict architectural isolation. Rather, as part of their baseline operations for managing route groups and lists, they typically synchronize and mirror their internal routing data structures directly to the outside host operating system. As comprehensively detailed by Suo et al. (2018), standard container networking relies heavily on the host OS; whether utilizing default bridge mode – which routes traffic through virtual ethernet (veth) pairs connected to a docker0 interface – or host mode – which completely shares the host's network namespace – these configurations force routing states and packet transmissions to traverse the standard Linux networking stack of the host. Consequently, this continuous synchronization across the container-host boundary introduces a critical attack surface. A compromised containerized router could theoretically manipulate these shared networking structures to achieve a container escape (jailbreak), a severe threat vector explicitly demonstrated by recent high-profile container runtime vulnerabilities (e.g., CVE-2024-21626, CVE-2025-31133, CVE-2025-52565).

However, the Cisco IOS XRd routers utilized in this architecture employ advanced forwarding optimizations that intentionally bypass – and effectively collapse – the host-level networking structures. Drawing an architectural parallel to Cisco's Converged SDN Transport framework, which simplifies the physical network infrastructure by collapsing distinct service and optical tiers into a unified network element, the XRd platform achieves similar network simplification by collapsing the intermediate virtualized networking layers. Consequently, attempting to verify routing integrity by statically checking the host's routing tables is fundamentally inadequate due to a profound lack of execution determinism. In this converged model, there is no deterministic guarantee regarding *who* updates the host routing tables or *when* these updates occur, as such state synchronization is typically offloaded directly to the underlying host networking hardware architecture rather than being explicitly and sequentially managed by the software routing engine.

This architectural reality fundamentally redefines the scope of the tracing methodology. Since the active routing state is realized within dedicated user-space memory domains that entirely bypass the host network stack, the framework cannot rely on the tracing of static state data structures within the host operating system. Instead, routing integrity must be guaranteed by continuously monitoring the dynamic execution flows and internal processes that occur strictly inside the containerized environment. By utilizing the eBPF trace unit from the shared kernel to capture the execution path, the framework provides an evidence-based guarantee that the internal, invisible forwarding tables were programmed by an authorized and mathematically verified protocol execution.

Furthermore, the tracing methodology established here represents the baseline requirement for defining the **critical path** of the routing protocol. By isolating the exact entry trigger (igp_lscache_update_link) and the exit realization point (igp_lscache_update_node), the eBPF trace unit successfully bounds the execution flow. However, this relies entirely on the dynamic symbols left exposed by the vendor. For fully opaque or heavily stripped libraries like librcmd.so, where even boundary functions are obfuscated or optimized away, eBPF uprobes reach their operational limits. Tracking the exact CPU instructions executed during the SPF calculation may eventually necessitate the integration of additional, lower-level

trace units, such as hardware-assisted execution tracing (e.g., Intel Processor Trace) or deep memory introspection.

However, pursuing this deeper level of visibility introduces profound architectural and security trade-offs. First, relying on specialized hardware extensions leads to a fragmentation of security solutions, as the tracing framework becomes strictly dependent on specific silicon availability rather than remaining platform-agnostic. Second, implementing deep memory introspection often necessitates highly privileged modifications or extensions within the kernel space. Ultimately, the deeper the introspection, the broader the threat model becomes; actively extending the trace and evidence space directly expands the attack surface. By deploying this additional tracing, the framework risks introducing new vulnerabilities into the very diagnostic mechanisms intended to secure the routing plane.

Given these security trade-offs, the current eBPF architectural implementation represents the optimal balance between visibility and safety. Rather than risking the expansion of the threat model by going lower in the system stack, the framework successfully captures the bounded critical path across the container-to-host boundary. Crucially, the raw sequence of events captured by these traces is inherently non-deterministic, as the exact execution path is strictly dependent on volatile routing inputs and network topologies. Therefore, to achieve reliable security guarantees, this highly variable execution footprint must be translated into a predictable framework. The necessary next step is to abstract this messy, input-dependent behavioural flow into a deterministic formal model. By mapping these raw system observations into a Finite State Machine (FSM), the framework can systematically evaluate whether the executed paths align with legitimate protocol operations, seamlessly bridging the gap between low-level system tracing and automated control plane verification.

Once this deterministic FSM is operational, its theoretical guarantees must be evaluated against actual, sophisticated exploits targeting the routing plane. A primary candidate for this future evaluation is the replay suppression attack – an exploit detailed in previous deliverables. In a high-traffic routing bottleneck scenario, an adversary could monitor network congestion via the link-map and exploit the router's internal load-balancing mechanisms. By launching a replay attack, the adversary essentially blinds the router to the true network state. This manipulation, whether achieved by artificially altering the link-state table or implicitly forcing a local node trigger, coaxes the routing engine into pushing traffic away from its secure, primary path and onto a vulnerable, transient link. Safeguarding against such an exploit serves as the primary use case for transitioning the FSM from a baseline model into an active defensive mechanism capable of detecting topology blinding and malicious path manipulation.

Within the specific context of the CASTOR framework, this attack scenario maps directly to the system's core trust designations. In CASTOR, persistent paths are elevated to **trust links**, while transient paths are classified as **alternate links**. Legitimate transitions between these states are strictly governed by the optimization engine – for instance, briefly directing traffic to an alternate link while a trust link undergoes resource provisioning or maintenance. The critical security challenge is determining how an attacker fraudulently convinces the router to make this shift, and how to detect it. Consequently, the potential next step in this research is to leverage the established tracing architecture to analyze systemcall evidence during these routing shifts. By monitoring the syscalls that trigger the load-balancer, the framework will investigate whether a transition from a trust link to an alternate link was the result of a valid optimization command, or an unauthenticated, unauthorized change. Identifying this unauthorized transition footprint vis syscall evidence will ultimately empower the FSM to distinguish between legitimate network engineering and malicious traffic redirection.

Chapter 8

CASTOR TNDE Key Management

Having described in detail all of CASTOR trust extensions and attestation mechanisms for enabling the runtime trust characterization of all elements comprising the routing plane, in the following table (Table 8.1), we summarize all crypto primitives (keys) that need to be established to the vRouters for achieving one of the core properties in evidence-based trust assessment theory [38]: this of **source-level validation and binding**. What this translates to is the *verifiability* of the trustworthiness evidence - ***how does the Orchestrator (acting as the Verifier) can validate the origin and binding of individual evidence traces when they are collected from multiple sources (Trace Units - Section 3.6) within an attesting routing element?*** Each evidence trust source (be it the Finite State Machine (FSM) or any additional verification unit - besides the Attestation Agent as this is the hardware-isolated trust source acting as the anchor (Layer 1) for enforcing the trust boundaries to the (user-space) instantiated routing functions) may operate under different trust models and may require individual validation with respect to their provenance, protection domain, and association with the Attester's identity. This creates the need of safeguarding the integrity of extracted traces, states and observations through a key management system that prevents their cloning or transfer. In practice, this implies cryptographic binding between evidence sources and the attestation function, supported by key management and endorsement models that enable composability and structured verification.

This is usually tasked as *device-binding* to ensure non-transferability among corrupt devices but also between (containerized) tenants (TNDIs) within the same virtualized environment. *CASTOR achieves this by binding all component-specific keys to the underlying (hardware-based) platform key*. The secure element (physical RoT as Layer-0) holds a **long-term endorsement key** $(sk_{\mathsf{EK}}, pk_{\mathsf{EK}})$ that constitutes its identity and is not used to produce attestation evidence, and a distinct **attestation key pair** $(sk_{\mathsf{AK}}, pk_{\mathsf{AK}})$ that is used to sign attestation evidence subject to policy. Trustworthiness evidence is always produced under an **attestation key** that is bound to usage policies safeguarded by the secure element. Such evidence capture the runtime configuration and behavioural state of either any trust source instantiated in Layer-2 or at the runtime TNDI layer (Figure 4.5).

Recall that all these keys (as described in Chapter 3) are generated and verified during the TNDI ONBOARDING Phase followed by the JOIN Phase of the attesting functions of CASTOR trusted computing base: As part of the former process, a similar to the $activateCredential$ semantic to a trusted platform module is executed for validating the underlying secure element and issuing the public part of its endorsement key (or the TNDE platform key as depicted in Table 8.1). The latter is focused on the generation and verification of the (policy-bound) attestation key pairs enhanced also with the generation of the asymmetric key pairs facilitating the advanced crypto protocols of *composite attestation* and *DORE* for the zero-knowledg sharing of path-centric trust metrics.

Table 8.1: Overview of Crypto Primitives (keys) established in CASTOR TNDE environment

Key Name	Type	Usage / Generation	Binding
RoT Key $(sk_{\text{EK}}, pk_{\text{EK}})$	asymmetric, long-term endorsement keypair	Exposed by the underlying secure element (RoT) (equivalent to an Endorsement Key) and used to authenticate/identify the device. Public part is issued/activated upon validation of the secure element by the respective vendor ($activateCredential$ process) Num:: [1 per RoT]	N/A
TNDE (platform) key $(pk_{\text{plat}}, sk_{\text{plat}})$	asymmetric, long-lived	Used by TN-DSM as SPDM identity key for the authentication of the TNDE to the Network Service Orchestrator. Generated by TN-DSM on initial startup and verified during the ONBOARDING phase. Num:: [1 per TNDE]	to RoT Key for device-binding (enforced in CASTOR TNDE v2)
TNDI key $(pk_{\text{tndi}}, sk_{\text{tndi}})$	asymmetric, long-lived	v1: Used by TN-DSM as the TNDI identity/authentication key, e.g., to authenticate the TNDI-SP data channels. v2: Used by TN-DSM to sign TNID-specific runtime evidence and trustworthiness clims and as authentication key for the TNDI-SP data channels (for the establishment of symmetric Diffie-Helman keys between the TN-DSM and the DLT Context Broker). Generated by TN-DSM during TNDI onboarding; pk_{tndi} shared with Network Service Orchestrator for verification. Num: [1 per TNDI]	to TNDE key (transitively to RoT)
Network Service Orchestrator key pair	asymmetric	Enables TN-DSM to authenticate the Orchestrator as a trusted entity. This is also used for establishing an authenticated and encrypted channel with each TN-DSM for the latter verification of the composite attestation key pairs per node Num:: [1 per Orchestrator]	N/A
BLS composite attestation key pair	asymmetric, long-lived	Used by each TNDI as the identity key for generating order-preserving proofs based on matrix-based constructions Num:: [1 per TNDI]	N/A
PS composite attestation key pair	asymmetric, long-lived	Used by TNDI as the identity key for generating composite attestation proof Num:: [1 per TNDI]	N/A
DORE key pair	asymmetric, long-lived	Used by network controller for constructing zero-knowledge proofs on trust metrics based on which computations (comparison) on the trust level can be conducted Num:: [1 per network controller]	N/A
Attestation Agent key pair	asymmetric	Key pair used for signing the extracted trustworthiness evidence by the Attestation Agent to be provided to the Local TAF per TNDE. Public part of AK is used for verification also from external Verifiers. Secret part is also binded to specific key restriction usage policies Num:: [1 per Attestation Agent]	to the RoT key
Data channel keys	symmetric session keys	Used as session keys for protecting the TNDI-SP data channels between TN-DSM and the Global TAF / DLT Context Broker; currently instantiated as TLS session keys. Num:: [per TNDI-SP data channel]	TNDI key authenticates channel, indirectly binding the derived session keys to the TNDI key

Continued on next page

Table 8.1 continued from previous page

Key Name	Type	Usage / Generation	Binding
Control channel keys	symmetric session keys	Used as session keys for the SPDM Secured Messages channel between the CASTOR Service Orchestrator and the TN-DSM. Num:: [per TNDI-SP control channel]	TNDE key authenticates channel, indirectly binding the derived session keys to the TNDE key (and transitively to the RoT key)

Chapter 9

Summary and Conclusions

This deliverable has advanced the CASTOR device-side trust architecture from the conceptual design established in D3.1 to a first integrated implementation of the key components on containerized routing platforms. D3.2 shows how the Trust Network Device Extension (TNDE), Trust Network Device Interfaces (TNDIs), and their security protocols can be realized on concrete vRouter platforms and tied into CASTOR's orchestration layer for trusted path routing. By grounding the abstract concepts of trusted device-side evidence collection in running code, protocol flows, and initial experiments, the deliverable demonstrates the feasibility of continuous, evidence-based trust assessment in the routing plane.

The document specifies and prototypes the TNDE services, the TNDI Security Protocol (TNDI-SP), and the JOIN and ONBOARDING flows that enroll TNDE platforms and onboard individual TNDIs into a CASTOR domain. It details how per-TNDI identities, lifecycle management, and data channels are combined with an eBPF-based Trace Unit and local trust sources to export trace and attestation evidence towards the Global TAF and Phala. In parallel, D3.2 instantiates CASTOR's tiered attestation architecture, providing concrete schemes for device-level configuration integrity verification, layered platform attestation, and composite path-level attestation that together extend trust from hardware roots of trust up to routing paths.

Beyond protocol and service design, the deliverable reports on the first integration and evaluation of trust sources and tracing mechanisms that operate on real XRd vRouter environments. The eBPF-based Trace Unit and associated experiments with FlexAlgo 128 routing updates show that CASTOR can extract both static and dynamic control-plane evidence from production-like setups, and that this evidence can feed into attestation checks and Finite State Machine (FSM)-based behavioral models. The FSM Trust Source chapter validates an end-to-end pipeline from trace collection through feature selection to model learning and highlights the potential of lightweight, explainable behavioral attestation, while also identifying open challenges around process attribution, dataset coverage, and runtime integration.

Overall, this deliverable provides the first concrete realization of CASTOR's device-side trust mechanisms across TNDE/TNDI-SP services, tiered attestation schemes, behavioral trust sources, tracing infrastructure, and key management. It captures the implementation status of version 1, clarifies which mechanisms are already realized in code and which remain at a proof-of-feasibility or exploratory stage, and sets out clear directions for version 2, including stronger hardware root-of-trust bindings, richer per-TNDI key hierarchies, deeper tracing and FSM integration, and further evaluation of composite attestation schemes. Together, these results complete the transition from the architectural principles of D3.1 to an operational prototype that the subsequent CASTOR work packages can build upon for end-to-end trusted path routing.

List of Abbreviations

Abbreviation	Translation
AK	Attestation Key
AS	Aggregate Signature
ATL	Actual Trust Level
BGP	Border Gateway Protocol
BLS	Boneh–Lynn–Shacham (signature scheme)
CC	Compute Continuum
CE	Compound Evidence
CBOR	Concise Binary Object Representation
CEF	Cisco Express Forwarding
CIV	Configuration Integrity Verification
CRL	Certificate Revocation List
DAA	Direct Anonymous Attestation
DICE	Device Identifier Composition Engine
DORE	Delegatable Order-Revealing Encryption
DPE	DICE Protection Environment
eBPF	extended Berkeley Filters
ECC	Elliptic Curve Cryptography
EK	Endorsement Key
FIB	Forwarding Information Base
FSM	Finite State Machine
fTPM	Firmware Trusted Platform Module
KDF	Key Derivation Function
LSPs	Link State PDUs
NIC	Network Interface Card
OCSP	Online Certificate Status Protocol
ORE	Order-Revealing Encryption
PCR	Platform Configuration Register
PKI	Public Key Infrastructure
RL	Revocation List
RIB	Routing Information Base
RoT	Root-of-Trust
RPKI	Resource Public Key Infrastructure
RTI	RoT for identity
RTL	Required Trust Level
RTM	RoT for measurement
RTR	RoT for reporting
RTS	RoT for storage
SAS	Sequantial Aggregate Signature
SGX	Software Guard Extensions

SLO	Service-Level Objective
SPF	Shortest Path First
SRK	Storage Root Key
TA	Trust Assessment
TAF	Trust Assessment Framework
TAF-API	Trust Assessment Framework – Application Programming Interface
TAF-DT	Trust Assessment Framework – Digital Twin
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCO	Tail Call Optimization
TDE	Trust Decision Engine
TDX	Trust Domain Extensions
TEE	Trusted Execution Environment
TLEE	Trustworthiness Level Expression Engine
TM	Trust Model
TMM	Trust Model Manager
TN-DSM	Trust Network Device Security Manager
TNDE	Trust Network Device Extensions
TNDI	Trust Network Device Interface
TNDI-SP	TNDI Security Protocol
TPM	Trusted Platform Module
TPR	Trusted Path Routing
TS	Trust Source
TSM	Trust Sources Manager
TVM	TEE Virtual Machine
VLR	Verifier-Local Revocation
VM	Virtual Machine
vTPM	Virtualized Trusted Platform Module
ZKP	Zero-Knowledge Proofs

Bibliography

[1] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi. Searchable encryption revisited: Consistency properties, relation to anonymous ibe, and extensions. In *Annual international cryptology conference*, pages 205–222, 2005.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[3] D.F. Aranha, C.P.L. Gouvêa, T. Markmann, R.S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[4] Diego F. Aranha, Benjamin Salling Hvass, Bas Spitters, and Mehdi Tibouchi. Faster constant-time evaluation of the kronecker symbol with application to elliptic curve hashing. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 3228–3238. ACM, 2023.

[5] P.S.L.M. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks (SCN)*, LNCS 2576, pages 257–267. Springer, 2002.

[6] Carsten Baum, Bernardo David, Elena Pagnin, and Akira Takahashi. Universally composable interactive and ordered multi-signatures. In *IACR International Conference on Public-Key Cryptography*, pages 3–31, 2025.

[7] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology — CRYPTO '96*, Lecture Notes in Computer Science, pages 1–15, Berlin, Heidelberg, 1996. Springer-Verlag.

[8] Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted aggregate signatures. In *International Colloquium on Automata, Languages, and Programming*, pages 411–422, 2007.

[9] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings 28*, pages 224–241. Springer, 2009.

[10] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual cryptology conference*, pages 578–595, 2011.

[11] Alexandra Boldyreva, Craig Gentry, Adam O'Neill, and Dae Hyun Yum. Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing. In *ACM CCS*, pages 276–285, 2007.

[12] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *International conference on the theory and application of cryptology and information security*, pages 435–464, 2018.

[13] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International conference on the theory and applications of cryptographic techniques*, pages 416–432, 2003.

[14] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 563–594. Springer, 2015.

[15] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[16] S. Bowe. BLS12-381: New zk-SNARK elliptic curve construction. Electric Coin Co. Blog Post, 2017. https://electriccoin.co/blog/new-snark-curve.

[17] E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology (CRYPTO)*, LNCS 6223, pages 237–254. Springer-Verlag, 2010. https://doi.org/10.1007/978-3-642-14623-7_13.

[18] Kyle Brogle, Sharon Goldberg, and Leonid Reyzin. Sequential aggregate signatures with lazy verification from trapdoor permutations. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 644–662, 2012.

[19] A. Budroni and F. Pintore. Efficient hash maps to G2 on BLS curves. *Applicable Algebra in Engineering, Communication and Computing (AAECC)*, 33(3):261–281, 2022. https://doi.org/10.1007/s00200-020-00453-9.

[20] David Cash, Feng-Hao Liu, Adam O'Neill, and Cong Zhang. Reducing the leakage in practical order-revealing encryption. *Cryptology ePrint Archive*, 2016.

[21] David Cash, Feng-Hao Liu, Adam O'Neill, Mark Zhandry, and Cong Zhang. Parameter-hiding order revealing encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 181–210, 2018.

[22] CASTOR. Operational landscape, requirements and reference architecture - initial version. Deliverable 2.1, The CASTOR Consortium, 14 2025.

[23] CASTOR. Architectural specification of dynamic enforcement of trust-/network-aware path establishments. Deliverable 5.1, The CASTOR Consortium, 15 2026.

[24] CASTOR. Conceptual architecture of castor trusted computing base & composable attestation model specification. Deliverable 3.1, The CASTOR Consortium, 15 2026.

[25] CASTOR. Trusted path establishment building blocks, optimization engine & crypto structures for trusted data sharing. Deliverable 4.2, The CASTOR Consortium, 18 2026.

[26] J. Chávez-Saab, F. Rodríguez-Henríquez, and M. Tibouchi. SwiftEC: Shallue-van de woestijne indifferentiable function to elliptic curves - faster indifferentiable hashing to elliptic curves. In *Advances in Cryptology (ASIACRYPT)*, LNCS 13791, pages 63–92. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-031-22963-3_3.

[27] Yanbo Chen and Yunlei Zhao. Half-aggregation of schnorr signatures with tight reductions. In *European Symposium on Research in Computer Security*, pages 385–404, 2022.

[28] Peter Cibik, Patrik Dobias, Sara Ricci, Jan Hajny, Lukas Malina, Petr Jedlicka, and David Smekal. Pushing aes-256-gcm to limits: Design, implementation and real fpga tests. In *Applied Cryptography and Network Security Workshops: ACNS 2024 Satellite Workshops*, pages 303–318, Berlin, Heidelberg, 2024. Springer-Verlag.

[29] Fernando J Corbató and Victor A Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, 1965.

[30] Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, pages 185–196, 1965.

[31] Distributed Management Task Force (DMTF). SPDM over TCP Binding Specification. Specification DSP0287, Distributed Management Task Force (DMTF), July 2024.

[32] DMTF. Secured Messages using SPDM Specification. Specification DSP0277, Distributed Management Task Force, Portland, OR, October 2025.

[33] DMTF. Security Protocol and Data Model (SPDM) Specification. Specification DSP0274, Distributed Management Task Force (DMTF), October 2025.

[34] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[35] Order-Revealing Encryption. Practical order-revealing encryption with limited leakage.

[36] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C.A. Wood. Hashing to Elliptic Curves. Internet Engineering Task Force (IETF) Request for Comments (RFC) 9380, 2023. https://www.rfc-editor.org/info/RFC:9380.

[37] Marc Fischlin, Anja Lehmann, and Dominique Schröder. History-free sequential aggregate signatures. In *International conference on security and cryptography for networks*, pages 113–130, 2012.

[38] Nikolaos Fotos, Koffi Ismael Ouattara, Dimitrios S. Karas, Ioannis Krontiris, Weizhi Meng, and Thanassis Giannetsos. Actions speak louder than words: Evidence-based trust level evaluation in multi-agent systems. In Jinguang Han, Yang Xiang, Guang Cheng, Willy Susilo, and Liquan Chen, editors, *Information and Communications Security*, pages 255–273, Singapore, 2026. Springer Nature Singapore.

[39] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. Rsa-oaep is secure under the rsa assumption. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, Lecture Notes in Computer Science, pages 260–274, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[40] S.D. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *Advances in Cryptology (EUROCRYPT)*, LNCS 5479, pages 518–535. Springer-Verlag, 2009. https://doi.org/10.1007/978-3-642-01001-9_30.

[41] R.P. Gallant, R.J. Lambert, and S.A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology (CRYPTO)*, LNCS 2139, pages 190–200. Springer-Verlag, 2001. https://doi.org/10.1007/3-540-44647-8_11.

[42] Simson Garfinkel. An overview of trusted computing technology. In *Trusted Computing*, pages 29–114. 2005.

[43] Georgios Giantamidis, Stylianos Basagiannis, and Stavros Tripakis. Learning Moore machines from input–output traces. *International Journal on Software Tools for Technology Transfer (STTT)*, 23:85–104, 2021. First online 2019.

[44] A. Guillevic. A short-list of pairing-friendly curves resistant to special TNFS at the 128-bit security level. In *Public-Key Cryptography (PKC)*, LNCS 12111, pages 535–564. Springer, 2020.

[45] D. Hayashida, K. Hayasaka, and T. Teruya. Efficient final exponentiation via cyclotomic structure for pairings over families of elliptic curves. Cryptology ePrint Archive, Paper 2020/875, 2020.

[46] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, Lecture Notes in Computer Science, pages 543–571. Springer, 2016.

[47] Kevin Lewi and David J Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178, 2016.

[48] Yuan Li, Hongbing Wang, and Yunlei Zhao. Delegatable order-revealing encryption. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 134–147, 2019.

[49] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 465–485, 2006.

[50] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential aggregate signatures from trapdoor permutations. In *International conference on the theory and applications of cryptographic techniques*, pages 74–90, 2004.

[51] Gregory Neven. Efficient sequential aggregate signed data. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 52–69, 2008.

[52] Jonas Nick, Tim Ruffing, and Yannick Seurin. Musig2: Simple two-round schnorr multi-signatures. In *Annual international cryptology conference*, pages 189–221, 2021.

[53] Jaehwan Park, Hyeonbum Lee, Junbeom Hur, Jae Hong Seo, and Doowon Kim. Utra: Universal token reusability attack and token unforgeable delegatable order-revealing encryption. In *European Symposium on Research in Computer Security*, pages 321–340, 2025.

[54] PCI-SIG. TEE Device Interface Security Protocol (TDISP). Engineering change notice, PCI-SIG, August 2022. Affected Document: PCIe Base Spec 5.0, 6.0.

[55] David Pointcheval and Olivier Sanders. Short randomizable signatures. In *Cryptographers' Track at the RSA Conference*, pages 111–126, 2016.

[56] David Pointcheval and Olivier Sanders. Reassessing security of randomizable signatures. In *Cryptographers' Track at the RSA Conference*, pages 319–338, 2018.

[57] Raluca Ada Popa, Frank H Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *IEEE symposium on security and privacy*, pages 463–477, 2013.

[58] Y. Sakemi, S. Kanno, and R. Wahby. Pairing-Friendly Curves. Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-12, Internet Engineering Task Force, 2025. https://datatracker.ietf.org/doc/draft-irtf-cfrg-pairing-friendly-curves/.

[59] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252, 1989.

[60] M. Scott. A note on group membership tests for G1, G2 and GT on BLS pairing-friendly curves. Cryptology ePrint Archive, Paper 2021/1130, 2021. https://eprint.iacr.org/2021/1130.

[61] Syh-Yuan Tan, Tiong-Sik Ng, and Swee-Huay Heng. Efficient fork-free bls multi-signature scheme with incremental signing. In *International Conference on Provable Security*, pages 250–268, 2024.

[62] Trusted Computing Group. TPM 2.0 Library Specification. Technical report, Trusted Computing Group, Portland, OR, USA, October 2013.

[63] F. Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory (TIT)*, 56(1):455–461, 2010.

[64] R.S. Wahby and D. Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019(4):154–179, 2019. https://doi.org/10.13154/tches.v2019.i4.154-179.

[65] Naoto Yanai, Masahiro Mambo, and Eiji Okamoto. An ordered multisignature scheme under the cdh assumption without random oracles. In *The 16th International Conference, ISC*, pages 367–377, 2015.