D3.1
Conceptual Architecture of CASTOR Trusted Computing Base & Composable Attestation Model Specification

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:	Report
Reference Number:	HORIZON-CL3-2021-CS-01-101167904/ D3.1 / v1.0
Workpackage:	WP3
Due Date:	31st December, 2025
Actual Submission Date:	9th February, 2026

Responsible Organisation:	SURREY
Editor:	Yalan Wang, Liqun Chen
Dissemination Level:	Public
Revision:	1.0

Abstract:	Deliverable 3.1 presents the archetype design of CASTOR Trusted Computing Base equipped with appropriate TEE Device Interfaces and all envisioned internal building blocks for enabling the trusted I/O virtualization and secure message exchange with any type of underlying secure element.

Funded by the European Union

Project funded by

Swiss Confederation

UK Research and Innovation

Funded by EU's Horizon Europe programme under Grant Agreement number 101167904 (CASTOR). Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority 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	R*	
	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

Yalan Wang, Liqun Chen (SURREY)

Contributors (ordered according to beneficiary numbers)

Nikos Fotos, Sofianna Menesidou, Georgia Basayianni, Stefanos Vasileiadis, 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 provide trusted path routing in a below-zero-trust model, enabling the dynamic enforcement of routing paths that meet specific network *and trust* requirements and can flexibly adapt to changes within the network. To achieve this, CASTOR elevates trust metrics to first-class citizens in routing decisions and traffic engineering, so that changes in the trust level of participating network nodes can trigger corresponding updates of the selected paths. In deliverable D2.1 [22], the overall CASTOR architecture and high-level requirements were introduced in the context of CASTOR's envisioned real-world use cases, outlining the orchestrator-driven trusted path routing framework and composite attestation schemes enabled by the Trust Network Device Extensions (TNDE). Building on that foundation, this deliverable focuses on the conceptual architecture and technical requirements of the device-side Trusted Computing Base (TCB) instantiated on the network elements, i.e., physical routers and servers hosting virtual routers, and explains how this TCB underpins CASTOR's trust network and routing assurances.

The device-side TCB and its TNDE are motivated by the need to enable *continuous* trust assessment of all network nodes that participate in the CASTOR trust network and trusted path routing. Existing efforts such as IETF Trusted Path Routing and TPM-based attestation for routers primarily rely on static or boot-time integrity evidence to establish trusted links between nodes, capturing only an initial trust state and not the dynamic fluctuations of trust throughout runtime [11]. Similarly, existing work on runtime attestation largely targets single applications or host systems and does not address the specifics of routing elements and trusted path routing [1, 27]. In contrast, D3.1 aims at a continuously evolving global view of the trust state of each network node, allowing the CASTOR orchestrator to react dynamically to changes in trustworthiness and to update trusted paths accordingly in a below-zero-trust model. Without TCB-anchored trust extension services on each node performing secure monitoring and local trust assessments, the orchestrator would lack trustworthy, fine-grained attestation and runtime information about the state of the routing elements and could not safely use trust as a metric in path selection and enforcement.

Within this scope, D3.1 systematically recaps and refines the concepts of the Trust Network Device Interface (TNDI), its Security Protocol (TNDI-SP), and the TNDE, and then develops their architectural roles and requirements in the context of the device-side TCB. The document lays out the TNDI-SP security mechanisms and the layers of the CASTOR device-side TCB that materialise these mechanisms on each network element, including Roots of Trust, isolation layers, and the trust extension components. It positions the TNDE, and in particular the TN-DSM, as the main component on each device that runs inside the TCB and is responsible for trust management and coordination of evidence, including device and TNDI enrolment, key management, configuration and reprogrammability of trust functions, and coordination of evidence collection and export based on the TNDI-SP control and data channels. These architectural choices are driven by a routing-plane threat model and a set of engineering stories that derive concrete device-side requirements from representative attack scenarios such as BGP hijacking, control-plane manipulation, and compromise of a TNDI's routing stack or network operating system.

In particular, D3.1 identifies two primary Trust Sources that operate under the control of the TNDE: an **attestation Trust Source**, which captures configuration, integrity, and platform state rooted in the underlying hardware Root of Trust; and a **finite-state-machine (FSM) Trust Source**, which expresses and monitors behavioural properties of routing and control-plane components as FSMs to reflect how the

device actually behaves at runtime. These Trust Sources are embedded into a multi-level runtime tracing architecture, centred around the CASTOR Tracing Hub and several Trace Units, that jointly provide systematic (system-level) and networking evidence about host integrity, control-plane behaviour, and forwarding state. In that way, the device-side TCB provides the required inspection capabilities to derive the trustworthiness evidence serving the requirements captured by the threat model

The document situates the CASTOR device-side TCB and its TNDI-SP security mechanisms in the broader context of trusted computing, runtime tracing, and network attestation, including TPM-based measurements, host-level runtime monitoring, and IETF work on trusted path and trusted link establishment that typically rely on static or snapshot-style evidence. The CASTOR-specific novelty introduced here lies in continuous runtime assessment of routing devices that combines attestation-based integrity and configuration evidence with FSM-based behavioural evidence from the routing stack, and in maintaining a global, dynamically updated trust state over all TNDIs in the trust network that serves as a first-class metric for orchestrator-driven trusted path routing. This enables the network to adapt paths when the trust level of individual nodes changes, aligning trust-aware traffic engineering with a “never trust, always verify” mindset applied directly at the level of routing elements.

D3.1 does not yet provide the full technical solution; rather, it introduces the key concepts, outlines high-level solution directions, and derives technical requirements for the device-side trust concepts and TCB, including the TNDI, TNDI-SP security mechanisms, and the TNDE (with a focus on TN-DSM, Tracing Hub, and Trust Sources) as the trusted device component implementing these mechanisms. Details on the technical implementation and concrete solutions for the CASTOR device-side TCB will be explored in the follow-up deliverable D3.2, while the architectural concepts and requirements of the trust assessment logic and trusted path composition are addressed in other work packages, including D4.1 of WP4 for trust assessment and D5.1 of WP5 for path establishment and enforcement. Overall, D3.1 advances CASTOR’s goal of below-zero-trust, trusted path routing by defining the device-side trust architecture that turns continuous, TCB-grounded evidence from routing elements into a first-class metric for orchestrator-driven path selection and adaptation.

Contents

List of Figures

List of Tables

Versioning and contribution history

Version	Date	Author	Notes
v0.1	19.11.2025	Yalan Wang (SURREY), Nikos Fotos (UBITECH)	Create Template
v0.2	03.12.2025	ALL	Listing and enumeration of the first version of the Engineering Stories capturing the requirements of the CASTOR Trusted Computing Base and Trusted Network Device Extensions (Chapter 6)
v0.3	18.12.2025	Fabian Schwarz (NVIDIA), Nikos Fotos, Stefanos Vasileiadis (UBITECH)	Finalization and Description of CASTOR TCB (Chapter 3)
v0.1	06.01.2026	Fabian Schwarz (NVIDIA), Alexandru Coles (ORO), Nikos Fotos, Stefanos Vasileiadis (UBITECH), Riccardo Orizio, Stelios Basayiannis (COLLINS)	First version of the CASTOR TNDE setup and early description of the tracing capabilities to be supported (Chapter 7 & 8)
v0.4	09.01.2026	Sofianna Menesidou (UBITECH), Anotnio Skarmeta (UMU), Riccardo Orizio (COLLINS)	First version of the considered threat model (Chapter 5)
v0.5	15.01.2026	Yalan Wang, Liqun Chen (SURREY), Jamie Pont, Theo Dimitrakos (UKENT), Nikos Fotos, Thanassis Giannetsos (UBITECH)	Final description of CASTOR's attestation and state-device monitoring capabilities (Chapters 4 & 9)
v0.6	22.01.2026	Sofianna Menesidou (UBITECH), Riccardo Orizio (COLLINS)	Final version of CASTOR's detailed threat model accompanied with possible type of evidence to be monitored for the successful detection (Chapter 11)
v0.7	26.01.2026	Fabian Schwarz (NVIDIA), Alexandru Coles (ORO), Antonio Skarmeta, Pablo Martinez (UMU)	Update of the CASTOR TCB so as to be able to consider the instantiation of TNDIs (vRouters) as containers sharing the same host kernel (Chapter 3)
v0.8	28.01.2026	Fabian Schwarz (NVIDIA), Nikos Fotos, Stefanos Vasileiadis (UBITECH)	Final description of CASTOR tracing capabilities clarifying the trade-off on the level of granularity and visibility (interpretation of monitored evidence towards context-aware observations for trust assessment) between the offered functionalities - kernel extensions vs. eBPFS (Chapter 8)
v0.85	30.01.2026	Yalan Wang, Liqun Chen (SURREY), Nikos Fotos, Stefanos Vasileiadis, Thanassis Giannetsos (UBITECH)	Final version of CASTOR's functional requirements on its crypto agility layer exposing composite attestation and DORE services (Chapter 9)
v0.86	02.02.2026	ALL	Final version of CASTOR Trust Extensions vocabulary
v0.9	05.02.2026	Antonio Skarmeta (UMU), Alexandru Coles (ORO)	Second round of review from the CASTOR consortium
v0.95	09.02.2026	Nikos Fotos, Sofianna Menesidou, Georgia Basayianni, Thanassis Giannetsos (UBITECH), Fabian Schwarz (NVIDIA), Yalan Wang (SURREY)	Final refinements and polishing related to the description of the threat model and cases of interest in CASTOR (Chapter 11)
v1.0	09.02.2026	Daphne Galani (UBITEH)	Submission of the deliverable

Chapter 1

Introduction

1.1 Towards Dynamic Trust Assessment in the Compute Continuum

The compute continuum encompasses a heterogeneous and highly dynamic set of environments, ranging from cloud data centres and edge platforms to programmable network devices operating across multiple administrative domains. In such settings, to achieve the vision of CASTOR and enable **trusted service graph chains across the compute continuum** requires **embedding explicit trust guarantees directly into the routing plane**. Traditional routing architectures rely on largely static trust assumptions, where network elements are implicitly trusted once deployed and configured. CASTOR departs from this approach by focusing on runtime trust assessment. It recognises that the trustworthiness of routing elements and network paths is not fixed, but continuously affected by software updates, configuration changes, workload dynamics, and adversarial behaviour. As a result, trusted path routing requires mechanisms that can assess trust dynamically, based on the current operational state of network components rather than static assumptions. CASTOR targets routing infrastructures operating at the edge and far edge, where traffic is steered across heterogeneous and potentially multi-domain environments of the continuum. In this context, **routing** is not merely a forwarding function but a **critical decision-making component** that directly impacts service trustworthiness.

CASTOR builds upon **modern routing paradigms** such as **segment routing** and **source routing**, which provide flexibility through self-organisation and self-configuration mechanisms (e.g., Flex-Algo with affinity constraints). Abstractly, these routing protocols expose two core functionalities: (i) route discovery, which identifies candidate paths based on network-level objectives, and (ii) route maintenance, which adapts paths in response to topology or performance changes. While these mechanisms are highly optimised for efficiency and scalability, they **do not natively account for trust-related properties** of routing elements or paths.

Introducing trust into routing therefore requires revisiting the design of routing elements themselves. CASTOR identifies the **need for routing-element-level extensions** that enable the generation of verifiable runtime evidence. However, all this evidence needs to be shared without affecting the performance of the network. In this context, to achieve an optimal interplay between operational routing flows (e.g., without impacting the bandwidth) and secure interactions for exchanging verifiable trust evidence, CASTOR provides all the necessary cryptographic primitives, security mechanisms, and trust extensions that operate alongside existing routing protocols. These mechanisms enable a relying party (e.g., the CASTOR Orchestrator) to securely collect, aggregate, and verify evidence produced by routing elements, and to establish trust decisions without disrupting normal routing operations.

As aforementioned, dynamic trust assessment fundamentally depends on the ability to collect, protect, and disseminate this runtime evidence in a verifiable and scalable manner. The construction of such verifiable evidence, and its secure use in trust decisions, constitutes a core innovation of the CASTOR. **CASTOR equips each routing element with a minimal Trusted Computing Base (TCB)**, anchored in hardware Roots of Trust, that enables secure introspection of both system- and network-level behaviour. This allows routing elements to actively participate in a distributed trust network by producing evidence that reflects their current configuration and runtime state.

Enabling this runtime evidence monitoring, and unlocking the dynamic trust assessment model described in Chapter 4, requires two key capabilities: the identification of appropriate **trust sources** and the provision of robust **introspection mechanisms**. CASTOR addresses these requirements by defining (i) a detailed architectural breakdown of the CASTOR TCB instantiated on each router to support runtime evidence collection, and (ii) the cryptographic primitives necessary to establish authenticated and encrypted communication channels for the secure exchange of trust-related evidence. These mechanisms ensure that evidence remains authentic, confidential, and tamper-resistant, even in adversarial environments.

In summary, D3.1 specifies the conceptual architecture of the CASTOR device-side TCB, including trust sources and introspection capabilities, and defines the architectural and cryptographic requirements that enable secure runtime evidence sharing form the routing infrastructure perspective. More details, from the orchestrator perspective, will be provided in the context of deliverable D5.1. Together, these elements provide the basis for the protocol design and detailed mechanisms that will be further developed in D3.2.

1.2 Scope and Purpose

This deliverable, D3.1, defines the core architectural foundations of CASTOR TCB for enabling runtime trust assessment in routing infrastructures. Its primary purpose is to specify the architectural artefacts that are instantiated on routing elements and that enable the secure generation, protection, and exposure of verifiable runtime evidence used to assess router trustworthiness prior to path creation.

In particular, D3.1 focuses on the conceptual architecture of the device-side TCB, including the identification of trust sources, introspection mechanisms, and their role in supporting dynamic trust assessment within the routing plane. It also defines the cryptographic primitives needed to securely exchange trust-related evidence across domains. The deliverable captures these requirements through a set of engineering stories.

1.3 Relation to other WPs and Deliverables

D3.1 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, and provides essential input to the trust evaluation mechanisms defined in D4.1. The threat models and engineering stories discussed in this document inform both the design of runtime monitoring mechanisms and the orchestration strategies developed in later WPs. Figure 1.1 depicts the relation of D3.1 with other WPs and deliverables.

1.4 Deliverable Structure

The remainder of this document is structured as follows.

Chapter 2 introduces the core concepts of trusted and confidential that underpin the CASTOR architecture. It reviews key concepts such as Trusted Computing Bases, Roots of Trust, and Trusted Execution Environments, and surveys relevant state-of-the-art technologies.

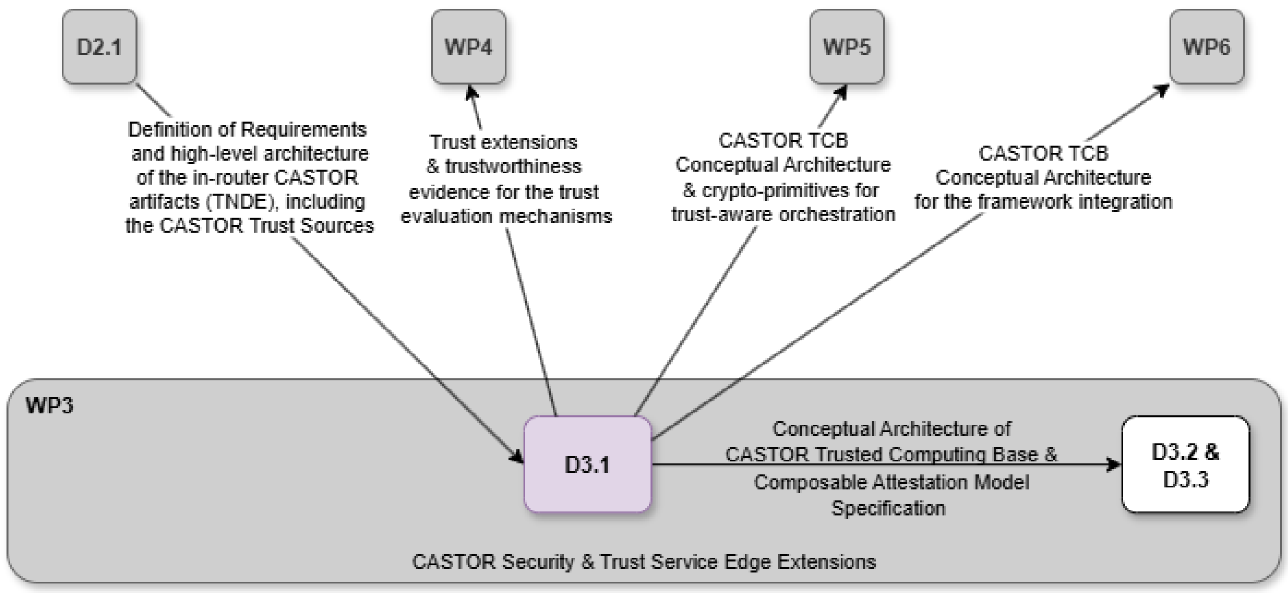

D2.1
WP4
WP5
WP6
Definition of Requirements and high-level architecture of the in-router CASTOR artifacts (TNDE), including the CASTOR Trust Sources
Trust extensions & trustworthiness evidence for the trust evaluation mechanisms
CASTOR TCB Conceptual Architecture & crypto-primitives for trust-aware orchestration
CASTOR TCB Conceptual Architecture for the framework integration
WP3
D3.1
Conceptual Architecture of CASTOR Trusted Computing Base & Composable Attestation Model Specification
D3.2 & D3.3
CASTOR Security & Trust Service Edge Extensions

Figure 1.1: Relation of D3.1 with other WPs and Deliverables

Chapter 3 presents the core contribution of this deliverable: the conceptual architecture of the CASTOR device-side Trusted Computing Base. It defines the scope and boundaries of the TCB, introduces the Trust Network Device Interface (TNDI) abstraction and the TNDI Security Protocol (TNDI-SP), and describes the internal components of the Trust Network Device Extensions (TNDE). The chapter also discusses different deployment models, including virtualised and physical routers, and outlines the design trade-offs associated with TCB minimization and reprogrammability.

Chapter 4 examines the trust sources used by CASTOR to assess network trustworthiness, distinguishing between systematic (system-level) and networking evidence. It discusses how configuration state, behavioural models, and runtime observations can be leveraged as trustworthiness signals, and relates these sources to CASTOR's overall threat model and trust assessment goals.

Chapter 5 provides an overview of threat modelling in the routing plane, identifying relevant attack surfaces and adversarial capabilities. This chapter contextualizes the security assumptions made throughout the deliverable and motivates the need for continuous, evidence-based trust assessment rather than static or implicit trust models.

Chapter 6 introduces a set of engineering stories that illustrate how CASTOR's trust management features operate across different phases of network operation, including proactive, preparedness, and reactive stages. These scenarios serve to connect the architectural concepts to concrete operational workflows and highlight key security requirements.

Chapter 7 focuses on the secure routing plane management, providing an overview of the TNDE and the TNDIs. It details the interaction between the TNDE, hardware Roots of Trust, and the CASTOR orchestrator, and describes device enrolment, attestation, configuration management, and secure communication mechanisms within the trust network.

Chapter 8 explores the design space for multi-level runtime tracing and observability in CASTOR. It presents the tracing architecture, Trace Units, and evidence richness considerations, analysing the trade-offs between visibility, security, and performance when collecting runtime evidence from network devices.

Chapter 9 introduces CASTOR's network attestation mechanisms, including composite and layered attestation models. It describes the cryptographic building blocks used to aggregate and verify attestation evidence across network paths, enabling trust assertions that span multiple devices and administrative domains.

Chapter 10 provides a detailed threat model and analysis of considered attacks against the routing plane.

Chapter 11 elaborates on the evidence collection mechanisms used for runtime threat detection, detailing both system-level and network-level evidence and their role in detecting compromise or misbehaviour.

Chapter 12 concludes the deliverable.

Chapter 2

Introduction to Trusted and Confidential Computing

Chapter 2 provides an overview of one of the core technologies where CASTOR's security features are anchored - that of a **trusted execution foundation** revolving around a (SW- or HW-based) secure element[1] for achieving **system-level security** and enabling the **efficient, runtime trust assessment** of the routing plane, converting all router elements into their (HW-backed) trusted equivalents. More specifically, the adoption of TEEs equip routers with the necessary trusted computing capabilities to provide guarantees that application workloads (application network traffic) traverses through nodes and paths that can exhibit the Required Trust Level (RTL) [20, 37] - as defined by the system model capturing the open, dynamic, and untrusted operating environments of today's "*cognitive, collaborating*" computing environments. In the CASTOR architecture [22], TEEs are used to verify not only remote components (e.g., adjacent routers) but also internal sub-components comprising the composite routing device. This translates to the enforcement of the below-zero-trust principle of "*never trust, always verify*" prior to establishing trust relationships: In today's network environments, it is very challenging to list, resolve and monitor the trust dependencies of our distributed infrastructures (**lack of trust dependency scoping**), and we have almost no control of what is the current trust level of a device. This is what CASTOR's fundamentally addresses through the provision of a **custom Trusted Computing Base (TCB - through the establishment of a a set of Trusted Network Device Interfaces (TNDIs) and Security Protocol (TNDISP) - cf. Section 3.2) enabling hardware-level isolation and enforcement of varying-level of trust profiles while improving engineering efficiency and agility so that it can be kept agnostic to the type and vendor of the underlying routing element.**

Through the adoption of Trusted Computing principles, CASTOR aims to establish a chain of trust that ranges from the infrastructure layer (i.e., the network topology) to the orchestration layer and eventually the application provider. The chain of trust originates in the router's hardware TEE as the Root of Trust and is extended through verification of the software layers forming the in-router Trusted Computing Base (TCB), as detailed in Chapter 3. This stack can then produce runtime trustworthiness evidence of critical network functions to CASTOR's Trust Assessment Framework (see D4.1 [20]). Eventually, this culminates in the realization of Actual Trust Level values (ATL) on key propositions that can be used to provision trust-aware traffic engineering policies.

One core driving factor behind CASTOR's overarching architecture (as described in D2.1 [22]) is to enable trusted path routing over one or more Autonomous Systems (ASes), potentially managed by different domain operators. The goal of such zero-trust architectures is to minimize the required trust and allow stakeholders to validate that their security requirements are met. The concept of trusted computing

[1] In CASTOR, as also elaborated in Chapter 3, the routing plance is elevated to a trusted digital infrastructure through the adoption and integration of Trusted Execution Environments (TEEs) as the foundational pillar for the offered system-level security and verifiable monitoring of trust-related evidence [22]

allows the provision of strong guarantees towards the proper management and behaviour of network functions. This chapter introduces key concepts of trusted computing that we will later use when outlining the CASTOR architecture and surveys the state of the art.

2.1 Basic Concepts of Trusted Computing

2.1.1 What is Trusted Computing Base (TCB)?

Secure communication protocols are continuously evolving, providing robust confidentiality and integrity guarantees across multiple layers of the network stack; from application-level protections such as HTTPS, to network- and link-layer mechanisms like VPNs, IPsec, and MACsec. However, as highlighted in the IETF's Trusted Path Routing specification [11], end-users may need stronger guarantees about the trustworthiness of the network elements that serve a highly-confidential workload. This introduces the need for the network elements to securely provide appraisals of their trustworthiness, allowing the formation of "Trusted Topologies". Through CASTOR's overarching framework [22], the continuous introspection of critical network functions (see Chapter 11) and the systematic trust quantification through the CASTOR Trust Assessment Framework (see D4.1 [20]) allow the establishment of such Trusted Topologies, as envisioned by the Trusted Path Routing paradigm. Going beyond that, CASTOR elevates such trust-related insights into Service-Level Objectives (SLOs) as it allows the realization of traffic engineering policies that can satisfy both network- and trust-related needs. In the context of CASTOR use cases [22], end users often rely on the transmission of critical information (e.g., radar application data in the context of aerospace surveillance systems or critical notification messages as part of a connected cooperative connected and automated mobility scenario) through communication channels that meet certain trust requirements (i.e., integrity, confidentiality, availability). In order for a domain operator to provide high level of assurances on an established application service, CASTOR needs to provide the mechanisms that ensure that certain critical network functions always behave as expected by the users (or else fail gracefully). These mechanisms form the Trusted Computing Base (TCB) of each network element.

In general, a TCB may encompass communication, storage, and computation components. Software systems often employ encryption to protect both data in transit and data at rest. For example, data traversing network channels is encrypted so that an adversary intercepting the traffic cannot learn sensitive information. Likewise, data is typically encrypted before being written to disk, ensuring that even if attackers gain access to stored files, the underlying secrets remain protected.

Formally, we define the **TCB for a given requirement of a service or function** as *the set of hardware, firmware, services, and software components that must operate correctly to ensure that this requirement is satisfied.*

2.1.2 Materializing a Re-Programmable TCB

An immediate implication of this definition is that different requirements for the same service can result in different TCBs. For example, the TCB for confidentiality in the establishment of IPsec tunnels primarily includes the components responsible for end-to-end encryption, whereas the TCB for availability of that same service may instead include mechanisms that enable failover at run time without interrupting operation.

TCBs are initially defined at design time. However, robust TCBs should also support reprogrammability, allowing the set of protected components to be adjusted according to the Required Trust Level (RTL) that each network element must achieve to support a given service. Traditional static TCBs, such as those based on a Trusted Platform Module (TPM), provide a fixed set of functions for integrity protection via attestation. In contrast, CASTOR envisions to implement an extensible TCB using modern TEEs, which allow new components to be dynamically configured at run time.

The TCB can grow very large and larger components are known to contain more bugs [69]. Consequently, the risk of defects increases- since a single bug in the TCB can compromise the requirement it is meant to protect- and poses a significant threat to critical services. To avoid that, designers aim to **minimize the TCB**, particularly for security-critical services. For example, a small hardware security module used for key storage typically presents a lower risk of compromise than a hypervisor-based virtualization system, which may result in a substantially larger TCB.

CASTOR's advantage lies in its streamlined TCB: It precisely limits the trust boundary to only those critical components (primarily the tracing hub and the key management layer (Figure 3.1) providing the verifiability properties required on the trustworthiness evidence to be provided to the Trust Assessment Framework (TAF)) and necessary runtime libraries to run in isolation (inside an enclave), excluding large and complex components such as the operating system, hypervisor, and even BIOS from the root-of-trust. This greatly reduces potential attack surfaces, making security audits more focused and feasible.

2.1.3 What is a Root of Trust?

The Root of Trust (RoT) represents the minimal set of security guarantees—typically provided by built-in hardware capabilities—required to protect a TCB. The design goal is to safeguard the (usually software-based) TCB under the assumption that a well-defined set of security functions is offered by the RoT. The RoT itself can vary in size: it may be a complete microcontroller within a Hardware Security Module (HSM) or a smaller component, such as an encryption engine with preloaded keys. One key advantage of a hardware RoT is that it is generally immutable by its owner, allowing it to serve as a foundation for establishing trust with remote users. In this scenario, a remote user can be confident that their application running in an untrusted environment executes as intended and that sensitive application data is protected. Hardware-based TEEs support this by providing remote attestation capabilities. Section 3.3.1 discusses the core requirements that the CASTOR RoT needs to exhibit.

2.1.4 What is a Trusted Execution Environment (TEE)

A Trusted Execution Environment (TEE) provides a secure, isolated environment for executing critical operations with high assurance. It achieves this by separating the host system into a “trusted” and an “untrusted” domain, ensuring that code and data within the TEE are protected from unauthorized access, disclosure, or tampering. Applications running inside a TEE operate within a confined domain that is opaque to other software on the system. In this way, TEEs can serve as a RoT for the secure execution of the CASTOR TCB, so as to securely monitor the required network functions of a router element.

One goal of a TEE is to remove the untrusted and large operating system from the TCB. Even after minimizing the TCB and separating the software TCB from the underlying hardware RoT, the TCB often still includes the operating system and much of the hardware, such as memory and storage. To further enhance security, designers minimize the trusted hardware and remove the operating system from the TCB, leaving only the trusted application and the underlying hardware.

In the context of CASTOR, the TCB provides the necessary capabilities for the runtime monitoring of network element characteristics (e.g., device configuration, integrity of the loaded software stack, behavioural assurance of critical network functions), constituting security claims, that need to be shared between entities that wish to establish a trust relationship. This can be done by the Service Orchestrator in order for the router to be managed remotely within a topology, or an adjacent router element requesting for attestation evidence (e.g., in the form of Stamped Passports as per [11]) before establishing any link connection. Validation properties may range from static properties such as integrity measurements of the router software stack, enabling the generation of static evidence on its correct configuration, to dynamic properties with respect to the fact that the router exhibits the expected behaviour when performing critical network operations (e.g., updating forwarding tables, or BGP configuration). All these security claims, serving as trustworthiness evidence for the node-level trust appraisal, need to be provided in a verifiable manner so as to not compromise the trust assessment process. Therefore, as presented in the Engineering Stories of Chapter 6, it is imperative that all processes involved in establishing these security guarantees are part of the network element's TCB (see Engineering Story I) and that the appropriate authenticated and authorized communication channels are established (see Engineering Story V).

2.2 Existing Roots of Trust & Security Functions

In the following, we will have a look at some existing technologies implementing RoTs and their respective security functions. We will focus on RoTs specifically relevant in the context of CASTOR. As we will explore in Chapter 3 and Chapter 7, CASTOR relies on RoTs in the network devices to establish trust in each router node by forming the CASTOR device-side TCBs and enabling the CASTOR orchestrator to remotely attest them.

2.2.1 Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) is typically a dedicated security chip or hardware component that complies with the TPM specification defined by the Trusted Computing Group (TCG) industry consortium [43]. A TPM implements several roots of trust, providing important security functionalities to serve as a platform hardware RoT:

RoT for measurement (RTM) A TPM can measure (i.e., hash) loaded platform code and securely store the resulting hash values in internal memory in so-called Platform Configuration Registers (PCRs). PCRs are stored in TPM-internal memory and operate in an extend-only mode via cumulative hashing; most of them are non-resettable during a boot cycle. This allows a TPM to securely record the system's boot chain (e.g., firmware, bootloader, and OS) in a tamper-evident way and even to record the loading of post-boot software such as loadable kernel modules or user services.

RoT for reporting/identity (RTR/RTI) A TPM contains a unique Endorsement Key (EK) identifying the TPM and one or multiple Attestation Keys (AKs) bound to that EK. This enables a TPM to (i) remotely prove the identity of a genuine TPM instance to a remote entity and (ii) support remote attestation. For remote attestation, the TPM signs the collected measurements (PCR values) with an AK bound to the EK. A remote verifier can then verify that the measurements originate from a genuine TPM by checking the TPM's EK certificate and the binding of the used AK to the EK, before checking the measurements against its verification policy or reference values.

RoT for storage (RTS) A TPM provides an asymmetric Storage Root Key (SRK) whose private part (or a seed for deriving it) is securely stored inside the TPM's non-volatile storage. The TPM can create a key hierarchy (symmetric or asymmetric) rooted in the SRK to support secure storage of keys and data on external storage. To prevent rollback attacks, TPMs provide monotonic counters that can be included in the cryptographic protection of stored keys/data or used to implement virtual monotonic counters on top of them.

Overall, a TPM can serve as a platform's hardware RoT, providing measurement and attestation capabilities for protecting the boot process, as well as secure key management and storage functions for platform and user services. In addition to hardware TPMs, there are also firmware TPM (fTPM) and virtualized

TPM (vTPM) implementations based on different forms of isolated execution environments, e.g., those provided by hypervisors or TEEs [81, 10].

2.2.2 Intel Software Guard Extensions (SGX)

Intel SGX is a set of CPU extensions that enables the creation of hardware-protected trusted execution environments (TEEs), called enclaves, within a process address space [26]. In contrast to a TPM-formed hardware RoT, which targets an entire platform including its boot chain, Intel SGX provides a RoT for the isolated execution of user-space services. Specifically, Intel SGX provides the following security functions / RoTs:

RoT for isolated execution SGX enclaves run application code and store data in a protected region of memory that is encrypted and integrity-protected by the processor, i.e., providing confidentiality and integrity guarantees. The CPU enforces that only the enclave itself can access its memory, even if higher-privileged software such as the operating system, hypervisor, or system BIOS/UEFI code is compromised.

RoT for measurement of enclaves When an enclave is created, the CPU cryptographically measures (hashes) the enclave's initial code and static data. The measurement is securely stored in protected memory only accessible by the CPU to protect it against tampering. This measurement (often referred to as *MRENCLAVE*) serves as an enclave identity precisely capturing the software the enclave runs.

RoT for attestation of enclaves SGX supports both local and remote attestation mechanisms to prove the existence and identity of an enclave to another party. For local attestation, the CPU creates an attestation report including the enclave measurement and associated metadata (e.g., enclave signer) and protects it with a MAC computed using a report key derived for the target (verifying) enclave. For remote attestation, the attestation report is transformed into a quote and signed with a platform-specific attestation key whose public part is linked, via a certificate chain rooted at Intel, to the underlying SGX platform and its TCB level. That way, a remote verifier can first check the quote signature and the binding of the attestation key to a genuine Intel SGX platform, before checking the included measurements and metadata together with the associated TCB information (e.g., microcode version) to decide whether to accept the enclave.

RoT for sealing of enclave data SGX provides sealing capabilities that allow an enclave to encrypt data in such a way that only the same enclave (or a defined group of enclaves) on the same platform can later decrypt it. This enables secure storage of enclave data on external storage. Note that Intel SGX does not have built-in, generally available non-volatile monotonic counters for rollback protection (in contrast to TPMs), but SGX platform services can provide monotonic counters or similar mechanisms to enclaves on some systems.

Overall, Intel SGX offers a CPU-based RoT focused on protecting selected user-space application components rather than the full system state. It complements platform-level RoTs such as TPMs by enabling fine-grained, enclave-level isolation, attestation, and secure storage (sealing) for sensitive workloads.

2.2.3 Others

In the following, we provide a brief overview of other technologies providing RoTs. However, note that in the rest of the document, we will mainly be focusing on TPMs and Intel SGX in the context of CASTOR's device-side TCB (see Chapter 3 and Chapter 7).

2.2.3.0.1 TEE Virtual Machines Platform security extensions like AMD SEV-SNP, Intel TDX, and Arm CCA provide TEE virtual machines (TVMs). They provide similar RoTs and security functionalities to Intel SGX but at a VM level rather than a user-space application level. That is, they provide isolated execution, measurement, and attestation functionalities. While SEV-SNP also provides key-derivation functionality to support data sealing to storage, Intel TDX and Arm CCA currently do not have built-in functionalities for that and rely on higher-level services or protocols.

2.2.3.0.2 Device Identifier Composition Engine (DICE) The DICE specifications of TCG describe concepts for a RoT that can be used on lightweight devices instead of a TPM, but can also be used on platforms that include a TPM. DICE provides a RoT for measurement (RTM) and reporting/identity (RTR/RTI). DICE defines a measured boot process in layers, where code from each layer is measured (hashed) and used to derive keys (especially attestation keys) passed to and only accessible by that layer. DICE roots its identity and other keys in a unique device secret. DPE (DICE Protection Environment) extends DICE with an isolated execution environment for managing and using the DICE secrets, decreasing the TCB size. In addition, it explicitly defines the derivation of sealing keys for a RoT for sealing/storage.

2.2.3.0.3 Arm TrustZone The Arm TrustZone security architecture partitions the platform into a secure and a non-secure world. That way, Arm TrustZone forms a trusted execution environment (TEE) for security-critical software which, together with secure boot functionality, can serve as a basis for RoT implementations. For example, Arm TrustZone can be used to host a firmware TPM (fTPM) or serve as a DICE Data Protection Environment (DPE).

Chapter 3

CASTOR Device-side Trusted Computing Base

In the context of CASTOR's trusted execution architecture, which seeks to augment the routing plane with trusted path routing capabilities (engrained trust extensions as part of the verifiable enforcement of routing policies), understanding the boundaries and formal definition of the Trusted Computing Base (TCB) is foundational. Recall that one of **CASTOR's main design principles is the minimization of the provided TCB compressing the trust boundary and complexity to the extreme** including only the necessary code for supporting the trust elevation of CASTOR-equipped network domains and minimized runtime dependencies, achieving theoretically maximum security.

From a security-engineering perspective, the TCB is not merely a conceptual subset of components. It is the minimal set of hardware, software, firmware, and cryptographic mechanisms whose correct operation is both necessary and sufficient to enforce the system's security properties. Any component outside this boundary must be treated as potentially untrusted, adversarial, or faulty. Consequently, CASTOR's assurance model must tightly constrain the TCB to reduce its attack surface and limit the number of components that must be trusted. CASTOR's innovation lies in the integration and extension of TEE technologies featuring both native compatibility (allowing the seamless integration with legacy trusted execution environments) but also the adoption of LibOS solutions to simplify the migration of existing applications. Formally, we define the TCB as follows:

> The TCB for a given requirement of a service or function is the distinct set of hardware, firmware, services, and software components that are **required to function correctly** in order to guarantee that the specific requirement is met.

This definition highlights an important nuance: the TCB is not the set of all components that influence a service, but the minimal set whose correctness is indispensable for maintaining a specific security or reliability property. If any component within this set fails—through compromise, misconfiguration, or latent defect—the corresponding security guarantee collapses. The precise delineation of this boundary is therefore a first-class security artifact.

There exists a fundamental tension between functionality and assurance: a larger TCB increases both complexity and risk. This relationship can be expressed as:

$$Risk_{defect} \propto Size(Component) \tag{3.1}$$

which reflects the empirical observation that larger components generally accumulate more state, interfaces, and interactions, all of which correlate with defect density and vulnerability likelihood. Because the TCB is the set of components that must never fail for security to hold, any expansion of this set magnifies systemic fragility.

3.1 CASTOR Trusted Computing Base (TCB) - Design Principles

In critical infrastructure environments, comprising elements managing critical services and application data workloads (such as in the case of inter- and intra-domain service-graph-chain establishment), the risk associated with large, complex TCBs becomes too high considering their core architectural principle which necessitates that such computing bases should be treated as hierarchies of "*functional blocks/layers of distinct privileges*" rather than monolithic entities. This, in turn, extends the threat model to be considered, capturing all the (trust) dependencies and vulnerabilities stemming from such hierarchical environments: It is usually only the firmware layer that can inherently be treated as trusted whereas as we are moving towards the outer (software) layers these trust assumptions are weakened and require a continuous attestation capability of their trust boundaries.

Attestation context has several dimensions. Environment attestation considers trustworthiness properties of the environments that protect and control the applications, data, and functionality that require operational integrity. Key attestation considers trustworthiness properties of the keys that are used when establishing secure protocols and securing distributed systems. Artifact attestation, also referred to as artifact provenance considers the origin and evolution of the components used to construct computing systems. These three dimensions of attestation context work together to reinforce attestation veracity in a security layering taxonomy, as the one that need to be supported by CASTOR TCB.

Computing environments can be structurally complex. Consider, for instance, the routing elements responsible for enforcing and maintaining *stable network connectivity* where they consist of multiple components (memory, CPU, storage, networking, firmware, software), and computational elements (including control-plane routing functionalities for establishing network topologies as well as data-plane routing processes for actually managing network traffic passed from the service layer) can be linked and composed to form computational pipelines, arrays, and networks. Not every computational element is expected to be capable of attestation and attestation capable elements might not be capable of attesting to every computing element that interacts with the computing environment. The attestation framework anticipates use of information modelling techniques that describe computing environment structure so that verification operations might rely on the information model as an interoperable way to navigate structural complexity.

An attestation capability itself is a computing environment. The act of monitoring trustworthiness attributes, collecting them into an interoperable format, integrity protecting, authenticating, and conveying them to the Global Trust Assessment Framework (TAF) [20] employs a computing environment - one that is separate from the one being attested. The trustworthiness of the attestation capability needs to also be considered when structuring the TCB. It should be possible for a verifier to understand the trustworthiness properties of the attestation capability for any set of assertions of an attestation flow. The attestation framework anticipates trust properties that depend on other trusted environments and the need for a root of trust that serves as the termination point. Ultimately, a portion of the computing environment's trustworthiness is established via non-automated means. For example, design reviews, manufacturing process audits, and physical security. For this reason, a trustworthy attestation mechanism depends on trustworthy manufacturing and supply chain practices.

Considering all the above, engineering CASTOR's TCB, one has to carefully design the minimal set of internal functional modules (especially for key management as well as the exposure of attestation capabilities) and monitoring, introspection tools allowing us to compress the trust boundary to the extreme so as to reduce the complexity and enable security compliance and auditability that can be proven through HW-provisioned guarantees. This essentially translates to the adoption of a tiered security model where CASTOR's TCB is architected to be minimal, static and tightly verified exposing narrow interfaces to the outside world - upper software layers responsible for the trustworthy management of traffic engineering policies. As will be detailed in the following sections, CASTOR will firmly choose the native development of its underlying TCB based on the following core considerations:

- **Minimizing Trusted Computing Base**: CASTOR focuses on reducing the number of elements that comprise the TCB to only those that are strictly necessary to *measure*, *store* and *report* the router's security state. This excludes complex components with large codebases such as the operating system, hypervisor (i.e., in the context of virtual routers on top of commodity infrastructure) and even BIOS from the root of trust.

- **Enabling Native Remote Attestation capabilities**: CASTOR aims at providing robust hardware-enabled remote attestation capabilities that provide fine-grained trustworthiness evidence (e.g., attestation reports) regarding the configuration [38, 29] and operational correctness [28] of a node (extending the concept of layered attestation, where the integrity of each system layer is dependent to integrity of the previous layer (Section 3.4.2.0.1)), a link or an entire path [33, 72]. This allows remote verifiers to derive trustworthiness appraisals that can be leveraged as part of the CASTOR Trust Assessment Framework throughout the lifecycle of the network topology.

- **Optimizing Cryptography Operators**: The CASTOR TCB exposes the hardware-accelerated cryptographic primitives that provide high-performance, low-latency operations. This is achieved while ensuring the confidentiality and integrity of the computation without assuming any trust to the underlying operating system.

- **Supporting Crypto Agility**: CASTOR's TCB designs are agnostic to the underlying root of trust. In principle, the main characteristics of the CASTOR TCB can be developed in different hardware RoT elements (see Section 7.1.1) and different instantiations of the router software stack (i.e., either deployed as containerized applications in commodity infrastructure or running as typical hardware equipment). Eventually, this increases deployment flexibility and portability, reducing the risk of vendor lock-in.

- **Fine-Grained Control over Enclave Behaviour**: Any components that need to run in isolation should be accompanied with a rich set of interfaces for allowing the complete control of their internal workings including memory layout planning, selection of key sealing strategies (and strict binding with the root-of-trust platform key), customization of remote attestation processes, etc. This fine-grained control is crucial for avoiding potential black box behaviours ensuring transparency, controllability, and auditability of high-risk core business logic - considering the high degree of fragmentation through the existence of a multitude of routing vendors.

- **Building Core Infrastructure that can Evolve Long-Term**: Considering the possible constant fluctuation of the trust level of a routing element which necessitates the easy and seamless configuration updates (fitting to various infrastructure environments), it should be the case that attestation mechanisms should allow the verifiable update (during runtime) of the upper layers of the underlying TCB without disrupting the operational profile of the node; i.e., without the need to onboard the routing element again so as to be issued with a new set of crypto primitives. This is an essential element for allowing CASTOR to cope within below-zero-trust architectures.

Within this model, routers do not become "trusted" simply by participating in forwarding; rather, they are trusted only as a minimal, well-defined TCB on each device can securely collect and protect runtime evidence about the router's behaviour and configuration, derive a trust assessment from this evidence, and expose both the assessment and the underlying evidence through authenticated interfaces. This information is exported to the CASTOR Orchestration Layer (through extended network exposure functions fine-tuned to also capture the secure communication of trust-telemetry data [21]), which uses it to construct a global, up-to-date view of the network's trust state.

This strict separation between the CASTOR TCB of a network device and all non-TCB routing and network services effectively eliminates an entire class of attacks in which adversaries exploit compromised intermediate routers that misrepresent or conceal their security state. Trusted Path Routing, grounded in a rigorously minimized and attested router TCB, enables continuous assessment, enforcement, and monitoring of path trustworthiness, rather than relying solely on static configuration or implicit trust in the infrastructure.

Ultimately, CASTOR guarantees that only a minimal, verifiable, and attested chain of trust—comprising CASTOR device-side TCBs and the orchestrator's decision logic (based on the collected runtime evidence)—can influence the routing trust state. Failures outside these TCBs must be fail-safe with respect to trust (for example, causing a router to be treated as untrusted and excluded from trusted paths), preventing misbehaving nodes from silently degrading or subverting the trusted path. This approach moves beyond conventional measures such as encryption, which protect confidentiality but not the trustworthiness of router nodes, to enforce true trusted path routing.

3.2 CASTOR's TCB on the Network Devices

In this section, we introduce the concepts and components of CASTOR's TCB on each network device. We lay out the relationship between the TNDIs and TNDI-SP that enable the secure participation of router nodes in CASTOR's trust network and trusted path routing service. Then, we provide an overview of CASTOR's TCB components and explain how each of them contributes to the implementation of the TNDI-SP mechanisms. That way, the CASTOR TCB enables the CASTOR orchestrator to establish trust into the router nodes and receive trustworthiness evidence to construct a global view of the network's trust state for calculating and enforcing trusted routing paths. In this context, we discuss how the concept of TNDIs and a device-side TCB can be instantiated in different ways, including a server with multiple virtual routers (vRouters)—our main setting for this report—or a physical router device. More information on the requirements and explored directions of the TNDIs, TNDI-SP, and device-side TCB are provided in Chapter 7.

3.2.1 Trust Network Device Interface (TNDI) and Security Protocol (TNDISP)

CASTOR seeks to provide the service of trusted path routing, i.e., CASTOR aims to establish a global, up-to-date view of the trust state of a network and to leverage network and trustworthiness telemetry information to make routing decisions. In this way, CASTOR can enforce trusted network paths that take into account not only network metrics but also trust information on routing nodes and their links. To achieve this, the CASTOR orchestrator must be able to securely onboard routing nodes into a trust network and establish a trusted monitoring framework on each node that enables continuous, evidence-based trust assessment of the network. That is, the CASTOR orchestrator needs to form a distributed trust network of routing nodes.

CASTOR's requirement to securely assign resources to a trusted domain (here: onboard network nodes) and monitor their security state has similarities to the PCI-SIG concept of TDIs (TEE Device Interfaces) and TDISP (TEE Device Interface Security Protocol). TDISP defines a protocol and mechanisms to securely assign TDI instances on PCIe/CXL devices to a trusted execution environment (TEE), extending the TEE's trust domain towards the device. In this context, a TDI is the unit of assignment—a slice of the device that is associated with a TDISP security state and on which security mechanisms are enforced to guarantee secure assignment and operation towards the TEE. This includes configuring and locking the TDI configuration, providing a secure channel between the TEE and the TDI, as well as monitoring the security-relevant state of the TDI for changes [67].

In CASTOR, we extend the ideas of TDIs and TDISP towards the domain of trust networks in the context of trusted path routing by introducing two new concepts: *TNDIs (Trust Network Device Interfaces)* and the *TNDI-SP (TNDI Security Protocol)*. The TNDI-SP is a new CASTOR-specific protocol inspired by TDISP but operates at the network level rather than at the PCIe/CXL device level. A TNDI is the fundamental unit of trust and assignment in a CASTOR trust network, i.e., an assignable, routing-capable unit of a network device with packet-forwarding capabilities that can join a trust network—specifically, a CASTOR network domain for trusted path routing. Depending on the platform, a TNDI can, for example, be one physical router device or one virtual router (vRouter). The TNDI-SP defines how trust is managed for a TNDI, i.e., securely established, monitored, and enforced. This includes a security state for each TNDI, TNDI onboarding into the trust network, runtime monitoring, trust assessment, and evidence exposure. The TNDI-SP splits its responsibilities into two sub-channels (or sub-protocols). First, a control channel is used for joining the device's TCB and onboarding its TNDIs, setting up the respective key hierarchies, configuring runtime monitoring and the trust assessment framework according to trust policies, and enforcing policy updates. Second, a data channel is used for sharing trustworthiness evidence with the CASTOR orchestration layer for remote verification and assessment. In addition, the TNDI-SP enables the sharing of trustworthiness evidence with neighbouring TNDIs to establish secure network links for secure communication within the trust network.

Thus, CASTOR forms the trust network from TNDIs, for which the TNDI-SP defines the respective security states and mechanisms to establish trust, enable secure communication links, and continuously provide the orchestrator with a global, up-to-date view of the trust state of each network node. Based on that, the CASTOR orchestrator can calculate trusted network paths and push respective configuration updates to the TNDIs to enforce these paths while the TNDI-SP mechanisms continuously monitor and share the TNDIs' trust states. To realize these concepts, CASTOR requires a minimal set of trusted services on each network device that implement the security mechanisms of the TNDI-SP and enforce them for the device's TNDIs. In the next section, we provide an overview of CASTOR's device-side TCB, which fulfils these requirements. We provide more details on the TNDIs, the TNDI-SP, as well as the CASTOR device-side TCB components in Chapter 7. Information on how CASTOR performs the construction and enforcement of trusted routing paths throughout the TNDI network will be provided in D5.1. Note that the TNDI-SP mechanisms will first focus (as part of the first CASTOR integrated framework detailed in D6.1) on the trust establishment into TNDIs and the collection and sharing of trust-related TNDI evidence between TNDIs and with the service orchestration layer. As part of version two, the TNDI-SP will then be extended to also manifest the enforcement of trusted route configurations (either explicit paths or segment routing policies) through the TNDI-located PCCs (path computation clients).

3.2.2 Router Extensions and TCB Components

The CASTOR Orchestration Layer (in particular the Network Service Orchestrator which is responsible for managing the secure deployment and communication with the network resources [21]) relies on a minimal TCB on each network device that is responsible for managing the device's TNDIs and implementing the TNDI-SP security mechanisms on them. CASTOR anchors these device-side TCBs in hardware roots of trust (RoTs) on each network device. That way, the orchestrator can securely establish trust in the network devices and onboard the associated TNDIs into the trust network to enforce trusted routing paths. Furthermore, CASTOR follows the principle of TCB minimization, i.e., CASTOR aims to limit the set of device-side components that are strictly necessary to guarantee CASTOR's security requirements to as few and small components as possible. Thus, CASTOR decreases the risk of compromise by keeping all other services, including, for example, the routing stack and network operating system (NOS), outside the device-side TCB.

Consequently, CASTOR requires each network device to be equipped with a HW RoT (see existing ones in Section 2.2) which serves as the foundation for the device-side TCB. The components of the device-side TCB implement the TNDI-SP, i.e., they are responsible for:

- managing the TNDIs of a device and enabling the CASTOR orchestrator to securely join the device into the domain and onboard its TNDIs into the trust network (including establishing secure communication links),

- establishing the necessary cryptographic keys required for secure communication across the trust network and for authenticating trustworthiness evidence from the device-side TCB and the managed TNDIs,
- enabling the orchestrator to configure and update security policies and trust models for the secure collection of trustworthiness evidence and trust reports on the TNDIs,
- coordinating the collection of trustworthiness evidence and reports on the TNDIs, and exposing this information to the orchestration layer and, as applicable, to neighbouring TNDIs.

In this way, they enable the CASTOR orchestrator to form a distributed trust network of TNDIs and maintain a global view of the trust states of the network, ultimately enabling the orchestrator to calculate and deploy trusted routing paths.

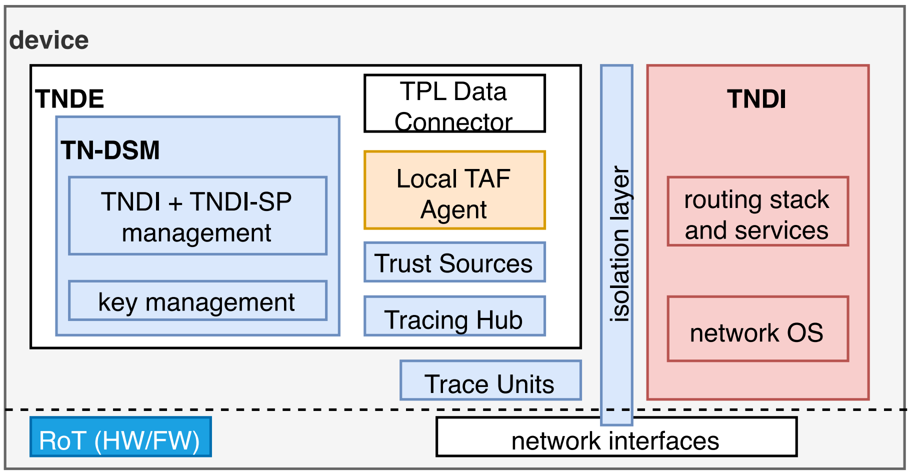


Figure 3.1: Generic overview of the trust extensions and CASTOR TCB on a network device. Each device includes a root of trust (RoT) that enables the CASTOR device-side TCB consisting of the TNDE components (excluding the TPL Data Connector) and the Trace Units. The light blue components are part of the CASTOR TCB, yellow components can be part of the CASTOR TCB depending on the concrete instantiation (as discussed later), and red components are explicitly not part of the TCB.

Figure 3.1 shows an overview of a CASTOR-compatible device including: (1.) a hardware RoT as the anchor for the device-side TCB, (2.) the device-side TCB components, and (3.) a TNDI managed by the TCB and ready to be onboarded into a CASTOR trust network. The RoT provides the necessary security foundation to layer the device-side TCB on top of it. In Section 3.3.1, we will describe what requirements CASTOR has on the RoT functionalities to enable the required security guarantees of the device-side TCB, and in Section 7.1.1, we will discuss concrete RoT instantiations. CASTOR's TCB on the network devices basically consists of the following components:

1. CASTOR's Trust Network Device Extensions (TNDE) services, except for the TPL Data Connector;
2. Trace Units managed by the TNDE;
3. an isolation layer that enforces separation between the TCB components and the remaining device components, especially the TNDIs. This layer is enabled by the RoT and its implementation not specific to CASTOR (see references below).

CASTOR introduces the new TNDE, which is a set of tightly scoped services that are responsible for managing a device's TNDIs and implementing the TNDI-SP security mechanisms on them, interfacing with the CASTOR orchestrator (and, potentially, neighbouring TNDIs) as needed. The TNDE instantiates a set of Trace Units for collecting traces on the TNDIs and uses these traces for trustworthiness monitoring and evidence generation. The isolation layer needs to separate CASTOR's device-side TCB components from the remaining device components, especially the TNDIs, even if the other non-TCB components become compromised or defective. This separation is mandatory to ensure the correct operation of the device-side TCB and thus that the security requirements still hold. Note that the concrete implementation of the isolation layer can be platform dependent (also see Section 3.3.1). We will briefly describe possible options that can be explored for CASTOR when discussing the instantiation options for the RoTs in Section 7.1.1.

In the following, we briefly describe each TNDE sub-component and outline how it contributes to the TNDI-SP functions of onboarding, key management, trust policy enforcement, and evidence handling. We will provide more details on the requirements and explored concepts of the TNDE components in Chapter 7 and follow up with a dedicated chapter on the runtime tracing architecture in Chapter 8.

Trust Network Device Security Manager (TN-DSM) The TN-DSM is the main control component of the TNDE and its TNDI-SP mechanisms. The TN-DSM is responsible for managing the TNDIs of a device and acts as the component that binds TNDIs into the trust network by maintaining their TNDI-SP security state and executing TNDI onboarding procedures on request from the orchestrator. It interfaces with the CASTOR orchestration layer services via TNDI-SP control and data channels (see Section 3.2.1), and locally enforces new configuration and trust policy requests on behalf of the orchestrator (with help from the TPL Data Connector). In addition, the TN-DSM coordinates the exposure of trustworthiness information on the TNDIs to the orchestration layer and performs key management for the TNDE and TNDIs, as required, for example, for authenticating attestation evidence and protecting TNDI-SP control and data channels.

Tracing Hub The Tracing Hub is configured by the TN-DSM and is responsible for collecting traces of the TNDIs as required for the monitoring and evidence generation of the TNDI-SP. It orchestrates a set of Trace Units, which act as the concrete tracing backends (e.g., memory inspection engines, programmable tracers, or software instrumentation), so that higher layers of the TNDE can remain independent of the specific tracing mechanisms used on a given device. The Tracing Hub needs to coordinate the sharing of traces with the Trust Sources and with the TN-DSM, which in turn exposes authenticated traces to the orchestration layer via the TNDI-SP data channels.

Trust Sources The Trust Sources are responsible for generating the trustworthiness evidence for the TNDIs based on the collected traces. In this way, they provide the evidence used to continuously assess the trustworthiness of a TNDI and allow the TN-DSM to share that information with the CASTOR orchestrator and, potentially, neighbouring TNDIs. As will be further outlined in Chapter 4, CASTOR's TCB will be required to support at least two types of Trust Sources to capture security-relevant configurational and behavioural information of the TNDIs.

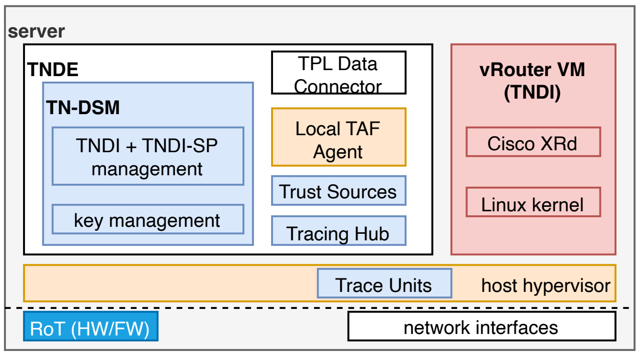
Local TAF Agent The Local TAF Agent is responsible for performing the local trust assessment of the TNDIs and generating respective trust reports that can be shared by the TN-DSM with remote verifiers, especially the CASTOR Global TAF which will use the information to construct the global, up-to-date view of the trust state of the network.

By default, as part of the first version of CASTOR's TCB, **fine-grained control over the Local TAF's behaviour will be provided:** The Local TAF will be considered as part of the trust model of the host device enabling full trust of its inner-workings; i.e., trust computations of the device's integrity to be then provided to the Global TAF. This translates to **strong trust transitivity properties** allowing for the manifestation of the overarching trust characterization of the produced routing path based on local trust opinions assigned with a full belief value [20]. This assumption includes an operational TAF model (in isolation) including memory layout planning, thread management, execution flow, definition of system call interfaces, selection of key sealing strategies. However, following CASTOR's principles towards the ultimate minimization of its custom TCB, this does not imply that the Local TAF is part of the host trusted computing base: This extension defines that this upper software layer is continuously measured and attested by the Global TAF prior to the ingestion of each produced trust related reports. This will be achieved by manifesting the produced trustworthiness claims to also include evidence on the trustworthiness of the Local TAF itself. It will be subject to a separate Attestation Key binded to additional attestation information ascertaining to the trustworthiness of the TAF itself.

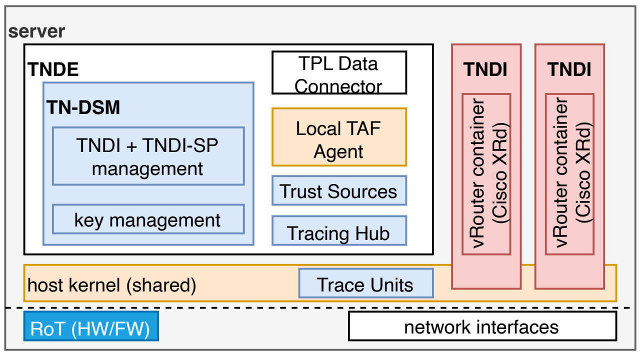
This assumption, nonetheless, will also be weakened (in the second version of the CASTOR TNDE) positioning the TAF outside of an enclave. This will also allow the evaluation of the trust assessment process when trust federation is based on local agents whose output cannot be inherently trusted - meaning that locally-computed trust opinions will need to be assigned with a certain degree of uncertaintly and/or disebelief which, in turn, will affect the trust characterization of the comprising path. On the other hand, this exclusion simplifies our deployment model reducing the runtime trust dependency management to primarily the tracing hub as the root-of-trust that will need to be enhanced with additional introspection capabilities for monitoring the execution of the Local TAF itself (Section 3.4.1).

3.2.2.1 TNDI Instantiations for Virtual and Physical Routers

Based on the described generic TNDE architecture, TNDIs can be instantiated either as virtual routers on a server platform, where a single device hosts multiple TNDIs, or as physical routing devices, typically exposing a single TNDI per device.



(a) A hypervisor-based server platform instantiating each TNDI as a virtual machine-based vRouter running a dedicated NOS and routing stack (e.g., Cisco XRd). The CASTOR TCB is located outside the VMs.



(b) A server platform instantiating each TNDI as a container-based vRouter. Each TNDI includes the container and relevant parts of the shared host kernel (e.g., routing tables, network interfaces, processes).

Figure 3.2: Two examples for TNDI instantiations as virtual routers (vRouters) on a server platform. The light blue components are part of the CASTOR TCB, yellow components can be part of the CASTOR TCB depending on the concrete instantiation, and red components are explicitly not part of the TCB.

3.2.2.1.1 vRouter Setup Figure 3.2a shows a server platform hosting a virtual router as a dedicated Virtual Machine (VM). The hypervisor managing the VMs forms an isolation layer between the TNDIs (vRouters) and the CASTOR TCB components (TNDE, Trace Units) running outside the VMs. In this case, the CASTOR TCB on the server platform can identify each vRouter as a separate TNDI and onboard them individually into CASTOR network domains. The different TNDIs can be exposed via the TNDI-SP to the CASTOR orchestrator accordingly. However, this puts a requirement on the CASTOR TCB to support multiple TNDIs and to isolate their security states from each other. That is, CASTOR requires the TNDE (especially the TN-DSM) to manage dedicated cryptographic keys per TNDI (see Section 3.3.2) and to ensure that a compromise of one TNDI cannot directly affect other TNDIs managed by the TNDE.

This modality is what will be adopted in CASTOR for supporting the evaluation of the envisioned use cases considering the need of a fully-featured and performant vRouter (i.e., xRd CISCO vRouter) capable of forwarding data traffic. This requires each (data-plane) vRouter image instantiated in each own VM so as to have access to its own distinct routing table. Control-plane vRouter images can be instantiated as a common functionality on the host server managing and sharing the iptable and nftable structures.

In principle, only control-plane vRouters do not have to be VMs but could also be isolated via different mechanisms, as long as the isolation guarantees satisfy the security requirements of the CASTOR TCB (e.g., see Section 3.3.1). For instance, Figure 3.2b shows an instantiation of vRouters as containers without any virtual machines. In this case, a TNDI captures a vRouter container with its associated host kernel resources (e.g., processes, network interfaces), as the host kernel serves as the network OS in this case. In this specific example, the host OS is shared across the TNDIs and CASTOR TCB and therefore responsible for isolating them from each other, demanding careful hardening of the host kernel. To keep the discussion focused, in CASTOR we assume that a vRouter (independent of its instantiation) consists of a single routing partition, i.e., each vRouter is identified via a single TNDI. We will discuss the TNDI management, RoT variations, and other aspects of the TNDE further in Chapter 7.

3.2.2.1.2 Physical Router The concept of TNDIs is not limited to virtual software routers but can explicitly also be instantiated on physical network devices, e.g., routers. In this case, typically, the TNDE would identify and manage the physical network device as a single TNDI. Compared to an instantiation with vRouters on a server system, on a physical network device, the isolation layer and the set of applicable Trace Units might be different depending on the platform capabilities and might face additional platform-specific requirements. However, the requirements and principles of the CASTOR TCB stay the same. While a single TNDI per physical network device (e.g., router) fits a majority of existing devices, neither CASTOR nor the concept of TNDIs enforces this 1-to-1 mapping. That is, if the network OS of a physical router forms multiple partitions (e.g., using a hypervisor, containers, or static partitioning), a physical router device could also expose multiple TNDIs, one per routing-capable partition. We provide more information on the TNDE and tracing architecture in Chapter 7 and Chapter 8.

CASTOR Custom TCB: In summary, the TNDE and its sub-components provide the device-side TCB functionality that implements the TNDI-SP, including the trust management of a device's TNDIs and the supply of the CASTOR orchestrator with trustworthy, evidence-based information on the TNDIs of the trust network. Thus, the TNDE (TNDI-SP) enables the orchestrator to securely calculate network paths based on network- *and trust*-related metrics (state information), allowing for the deployment of trusted path routing. To actually realize these functions securely on concrete platforms, the device-side TCB must be rooted in appropriate hardware support on each device. The following section therefore outlines the requirements that CASTOR places on the device-side TCB and the underlying RoT in order to support these mechanisms.

3.3 CASTOR Requirements

In this section, we start by outlining CASTOR's functional specifications, of the device-side RoT, as required for both supporting the **secure boot-up process** of the entire TCB (predicating the successful onboarding of the host routing element in the target domain) but also the runtime provision of verifiable evidence on the correct configuration and operational state of each core internal computing element - primarily the unaltered configuration of each TNDI. This allows remote verifiers to receive trustworthy confirmation that specific, unmodified code is indeed running inside each isolated island managed by the TCB on the target platform. This is crucial for verifying the correctness of trust computation results and establishing trust roots in a federated manner. We will then provide a brief overview of CASTOR's device-side key management approach, including a summary of the relevant key types used by the CASTOR

TCB to satisfy the security requirements needed for CASTOR's trusted path routing.

3.3.1 CASTOR RoT Requirements

The CASTOR orchestrator depends on the secure operation of the device-side TCB components (implementing the TNDI-SP) to enable trusted path routing, as explained in the previous section. Otherwise, CASTOR cannot reliably establish trust into each router node and receive the runtime evidence to get a global notion of the trustworthiness of the network as required for selecting trustworthy network paths. Therefore, CASTOR requires each network device to be equipped with a root of trust (RoT) which serves as the fundamental security anchor of the device-side CASTOR TCB. To satisfy CASTOR's security requirements, CASTOR requires the device RoT(s) to provide a specific set of security functions necessary for the CASTOR TCB to securely implement the TNDI-SP security mechanisms, including the TNDI trust establishment and reliable evidence collection and sharing. In the following, we provide a list of requirements for CASTOR's TCB on the network devices that need to be met by each device based on the capabilities of the device RoT(s) in order to allow participation in CASTOR's trust network domain.

R1: Isolation of TNDE TCB components from TNDIs Most of CASTOR's TNDE services are part of the device-side TCB and therefore must be strongly isolated from the TNDI (e.g., the routing stack and network OS) they are managing and monitoring. Otherwise, a compromise of a TNDI might spread to the TNDE which would break the security guarantees of CASTOR's device-side TCB. It should also support enhanced authorization on which process, running on a device, can access the trusted component

R2: Isolation of Trace Units from TNDIs The Trace Units are also part of the device-side TCB and therefore must be isolated from the TNDIs to guarantee the reliability of the evidence monitoring.

R3: Measurement of TCB software/firmare components CASTOR requires a secure way to measure the device-side TCB components in order to establish trust in the router nodes (RoT for measurement). This includes the respective CASTOR TNDE services and Trace Units (see Section 3.2) and might additionally include parts of the platform software, e.g., depending on the instantiation of the isolation mechanism (R1 and R2). Note that this is not to be confused with the runtime evidence on the TNDIs provided by the CASTOR TNDE.

R4: Reporting of Measurements CASTOR requires a secure way to report the measurements (of R3) to remote verifiers and have them linked to that specific network device via a unique identity (RoT for identity/reporting). That is, in particular, the CASTOR orchestrator must be able to perform remote attestation to verify the CASTOR TCB and its security guarantees as required to establish trust in the network device and its TNDIs. Note that this is not to be confused with the reporting of the runtime evidence on the TNDIs provided by the CASTOR TNDE.

R5: Secure storage capabilities CASTOR's device-side TCB components require a way to securely store data (e.g., keys) on non-volatile storage (RoT for storage/sealing), optionally binding the data to specific access policies (e.g., measurement-based). This will be used to store essential keys and for a small set of platform configuration registers used to hold the software measurements and traces, extracted by the CASTOR Tracing Hub, based on which the trust assessment process will be executed. Such measurements need to be accompanied with strict verifiability guarantees on their integrity, freshness and authenticity.

R6: Secure entropy CASTOR's device-side TCB components require access to secure entropy sources, e.g., for the generation of keys or cryptographic nonces.

R7: Secure notion of time CASTOR's device-side TCB components require a secure notion of time, e.g., for certificate verification or to establish an ordering scheme for evidence.

CASTOR's device-side TCB requires the above requirements to be met by the network device platform in order to securely operate and enable the orchestrator to establish trust into the network node(s) based on the TNDI-SP mechanisms. As mentioned, these requirements require respective RoT functionalities to anchor their fulfilment into trusted hardware/firmware. In Section 2.2, we have provided a list of existing technologies that implement RoTs with relevant functionalities to meet CASTOR's requirements. Note that not all of the requirements (e.g., the isolation requirements R1/R2 or the secure notion of time R7) need to necessarily be directly satisfied by the device RoT itself. Instead, they could be fulfilled by additional TCB components (with reasonable complexity) whose trust is anchored in the device RoT (e.g., a measured hypervisor providing isolation). In Section 7.1.1, we will discuss some of the options that CASTOR will explore and analyse their expected tradeoffs regarding, e.g., security and deployment.

3.3.2 CASTOR Key Management

As aforementioned, one of CASTOR's TCB design principles is to offer strong **key management capabilities** for establishing and maintaining the appropriate crypto primitives (secret keys) needed to not only secure the communication between the routing elements (comprising a path) but to also establish (run-time) trust relationships between connected nodes, as determined by the control plane. *In the current definition of IETF's Trusted Path Routing specifications [11] such available paths are implicitly trusted.* Communicating entities have no means yet to establish secure sessions, amongst them, that can enable the secure interactions and conveyance of trust-related evidence supporting the continuous execution of trustworthiness appraisals - both at a **node-level**, allowing each routing element to securely exchange "stamped passports" on the trust level of its adjacent routers, but also at a **topology-level** enabling the Orchestrator to maintain a trust network adaptive to changes in the trust level of each element, link and/or path. What is the **common denominator when determining they key management system needed to support such a domain-wide trust quantification is the need for the provision of fresh and verifiable evidence capturing the operational state of each element anchored to its decentralized root-of-trust (and its host CASTOR TCB).**

This essentially translates to the need to decouple those crypto primitives that are already foreseen for guaranteeing the confidentiality, integrity and authenticity of data-plane communications (already provided through the employment of traditional Public Key Infrastructures (PKIs) in the form of certificates and asymmetric private-public key pairs during the activation of a router) from those needed to resolve the **trust dependencies between all core components comprising the network's infrastructure:** This includes both the secure interactions and evidence needed between the dependent layers of a routing/infrastructure element (layered attestation) as well as path-comprising elements (routing plane) all the way to topology-comprising infrastructure nodes (infrastructure plane). In CASTOR, this comes in the form of an auxiliary key management system (complementary to the traditional trust router configuration certificates), managed by the TN-DSM and as part of the overarching TCB towards unlocking services beyond the pure encryption-based data security: (i) better visibility of the trustworthiness characteristics of each one of software layers of the target network elements; (ii) construction of a "*Network-based RoT*"extending the attestation model of *(Single Prover-Single Verifier)* to that of *(Multiple Provers-Multiple Verifiers)* allowing for the scalable and (order-preserving) composite attestation of all path-comprising elements that might adhere to different appraisal policies (enforced by the different routing vendors); and (iii) secure and privacy-preserving sharing of evidence on the trustworthiness characteristics and capabilities of a domain infrastructure without though revealing any details on the specific trust scors of the established path profiles (i.e., order-revealing encryption). To support these operations, Table 3.1 provides an overview of the relevant cryptographic keys, their scale and usage, as well as their binding requirements.

For the latter, it is important to highlight the **key binding considerations** needed,for attesting nodes, that enable association of runtime trustworthiness evidence and reference values (per the IETF and TCG Attestation Model) with the specific attester's (HW-based) identifier. A model of an example Attester helps

Table 3.1: Overview of key types relevant for the CASTOR device-side TCB.
(*in a specific implementation, the TNDI-SP control and data channel could share keys)

Key Description	Type	Usage	Number	Binding
RoT Key	asymmetric	exposed by the underlying secure element (RoT) and used to authenticate/identify the device	1 per RoT	N/A
TNDE (platform) key	asymmetric, signing	used by TN-DSM to sign attestation evidence on the TNDE and its underlying platform	1 per TNDE	bound to/child of the RoT key; bound to device platform (RoT, isolation layer) and TNDE
TNDE sealing key	symmetric, authenticated encryption	used by TNDE to securely store data on storage	≥ 1 per TNDE	bound to the device platform and TNDE
TNDE internal communication key (*optional*)	symmetric, authenticated encryption	used by TNDE components to communicate securely with each other — *if required*	each TNDE component ≥ 1 per other component	implementation-specific (e.g., to component IDs and/or the TNDE key)
TNDI (attestation) Key	asymmetric, signing	used by TN-DSM to sign (TNDI-related) trustworthiness claims from L-TAF; and/or attestation reports produced from the trust sources to be shared with the CASTOR orchestration layer	1 per TNDI	bound to the TNDE key and, optionally, the TNDI
Trust Source Key	asymmetric, signing	used by either of the Trust Sources (Attestation; and/or FSM) for signing the produced attestation report	1 per TNDI	bound to the reference value representing the "expected state" of the TNDI based on the property of interest defined during onboarding
TNDI trace key	symmetric, signing	used by Tracing Hub or Trace Units to sign TNDI runtime traces	1 per TNDI	bound to the TNDI key (and thus also to the TNDE)
Orchestrator authentication key	asymmetric, signing	public key is used by TNDE to authenticate CASTOR orchestrator taking ownership of the TNDE (device) / TNDIs. This is issued by the mployed PKI and is included in the router's trust configuration manifest	≥ 1 per CASTOR orchestrator	n/a
TNDI-SP control channel key (*)	symmetric, signing / encryption	used by TN-DSM to securely communicate with CASTOR orchestrator	≥ 1 per control channel	bound to the TNDE key
TNDI-SP data channel key (*)	symmetric, signing / encryption	used by TN-DSM to securely communicate (TNDI evidence) to the CASTOR DLT and Global TAF	≥ 1 per data channel	bound to TNDI-SP control channel and, optionally, to the associated TNDI
TNDI-to-TNDI secure link key	symmetric, signing / encryption	used for secure communication (e.g., MACsec) between two neighbouring (topology-wise) TNDIs	≥ 1 per TNDI-to-TNDI link	bound to the TNDI key (and thus also to the TNDE)
Composite & (D)ORE key	both asymmetric and symmetric key pairs	Used for supporting the multi-ordered signature scheme of CASTOR towards path-level device attestation as well as the sharing of domain-specific trust characteristics in a privacy-preserving manner	1 per TNDI	bound to the TNDE key

illustrate that evidence can be bound to an Attester to show trustworthiness claims that build on top of each other (Figure 3.3).

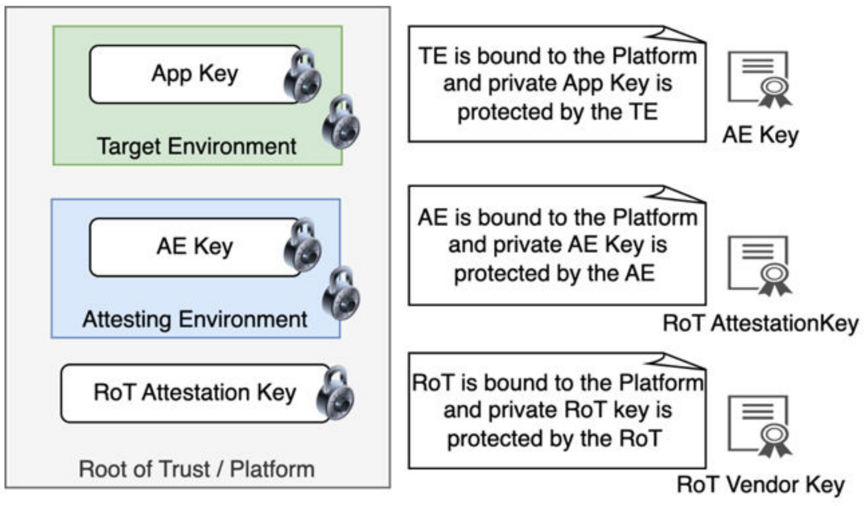

App Key
Target Environment
AE Key
Attesting Environment
RoT Attestation Key
Root of Trust / Platform
TE is bound to the Platform and private App Key is protected by the TE
AE Key
AE is bound to the Platform and private AE Key is protected by the AE
RoT AttestationKey
RoT is bound to the Platform and private RoT key is protected by the RoT
RoT Vendor Key

Figure 3.3: Key Binding Considerations of CASTOR's Key Management System.

An example Attester (Routing Element - TNDI) is instantiated in an infrastructure element (Platform) successfully integrated with RoT components. A **RoT Key** is securely provisioned to the RoT. An Attesting Environment (AE) firmware is securely loaded into the platform, and an Attestation Key (**TNDE Platform Key**) is securely provisioned to the AE. A Target Environment (TE) software is securely loaded into the platform, and Application Keys (in the case of CASTOR such trusted applications are considering the trust sources providing evidence on the configuration and execution state of the target TNDI - depicted as **Trust Source Keys** in Table 3.1) are securely provisioned. These keys are all binded to the (device-bound) RoT Key - furthermore, the TNDE is also exposed to a separate **TNDE Sealing Key** for allowing the establishment of secure sessions with other (infrastructure-oriented) TNDEs.

There are three pieces of evidence accompanying the Attester that convince a Verifier to trust the Attester. The first is an Endorsement that is signed by the RoT vendor (RoT Vendor Key) that claims the RoT component is securely integrated with the platform and the private RoT Attestation Key is protected by an appropriate protection capability of the RoT. The second piece of (attestation) information is Evidence that is collected by the RoT environment (Tracing Hub and signed through the distinct **TNDI-specific Tracing Keys**) that claims the AE is securely bound to the platform and that the private AE Key is protected by the AE. The third piece of attestation information is Evidence that is collected by the AE that claims the TE is securely bound to the platform and that the private TE Key is protected by the TE. The AE signs the Evidence with its private AE key to prove the claim is legitimate.

In addition to those crypto primitives needed for capturing the trust dependencies of static (at a router-level) and dynamic (at a path-level) dependencies so as to allow the provision of fresh and authentic trustworthiness claims, the CASTOR KMS needs to also expose capabilities for allowing the composite attestation of nested AEs: Attestation of path-comprising TNDEs (managing multiple TNDIs) verifying not only the trust level of each TNDE but also ascertaining the maintenance of the correct order of nodes/routers in the path enforced by the control-plane. Such **Composite Attestation** key constructions and certificates will need to be issued to each Attester, during onboarding, by the Verifier (Orchestrator). As will be seen in D3.2, CASTOR aims to investigate the integration of multi-ordered aggregation signature schemes and/or leverage the commutativity property of matrix-based key constructions for enforcing verifiable order preservation into the TCB of each node.

Within this design, the hardware-isolated trusted computing base functions as the origin of authoritative evidence, in a manner analogous to a RATS Attester, with cryptographic keys distributed across its internal sub-components. Each trace unit (per TNDI) possesses its own key material, enabling it to produce fresh and authentic execution traces that are cryptographically bound to a specific hardware-protected context. This fine-grained key allocation ensures that evidence remains attributable and independently verifiable, while preventing the collapse of the trust chain in the presence of partial compromise. As aforementioned, TNDIs are provisioned with dedicated cryptographic keys that support the establishment of secure channels with the orchestration layer These keys enable the generation of authenticated network-level evidence that captures the observable behaviour of a node in the routing plane without asserting claims about its internal execution state.

3.4 Design Variations and Tradeoffs

Describing variations to the CASTOR TCB and their respective tradeoffs. In subsequent deliverables (D3.2, D3.3), results will be presented based on the concrete choices made for the prototypes.

3.4.1 Runtime Evidence Monitoring for Local TAF Agent

In CASTOR, the Local TAF Agent - a standalone trust assessment modality - operates at the node level in the network such as on a router and is a **design choice not to be part of the CASTOR TCB** (i.e., it is assumed to be potentially compromised at runtime), since there is a security requirement (see also the description of SR.2 of D2.1) to keep the size of CASTOR TCB as minimal as possible in order to decrease the risk of vulnerabilities and allow for easier code audits.

Instead, the Local TAF agent is treated as a **trusted application that runs in isolation**, quantifies the node's trustworthiness based on integrity-related attestations, and forwards its self-assessment (trust opinion) to the Global TAF - a federated trust assessment modality - in the orchestration layer. As described in D4.1 [20] the Local TAF agent is focused on integrity-related trust proposition avoiding the use of subjective logic which is complex, making it unsuitable for resource-constrained devices such as routers. Subjective logic will be utilized only in the Global TAF, which consumes routers' trust opinions.

Keep in mind that if the Local TAF agent does not operate in isolation, then the Global TAF must also account for the additional uncertainty, or reduced belief, introduced by that lack of isolation. This uncertainty is directly tied to the strength of the Local TAF's isolation guaranties. In particular, the **internal mechanisms of the Local TAF** (e.g., the TLEE and TSM) **must be proven correct** for the Global TAF to maintain maximal confidence. Otherwise, the Global TAF should incorporate the corresponding uncertainty or disbelief arising from the risks inherent in the underlying environment.

To address this, CASTOR requires the Local TAF to **attach additional trustworthiness claims (e.g., an attestation report)** each time it computes a trust opinion and sends it to the Global TAF, ensuring the integrity of the operation. Apart from the trust opinion, the Local TAF provides an attestation report that offers evidence about the integrity of the routing element at the time the trust calculation is executed. These claims characterize the operational state and correctness of the Local TAF and can be independently verified by the Global TAF.

The Global TAF then applies subjective logic to adjust the Local TAF's assessment, for example, by reducing confidence or increasing uncertainty, based on the associated trustworthiness evidence. For this reason, as described later in Chapter 8, we introduce a **dedicated tracing unit within the CASTOR Tracing Hub** that is entrusted with introspecting the isolated memory space where the Local TAF is instantiated.

3.4.2 Towards Layered or Composite Attestation

According to the Trusted Computing Group [78], the main three components participating in an attestation process are: (i) the attester that provides a set of evidence with respect to its state, ii) the verifier that consumes the attestation evidence and is able to determine the trustworthiness of the attester, and iii) the relying party that uses the results of the verifier, evaluates them and potentially performs an action based on them. For instance, the CASTOR TCB running in a TNDI should be able to provide such evidence to a requester, e.g., the Local TAF agent- in order to verify the collected evidence and evaluate the trustworthiness of the target trust properties. However, in the context of the CASTOR architecture, the system model is more complex both within the boundaries of a router element and outside it.

When it comes to attester elements, they may come in different shapes and forms. First, starting from the complex router software stack, it consists of multiple processes that interact with each other in order to realize vital network functions. Along with the need to cater different levels of assurance and different target security properties, CASTOR needs to explore flexible and efficient mechanisms that allow the CASTOR TCB to expose attestation reporting primitives to external verifiers. The need for attestation mechanisms capable for handling multiple provers is also expanded beyond the boundaries of a router element. Specifically, the requirement to continuously evaluate the state of provisioned network paths — in order to ensure service fulfillment in accordance with established service-level agreements — creates a need for robust methods that offer path-level evidence without compromising the operational profile of the established communication channels.

On a similar note, the verifier elements play a crucial role in the attestation process as they appraise Attesters and produce attestation results for the Relying Party. In CASTOR's highly heterogeneous and multi-tenant environment it may be likely that a single verifier cannot process all attestation evidence that are transmitted by the network topology. In addition to that, CASTOR considers verifiers in different layers in the compute continuum: ranging from the infrastructure layer (e.g., adjacent routers verifying the attestation quote as part of the link layer authentication process as mentioned in IETF's Trusted Path Routing [11]) all the way to the Orchestration layer (e.g., Global TAF Trust Source Manager mentioned in D4.1 [20]) and the CASTOR DLT (data veracity checks by off-chain worker nodes as mentioned in D5.1 [21]). Regardless of its placement in the CASTOR framework, the complexity in the attested environments introduces several challenges to considering a single verifier [30]:

1. The set of participating attesters may change over time. In the context of service assurance, the Service Orchestrator may decide to migrate a workload traffic from one path to another; hence, the set of attesters need to be revised.

2. The existence of multiple services that verify different components of the router software stack and the CASTOR TCB may not favour the development of a "monolithic" verifier (i.e., due to a lack of knowledge, complexity, regulations or associated cost).

3. The appraisal policy owner or the reference value provider may not want to disclose any sensitive information to an external verifier. In the context of CASTOR, it may be likely that router vendors do not want to disclose any data pertaining to the "correct" state of a router to verifiers running within an administrative domain.

Overall, it becomes clear that CASTOR necessitates the design and implementation of attestation primitives that are able to accommodate the transition towards multi-prover/multi-verifier system models in a secure and efficient manner. To this end, CASTOR envisions to focus on different attestation variations that are able to cope with intrinsic challenges on the continuous trust evaluations within and beyond the boundaries of a TNDI. In what follows, we consider the instantiation of these variations in intra-domain scenarios; the details and challenges of cross-domain exchange of trustworthiness evidence are mentioned in Section 9.2.2.

3.4.2.0.1 Layered Attestation As networks transition from fixed hardware routers to virtualized routing platforms, trust must be established across components that are deployed, upgraded, and orchestrated independently. A virtualized routing stack benefits from a reconfigurable and clearly bounded TCB so that changes in software or infrastructure do not require re-establishing trust in the entire system. It must also remain agile enough to measure a varying set of target environments, from the underlying host to the virtual router image and its vital network functions. Depending on the required trust level that needs to be attained, and the level of assurance that a network element needs to exhibit, the system must be able to extract a correspondingly varying level and volume of evidence. In addition, identities naturally align with system layers — hardware, platform, virtual router, and applications — and must be bound to the evidence produced at each of those boundaries. Taken together, these needs motivate an approach in which trust is constructed incrementally and composed across layers. To this end, CASTOR adopts a layered attestation model that is able to accommodate the needs of the overarching trust assessment process.

The design and implementation of the different layers that comprise this attestation primitive are highly dependent on the system model and the underlying trust assumptions. In all cases, the foundational layer- i.e., layer 0 - is the hardware-based RoT. Subsequently, as part of the first integrated framework of CASTOR, we consider the overarching CASTOR TCB which is assumed to be trusted and is able to attest all upper layers, including the configuration integrity and operational behaviour of vital network functions of a TNDI. However, in the second version of the CASTOR integrated framework, CASTOR aims to weaken the trust assumptions on the TCB and minimize it further. As shown in Figure 3.4, CASTOR will investigate layered attestation approaches in which Secure Loaders- acting as Layer 1- securely measure and provision subsequent layers, enabling the secure launch of TNDE artifacts and, ultimately, the target TNDI environments. This approach ensures that all components of the initial TCB are measured at least during their initialization, while the Hardware RoT elements remain the only components inherently trusted throughout the lifecycle of the TNDIs. This enables the compartmentalization of the attestation process, as each TNDI may require a different Secure Loader as a parent process (e.g., Intel SGX parent enclave process) to provision the necessary TCB elements (e.g., the appropriate Trace Units for runtime introspection) and trusted applications (e.g., Local TAF agent) for specific set of critical network functions that need to be monitored under a given threat model.

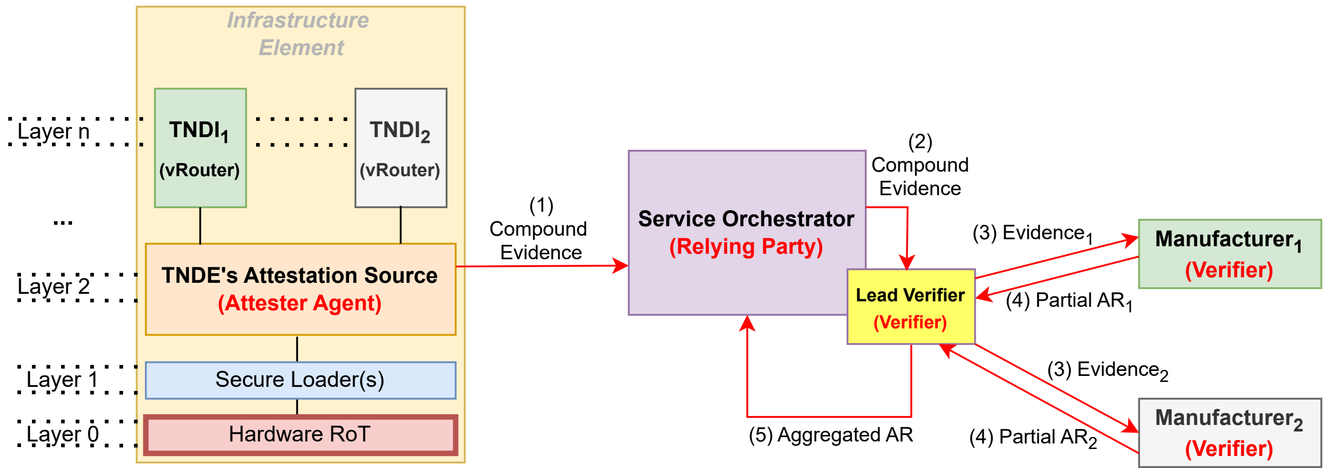


Figure 3.4: Layered attestation in a hierarchical multi-prover/multi-verifier scenario. Compound Evidence (CE) is efficiently shared to the appropriate manufacturer's Verifier, while the Aggregated Attestation Result (AAR) is returned back to the Relying Party by the Lead Verifier.

Going beyond the in-router multi-prover model, the Attestation Source generates attestation evidence using TNDI-specific attestation keys and shares it with the Service Orchestrator, which serves as the relying party. In this variation, we also consider the case where the Service Orchestrator does not have access to the necessary reference values and cryptographic material to verify the compound evidence that is produced by an infrastructure element. In fact, this can be a realistic scenario, given that router manufacturers may not disclose such information to any domain operator. In this context, CASTOR aims to investigate hierarchical multi-verifier models [30] where the Service Orchestrator acts as the lead verifier and delegates the necessary attestation evidence (e.g., attestation quote) to the appropriate router manufacturer in order to conduct the verification process and return the partial attestation results (PAR) back to the orchestrator's lead verifier in order to construct the final aggregated attestation result (AAR). Eventually, this result is forwarded to the Service Orchestrator (i.e., Relying Party) and can be used to monitor the trustworthiness of the router element's trust property under evaluation.

3.4.2.0.2 Composite Attestation As described in the distinct phases of the overarching CASTOR framework in D2.1 [22], the Global Trust Assessment Framework in the orchestration layer aggregates trustworthiness evidence and local trust evaluations from the infrastructure layer (i.e., the network topology) to enable trust-aware traffic engineering for the Service Orchestrator. Once network paths are selected—based on the recommendations of the CASTOR Optimization Engine, as detailed in D4.1 [20] — the Service Orchestrator must be able to efficiently monitor the trustworthiness of those paths as part of the service-assurance processes that uphold the established Secure Service Level Agreements (SS-LAs). To this end, CASTOR envisions extending these mechanisms with a composite attestation scheme capable of generating contextual, path-aware trustworthiness evidence that can attest to the operational correctness of multiple TNDIs (i.e., network elements) across an entire segment or even an end-to-end path. Unlike device-level attestation schemes, this variation additionally guarantees that the sequence of network elements along the path is preserved. This property provides strong evidence that the path remains intact throughout its lifecycle, enabling verifiers to detect any reordering, insertion, or removal of nodes.

Figure 3.5 captures a high-level example of the composite attestation system model in an established path, allowing the transmission of workload traffic between two endpoints. In this variation, the Attestation Source- attached to each TNDI- produces compound evidence (CE) that characterizes its own state and then forwards it to the Attestation Source of the subsequent TNDI. In this context, the receiving Attestation Source shall be able to produce its own compound evidence and aggregate it with the received CE before relaying the final result to the subsequent TNDI. Typically, each TNDI is equipped with the necessary TNDE capabilities to expose the functionalities of the Attester Agent and, potentially, the Verifier. However, as shown in the case of $TNDI_1$ and $TNDI_2$ in Figure 3.5, it is possible for multiple TNDIs along a single path to be collected within the same infrastructure element. This underscores the need for a robust and efficient TNDE capable of managing TNDI-specific cryptographic material, since each TNDI must sign its own CE using its corresponding TNDI-bound attestation keys. Through TNDI-specific cryptographic keys, CASTOR is able to derive trustworthiness evidence about the number and order of network elements that comprise an established path in the network topology. The details of the requirements of the composite attestation scheme are documented in Chapter 9.

Finally, an important parameter in the realization of the composite attestation scheme concerns local verification at the Attestation Source (i.e., the Attester Agent) associated with each TNDI along the path. In contrast to the layered attestation model, this approach would introduce a cascading verifier scheme: each Attestation Source validates the previously received compound evidence (e.g., CE_{1-2} received by the TNDE of Infrastructure Element B), before aggregating it with its own evidence in order to encode and transmit the derived compound evidence to the subsequent node in the path (e.g., CE_{1-2-3} produced by the TNDE of Infrastructure Element B). Of course, the hierarchical model of Figure 3.4 is also applicable in this context: when local verifiers of a TNDI need to verify the compound evidence of another vendor's router element, they may have to reach out to the manufacturer's verifier in order to get the necessary attestation results (e.g., Figure 3.5 shows $TNDI_4$ with a different colour than the rest of the topology, capturing the fact that the end-to-end paths may consist of network elements from different router vendors). However, local verification is not a strict prerequisite. The Attestation Sources could simply aggregate the compound evidence and forward it directly to the Service Orchestrator, where verification would take

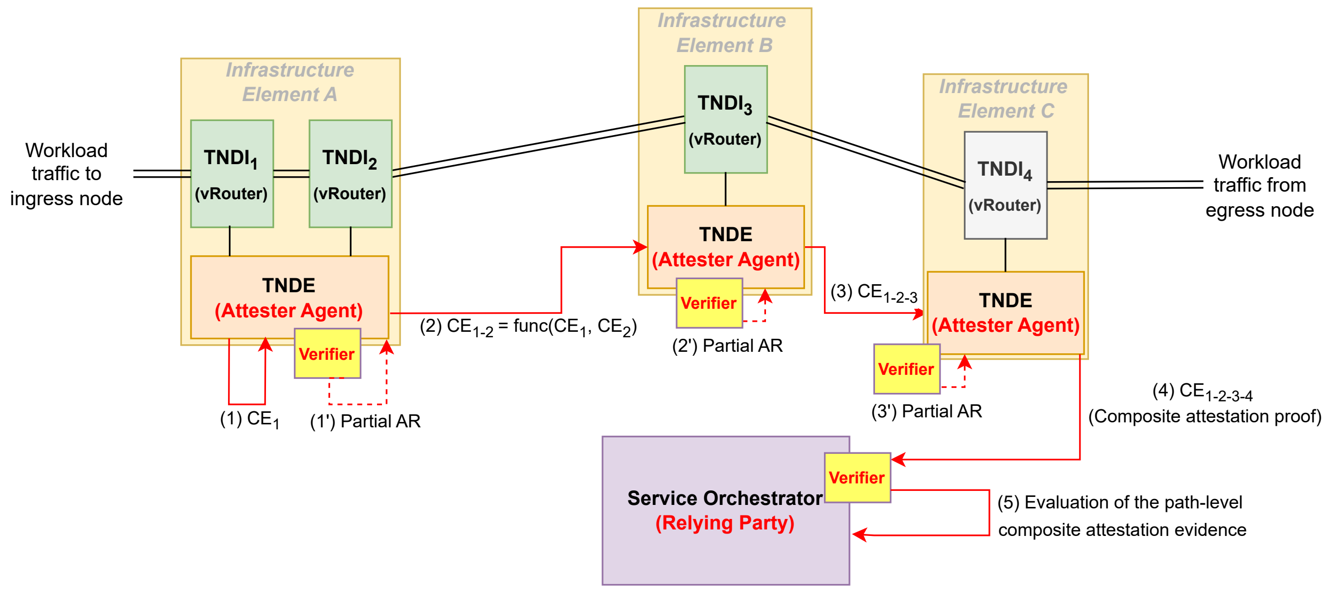

Infrastructure Element A
Infrastructure Element B
Infrastructure Element C
Workload traffic to ingress node
Workload traffic from egress node
TNDI$_1$ (vRouter)
TNDI$_2$ (vRouter)
TNDI$_3$ (vRouter)
TNDI$_4$ (vRouter)
TNDE (Attester Agent)
Verifier
(1) CE$_1$
(1') Partial AR
(2) CE$_{1-2}$ = func(CE$_1$, CE$_2$)
(2') Partial AR
(3) CE$_{1-2-3}$
(3') Partial AR
(4) CE$_{1-2-3-4}$ (Composite attestation proof)
Service Orchestrator (Relying Party)
(5) Evaluation of the path-level composite attestation evidence

Figure 3.5: Composite attestation in a cascading multi-prover/multi-verifier scenario. The TNDE's Attestation Source of each path element contribute to the creation of a composite attestation proof that captures the state of an established network path.

place. The resulting assessment could then be consumed by the Global TAF as additional evidence, contributing to a more confident path-level trust characterization throughout the operational lifecycle of the network topology.

Chapter 4

CASTOR-enabled Trust Sources for Traffic Engineering Operational Assurances

4.1 Systematic vs. Networking Evidence

In the CASTOR trust framework, trust is not a static property but a dynamic state governed by the Below-Zero-Trust model as conceptualized in IETF frameworks such as Remote Attestation Procedures (RATS). This paradigm shifts beyond the foundational ”never trust, always verify” approach by acknowledging that an entity’s trustworthiness is subject to constant fluctuation. To account for this volatility, the system moves away from a priori trust assumptions and instead requires continuous runtime observability and deep introspection. This persistent monitoring allows the system to perform real-time trust assessments that reflect the current operational reality of each component.

To achieve this level of assurance, the framework integrates two primary streams of evidence designed to justify the integrity and reliability of the entire system. Networking evidence serves as a behavioural and functional record of the infrastructure, moving beyond simple attestation of network operations to include any evidence related to network routines and functions. This encompasses verifiable proofs of routing paths, as well as operational telemetry that ensures network elements are adhering to intended protocols. By analyzing these functional artifacts, the framework can characterize trust along specific service paths and detect anomalies in traffic handling.

Simultaneously, systematic evidence provides the necessary visibility into the internal state of the endpoints themselves. This system-level evidence consists of runtime measurements—such as memory states and CPU behaviour—that enable the platform to perform introspection. By verifying the internal integrity of the devices, the framework ensures that the hardware and software layers remain a reliable foundation for all higher-level operations. These two streams converge within the Trust Assessment Framework, where local Trust Assessment Functions at the edge nodes ingest networking data, while systematic artifacts are used to validate device-level integrity. In practice, CASTOR’s attestation enablers generate these essential trustworthiness artifacts through secure telemetry and on-device attestation, providing the necessary data to sustain the Below-Zero-Trust lifecycle.

4.1.1 Network-Level Evidence

Networking evidence is the primary input to the local trust characterization process. In CASTOR, network devices and paths produce cryptographic attestations or telemetry that prove their security properties. For example, the NASR (Network Attestation for Secure Routing) approach provides “appraisable evidence of the trust or security properties” of a routing path”, essentially a proof that the packets traversed an agreed secure path. CASTOR extends PCI-SIG’s TEE-Device Interface (TDISP) so that network elements can securely transport trust evidence from the link layer up to the orchestration plane. A Local TAF uses this evidence (e.g. authenticated path attestations, hop-by-hop integrity proofs, or enriched flow metadata) to compute a trust score for each route or service path. In other words, network telemetry and path attestations let CASTOR quickly identify malicious or compromised links (such as a hijacked route) and adjust trust ratings or routing decisions accordingly.

4.1.2 System-Level (Systematic) Evidence

Systematic evidence verifies the trustworthiness of the data sources themselves by inspecting their internal state. In CASTOR, dynamic in-kernel tracing and hardware attestation extract fresh device-level evidence for trust evaluation. For instance, CASTOR's dynamic tracing modules focus on kernel extensions that capture runtime state of processes and memory. A concrete example from trusted computing is the CoPilot system: it runs on a separate hardware bus and periodically computes hashes of key memory regions in the kernel, comparing them to known-good values and reporting any discrepancies. Such memory-inspection evidence confirms whether the device is running unmodified, expected software, thereby ensuring that its outputs can be trusted. Likewise, integrating TEEs (Trusted Execution Environments) allows CASTOR to collect signed measurements of system configuration and software (from boot to runtime) as evidence. By including these systematic checks, CASTOR can detect if an endpoint has been tampered with or infected – verifiability that purely network-based evidence would miss. By including these systematic checks, CASTOR can detect if an endpoint has been tampered with or infected – verifiability that purely network-based evidence would miss.

4.1.3 Introspection vs Observability

In the CASTOR trust framework, the increasing complexity and interconnectedness of modern computing platforms have transformed the security landscape into one where attacks can originate from multiple layers simultaneously. These threats are not confined to a single device but stem from within a device's internal architecture—such as software vulnerabilities and malicious code injection—as well as across the different layers of the network stack, including networking-driven intrusions and supply-chain tampering. In this environment, system behavior must be continuously scrutinized because static defenses are no longer sufficient. Relying solely on static policies, such as fixed configurations to prevent device misstatement or predefined rules for capturing the expected state of a routing device, fails to account for the dynamic nature of runtime operations.

To bridge this gap, CASTOR utilizes Trace Units to move beyond static snapshots toward continuous assurance. Ensuring trustworthiness requires mechanisms that can reveal the internal dynamics of a system while it is operating, which is the specific domain of observability and introspection. While these terms are often used interchangeably, CASTOR recognizes a thin but critical line between them. **Introspection** is defined as the process of raw trace extraction, providing direct, low-level examination of runtime execution—including memory structures, control flow, and kernel-level events. **Observability** is the interpretation of these raw traces into a human-readable and actionable format, allowing the system to infer its internal state from the signals and metrics emitted during execution. Together, these mechanisms provide the foundation for detecting deviations from expected semantics and preserving system integrity across both the device and the network layers.

4.1.4 The CASTOR Case

The rationale for introducing CASTOR evidence as a distinct evidence category stems from a functional limitation inherent in conventional observability approaches when applied to Below-Zero-Trust (BZT) environments. In such models, the objective is not merely to attest the integrity of isolated components, but to continuously contextualize the operational health of a service as it evolves over time. Traditional systematic and networking tracing mechanisms, while individually exhaustive within their respective domains, fail to provide the semantic cohesion required to assess the integrity of a dynamic entity such as a Virtual Network Function (VNF). In the IETF-aligned Below-Zero-Trust paradigm, trust is neither static nor binary; rather, it is a continuously evaluated function derived from the interaction between multiple architectural layers. CASTOR evidence is therefore explicitly designed to capture the surface of interaction where network logic intersects with underlying compute resources, ensuring that any trust assessment reflects the VNF's actual execution lifecycle and runtime environment, rather than a fragmented snapshot of its components.

As CASTOR aims to unify the systematic and networking tracing domains, it is essential to account for the execution context in which Virtual Network Functions (VNFs) operate. Although network tracing provides critical visibility into service connectivity and data transit, it lacks insight into the underlying computational context that ultimately governs how packets are processed. Systematic and networking evidence are therefore fundamentally disjoint: they originate from different extraction points, encode distinct semantics, and belong to separate architectural planes.

The CASTOR framework bridges this semantic and architectural gap by defining its evidence set through a VNF-centric operational lens. Rather than treating systematic and network traces as independent telemetry streams, CASTOR explicitly couples them through the execution semantics of the VNF itself. Formally, CASTOR evidence is defined as the union of the complete networking tracing set and a carefully scoped subset of systematic tracing artifacts that are intrinsically tied to the VNF's execution lifecycle. This lifecycle encompasses the critical phases of instantiation, runtime configuration, packet processing, scaling, and termination. Only those system-level events that are causally or temporally linked to these phases are incorporated, ensuring that the resulting evidence stream remains relevant and interpretable.

By isolating systematic traces that correlate with VNF-specific behavior—such as transient memory pressure during packet bursts, kernel-level interactions triggered by dynamic scaling, or scheduling anomalies affecting packet processing latency—CASTOR produces a high-fidelity, service-aware evidence stream. This enables a form of holistic observability in which network-level symptoms can be directly correlated with system-level causes. For example, a localized degradation in throughput observed at the network layer can be justified by concurrent evidence of hardware resource contention, misconfiguration, or malicious memory manipulation at the system layer. In doing so, CASTOR transforms raw telemetry into actionable trust signals, providing a principled and continuous justification for the current trust state of the service path under the Below-Zero-Trust model.

4.1.5 Relation to Threat Model

The integration of these evidence streams is further justified by the need to counter specific protocol-level vulnerabilities that threaten the communication fabric. Within the networking domain, CASTOR must account for sophisticated attacks targeting core routing protocols such as BGP (Border Gateway Protocol) and OSPF (Open Shortest Path First). In BGP, adversaries may exploit the lack of inherent trust verification to perform prefix hijacking, route leaks, or AS PATH tampering, effectively misdirecting global or inter-domain traffic. Similarly, in OSPF environments, internal or compromised actors can engage in Link State Advertisement (LSA) falsification or spoofing, evading the protocol's standard ”fight-back” mechanisms to persistently poison the topology view of an entire routing area.

Networking evidence addresses these threats by providing the verifiable proofs and telemetry needed to detect such control-plane anomalies. Local TAFs utilize this data to ensure that routing decisions are not only based on protocol advertisements but are backed by valid attestations. Meanwhile, systematic evidence serves as a defense against the underlying cause of such protocol attacks—the compromise of the routing entity itself. By verifying at runtime that a router's or VNF's hardware and software states have not been altered by malware or counterfeit firmware, CASTOR ensures the integrity of the ”BGP speaker” or ”OSPF neighbor.” Ultimately, under CASTOR’s threat model, trust in a service requires both a trustworthy path (resilient to BGP/OSPF manipulation) and trustworthy endpoints (resilient to system-layer exploits). The Local TAF thus combines these streams to derive an accurate trust characterization, ensuring that networking evidence drives route-level trust while systematic evidence validates the integrity of the nodes participating in those routes.

4.2 Extended Trust Models for Segment Routing

In the IETF’s Trusted Path Routing (TPR) paradigm [11], routers can establish links with each other only if they exchange sufficient evidence of their trustworthiness. This enables the formation of ”Trusted Topologies,” over which the domain operator can provision traffic engineering policies, such as Segment Routing, that traverse only ”trust-verified” network devices. A key feature of TPR is the establishment and continuous maintenance of trust relationships between adjacent routers. However, CASTOR has already highlighted [50] the main challenges that have not been addressed yet by the current TPR specification with respect to runtime trust evaluation that goes beyond static properties (e.g., router is equipped with a TPM processor or it has securely booted). Specifically, the current TPR system model assumes a single assessment instance, uniform evidence semantics, and a clear Verifier–Attester boundary. By lifting these assumptions, the CASTOR framework enables the trust engineering process which, in turn, enables the CASTOR Optimization Engine to generate timely recommendations for candidate paths that satisfy both network- and trust-related requirements defined in the established Secure Service Level Agreements (SSLAs).

As detailed in D4.1 [20], the Global TAF evaluates various trust properties, including integrity, availability, confidentiality, and robustness of nodes, links, or entire paths. By grouping all trust relationships by trust property and creating trust networks, called trust models, the Global TAF can update trust relationships as the network topology evolves, and derive the corresponding ATL values. To compute the final ATLs, each trust relationship is assigned a trust opinion, expressed as a subjective logic opinion. This opinion captures the level of belief and uncertainty regarding an assertion about a supported trust property (e.g., whether the integrity of the ingress router is preserved). The derivation of these trust opinions relies much on the trustworthiness evidence collected from the target router functions being monitored. The impact of each type of evidence on the belief and uncertainty of a trust opinion depends on the threat model and risk posture of each router element. Consequently, robust mechanisms—namely, Trust Sources—are required to securely evaluate the collected traces from the target environment and provide trustworthiness evidence that can be consumed by the CASTOR Trust Assessment Framework, either by the Global TAF or the Local TAF agents.

Overall, there can be different types of Trust Sources that can interact with a Trust Assessment entity (i.e., with the Trust Source Manager subcomponent of a Local TAF agent or the Global TAF) depending on the type of evidence (e.g., evidence on the host routing element operation and/or at the networking/application layer) that they produce or their placement network architecture (i.e., locally instantiated as part of a router or maintained at the orchestration layer). In what follows, we present the two default Trust Sources that we consider as part of the CASTOR TNDE ecosystem for capturing the minimal yet sufficient set of static and runtime properties pertaining to both the host router operation and the network functions and routines[1].

[1] In CASTOR, in addition to these two Trust Sources, the Local TAF agents are also considered a Trust Source for the Global TAF within the federated trust assessment framework; further details are provided in D4.1.

4.2.1 "*Device State*" Configuration as a Trustworthiness Source

As highlighted in the threat model analysis of Chapter 5, some of the most critical attacks involve tampering with router configurations — such as TM.3.1, where a router is compromised to inject crafted routes, or TM.3.7, where a configuration downgrade could lead to insecure communication channels — or compromising the router software stack, as in TM.3.4, where a vulnerability may be exploited to undermine software integrity. These threats underscore the need for mechanisms to securely collect and report the overall state of a router, enabling external verifiers—such as the Service Orchestrator or adjacent routers—to confirm that the router has not been tampered with.

To address this, CASTOR leverages the Attestation Source as a key Trust Source, capable of detecting such violations and notifying the CASTOR Trust Assessment Framework to update trust evaluations accordingly. The foundation of the Attestation Source is a novel attestation scheme, namely the Configuration Integrity Verification (CIV), providing verifiable evidence on the integrity and correctness of deployed TNDEs and TNDI router functions. This implicit attestation scheme allows a prover TNDI to share evidence to an external verifier without disclosing sensitive information pertaining to its platform or router state. The key characteristic of this protocol is the provisioning of flexible key restriction usage policies that allow the correct signing of an attestation report if and only if the prover network element is in a correct state. Specifically, by conditioning the TNDI's ability to attest to whether its configurations are authorized by the Service Orchestrator, this scheme allows arbitrary verifiers to assess the integrity of other TNDIs without knowing their internal state, thereby alleviating critical trust assumptions regarding the verifiers' trustworthiness (see **??** for challenges in multi-verifier settings). An initial design of this attestation scheme is already published as part of CASTOR's work [38]. In D3.2 [23], the goal is to document the detailed attestation protocol with enhanced capabilities on providing additional evidence on the key restriction usage policy that has been enforced.

As CASTOR moves toward runtime attestation capabilities, the Attestation Source must interact with the Tracing Hub to collect critical traces from the Trace Units. These measurements are aggregated to construct a single digest that captures all evaluated configuration and integrity data. The resulting digest can then be compared against the key restriction usage policy enforced in the TNDI by the Service Orchestrator and check whether it can access the corresponding TNDI-specific attestation key. To achieve even higher levels of assurance, CASTOR envisions that the supported Trace Units will be capable of tracing low-level events that reflect the overall behaviour of a TNDI while executing critical network functions. This goes beyond configuration integrity verification and revolves around a more demanding attestation primitive that focuses on capturing the overall control flow of a target network function. As such attestation schemes would increase the level of complexity in the Attestation Source and would pose a significant performance overhead in the overall router operation, we need to find alternative Trust Sources that are able to capture such threats that could potentially pass undetected from the Attestation Source as they may not tamper with the configuration. An example of such a threat, discussed in Chapter 10, involves the BGP Update process of a router. An incoming crafted packet could cause the router to establish a malicious route. This type of attack would not be detected by the Attestation Source, since the BGP Update message does not compromise the integrity of the router's software stack. This requires a Trust Source capable of processing networking evidence—such as traces from continuous monitoring of the BGP Update process—and detecting any divergence from the expected nominal behaviour.

4.2.2 "*Finite-State Machine-expressed*" Behaviour as a Trustworthiness Source

While the standard attestation source provides a static evaluation of the configuration of the router in its current state, the FSM-based Tust Source provides a dynamic integrity evaluation of the router, considering the actions, their relations and evolution in time performed within it. In CASTOR, these actions are identified in the form of system calls collected by the tracing layer at runtime, thoroughly discussed in deliverable D2.1 [22]. While the decision of working with system calls is a design choice made in CASTOR, the behavioural trust source is not limited to them and can be extended to include, for example, system logs, user defined function calls, and system commands, but this go beyond the scope of the current project. The adoption of runtime behavioural monitoring solutions is strongly recommended to provide ongoing assurance of the integrity and correct operation of a router throughout its deployment.

By enabling continuous verification during normal operation, this approach improves the security of the device itself, as well as can contribute to the resilience of the surrounding network ensuring that the router and its services remain reliable over time helps maintain its trust and the trust across the full network. To ensure that the FSM-based trust source is effective and efficient, ad-hoc tailored models will be trained keeping in consideration the computational constraints tied to the nature of the routers under analysis as well as the threat models designed during the project, discussed in deliverable D3.2 [23]. The training will be done offline, not on the router itself to not impact their standard operational behaviour, based on real runtime traces collected in a safe environment and optimised based on the original work presented in [6] and extended to one of its most recent version in [42] in which the L* algorithm and state models minimisation are described. Each model generation process must strike an appropriate balance between accuracy and efficiency, while also preventing over-fitting to specific scenarios. Achieving higher accuracy often requires collecting a larger amount of runtime trace information, which can increase evaluation costs and introduce additional system overhead. At the same time, sufficient model accuracy is essential to produce reliable and meaningful attestation evidence, making this tradeoff a central consideration in model design. We envision to be able to validate the integrity of the operations performed at system calls level when configuration of the router are been requested, for example, modifications of the ip tables and routing configurations. More information are described in Chapter 11.

Figure 4.1 illustrates an example of a trained FSM on the behaviour of a service deployed on a Linux machine. The service under evaluation performs a series of file management routines (i.e., open/read/write actions). This exemplary scenario is extremely relevant in the context of networking functions, as this resembles the management operations that are carried out throughout different networking operations and have as a consequence the update of critical network configuration artifacts such as route and forwarding tables, Access Control Lists (ACLs), BGP policy files etc.

The state model in the figure captures the flow of interactions of the system calls performed by the operating system representing the expected behaviour of the service under analysis. Each transition between the states represents a valid system call of the captured behaviour, while each state is an abstract internal representation of the current state of the service. While the *ACCEPT* state is a state that can be considered a proper termination of the correct behaviour of the service, the *OK* expects more system calls to be executed to be accepted as a correct behaviour. The transitions represented in the image are the ones that are captured as part of a correct behaviour, any other transition, and therefore system calls, that are not printed in the flow are assumed to lead to the *REJECT* state which represents a malicious behavioural execution of the service.

In the context of CASTOR, this learning process will be investigated for critical network operations. Starting from the full design space of possible routing behaviors, captured by the critical network functions identified in the Threat Model (Chapter 10), CASTOR aims to derive finite-state models that enable efficient monitoring of router behaviour. Building on the previous example, which illustrates FSM-based detection for a generic Linux service, a concrete manifestation of this paradigm can be identified in the context of CASTOR and trusted path routing. Specifically, a TNDI implementing routing functionality processes and maintains a link map file whose evolution follows well-defined patterns under normal operation. Any modification of this file—such as removing trace links—can indicate manipulation associated with in-network replay suppression attacks, as described in [52]. CASTOR can detect such modifications using FSM to introspect the link map file. As can be seen from Figure 4.1, the number of states required to characterize a given function could explode even when monitoring relatively simple operations. Therefore, the key challenges to be addressed within CASTOR will be: first, to accurately capture the low-level interactions that characterize critical routing functions; and second, to assess how state-model abstraction influences both detection accuracy and the overhead imposed.

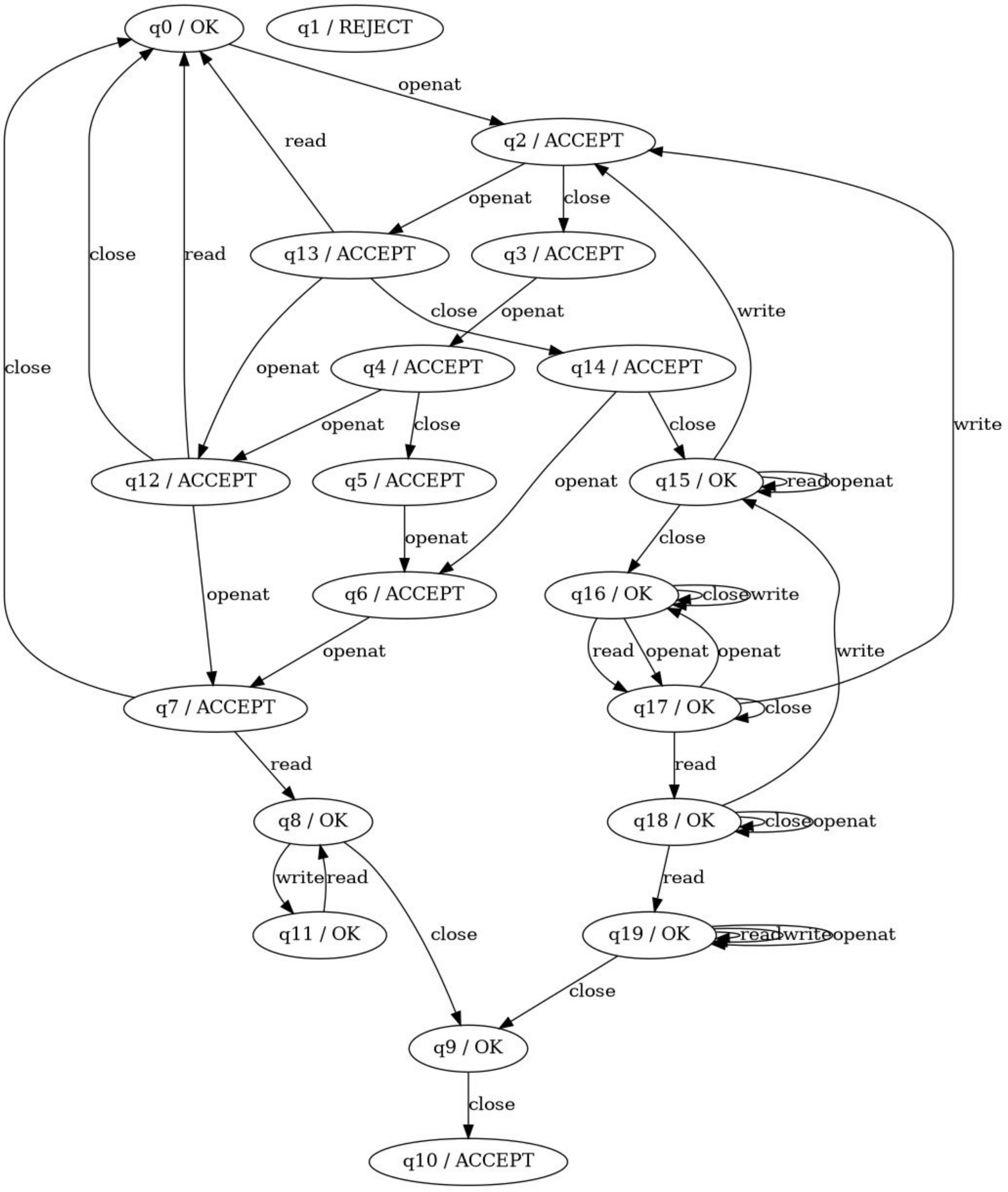


Figure 4.1: Trained FSM of a service behaviour

Chapter 5

High-Level Overview of Threat Modelling in the Routing Plane

The CASTOR project aims to develop an innovative system for **securing routing paths in next generation environments**, such as B5G/6G networks and cloud infrastructures [4]. A network topology may consist solely of physical routers, solely of virtualized routers (vRouters), or a hybrid combination of both. However, given the community's ongoing shift towards virtualization [5], our focus is primarily on routing functionality instantiated as virtual functions deployed on infrastructure elements. This design choice is made deliberately to support experimentation and exploitation scenarios, and it directly informs the threat model used in CASTOR. The adoption of virtualization introduces additional attack surfaces and opens up new threats, which are therefore explicitly considered as part of the overall threat model, where the security of a vRouter now depends not only on the routing software but also on the entire underlying virtualization platform [32]. The **CASTOR threat model serves two purposes**. First, it **defines the RTL for Trust Assessment**, as documented in Deliverable D4.1 [20], based on identified threats, risks, and their impact and likelihood, as well as evidence of the correct execution of mitigating security mechanisms. Second, it **determines the type of evidence to be collected** by the trace units (see more details regarding the network and systematic evidence types in Chapter 4). Based on this scope, CASTOR initially focuses on the routing and network layers.

CASTOR's virtualized routers can be implemented with two architectures: (a) **Virtual Machine (VMs)-based:** Each virtualized router runs inside a complete Virtual Machine, with its own operating system (kernel), isolated from other VMs through a Hypervisor (like KVM, VMware ESXi). This approach provides strong isolation [62]; (b) **Container-based:** Each virtualized router runs as a container (e.g., Docker, Kubernetes pod) that shares the kernel of the host's operating system (OS) with the other containers. The isolation is provided from OS's mechanisms (such as namespaces, cgroups). This approach is lighter and faster on booting, but isolation is considered to be less strong, in general, than that of VMs [56]. VMs and containers both enable routing software (e.g., BGP, OSPF) to run on general-purpose computing platforms instead of specialized hardware. In both cases, they rely on the host's physical network, typically using virtual switches (such as Open vSwitch or Linux Bridge) and virtual network interfaces (vNICs) to connect the vRouter to the rest of the network [4]. From a management perspective, both architectures also have orchestration systems, such as Kubernetes for containers, OpenStack for VMs, or NFV MANO for broader network-function orchestration [5]. The security and performance of this virtual networking stack are critical in both architectures. While the transition to virtualized router instances provides huge advantages in flexibility, scalability and fast deployment, concurrently expands dramatically the attack surface and imports new, complex threats [4]. The CASTOR model does not have to face only the **classic threats** of routing protocols but the **entire threat model on modern computing systems and cloud environments**.

As defined by CASTOR's goals - focusing on routing and the corresponding services - the threat analysis mainly concerns the **Network Layer (Layer 3)** and the **upper layers that support functions on man-**

agement and orchestration. This layer is responsible for routing and Traffic Engineering (TE), which together establish network paths. As defined in D2.1 [22], this is also where the CASTOR Traffic Engineering Policy Engine operates. The **Traffic Engineering Policy Engine** is CASTOR's main **control plane** infrastructure component offering **control services for the path exploration** process resulting to the enforcement of the optimal traffic engineering policies to enable continuous maintenance of (S)SLA compliance. In a nutshell, Layer 3 is responsible for logical addressing, routing, packetizing, fragmentation and reassembly. Usually in cloud environment threat models, threats regarding the Physical Layer (Layer 1), the Data Link Layer (Layer 2) and the physical compromise attacks (e.g., stolen equipment, physical cable monitoring) are considered out of scope. CASTOR similarly presupposes that the security of these lower layers is ensured by the infrastructure provider [4]. Consequently, the focus is on the logical, software and configuration threats that can compromise the integrity of virtualized nodes.

Routers are the key-devices of the Network Layer, because they function as a transport node between different networks. Routing tables can be formed manually (static routers), but in most networks they are dynamically updated through **routing protocols**. These protocols allow routers to exchange information about the network topology and calculate the optimal path. They are distinct from Interior Gateway Protocols (IGPs) and Exterior Gateway Protocols (EGPs). IGPs are used for routing inside an Autonomous System (AS - a network under a unified management), such as Open Shortest Path First (OSPF) - a link-state protocol where each router builds a full map of the AS, and Intermediate System to Intermediate System (IS-IS) - another link-state protocol, similar to OSPF, often in server's networks [32]. In contrast, EGPs are used to route between different ASs. The dominant protocol is the Border Gateway Protocol (BGP) - a path-vector protocol, which does not focus on just the fastest path but mostly on applying routing policies between domains (ASes) that constitute the web. It is the protocol that keeps the global Internet unified [4].

Our first step towards providing a detailed threat modelling is to primarily focus on the **threat modelling of the Network Layer** (Layer 3), which is the core of connectivity. Its basic function is routing: determination of optimal path for data packets based on logical IP addresses, and is based on routers, which communicate through routing protocols (BGP, OSPF) and maintain dynamic maps - routing tables. However, this logical function is implemented on computing systems (hosts). Routers (either physical or VNFs), controllers (SDN microcontrollers, PCEs) and attestation systems are basically computing systems. This dependency means that the **security of the Network Layer is inseparably connected with the security of the hosts**. Consequently, secure routing demands analysing threats that aim for these hosts and thus a point of view from the **host-level threat analysis** has to be adopted. For systemically analysing this attack surface, six distinct **Threat Domains** are classified and cover the whole technological stack [5]. Refer to Table 11.1 in Chapter 11 for the detailed threat catalogue ordered by these six threat domains.

- **Virtualization** - Virtualization & Container Layer Threats
- **Host OS** - Host OS & Software Layer Threats
- **Application** - Application & Service Layer Threats
- **Auth** - Authentication, Access & Identity Threats
- **Network** - Network & Protocol (Control/Data Plane) Threats(Focusing on host interactions)
- **Config** - Configuration & Information Threats

CASTOR acknowledges this threat model, but it does not aim to completely eliminate threats, as such a thing is impossible. In contrast, its goal is to build a framework that allows the establishment of **trustworthy routing paths despite the danger of compromise**, by using advanced security mechanisms. Detailed analysis of the threats that follow is the necessary first step in designing defence mechanisms.

Thus, this chapter focuses on providing an **overview regarding the detailed threat model in the control service and routing and network layer**. In this first analysis, we list the key threats that target functionalities relevant to the network mechanisms (e.g., establishment, advertisement, re-establishment of a routing path) either within an AS (intra-domain) or different ASs (inter-domain). More information regarding our detailed threat model is documented in Chapter 10. Also, CASTOR focuses on the source-based routing and especially to the segment routing (a source-based routing technique). Thus, CASTOR must be well aligned to the latest industrial efforts and to consider threats and vulnerabilities that can affect the most prominent protocols of both the inter-domain (such as the PGP) and the intra-domain (such as the Flex-Algo) relevant to segment routing. The reasoning behind CASTOR's threat model is that it will **guide and dictate the type of traces and evidence that needs to be monitored during runtime** with the CASTOR Tracing Hub (see Chapter 8). Essentially, this threat modelling will lead to the definition of the Required Trust Level (RTL) per router for a specific type of property (e.g., integrity, resilience, etc.), as already defined in the D4.1, since the definition of the RTL is directly interlinked to evidence for threats and applied security controls. Also, since CASTOR focuses on segment routing and is agnostic to the underline routing protocol, in our detailed threat model we cluster our threats in order to perform the evidence mapping and we do not particularly focus on covering all underline protocol modalities.

Particular emphasis will be placed on the **router-reflection attack** (TM3.13) [52]. Such an attack can be launched using a compromised router where the adversary leverages services that do not perform end-to-end replay detection such as UDP-based services (e.g., DNS, NTP), finds the routing bottlenecks of the target region and replays requests whose responses will target these bottleneck links on the return path.

Tables 5.1 and 5.2 summarize the identified threats in the control service and network layer respectively. Threats of Table 5.2 will further elaborated in Chapter 10.

Threats in the Control Service		
ID	**Title**	**Description**
TM.2.1	SDN Spoofing	The controller provides an abstraction to applications so that the applications can read/write network state, which is effectively a level of network control. If an attacker impersonated a controller/application, it could gain access to network resources and manipulate the network operation.
TM.2.2	SDN Tampering	If an attacker is able to hijack the controller, then it would effectively have control over the entire system. From this privileged position, the attacker can insert or modify flow rules on packet forwarding in the network devices - and general fraudulent rule insertion, which would allow packets to be steered through the network to the attacker's advantage.
TM.2.3	SDN Information Disclosure	It is possible for an attacker to determine the action applied to specific packet types by means of packet processing timing analysis. The attacker can therefore discover the proactive/reactive configuration of the switch.
TM.2.4	SDN DoS	In the case of SDN, the network devices require access to the control plane to receive traffic management instructions and traffic across the network requires access to the network device flow tables to dictate traffic management policies. The data-control plane interface and the network device flow table are therefore points of vulnerability to DoS attack.
TM.2.5	SDN Elevation of privilege	The controller provides an abstraction to applications so that the applications can read/write network state, which is effectively a level of network control that can be heavily impacted when an elevation of privilege occurs.
TM.2.6	Malicious North-bound applications	Given that the controller acts as an abstraction from the data plane for the applications and that SDN enables 3rd party applications to be integrated into the architecture, a malicious application could have as much of a detrimental effect on the network as a compromised controller.

TM.2.7	SDN Improper configuration	Network security policies and protocols are continuously updated as new network vulnerabilities are detected. Many of these will apply to the layers and interfaces of the SDN framework. If such security policies are not implemented properly or, are disabled without understanding the security implications of the deployment scenario it may lead to network breakdown. In an SDN-based network, it will be important for network operators to enforce the implementation of policies such as Transport Layer Security (TLS) and regularly update those policies.
TM.2.8	SDN Repudiation	In an SDN architecture the interception and alteration of SDN control plane packets, by a rogue SDN controller that attempts to alter configurations of network elements can impact fundamental attributes of a secure communication network and can be considered a repudiation threat.
TM.2.9	VNF Improper Input Validation	When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.
TM.2.10	VNF Memory-related vulnerabilities	Memory-related vulnerabilities, depending on which part of the code it exists, may allow attackers to read sensitive information from other memory locations or cause a crash. As a result, confidentiality is breached and penetration phase commences.
TM.2.11	VNF Dynamic memory deallocation	The usage of previously freed memory can have any number of adverse consequences, ranging from the corruption of valid data to the execution of arbitrary code, depending on the instantiation and timing of the flaw.
TM.2.12	VNF Vulnerable software components	In VNFs vulnerable software include general software vulnerabilities and weaknesses that are escalate by the dynamic nature of virtualization.
TM.2.13	VNF Memory buffer bounds	An application performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer (CWE-119). As a result, an attacker may exploit such operations and execute malicious code in order to change the flow control, read sensitive data or even cause the system to crash.
TM.2.14	VNF Exposure of Sensitive Information	Exposure of sensitive information on a VNF code happens when the system exposes sensitive information to an actor that is not explicitly authorized to have access to that information (CWE-200).
TM.2.15	VNF Authentication & Authorization broken access	Improper authentication and broken access authorization happen when the software application does not perform an authorization control check every time an actor attempts to access a resource or perform an action. As a result, users might be able to access data or perform actions that should not be originally allowed to.
TM.2.16	VNF Insufficiently Protected Credentials	When an application transmits or stores authentication credentials using an insecure method, unauthorized interception, retrieval and Man-in-the-Middle (MiTM) attacks may occurs.
TM.2.17	VNF Insufficient logging & state monitoring control	Insufficient logging of software execution stages – events (e.g. modification, tracking, or checking privileges for an actor) and monitoring its state (e.g. allocation and control of a limited resource) together with an ineffective policy of incident response, allows attackers to continue attacking a system that is already under attack.
TM.2.18	VNF Insecure deserialization of untrusted data	Remote code execution can occur when the application uses code that deserializes untrusted data without sufficiently verifying that the resulting data will be valid. The data Integrity and service availability is impacted by this threat (CWE-502).

TM.2.19	VNF Security misconfiguration	A security misconfiguration threat typically arises from various oversights and weaknesses in system setup and maintenance. These include insecure default configurations that are often left unchanged, incomplete or ad hoc configurations that fail to follow standardized security practices, and open cloud storage that can expose sensitive data to unauthorized access.
TM.2.20	VNF Container Escape	In VNFs deployed as containers, attackers may exploit container runtime flaws to escape to the host, gaining unauthorized access to the broader system.
TM.2.21	SDN Controller Resource Exhaustion	Attackers may overwhelm the SDN controller with API calls or control messages, degrading performance or triggering a crash.
TM.2.22	Trust Boundary Violation in Multi-Tenant Environments	In cloud environments with multiple tenants sharing SDN/VNF components, insufficient isolation can lead to unauthorized data access or interference between tenants.
TM.2.23	VNF Dependency Manipulation	VNFs depending on libraries or services may be compromised through upstream manipulation (e.g., poisoned libraries or outdated services).
TM.2.24	Misconfigured VNF Chaining (Service Function Chaining)	Misrouting or configuration errors in chained VNFs can allow traffic to bypass security functions (e.g., IDS/IPS), or expose it to unauthorized inspection.
TM.2.25	Time Synchronization Attacks	Desynchronizing time sources across infrastructure components can disrupt logs, break certificates, or enable replay attacks.

Table 5.1: Threats in the Control Service

Threats in Network and Routing Layer (Network Protocols)			
ID	**Title**	**Description**	**Security Controls**
TM.3.1	BGP Prefix Hijacking	Routers maliciously announce IPs/paths of other domains	tba. (prefix / AS path filtering)
TM.3.2	Active Wiretapping	Unauthorized interception and modification of BGP session data (MITM)	Session Encryption, Authentication (e.g., TCP-AO)
TM.3.3	Attacks on BGP Routers	Exploiting router vulnerabilities to inject malicious/unauthorized route attributes	Router Hardening, BGP Security Extensions (e.g., BGPsec)
TM.3.4	Compromise of Management Systems	Unauthorized access to routing management systems	Secure Management Protocols, Access Control
TM.3.5	RPKI Repository Attacks	Tampering with repositories hosting RPKI data	Repository Integrity Checks, Repository Redundancy, Validation Caching
TM.3.6	Protocol DoS	Injection of crafted packets spoofing OSPF/ IS-IS/ LDP hellos, IGMP/ PIM/ Spanning-Tree Messages	Infrastructure ACLs, FlowSpec Drops, Disabling Protocols per Interface
TM.3.7	Inter-Domain Border Security Exploitation	Inter-Domain Border Attacks specifically target the critical transition points between different administrative domains in the computing continuum. These boundaries represent natural security perimeters where trust models, security policies, and enforcement mechanisms often change significantly.	CASTOR's inter-domain FAD extensions verify and enforce trust requirements across domain boundaries, ensuring continuous trust verification during cross-domain communications.

TM.3.8	Flex-Algo Manipulation	Malicious actors exploit the Flexible Algorithm (Flex-Algo) mechanism in modern routing protocols by tampering with algorithm definitions or their advertisements. These attacks target the integrity of path selection by manipulating parameters that influence traffic engineering decisions, causing traffic to be routed through compromised network segments.	Cryptographically authenticated and integrity-protected Flex-Algo advertisements, pre-deployment validation of algorithm definitions, CASTOR's verifiable trust extensions, monitoring for anomalous routing patterns, and multi-source validation of received routing information.
TM.3.9	Interception of PCE requests or responses	Unauthorized access to monitor, alter or inject messages between the PCE and PCC in order to obtain sensitive information and influence path decisions.	Authentication, Integrity Checks, Session Encryption
TM.3.10	Impersonation of PCE or PCC	Unauthorized impersonation of an attacker to participate in path computation element communication protocol communications.	Authentication, Monitoring
TM.3.11	Falsification of TE information, policy information, or capabilities	False or manipulated TE data, policies, advertisements to PCE/PCC such as incorrect bandwidth availability, misleading topology attributes, forged policies etc.	Authentication, Monitoring, Configuration Validation
TM.3.12	Denial-of-service attacks on PCE or PCE communication mechanisms	DoS attack disrupting the PCE/PCC mechanism leading to traffic setup failures, degraded network peformance or loss of service.	Monitoring, Configuration Validation
TM.3.13	Router-reflection attack	Replay attacks launched using a compromised router that degrades or blocks the connectivity (legitimate traffic) of a remote Internet region just by replaying packets.	Monitoring

Table 5.2: Threats in the Network Layer

Chapter 6

Engineering Stories for Trust Management Features of CASTOR

CASTOR's trust-aware approach not only considers routing and control-plane decisions, but also depends critically on robust lifecycle management of network elements. As part of the overall CASTOR framework in D2.1 [22], three out of the four defined phases relate to the overall management of the underlying infrastructure layer (i.e., network topology): i) Proactive, ii) Preparedness and iii) Reactive. The *Proactive* phase addresses all the aspects of the secure enrolment of a new network element in the topology: from the initial attestation of the underlying platform up to the provisioning of the necessary cryptographic material in order to interact with the existing network topology and establish communication links. Subsequently, the *Preparedness* phase focuses on the initial provisioning of the orchestration layer to enable the offering of traffic engineering policies with certain network and trust guaranties. From the perspective of the network elements, this phase may include the re-configuration of specific policies (through the Trust Policy as explained in D4.1 [20] allowing the re-programmability of the CASTOR TCB in order to attain an updated level of assurance (e.g., in case of the identification of a zero-day vulnerability that requires more evidence to be collected from the underlying network element, or in case of configuration updates in order to satisfy an SSLA with higher trust guaranties). Finally, the *Reactive* phase refers to the mitigation strategies that are applied to a network element when there is a network- or trust-driven change in the topology.

In what follows, this chapter specifies important required behaviours of the CASTOR architecture in the form of engineering stories. These stories document the requirements of different entities within the CASTOR framework [22]. Each engineering story describes well-defined usages by specific roles or groups, together with their associated security and functional requirements. The goal is to provide a high-level description of the requirements that must be satisfied by the in-router CASTOR artifacts that will be designed and detailed in D3.2 [23]. In this deliverable, we capture the core TNDE capabilities, while all engineering stories covering the rest of the layers of the CASTOR architecture are presented in D4.1 [20], and D5.1 [21]. The engineering stories are grouped by the overall CASTOR phases that are defined in D2.1 [22]: Starting from the general/preliminary assumptions, the chapter delves into the engineering stories that are relevant for the proactive phase, focusing primarily on the secure onboarding of a new network element. Finally, this chapter presents the engineering stories that are relevant for both the preparedness and reactive phases, as they show how the TNDE would be reconfigured either in a proactive or a reactive manner.

NOTE: This deliverable focuses on the engineering stories that highlight in-router TNDE requirements. For this purpose, the CASTOR orchestration layer is represented by the term "Service Orchestrator", which serves as the primary Relying Party within an administrative domain, emphasizing that it constitutes a trusted entity. A detailed breakdown of all orchestration layer components is provided in D5.1 [21], while a critical discussion of the impact of weakening trust assumptions is included in D3.2 [23] as part of the design of the CASTOR cryptographic protocols.

6.1 Preliminary assumptions

Engineering Story-I

As a network device, I want to be equipped with a trusted computing base (TCB) so that I can enable the verifiable monitoring and sharing of evidence regarding my expected state.

Objective Each network device requires a trusted computing base (TCB) anchored in a (typically) hardware-based root of trust (RoT) to enable the secure collection and sharing of trustworthiness evidence on its routing nodes participating in a CASTOR trust network domain.

Motivation The Service Orchestrator needs to form a distributed trust network of network nodes and maintain a global, up-to-date view of their trust states in order to provide trusted path routing as a service. To enable this, the orchestrator must be able to remotely establish trust in each network device and continuously receive verifiable trustworthiness evidence on the routing nodes, i.e., TNDIs, of the network device. Therefore, CASTOR requires each network device to be equipped with a TCB anchored in a HW RoT. The TCB needs to manage the device's TNDIs participating in CASTOR's trust network and implement the required security mechanisms for trusted path routing, including the verifiable runtime monitoring and evidence generation on the TNDI's security state. As described in Section 3.2, CASTOR introduces the new notion of TNDIs and the TNDI-SP (the associated security protocol and mechanisms) to capture these concepts and implement them in the per-device CASTOR TCB. More information will be provided in Chapter 7 and Chapter 8.

Requirements As described in Section 3.3.1, each network device participating in a CASTOR trust network requires a platform/hardware root of trust that provides the security functions necessary to layer the CASTOR TCB on top of it (also see Engineering Story III).

Engineering Story-II

As a network device (e.g., router), I want to be configured with the necessary cryptographic keys so that I can authenticate myself to any intended party.

Objective To establish a cryptographically verifiable, unique digital identity that serves as the root of trust for the secure lifecycle management of a router within an administrative domain.

Motivation Establishming an initial set of cryptographic keys- provisioned at manufacturing time or first during initialization- is essential for the overall lifecycle management of a network element. Such key material, when anchored in hardware, provides a root of trust that enables the secure generation, protection and management of derived cryptographic keys that can be used for critical operations ranging from identity and authentication to attestation and secure communications. First, in accordance with Engineering Story I, the initially configured key material is associated with the hardware identity of the root of trust on top of which the CASTOR TCB is realized. This RoT identity enables the network element vendor to securely provision a device identity - e.g., during manufacturing - which can be then used for authentication purposes within a network topology. In addition, as presented in Section 3.3.2, this allows the derivation of all CASTOR-related key hierarchies either at the platform (i.e., TNDE) or at the router level (i.e., TNDI), thereby ensuring the secure lifecycle management of a router. Finally, as part of the initial cryptographic keys, we also consider additional application-related keys that are provided by the network element manufacturer or the domain operator to ensure the secure management of the element (secure firmware/software updates) or its system/network configuration.

Requirements Strong guarantees on the trustworthiness of a CASTOR enabled router are associated with the secure storage and management of all CASTOR-related keys throughout the router's lifecycle. For this purpose, the root of trust for storage is one key requirement in CASTOR's TCB as presented in

Section 3.3.1.

Engineering Story-III

As a network device (e.g., router), I want to be equipped with the necessary root of trust (RoT) functionalities so that I can securely measure, store, and present evidence on a TCB as required by CASTOR trust networks.

Objective Each network device joining a CASTOR trust network requires a platform/hardware RoT that provides the necessary capabilities to perform secure measurement, reporting, and storage to layer a CASTOR device-side TCB on top of it.

Motivation As described in Engineering Story I, the Service Orchestrator requires each network device to be equipped with a TCB that enables the orchestrator to establish trust in each device (see Engineering Story VII) and securely receive verifiable trustworthiness on the security states of the device's TNDIs. However, to enable trust establishment and secure operation of the TCB, the protection of the TCB must be anchored in a platform/hardware RoT that provides the necessary security functions, forming the basis for Engineering Story I. As detailed in Section 3.3.1, CASTOR requires the RoT in each network device to support secure measurement of the CASTOR TCB (e.g., trust extension services), verifiable reporting on the measurements to the Service Orchestrator, isolation of the TCB from untrusted components (e.g., the router stack and NOS of the device's TNDIs), and necessary secure storage, time, and counter functionalities.

Requirements This is a fundamental base requirement for each network device aiming to participate in a CASTOR trust network and thus trusted path routing.

Engineering Story-IV

As a TNDE, I want to generate a TNDE (platform) key bound to the underlying device platform and TNDE identity to enable authentication of TNDE and platform evidence.

Objective The TN-DSM of the TNDE must be able to securely generate a TNDE (platform) key for attestation purposes (see Section 3.3.2) that is bound to the underlying platform (RoT and isolation layer) and the TNDE identity (e.g., measurement), utilizing functionalities provided by the device's platform RoT.

Motivation CASTOR requires to establish trust in each network device that forms its trust network domain, as we will outline in Engineering Story VII. To enable the orchestrator to remotely identify a network device and verify the state of its TCB, the device must be able to authenticate using a cryptographic key bound to the platform and TCB running on the device. Otherwise, the orchestrator cannot securely verify that evidence on the device's state is trustworthy. Therefore, CASTOR will explore different ways for the CASTOR TCB to derive a TNDE platform key that is bound to the device platform (RoT, isolation layer) and the TNDE, based on the device's RoT. As the Trace Units will be dynamically managed by the TNDE (Section 3.2 and Chapter 8), they are not included in the binding. The specific way the bindings are achieved can depend on the available RoTs and TNDE implementation, and the binding to the TNDE might be achieved differently than the platform (device) binding. Therefore, CASTOR will explore multiple schemes based on different RoT instantiations (see Section 7.1.1 in Chapter 7).

Requirements This requires each device to be equipped with a platform/hardware RoT that provides the necessary measurement, reporting, and binding functions (Engineering Story III) and has been provisioned with respective cryptographic device keys (Engineering Story II). Furthermore, it requires a secure CASTOR TCB on the device (including the TNDE and thus TN-DSM) that will perform the key management operations to securely create the TNDE key and establish its binding policy. An overview of the device-side TCB and RoT requirements is provided in Section 3.2 and Section 3.3.1.

6.2 Proactive phase

Engineering Story-V

As a TNDE, I want to generate TNDI attestation and TNDI-SP communication keys that are bound to the TNDE, so that I can have fine-grained control over the lifecycle of TNDIs co-located on the same platform.

Objective The TN-DSM of the TNDE must be able to generate an attestation key for each TNDI of the underlying device and bind it to the TNDE key and, optionally, the TNDI, such that the TN-DSM can authenticate runtime evidence about the TNDIs. Furthermore, the TN-DSM must be able to securely establish TNDI-SP control and data channel communication keys and bind them to the TNDE key and, optionally, the associated TNDI (see Section 3.3.2), for secure communication with the CASTOR elements at the orchestration layer (especially Service Orchestrator, Global TAF, and DLT).

Motivation To provide trusted path routing, the Service Orchestrator must form a trust network of TNDIs for each of which it continuously receives trustworthiness evidence on the security state. This constitutes the primary step before allowing the network elements to, then, establish trusted links between each other, adhering to the IETF Trusted Path Routing principles [11] (e.g., routers sharing Stamped Passports with its adjacency before establishing a communication link). Forming a trust network and deploying trusted routing paths requires the Service Orchestrator to be able to securely communicate with the device TNDEs managing the TNDIs and receive verifiable evidence from them to maintain an up-to-date view of the TNDI trust states. Therefore, the TNDE of an infrastructure element must be able to securely derive attestation keys for its TNDIs and communication keys for the required TNDI-SP communication channels. Furthermore, the TNDE must be able to bind the keys to the TNDE and, optionally, the associated TNDIs (e.g., to the measurement or attestation key). CASTOR explores respective key generation and binding strategies (Section 3.3.2) in the context of its new TNDI and TNDI-SP concepts to satisfy these requirements, based on the functions of the device RoT. CASTOR explores different options for deriving and binding keys for the TNDI-SP control channels (for TNDI management and configuration) and data channels (for sharing TNDI evidence).

Requirements To allow for the generation of TNDI and TNDI-SP keys with the required bindings, each network device must be equipped with a CASTOR TCB (Engineering Story I) handling the key management for the TNDIs and TNDI-SP endpoints and an underlying platform RoT (Engineering Story III) supporting the required binding capabilities.

Engineering Story-VI

As a TNDE, I want to securely manage the TNDIs of a network device and enable the Service Orchestrator to establish a trusted connection to me in order to allow for the device platform and its TNDIs to join and onboard into the CASTOR domain.

Objective The TN-DSM of the TNDE must be able to securely identify and manage the TNDIs of the underlying network device. In addition, the TN-DSM must enable the Service Orchestrator to establish a trusted connection based on which it enrols the device (join phase) and its TNDIs (onboarding phase) into the CASTOR network domain.

Motivation The Service Orchestrator forms its trust network based on routing nodes for which a device-side TCB can enforce the necessary security mechanisms required for trusted path routing. As explained in Section 3.2, CASTOR introduces the notion of TNDIs for such routing nodes and the TNDI-SP for the respective security mechanisms, including the TNDI security state monitoring based on trustworthiness evidence. As a prerequisite, CASTOR therefore explores how the CASTOR device-side TCB can securely identify and manage TNDIs of the underlying network device. Furthermore, CASTOR explores how the

TCB can then expose a secure communication endpoint to the orchestrator through which subsequent JOIN (Engineering Story VII) and ONBOARDING (Engineering Story VIII) procedures to a CASTOR trust network can be performed.

Requirements The secure management of a device's TNDIs requires the device to be equipped with a secure CASTOR TCB (Engineering Story I) and underlying RoT (Engineering Story III). Furthermore, the device TNDE must be able to establish secure TNDI attestation and TNDI-SP channel keys (Engineering Story V) to enable verifiable communication with the Service Orchestrator.

Engineering Story-VII

As a Service Orchestrator, I want to establish trust in a network device so that I can add it to my domain (join phase) before onboarding any of its TNDIs into my network topology.

Objective To form a trust network, the Service Orchestrator must be able to establish trust in the TCB of a device and join it to its domain before starting to onboard the device's TNDIs into the network topology.

Motivation The Service Orchestrator forms a trust network of TNDIs to securely provide trusted path routing as a service. However, before the orchestrator can onboard TNDIs into the network topology (see Engineering Story VIII), the orchestrator must be able to establish trust in the underlying network devices and join them into the domain. Joining the device into the domain is the prerequisite as the device's TCB is responsible for managing the TNDIs and enforcing the security mechanisms (e.g., runtime monitoring) and evidence reporting on them. Therefore, CASTOR explores a new JOIN protocol as part of the CASTOR TNDI-SP control channel (see Section 3.2.1) that enables the orchestrator to securely establish trust in a network device's TCB based on a device's RoT and bootstrap necessary configurations and cryptographic keys to allow for secure onboarding of the device's TNDIs.

Requirements To enable the Service Orchestrator to establish trust in a network device and join it to its domain, each network device must be equipped with a TCB (Engineering Story I) that exposes a TNDI-SP control channel endpoint (Engineering Story VI) and that can report evidence on its security state in a verifiable way, based on an attestation key (Engineering Story IV). These device-side TCBs must be anchored in a trusted platform/hardware root of trust (Engineering Story III) that supports the required security functions, e.g., to measure and report the platform and/or TCB state.

Engineering Story-VIII

As a Service Orchestrator, I want to securely onboard a TNDI into my network domain before letting it participate in the trusted path routing so that I can enable secure TNDI monitoring and provision trustworthy links across the underlying infrastructure layer.

Objective The Service Orchestrator must be able to securely onboard TNDIs into its trust network topology, which sets up the required security mechanisms for each TNDI needed for CASTOR's trusted path routing.

Motivation The Service Orchestrator forms a distributed trust network of TNDIs for which it maintains a global and up-to-date trust state (i.e., through the Global Trust Assessment Framework). In this way, the orchestrator can provide trusted path routing as a service. However, this requires the orchestrator to be able to continuously receive trustworthiness evidence on the security state of each TNDI and to be able to deploy trusted paths through the TNDIs. Therefore, CASTOR explores a new ONBOARDING protocol as part of the CASTOR TNDI-SP control channel (see Section 3.2.1) that enables the orchestrator to securely integrate TNDIs of trusted network devices into its trust network topology. As part of the onboarding, the Service orchestrator must be able to configure the TNDE component of each infrastructure element to enforce the required TNDI-SP security mechanisms (e.g., runtime monitoring) and deploy a trust policy that defines the tracing and trustworthiness evidence required for the TNDI. Furthermore, it needs to enable the establishment of the TNDI-SP data channels for sharing the TNDI runtime evidence with the Service Orchestration layer services, and prepare the setup of secure links with neighbouring TNDIs (Engineering Story IX).

Requirements Before the Service orchestrator can onboard a TNDI into its trust network, the orchestrator needs to establish trust into the underlying device (Engineering Story VII). To enable that, the device must be equipped with a CASTOR TCB (Engineering Story I) that exposes a secure TNDI-SP control channel endpoint (Engineering Story VI) and the capability to establish the necessary cryptographic keys (Engineering Story V).

Engineering Story-IX

As a TNDI, I want to exchange authenticated trustworthiness evidence with neighbouring TNDIs so that I can enable the establishment of secure communication links.

Objective The objective is to enable a decentralized, peer-to-peer verification mechanism where TNDIs directly exchange and validate trustworthiness evidence before establishing or maintaining a data plane connection. This ensures that secure communication links are only formed between nodes that mutually meet the defined security policies, effectively extending the trust boundary hop-by-hop.

Motivation In a distributed routing environment, waiting for a central authority to validate every link can introduce unacceptable latency and a single point of failure. Therefore, TNDIs must be capable of establishing local trust relationships. If a neighbouring router has been compromised (e.g., running malicious firmware), establishing a standard OSPF/BGP session with it poses a severe risk of route hijacking or traffic interception. By mandating the exchange of authenticated evidence (such as Actual Trust Levels or attestation quotes) during the link negotiation phase, a router can autonomously reject connections from untrustworthy peers, as dictated by the IETF Trusted Path Routing framework. This capability is essential for creating a ”Trust Aware” topology where the physical link state is gated by the logical security state, ensuring the integrity of the data path at the very edge of the network. To this extent, CASTOR aims to extend the current Stamped Passport data model and incorporate additional runtime trustworthiness claims that can be exchanged between neighbouring network elements.

Requirements The primary requirement is the extension of standard link-establishment protocols (such as LLDP, BFD, or IKEv2) or the introduction of a dedicated ”Trust Handshake” protocol to carry signed trustworthiness evidence. This exchange must occur prior to the adjacency formation of routing protocols. The incoming evidence act as another Trust Source in the context of the Local TAF agent residing in the receiving TNDE and allows it to form an opinion on the trustworthiness of the transmitting TNDI. Furthermore, the system must support periodic re-verification to handle runtime state changes; if a neighbor's Actual Trust Level (ATL) drops below the threshold during an active session, the TNDI must be capable of dynamically tearing down the secure link and alerting the orchestration layer.

Engineering Story-X

As a Service Orchestrator, I want to have access to trust-related data from the underlying topology so that I can ensure the satisfaction of the established SSLAs in a secure and auditable manner.

Objective The objective is to maintain a holistic trust characterization of the entire network topology, allowing the Service Orchestrator to dynamically select the most trustworthy routing paths based on real-time network and trust information.

Motivation In the CASTOR framework, local node integrity is necessary but insufficient for secure routing; the Orchestrator requires a comprehensive, network-wide view to make informed path selection decisions. Without reliable receipt of the Actual Trust Levels (ATLs) and supporting evidence from every node, the Global TAF cannot accurately weight the trustworthiness of a specific route or detect nodes whose security posture has degraded. Furthermore, to ensure transparency and accountability, these trust levels must be immutably recorded in the DLT. This enables authorized third parties (e.g., manufacturers, domain providers) to get access to trust-related data exposed with the necessary level of abstraction. Eventually, this is what helps establish end-to-end services that span across multiple domains and maintain a common level of network and trust guarantees.

Requirements The system requires a secure, high-integrity transport mechanism capable of ingesting diverse telemetry streams, including attestation quotes, operational traces, and the calculated Actual Trust Levels (ATLs) from distributed TNDIs Engineering Story V. The Global TAF must be equipped to parse these incoming trust levels to dynamically update the network topology map in real-time. Simultaneously, an integration layer with the DLT is required to asynchronously commit these signed ATLs to the ledger, creating a permanent and tamper-evident history of the network's trust evolution without introducing latency to the routing decision process.

6.3 Preparedness and Reactive phase

Engineering Story-XI

As a TNDE, I want to be able to dynamically set up and (re-)configure my components (e.g., Tracing Hub, Local TAF Agent) to enable and adjust the enforcement of the TNDI-SP security mechanisms, including the secure generation of authenticated TNDI evidence, based on requests by the Service Orchestrator.

Objective The TNDE must be able to set up and dynamically (re-)configure all of its components and Trace Units based on requirements and information provided by the Service Orchestrator. The orchestrator must be able to push these configuration requirements as part of the device enrolment (join) and TNDI onboarding process, e.g., via the deployed trust policy (Engineering Story VIII), and via runtime configuration and policy update requests through the TNDI-SP control channel, post-onboarding.

Motivation The Service Orchestrator provides trusted path routing as a service by forming a trust network of TNDIs and enforcing trusted routing paths through them. This requires the orchestrator to be able to securely configure the TNDIs and their managing TNDEs as part of the device and TNDI enrolment (join and onboarding). Otherwise, the Service Orchestrator cannot establish trust into the network devices and enforce the TNDI-SP security mechanisms (e.g., TNDI monitoring and evidence generation) required for CASTOR's trusted path routing. In addition to the setup and configuration steps as part of the enrolment (Engineering Story VII and Engineering Story VIII), the Service Orchestrator also needs to be able to *dynamically* update TNDI-related configurations and policies of the TNDE (e.g., the required system- and network-level evidence) post-enrolment in order to react to network- or trust-related changes in the trust network (e.g., trust model updates or a TNDI compromise). Therefore, CASTOR explores different ways of how the TNDE can dynamically update the configurations and enforced security policies of its components, especially per-TNDI ones, based on TNDI-SP control requests by the Service Orchestrator. CASTOR aims at enabling the orchestrator to push configurations to the TN-DSM of the TNDE as part of the TNDI-SP join and onboarding processes and via TNDI-SP control messages at runtime, post-onboarding. CASTOR explores a variety of configuration dimensions for the TNDE, ranging from more general configurations, e.g., network addresses of orchestration layer services (e.g., Global TAF), to trust policies for the TNDI-SP security mechanisms (e.g., the required system- and network-level evidence per TNDI). Further information on the considered configuration options - e.g., to select Trace Units, tweak the tracing performance-security tradeoff, or adjust the evidence sharing (via the TNDI-SP data channels) - is provided in Chapter 7 and Chapter 8.

Requirements The Service Orchestrator requires each network device to be equipped with a CASTOR

TCB (especially TNDE) that implements the TNDI-SP (Engineering Story I), manages the device's TNDIs (Engineering Story VI), and enforces the orchestrator's configuration requests. Otherwise, the orchestrator cannot enrol the device (Engineering Story VII) and its TNDIs (Engineering Story VIII) as required for configuring their TNDI-SP security mechanisms for trusted path routing. Furthermore, the Service Orchestrator requires an active TNDI-SP control channel to the TN-DSM of a device to send configuration messages at runtime for one or multiple of the device's TNDIs.

Engineering Story-XII

As a network device (e.g,. router), I want to be equipped with the required lightweight monitoring technologies so that I can generate security evidence regarding my operational behaviour.

Objective To ensure the router's runtime operational integrity, each router will be deployed with ad-hoc lightweight monitors capable of validating the correctness and integrity of crucial router operations under analysis, leveraging runtime information collected by the tracing layer.

Motivation As previously presented in Section 4.2.2, these integrity monitors will represent one of the trust sources of the CASTOR Trust Assessment Framework. While the integrity monitors will focus on the dynamic behavioural side of the router, the attestation source will aim for its runtime - yet static - configuration. Eventually, this highlights the complementarity of the CASTOR Trust Sources which eventually offer wider perception to the overall Trust Assessment process. An important note to highlight is related to the router itself and what functionalities the monitors are addressing. When this integrity monitors are mentioned, their deployment is meant on each single router, which in the CASTOR framework refers to the virtual router, of which multiple instances can be instantiated in a single hardware server. With respect to the computational constraints imposed by the nature of the current system under analysis, these integrity monitors will be trained and developed in the form of Finite State Machines (FSMs). The choice of using this specific model comes from two main factors: (i) their intrinsic simple structure, which makes them computationally light to evaluate at runtime, and (ii) the optimisation mechanisms used during their training, described in [42], which reduces the number of states required to accurately depict the nominal behaviour being analysed. This will lead to the minimum computational overhead that the router is subjected to, while ensuring effective integrity measures to ensure the security of the router.

Requirements To enable the runtime integrity monitors, the tracing layer is required to capture the relevant information needed by the FSM-monitors to evaluate the behaviour of the operations performed by the router. The tracing layer will work in close collaboration with these monitors to collect the minimum set of the most relevant information needed to best capture the behaviours of the router to be monitored. This is crucial to not overwhelm the router's standard operational functionalities by collecting information that will not be used while still collecting enough information to ensure the validation of its integrity. The relevance of the information collected by the tracing layer needed to generate an effective FSM-based monitor is given not only by the system calls names that can be captured, but also, for example, by their moment in time in which they have been executed, information related to the arguments that were passed during their invocation, their returning values once their execution has been completed, information about the user executing the commands and their privileges on the router. All of this information forms the contextual information surrounding the system calls, enhancing the monitors training and runtime efficacy and efficiency.

Engineering Story-XIII

As a network element, I want to feature dynamic configuration capabilities for the correct installation of a software stack and establishment of the necessary keys so that I can participate in the overall network topology.

Objective The router needs to be equipped with the updatable and robust attestation primitives in order to share evidence about its configuration and software/firmware integrity at any point in its operational lifecycle. The CASTOR's Attestation Source is tasked to securely process and evaluate raw traces (Engineering Story XI), as dictated by an enforced Trust Policy. This allows for the generation of trustworthiness evidence that can be consumed by the Local TAF agent, the Global TAF at the Orchestration layer or even the CASTOR DLT for auditability purposes by authorized stakeholders.

Motivation As highlighted in the functional requirements of deliverable D2.1 [22], one of the CASTOR key functionalities relates to the possibility of the network elements to provide verifiable guarantees on the correctness of the routers configuration state. Through the CASTOR Attestation Source, such attestation evidence acts as a Trust Source in the overall Trust Assessment Framework, allowing the formation of trust opinions on router, link or even path level. Acting throughout the operational lifecycle of a TNDI, the Attestation Source needs to work in tandem with the tracing layer (i.e., the CASTOR Tracing Hub and the attached Tracing Units) in order to securely collect runtime traces that can be then used to provide attestation results with respect to the target security properties that need to be evaluated by the TAF (i.e., either the Local TAF agent within the TNDE or the Global TAF at the orchestration layer). Instead of engaging into a typical remote attestation scheme - where the Attestation Source shares its measurements with the requested verifier -, CASTOR envisions to explore novel implicit attestation mechanisms that aims to provide the necessary attestation evidence - i.e., quotes - without disclosing any sensitive information of the current state of the TNDI (or the associated TNDE). This is achieved through the appropriate provisioning of scoped attestation keys that can be used in the attestation protocol if and only if the TNDI is at the expected state. On one hand, through the enforcement of such key restriction usage policies, the TNDI is able to share its attestation evidence to any intended party: within the TNDE (e.g., Local TAF agent) and beyond it (e.g., Service Orchestrator, Global TAF, CASTOR DLT, neighbouring TNDIs). Within this multi-prover multi-verifier scheme, such an Attestation Source will provide robust attestation mechanisms for various reasons. First, given the heterogeneity in the infrastructure layer - i.e., multiple router vendors offering routers with different firmware/software characteristics - each verifier does not have to keep track of the expected state of its possible prover that it interacts with. In addition, any changes in the TNDI's state (e.g., software update or re-configuration) do not affect the attestation process except from the fact that the prover's attestation keys need to be re-scoped: updating the key restriction usage policy does not introduce the need to re-provision new attestation keys during runtime. Finally, as explained in Chapter 9, such attestation primitives can be used and extended in order to construct efficient proofs that characterize an entire sequence of TNDIs, forming trustworthiness evidence at a path level. Details on the attestation scheme are provided in D3.2 [23].

Requirements The TNDIs need to be backed by a hardware Root of Trust Engineering Story III, such as a Trusted Platform Module (TPM) or a Trusted Execution Environment (TEE). This RoT serves as the immutable anchor for the system, strictly required to provide secure, tamper-resistant storage for private keys and to execute cryptographic capabilities within a protected boundary. Building upon this RoT, the system must generate asymmetric key pairs locally to ensure non-repudiation, preventing private keys from ever being exposed in plaintext Engineering Story II. Finally, the realization of the implicit attestation scheme that is envisioned in the TNDE's Attestation Source implies the need that all verifiers have access to the necessary cryptographic material (e.g., public keys, X.509 certificates) in order to verify the attestation evidence - i.e., quotes - produced by a prover entity. In the CASTOR framework, this capability is provided by the Service Orchestrator.

Engineering Story-XIV

In CASTOR, as a global TAF, I want to form a single proof of all TNDEs' proofs (each TNDE can be associated with a physical router), in which all routers already formed a trusted routing path. In the single proof, the order of all corresponding routers in the path should be maintained.

Objective A router in an enforced trusted routing path (which has been established after ONBOARDING in Engineering Story VIII) needs to make use of a cryptographic primitive to prove the trust evidence to be collected by the global TAF. The global TAF should form a single claim after collecting multiple routers' proofs. In the single claim, the order of all routers in the path should be maintained. If a router is rogue or compromised in the path, the global TAF should be able to revoke it. When the setting is extended to multiple segments, routers at boundaries may selectively verify a certain subset of the aggregated claim. To prevent impersonation attack, a router's secret signing key for generating proofs should be bound to the identity key (TNDE's key or the TNDI).

Motivation CASTOR needs to establish a trusted routing path, in which there are multiple routers (act as provers) and verifiers. In a path, all routers' proofs must be collected by the global TAF to form a single proof, in which the order of all routers should be maintained. This proof serves as a cryptographic attestation that the router has correctly processed and forwarded the data according to the specified routing policy.

In inter-domain setting, partial verification of combined proofs should be guaranteed at each segment boundary to improve efficiency of proofs. Moreover, partial verification can prevent highly capable adversaries from invalidating trust by targeting isolated components. Furthermore, to ensure a whole trusted routing path, any corrupt router should be detected and subsequently revoked from the network. This necessitates mechanisms for identifying misbehaving routers through proof verification failures and enforcing their exclusion, thereby maintaining the overall integrity and trustworthiness of the routing infrastructure. To prevent impersonation attack, a router's secret key for generating a proof should be bound to TNDE's platform key.

Requirements In composite attestation, there are multiple functions. Here we split all requirements as two types, i.e., functional requirements and security requirements. For functional requirements, revocation and partial verification of proofs should be supported; To reduce communication cost, the size of aggregated proof should be as small as possible. For security requirements, we need to involve correctness and unforgeability described in detail as follows:

- Correctness of this scheme is required, which means a valid (aggregated) proof can pass the verification.
- Unforgeability: a proof/aggregated proof is required, which means an adversary without a valid secret key cannot generate a signature that can pass the verification. Any adversary cannot pretend to be a valid signer to pass the verification, which is associated the key-binding property, being against the impersonation attack.

Engineering Story-XV

As a network device (e.g., router), I want to encrypt ATL/any trust evidence and send the ciphertext to the orchestrator, who can compare ciphertexts to get the order of different ATLs without revealing the real values. Further the orchestrator should be able to give the lowest ATL for each domain.

Objective In order to achieve order of ATLs without revealing real values, firstly, each router should encrypt their ATL to get ciphertexts, which can protect the real ATLs. Then, the orchestrator can compare every two ciphertexts from two routers. This process can be repeated until the order of all routers' ATLs are obtained. The lowest ATL can be revealed.

Motivation In a trusted routing environment, each router maintains an ATL. When selecting or validating a routing path, it is often necessary to compare the ATLs of participating routers to make informed routing decisions. However, directly exposing the exact ATL values raises significant privacy and security concerns. Revealing precise trust levels could enable adversaries to target routers with lower ATLs, or allow competing network operators to gain insights into the internal trust posture of other domains. Therefore, a mechanism is required that enables meaningful comparison of ATLs while preserving the confidentiality of their actual values. (Delegatbale) Order-revealing encryption (DORE) provides a solution to this challenge by allowing ciphertexts to be compared without decryption. Under this approach, each router encrypts its ATL using an ORE scheme, producing a ciphertext that conceals the underlying value yet permits order comparisons. The orchestrator, responsible for coordinating the routing path evaluation, can then perform bitwise comparisons between any two ciphertexts to determine which router possesses the higher or lower ATL. By iteratively applying these comparisons, the orchestrator can establish a complete order of all routers' ATLs without ever learning their exact values. Once the ordering is determined, only the lowest ATL—representing the weakest link in the routing path—needs to be revealed, enabling trust-based decisions while minimising unnecessary information disclosure.

Requirements In this setting, there are some requirements, correctness, IND-OCPA and key-identity binding described as follows:

- Correctness of the ORE scheme is required, which means two valid ciphertexts associated with two messages can be compared.
- Indistinguishability under ordered chosen plaintext attack (IND-OCPA), which means an adversary cannot distinguish between a real ciphertext and a simulated ciphertext under the help of the information leakage.
- Key-binding means a router's ORE key should be bound to the TNDI/TNDE's platform key.

Chapter 7

CASTOR Secure Routing Plane Management via Trust Network Device Extensions (TNDE)

7.1 TNDE Overview and the Role of the TN-DSM

The CASTOR Service Orchestrator forms a trust network of TNDIs in order to provide trusted path routing. It cooperates with trusted components on each network device, which implement the TNDI-SP security mechanisms (Section 3.2.1) to manage the device's TNDIs and to continuously provide the Orchestrator with trustworthiness information about them. The trusted onboarding of TNDIs and the secure stream of authenticated TNDI trust information enable the Orchestrator to maintain a global, up-to-date view of the trust network and, based on this, to calculate and deploy trusted network paths. Therefore, each network device joining the trust network needs to be equipped with **CASTOR's Trust Network Device Extensions (TNDE)**, which form the main component of CASTOR's on-device trusted computing base (TCB), building on an underlying platform root of trust (RoT), as outlined in Chapter 3. In the remainder of this chapter, we provide an overview of the roles and technical requirements of the TNDE, specifically focusing on the central role of the Trust Network Device Security Manager (TN-DSM) for managing the device TNDIs and TNDI-SP security mechanisms. In Chapter 8, we then focus more on the multi-level tracing architecture, with the TNDE's Tracing Hub as the central component in control.

As described in Section 3.2, the TNDE is the main trusted component on each network device, enabling the CASTOR Service Orchestrator to establish trust (bootstrap and maintain the *required trust level*) in the device and onboard it in the domain topology. The TNDE is responsible for managing the TNDIs of the underlying device (e.g., physical router or vRouter-hosting server) and implementing the TNDI-SP security mechanisms on these TNDIs. That is, the TNDE enables the secure enrolment of the device and its TNDIs into the trust network; the monitoring and sharing of TNDI trustworthiness evidence; and the establishment of secure communication links for each TNDI, interfacing with the CASTOR Service Orchestration Layer and neighbouring TNDIs as needed. As shown in Figure 3.1, the TNDE consists of multiple sub-components that are part of the TCB and contribute to the TNDI-SP security mechanisms, including the Trust Network Device Security Manager (TN-DSM), the Tracing Hub, a set of Trust Sources, the Local Trust Assessment Framework Agent (Local TAF Agent), and the auxiliary TPL Data Connector (not part of the TCB).

The TN-DSM is the central entity of the TNDE exposing the main communication interfaces of the device through which the CASTOR Service Orchestration Layer interacts with the TNDE and its TNDIs. The TN-DSM is responsible for managing the device TNDIs and maintaining their security states. Furthermore, the TN-DSM handles the TNDI-SP control channel messages (as the device-side control channel endpoint), performing the necessary device enrolment (join) and TNDI onboarding steps, including the configuration of the TNDIs and other TNDE components, on request from the Service Orchestrator. This also includes the necessary key management steps required to provide verifiable evidence on the platform, TNDE, and TNDIs (see Chapter 6) and to establish the necessary TNDI-SP data channels to expose the evidence and trust reports to the service orchestration layer (including the CASTOR blockchain infrastructure / DLT) or neighbouring TNDIs. That is, the TN-DSM serves as the central coordinator of the TNDE, configuring and managing the operations and interactions of the Tracing Hub, Trust Sources, and Local TAF Agent to implement the TNDI-SP and thus enable participation in CASTOR's trusted path routing domain.

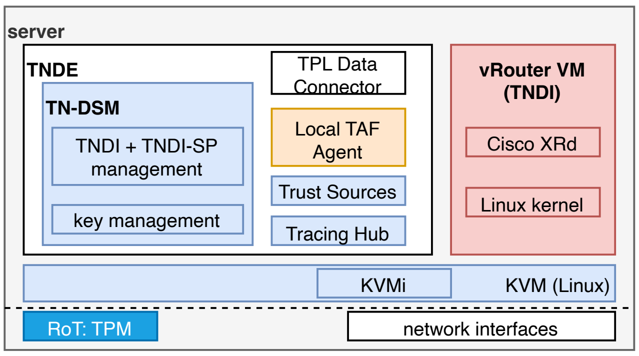
Information on how CASTOR performs the construction and enforcement of trusted routing paths throughout the TNDIs network will be provided in D5.1. Note that the TNDI-SP mechanisms will first focus (as part of the first CASTOR integrated framework detailed in D6.1) on the establishment of trust in TNDIs and the collection and sharing of trust-related TNDI evidence between TNDIs and with the service orchestration layer. As part of version two, the TNDI-SP will then be extended to also manifest the enforcement of trusted route configurations (either explicit paths or segment routing policies) through the TNDI-located PCCs (path computation clients).

Note that, in this chapter, TNDI-SP operations and requirements are described from the perspective of the TNDE, and in particular its TN-DSM component, even though some of the listed requirements originate from the TNDI-SP concept itself rather than from any specific TNDE realisation. Furthermore, as stated in Section 3.2.1, the TNDI-SP currently focuses on establishing trust in TNDIs and providing trust-related runtime evidence on them to the service orchestration layer and neighbouring TNDIs. As part of version two of the CASTOR integrated framework, CASTOR will explore extending the TNDI-SP to also handle the enforcement of trusted routing configurations (trusted paths) through the TNDI-located PCCs (path computation clients). Information on CASTOR's trusted path construction and deployment is out of scope for D3.1 and instead covered in D5.1.

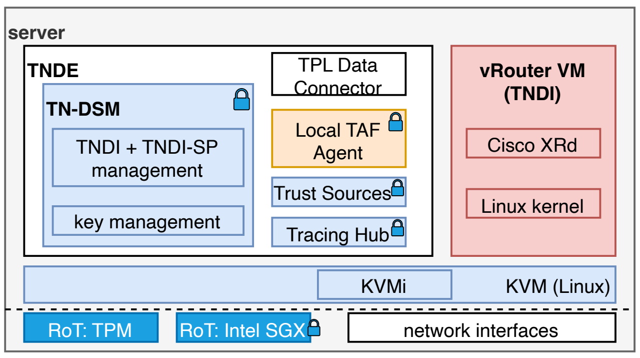
7.1.1 Choice of Hardware Root of Trust (and Isolation Layer)

The device-side CASTOR TCB is layered on top of a platform RoT in order to anchor its security in hardware and enable verifiable evidence reporting. In Section 3.2.2, we have described the TCB architecture based on a generic RoT and isolation layer and presented a hypervisor and a host OS with container support as two possible instantiation examples for the isolation layer between the device TCB and its TNDIs. In the following, we now discuss possible RoT and isolation layer choices that CASTOR explores for the network devices and describe their technical requirements. In that context, we refer to the requirements outlined in Section 3.3.1 that the CASTOR TCB imposes on the device RoT to provide the necessary security guaranties and functionalities for the secure operation of the device TCB. As CASTOR intends to support multiple network device vendors as well as the instantiation of TNDIs as physical and virtual routers (see Section 3.2.2), we will also briefly outline additional directions for compatible RoTs and isolation layers that are not further explored as part of the CASTOR prototype.

Figure 7.1 shows two instantiations of the RoT and isolation layer that CASTOR will explore as the main options in version 1. Note that some of the concrete instantiations in the figure serve as examples considered as options for the CASTOR version 1 prototyping: Linux with KVM hypervisor support serving as the host OS of a server system which instantiates TNDIs as virtual routers (vRouters) based on virtual machines (VMs). In this instantiation, each TNDI VM runs a Linux guest OS as the vRouter's network OS and a Cisco XRd container providing the actual network and routing stack, while the TNDE runs isolated from the TNDIs (outside the VMs). In this context, KVMi represents one specific implementation of a Trace Unit that CASTOR will consider but is not the main focus of this discussion (Section 8.2.1). We will discuss CASTOR's multi-level tracing architecture and the types of considered Trace Units in Chapter 8. Next, we focus on discussing reasonable RoT and isolation layer instantiations (see Section 2.2 for an overview on existing RoTs) and to what extent they can satisfy the requirements of the CASTOR device-side TCB.



(a) A TPM forms the RoT providing measurement, attestation, and secure storage capabilities. The hypervisor, CASTOR TNDE (except TPL Data connector), and Trace Units form the TCB. The hypervisor provides isolation from the TNDIs.



(b) Here the device is extended with a second RoT: Intel SGX. Intel SGX provides an additional layer of isolation and attestation at the application layer for the CASTOR TNDE components. Components with a blue lock are isolated and measured by Intel SGX.

Figure 7.1: Two possible RoT choices for CASTOR's device side. Dark blue components are the RoTs (typically hardware and/or firmware), light blue components form the TCB, and yellow ones might or might not be part of the TCB. Red components are explicitly not part of the CASTOR device-side TCB.

7.1.1.0.1 TPM as RoT (preferred option a) Equipping each device with a hardware TPM (or a chip with TPM-equivalent interfaces) provides a strong anchor for the CASTOR device-side TCB. A hardware TPM is physically isolated from the host device and directly satisfies most of the security requirements of the TCB outlined in Section 3.3.1. The TPM includes a RoT for measurement (RTM) to measure the platform software and TNDE, a RoT for reporting (RTR) to expose the measurements as verifiable evidence, and a RoT for storage (RTS) to store cryptographic keys and other critical data. In addition, a TPM provides secure entropy and monotonic counters for secure key generation, with optional binding to access policies (e.g., based on the platform and/or TNDE measurements). In this way, a TPM forms the necessary foundation for the CASTOR TCB (especially the TNDE) to form a trusted identity, report verifiable evidence, and perform secure key management (Section 3.3.2). That is, a TPM directly satisfies requirements R3 to R7 of Section 3.3.1. While a TPM does not provide a RoT for isolation applicable to the TNDE and Trace Units (R1 and R2), the TPM enables the measurement and attestation of a software-based isolation layer, e.g., based on a host hypervisor (described above) or host OS partitioning scheme. For example, when combining a TPM with dynamic RTM technologies (DRTM) such as Intel TXT, a TPM can securely capture the measurement of the loaded hypervisor, enabling the CASTOR Service Orchestrator to remotely verify its identity and the isolation of the TNDE and Trace Units. This specific setup is illustrated in Figure 7.1a for a device using the KVM hypervisor of Linux as the isolation layer and instantiating TNDIs as VMs. In Section 3.2.2, we also briefly described a scenario where TNDIs are instantiated as container-based vRouters on top of the host OS (see Figure 3.2b), where the host OS forms the isolation layer and therefore must be captured by the TPM measurements as part of a secure boot process.

7.1.1.0.2 Intel SGX as RoT Choosing an Intel CPU with support for Intel SGX-based enclaves (user-space TEEs) as the management CPU of the network device provides a RoT with a very small RoT TCB size in terms of interfaces and firmware. Intel SGX provides secure measurement (RTM), attestation (RTR), and storage (RTS) capabilities for its user-space enclaves (R3 to R5), and some device platforms additionally provide secure monotonic counters and timestamp functionality for SGX enclaves (R6 and R7). In contrast to a TPM, Intel SGX also provides a RoT for isolated execution (Section 2.2), allowing the isolation of the TNDE and (user space parts of the) Trace Units within SGX enclaves (R1 and R2) without relying on an additional, separate software isolation layer, further decreasing the RoT TCB size. However, relying solely on Intel SGX introduces additional challenges due to limitations implied by SGX's scope being restricted to user-space enclaves, excluding the platform code (e.g., host OS or hypervisor). In particular, Intel SGX does not have direct access to the underlying network hardware, e.g., to isolate the network interfaces of multiple TNDIs or access NIC counters, and cannot isolate privileged code (e.g., the host kernel), which impacts the reliability of TNDI tracing. Without isolated privileged code, the CASTOR TCB can host Trace Units securely only within SGX enclaves, requiring them to rely on the no-longer protected (or measured) host OS to set up access to TNDI-related data structures and control-flow information, or on dedicated hardware devices. Furthermore, as mentioned above, the counter and time functionalities depend on respective platform support and are not built in to SGX, i.e., they require additional platform support on the device. Last but not least, note that today Intel SGX support is mostly limited to server-grade CPUs, as support for smaller (client) devices has been deprecated for several years.

7.1.1.0.3 Two RoTs: Combining a TPM with Intel SGX (preferred option b) The TPM-based design directly satisfies most of the requirements imposed by the CASTOR TCB on the platform RoT. However, the software-based isolation layer, e.g., a host hypervisor, has a significant impact on the TCB size. Therefore, another option that CASTOR considers is the combination of a TPM with Intel SGX enclaves to provide defense in depth for the TNDE service isolation based on a smaller RoT, without sacrificing privileged Trace Units and direct control of networking hardware. Figure 7.1b illustrates one possible instantiation of this scheme, with a host hypervisor as the additional software-based isolation layer covered by the TPM measurements. The hypervisor isolates the Trace Units from the TNDIs and the TNDIs from each other. Furthermore, the TPM can provide platform measurements and key bindings for the TNDE. Intel SGX augments this with a strong RoT-based isolation of the TNDE user-space components (e.g., TN-DSM) and separate measurements and attestation reports for them. As the platform measurements and TNDE measurements originate in different RoTs, the TNDE (more specifically, the TN-DSM) must interact with both RoTs and bind necessary cryptographic keys in a reasonable way to the RoTs (also see Section 7.1.2). Some functionalities are overlapping, e.g., as both provide secure storage capabilities, such that the TNDE requires policies that decide what binding is reasonable for what type of data.

7.1.1.0.4 Further Directions As outlined in Section 2.2 and Section 3.2.2, there are many technologies providing RoT functionalities and hardware- or software-based isolation layers. As CASTOR and its TNDI concepts are designed to capture a variety of different device vendors and device types (including physical and virtual ones), there are several additional options that CASTOR could explore to instantiate the RoT requirements of CASTOR's device-side TCB. For example, the isolation layer could be based on containers or a static partitioning scheme provided by the network OS of a physical router, or parts of the TNDE could be offloaded to a dedicated co-processor with strong physical isolation. Furthermore, RoTs implementing the DICE specifications or TEE virtual machines (e.g., AMD SEV, Intel TDX) could be considered to replace the roles of the TPM and/or SGX enclaves outlined above. Likewise, solutions based on system TEEs like Arm TrustZone or RISC-V equivalents could be considered for non-x86 network devices. That is, CASTOR is by far not limited to supporting network devices with the above TPM- or SGX-based RoTs, but can enrol network devices with different RoTs into the CASTOR trust network. As long as the RoTs satisfy the requirements imposed by CASTOR's device-side TCB for securely implementing the TNDI-SP, CASTOR can include the devices in its trusted path routing domain.

7.1.2 TNDE (and TN-DSM) Interaction with RoT

The TNDE builds on top of the platform RoT to securely implement the TNDI-SP security mechanisms required for CASTOR's trusted path routing. The TN-DSM is responsible for interacting with the underlying RoT in order to bind cryptographic keys, especially the TNDE platform key (see Section 3.3.2), to the underlying platform and TNDE (code) identity. Considering a TPM as the RoT, CASTOR can, for instance, explore key access policies based on the platform and TNDE measurements performed by the TPM. Similarly, if Intel SGX or a combination of a TPM and SGX is used, the TN-DSM needs to interact with each RoT to establish a cross-RoT binding of the cryptographic identities and keys anchored in the platform RoTs.

In addition, the TN-DSM is responsible for collecting measurement evidence on the platform and TNDE from the RoTs in order to report verifiable attestation evidence towards the CASTOR orchestration layer as part of the device enrolment (see Section 7.2.1). The TNDE also needs to interact with the other necessary security functions of the device RoTs (Section 3.3.1), either directly or via the TN-DSM as an abstraction layer, for example to securely store data bound to the TNDE.

7.2 Device and TNDI Management

The CASTOR Service Orchestrator manages a distributed trust network of TNDIs, using the TNDI-SP to configure, monitor, and assess the trustworthiness of each TNDI in the context of trusted path routing. In this setting, a TNDI represents the unit of trust and assignment on a network device, while the TNDI-SP defines the security mechanisms and the control and data channels used to securely onboard and operate these TNDIs. The TNDE, and in particular the TN-DSM, is responsible for managing the TNDIs on each device and for implementing the TNDI-SP security mechanisms on these TNDIs (see Section 3.2).

Before a device can participate in a CASTOR-managed domain, the TNDE must perform initial provisioning steps to establish its own cryptographic foundation. In particular, a TNDE platform key needs to be generated and bound to the device's platform RoTs and TNDE identity, and additional key hierarchies may be prepared, such as secure sealing keys for TNDE-local storage and, optionally, TNDE-internal communication keys for protected interactions between TNDE components (see key overview in Section 3.3.2). These provisioning steps prepare the TNDE to expose trustworthy TNDI-SP control and data channel endpoints and to authenticate itself towards a CASTOR Service Orchestrator during the subsequent device JOIN procedure (detailed in Section 7.2.1).

The TN-DSM is the central entity in the TNDE that manages the TNDIs of a device and coordinates the TNDI-SP security mechanisms on them, handling the TNDI-SP control and data channels (as their endpoints). It maintains the TNDI security states and coordinates the execution of device and TNDI enrolments (join and onboarding), configuration updates, and the interaction between the Service Orchestrator and local TNDE subcomponents such as the Tracing Hub, Trust Sources, and Local TAF Agent. Conceptually, and as part of each TNDI's security state, the TN-DSM maintains a TNDI-SP finite state machine (FSM) instance that reflects, for each TNDI, both its lifecycle progression (e.g., from teardown to onboarding to runtime) and the status of the TNDI-SP security mechanisms, including whether they are disabled, in configuration, or fully enabled for secure participation in a CASTOR trust network. This FSM concept is inspired by the PCI-SIG TDISP [67] state machine for TDIs and plays a similar role, but tailored to TNDIs of the CASTOR TNDI-SP and its trust requirements in the context of trusted path routing (see Section 3.2.1). The definition of the TNDI-SP FSM is out of scope of this document and will be specified in D3.2 and D3.3. Through these interfaces and state machines, the TN-DSM allows the Service Orchestrator to treat each TNDI as an individually addressable and manageable trust object.

To manage TNDIs securely, the TN-DSM must be able to identify and discover the TNDIs present on a device. This can occur at boot time, for example when the device's platform or network OS statically instantiates one or more TNDIs during startup; post boot under local control by the device operator, who may create additional TNDIs using device-specific management interfaces (e.g., router NOS or server CLI accessed via SSH through a management network), for example by spawning additional vRouters on a server platform or by configuring additional partitions on a physical router; or post boot under the control of an external Orchestrator or management system that deploys and tears down network functions on demand (e.g., dynamic vRouter deployment using a VM or container orchestration platform). One concrete option that CASTOR will explore is having the CASTOR orchestration layer itself deploy and manage vRouter VMs forming TNDIs on suitable server platforms (see Figure 7.1). Which scenarios apply depends on the device and platform type (e.g., physical router versus server platform with vRouters) and on vendor-specific RoT and isolation-layer instantiations (see Section 7.1.1). The operation of the CASTOR TNDE and its TN-DSM is designed to be agnostic to these details: in all cases, the TN-DSM must associate each TNDI with a stable identifier and maintain the mapping between TNDI identifiers and their corresponding TNDI-SP contexts.

To support attestation and evidence binding, each TNDI requires a cryptographic identity. CASTOR associates a TNDI attestation key with each TNDI, which serves as its cryptographic identity in the CASTOR trust network, and the TN-DSM is responsible for provisioning, binding, and managing these keys, for example by deriving or provisioning them under the device's RoT and binding them to the TNDE key and, optionally, the TNDI's code and configuration state (Section 3.3.2). The exact lifecycle and derivation of TNDI keys (e.g., whether they are established during TNDI identification or only at onboarding, and whether they are generated randomly or derived via a KDF from TNDE key material, TNDI measurements, and/or vendor or device metadata) is implementation-dependent and may be adapted to different deployment and trust models. Depending on these choices, TNDI keys may change frequently (e.g., with TNDI software or configuration updates), less frequently, or in the extreme case not at all for long-lived, single-TNDI devices. Once the TNDE has been provisioned and enrolled via device JOIN (Section 7.2.1), these TNDI-specific identities can be onboarded into CASTOR-managed domains.

Each TNDI passes through a sequence of lifecycle phases that determine how it can participate in the CASTOR trust network. At a high level, these phases include, and are conceptually reflected in the corresponding TNDI-SP FSM instance maintained by the TN-DSM:

Initialization and discovery The TNDI is created or detected by the TN-DSM, and a local TNDI context is established, including the association of the TNDI with a local identifier and a TNDI attestation key.

TNDI ONBOARDING The CASTOR Service Orchestrator requests the onboarding of one or more TNDIs into a CASTOR network domain from the TN-DSM via the TNDI-SP control channel (Section 7.2.2), thereby admitting them as managed members of the domain's trust and routing context. As part of onboarding, the TN-DSM and Orchestrator configure TNDI-specific trust policies, tracing and evidence collection settings, and other parameters for the TNDI-SP security mechanisms. Thus, the TNDE can produce domain-relevant evidence for that TNDI, enforce the required policies, and enable the TNDI to participate in trusted path routing. In parallel, the TN-DSM transitions the TNDI's TNDI-SP FSM through the corresponding onboarding-related states until the TNDI-SP mechanisms for that TNDI are fully configured and enabled for secure operation.

Runtime operation and reprogrammability After onboarding, the TNDI enters runtime operation, in which it actively participates in the CASTOR trust network under the configured policies. At the start of this phase, neighbouring TNDIs establish secure links as part of the transition from onboarding to runtime operation, and during runtime these links may be rekeyed or otherwise maintained as required. In this phase, according to the per-TNDI TNDI-SP configurations, the TNDE needs to continuously collect runtime traces from the TNDI via the Tracing Hub, generate attestation and behavioural evidence via the Trust Sources, and let the TN-DSM coordinate the sharing of the resulting evidence and local trust reports (from the Local TAF Agent) via the TNDI-SP data channels with the CASTOR orchestration layer (Section 7.4). The Orchestrator and TN-DSM may need to reconfigure the TNDI over time to react to trust- or network-related changes in the network (Section 7.3), for example by adapting trust policies, network configuration, tracing granularity or active Trace Units, or evidence collection parameters, while preserving a consistent TNDI state and evidence history. During secure runtime operation, the TNDI-SP FSM reflects that the TNDI-SP mechanisms for the TNDI remain enabled and operational, and only transitions away from this secure operational state when security mechanisms are intentionally disabled or the TNDI moves towards teardown.

Teardown and ownership transfer Eventually, a TNDI may be removed from service or transferred to a different owner or CASTOR domain, requiring the TN-DSM to tear down local state, revoke or rotate TNDI keys as appropriate, and update the TNDI's lifecycle state accordingly. Teardown concludes the TNDI's participation in the current CASTOR trust network but may be followed by re-initialization and onboarding under a different trust or ownership context. The TNDI-SP FSM for the TNDI reflects this by transitioning into a teardown or disabled state, in which TNDI-SP security mechanisms are no longer available for trusted path routing.

The precise set of states and transitions in the TNDI lifecycle, and the exact points at which keys, measurements, configurations, secure links, and evidence flows are established or updated, may differ across implementations. CASTOR does not mandate a specific, globally fixed realization of this lifecycle but requires that any implementation defines well-specified, auditable, and authenticated transitions that implement the semantics of initialization, onboarding, runtime operation (including secure link management and continuous evidence sharing), and teardown. In contrast, the TNDI-SP FSM is intended to provide a more concrete, protocol-level state machine for the TNDI-SP security mechanisms, similar in spirit to the PCI-SIG TDISP state machine for TDIs, and will be specified in detail in subsequent CASTOR deliverables (D3.2 and D3.3). Device JOIN remains a prerequisite that enrols the device and its TNDE into a CASTOR-managed domain and establishes the initial trust relationship and attestation baseline between the CASTOR orchestration layer and the device platform plus TNDE, before any of the device's TNDIs can be onboarded into a CASTOR trust network; the following subsection details this device JOIN phase.

7.2.1 Device Enrolment and Attestation (TNDE Join)

Before any of a device's TNDIs can be onboarded into a CASTOR trust network, the CASTOR Service Orchestrator first needs to establish trust in the network device platform and its TNDE. Without a verified and enrolled device, the secure enforcement of TNDI-SP mechanisms cannot be guaranteed, which is crucial for CASTOR's calculation and enforcement of trusted routing paths. The TNDE Join phase therefore acts as a prerequisite for subsequent TNDI onboarding, establishing the initial trust relationship and attestation baseline between the CASTOR orchestration layer and the network device.

The Join procedure conceptually operates over the TNDI-SP control channel, with the TN-DSM of the TNDE acting as the device-side endpoint. CASTOR requires that, to initiate the Join phase, the Orchestrator can reach the TN-DSM over a suitably protected network path, for example via a dedicated management network or through the in-band operational network used for regular device traffic. Depending on the device platform and implementation, the TNDE may have direct access to a network interface or may need to send and receive control traffic via the NOS if the NOS owns all physical interfaces. The Orchestrator and TN-DSM need to jointly establish communication that ensures at least message authenticity and integrity during the initial exchange and can later be upgraded to additionally provide end-to-end confidentiality once trust is established, independent of the underlying transport.

During the Join phase, the CASTOR Service Orchestrator authenticates the device identity and requests attestation evidence for the device platform and TNDE. To produce this evidence, the TN-DSM interacts with the device's root(s) of trust (RoTs) and uses the previously established TNDE platform key, which is already bound to the device platform and TNDE code identity (see Section 7.2). Depending on the device architecture, the TN-DSM may also need to communicate with other TNDE subcomponents to collect and aggregate their respective measurements or reports before generating the overall attestation evidence. For example, if multiple components execute in isolated environments such as SGX enclaves

(Section 7.1.1), their reports need to be consolidated under the TNDE's attestation context. The TN-DSM signs the combined evidence with the TNDE platform key before returning it to the Orchestrator.

CASTOR requires that the attestation evidence enables the Orchestrator to verify its authenticity and to link the TNDE to the device's hardware-rooted identity and trusted platform state. CASTOR will further explore mechanisms where this verification can rely on cryptographic bindings established via the TNDE platform key and its anchoring in the device's RoT(s), allowing the Orchestrator to assess both the integrity and origin of the attestation data.

Once verification succeeds, the Orchestrator and TN-DSM are expected to operate over an end-to-end encrypted and authenticated channel for all subsequent TNDI-SP control communication. The Join phase must also support the enrolment of the device by allowing the Orchestrator to provide initial configuration and metadata through the TNDE, such as service endpoints for orchestration layer components, credentials for (mutual) authentication, and metadata describing device capabilities or high-level management constraints (as determined by the ownership policies discussed below), without yet assigning specific TNDIs to the Orchestrator.

Upon completion of the Join phase, the CASTOR Service Orchestrator has an authenticated, attested, and securely connected device. The TNDE can support different ownership policies, allowing the entire device to be controlled by a single CASTOR domain or enabling subsets of its TNDIs to be onboarded into separate CASTOR domains. The specific ownership policy, such as whether multiple Orchestrators may coexist, is determined by a combination of device capabilities and deployment-time configuration. This flexibility in ownership and orchestration boundaries does not modify the Join procedure itself but enables diverse deployment models. The Orchestrator can then initiate TNDI onboarding via TNDI-SP control channel messages to enrol individual TNDIs into the CASTOR trust network and enable their participation in trusted path routing, as described in the following subsection. Note that in the second version of the CASTOR framework we will also explore a variation of the device attestation model which is based on a secure loader (below the TNDE) and a layered attestation approach, as introduced in Paragraph 3.4.2.0.1.

7.2.2 TNDI Onboarding and Runtime Transition

Following the completion of the Join phase, the CASTOR Service Orchestrator has established a verified trust relationship and a protected TNDI-SP control channel with the TN-DSM of the device's TNDE. The keys protecting this control channel are bound to the TNDE through the TNDE platform key (see Section 3.3.2), ensuring continuity of trust between device-level attestation and subsequent onboarding operations. The onboarding of individual TNDIs builds on this foundation and extends trust from the device and TNDE level to the routing-capable units that will participate in a CASTOR trust network.

The main objective of TNDI onboarding is to enrol selected TNDIs into the CASTOR trust network and to enable them to operate under CASTOR's TNDI-SP security mechanisms as managed participants in trusted path routing. During onboarding, the Orchestrator sends request messages via the TNDI-SP control channel that specify per-TNDI configurations and trust policies, while the TN-DSM, acting as the TNDE's TNDI-SP coordinator, enforces these requests by configuring TNDE sub-components, updating per-TNDI security context, and preparing each TNDI for the establishment of secure links with other TNDIs. In this way, the TNDI-SP security mechanisms are applied according to Orchestrator-provided security policies, and onboarded TNDIs become managed entities whose evidence and trust reports can be consumed by the CASTOR orchestration layer for the selection and deployment of trusted paths through the network.

The CASTOR Service Orchestrator first uses TNDI-SP control messages to query the TN-DSM for the set of TNDIs managed by the TNDE and to obtain information about them in order to decide which TNDIs to onboard. The TN-DSM must be able to expose metadata for each TNDI based on the internally maintained state and security context (see Section 7.2), for example manufacturer identifiers, model and firmware information, and a summary of integrity information associated with each TNDI. This integrity information is expected to be derived by the TNDE from boot-time (or load-time) measurements on the TNDIs, for example captured via the device RoT(s) or by TNDE-managed measurement mechanisms. On this basis, the Orchestrator can determine which TNDIs are eligible for onboarding under the applicable CASTOR policies and select appropriate trust models for each TNDI. The Orchestrator can then start onboarding a TNDI by sending control messages with the respective configuration parameters such as trust policy settings and tracing and evidence-collection configurations. Furthermore, the Service Orchestrator must be able to request the establishment of TNDI-specific TNDI-SP data channels towards the CASTOR orchestration-layer components (Global TAF and DLT) for evidence and trust-report sharing (see Section 7.4). The TN-DSM needs to be able to set up the other TNDE components and the data channels accordingly, ensuring that onboarded TNDIs are placed into a secure operational state and become addressable and identifiable within the CASTOR trust network. Throughout this process, the TN-DSM maintains the per-TNDI TNDI-SP security state (including the FSM), capturing the TNDI lifecycle, onboarding status, and the status of the associated security mechanisms.

During onboarding, the TN-DSM must further be able to derive, access, and manage the cryptographic keys required for the TNDI key sub-hierarchy associated with the TNDE key hierarchy (see Section 3.3.2 and Section 7.2). In particular, the TN-DSM must ensure that the TNDI attestation key and any additional functional keys for that TNDI (for example, keys for signing runtime traces or protecting TNDI-SP data channels) are available and correctly bound to the TNDE and/or TNDI. Note that, as explained in Section 7.2, CASTOR may explore different variations for the lifecycle of some of the keys, for example for the TNDI attestation key; in all cases, CASTOR requires that resulting keys remain cryptographically anchored in the TNDE trust chain, ensuring end-to-end accountability of TNDI evidence and control operations, while leaving room for different deployment- and implementation-specific variations.

7.2.2.0.1 TNDI Runtime Transition After successful onboarding of a TNDI, the TNDI enters the runtime phase where it begins operation in the CASTOR trust network, i.e., the TNDI participates in secure network operations for trusted path routing and the TNDE continuously shares evidence and trust reports on the TNDI with the orchestration layer. At this stage, TNDIs need to be able to establish authenticated and encrypted links with other TNDIs in the CASTOR trust network for secure communication. For each potential TNDI-to-TNDI link, peers must exchange and appraise evidence about their respective trust states to authenticate each other and establish trust before link activation. CASTOR will take into account existing work on Trusted Path Routing (TPR), where devices exchange attestation-based "stamped passports" (for example, using TPM 2.0 evidence) [11], and extend these ideas with CASTOR's concepts of TNDIs and TNDI-SP to support both static platform evidence (for example, boot- or load-time measurements) and, where available, runtime trust reports from the TNDE. That is, the TNDE needs to assist and coordinate the generation and mutual exchange of authenticated stamped passports between (topology-wise) neighbouring TNDIs of the trust network. This mutual evidence exchange and validation enables the establishment of secure links (for example, using link-layer encryption mechanisms such as MACsec) and enables restricting communication to TNDIs that meet the required trust criteria.

7.3 Configuration and Reprogrammability

To support trusted path routing in dynamic environments, CASTOR requires the TNDE to be configurable and reprogrammable across multiple stages of the device and TNDI lifecycles. Configuration is not an one-time action but an ongoing process that underpins device enrolment, TNDI onboarding, and runtime adaptation of TNDI-SP security mechanisms as network and trust conditions evolve.

The CASTOR Service Orchestrator configures the TNDE by sending TNDI-SP control messages to the TN-DSM, which acts as the central coordinator for TNDI-SP on the device. Through these messages, the

Orchestrator provides configuration parameters and security policies, while the TN-DSM deploys them on behalf of the Orchestrator by updating TNDE sub-components and per-TNDI security context. Conceptually, three configuration phases can be distinguished: first, device-level configuration of TNDE and TNDI-SP functions during device Join; second, per-TNDI configuration of TNDI-SP security mechanisms (for example, tracing, Trust Sources, and local trust assessment policies) during TNDI onboarding; and third, reconfiguration of these mechanisms for already onboarded TNDIs during their runtime phase, for example to adjust to new trust requirements or changes in the CASTOR trust network.

The TNDI-SP control sub-protocol must therefore support the transport of configuration and policy information, but it does not expose or constrain the internal configuration logic of the TNDE. The way in which individual TNDE sub-components interpret and apply configuration parameters remains implementation-specific and opaque to the TNDI-SP control channel. This includes, for example, how a specific Local TAF Agent updates its appraisal model or how particular Trust Sources change their evidence-collection behaviour. These internal choices are acceptable as long as the externally visible behaviour and security guaranties of the TNDI-SP interfaces are preserved. Where possible, CASTOR will explore the use of existing configuration and policy languages to express these parameters, aiming for compatibility with relevant standards and ecosystems rather than introducing new formats without clear benefit.

Across the different lifecycle phases, the TNDE is expected to support a range of configuration types, including but not limited to:

- credentials and network addresses of CASTOR orchestration-layer services (for example, Global TAF and DLT endpoints),
- tracing configurations (for example, selection and enablement of specific Trace Units and the granularity or sampling level of collected traces),
- trust-model and trust-policy configurations (for example, selection of Trust Sources, choice of relevant evidence types, and parameters of the Local TAF Agent's assessment model), and
- TNDI-SP data-channel parameters (for example, push versus pull models, batching strategies, and reporting intervals for evidence and trust reports).

Dynamic reconfiguration may be triggered by different classes of changes in the CASTOR trust network. Examples include changes in security requirements or service-level objectives that necessitate an updated trust model (e.g., evidence selection), lifecycle events such as the creation or teardown of TNDIs on a device, or new global or local trust-assessment policies that call for a different security–performance tradeoff and therefore require adjusting tracing granularity or enabling or disabling specific types of Trace Units. Within this framework, CASTOR will explore different configuration parameters and profiles to support diverse deployment scenarios and to enable adaptation to such changes without, in the general case, requiring the device to repeat the Join phase or individual TNDIs to be torn down and re-onboarded. However, if drastic events or changes in the CASTOR trust network undermine the TNDI-SP security guarantees for a TNDI or device, CASTOR may require the corresponding lifecycle transitions, including tearing down and subsequently re-onboarding affected TNDIs or repeating device enrolment.

7.4 TNDI Data Channels: Trace and Evidence Sharing

As part of a TNDI's onboarding process, the TN-DSM establishes TNDI-specific TNDI-SP data channels from the TNDE to the relevant CASTOR orchestration-layer services, in particular the Global TAF and the CASTOR DLT. These data channels are rooted in the existing TNDI-SP control channel and bound to the TNDE key hierarchy, and may additionally be bound to the respective TNDI according to the key types and bindings described in Section 3.3.2. Once a TNDI has been successfully onboarded and entered its runtime phase, these TNDI-specific data channels form part of the active TNDI-SP security mechanisms and are used for the continuous sharing of traces, trustworthiness evidence, and trust reports from that TNDI with the orchestration-layer services.

The primary motivation of the TNDI-SP data channels is the continuous, secure transport of trustworthiness information produced by the device-side TCB, in particular the TNDE, to the orchestration layer throughout the runtime phase of a TNDI. This includes trust reports generated by the Local TAF Agent and, where applicable, trustworthiness evidence derived by the Trust Sources from TNDI runtime traces that serves as input to the Global TAF's global trust assessment of the CASTOR trust network. In addition, TNDI runtime traces collected by the Tracing Hub via its Trace Units and, optionally, the corresponding derived trustworthiness evidence may be shared with the CASTOR DLT. The DLT provides a secure and auditable storage backend for inspection by authorized parties such as the CASTOR Service Orchestrator or, subject to policy, device vendors and other stakeholders. Within the TNDE, traces are first collected by the Tracing Hub, transformed into trustworthiness evidence by the Trust Sources, and then consumed by the Local TAF Agent to generate trust reports, before the TN-DSM coordinates the sharing of the resulting traces, evidence, and reports via the TNDI-SP data channels. Together, these data flows enable the CASTOR orchestration layer to maintain an up-to-date view of the trust state of TNDIs, which is essential for calculating and enforcing trusted paths for trusted path routing, and to support verifiable diagnostics and post-incident analysis.

The main goal of the TNDI-SP data channels is to provide secure transport with reasonable performance overhead in terms of both latency and bandwidth. Low latency is important to enable timely detection of and reaction to changes in the CASTOR trust network. Examples include situations where updated trust reports indicate a drop in a TNDI's trust level that requires rerouting, or where new indicators of compromise motivate an increase in tracing granularity as discussed in the previous section. At the same time, the bandwidth overhead of trace, trustworthiness-evidence, and trust-report traffic must remain sufficiently low to avoid saturating the network. Otherwise, CASTOR risks degrading the capacity available for regular data-plane traffic. In addition, the TNDE must ensure that TNDI-SP data channels and the data transported over them are authenticated and verifiable. CASTOR will explore options such as cryptographically binding the data channels to the TNDE and device identities via the TNDI-SP control channel and ensuring that trust reports, trustworthiness evidence, and traces are signed with the respective TNDI attestation and tracing keys by the TN-DSM and the Tracing Hub (see key overview in Section 3.3.2). CASTOR will therefore explore different realization options for the TNDI-SP data channels that satisfy these security and performance requirements while leaving room for deployment-specific tradeoffs.

As outlined in Section 7.3, TNDI-SP data channels themselves are subject to configuration and reprogrammability via TNDI-SP control messages. This includes, for example, parameters that influence how and when traces, trustworthiness evidence, and trust reports are sent over these channels, such as push versus pull behaviour, batching strategies, and reporting intervals, allowing CASTOR deployments to tune the balance between timeliness of information and resource usage.

To support interoperability across vendors and service implementations, the TNDE must ensure that traces, trustworthiness evidence, and trust reports follow formats that are processable in a consistent way. Where possible, CASTOR will investigate the use of existing encoding and data-modelling technologies, such as CBOR and YANG, to represent this information in an efficient yet standards-aligned manner, enabling external verifiers to correctly interpret and process the shared evidence without requiring CASTOR-specific encodings.

Chapter 8

CASTOR Design Space for Multi-level Runtime Tracing & Observability

8.1 Multi-level Tracing Overview and CASTOR Tracing Hub

To support trusted path routing, CASTOR requires continuous visibility into the runtime behaviour and configuration state of each TNDI. For this purpose, the TNDE on each network device must collect configurational and behavioural runtime traces for every managed TNDI. These traces are provided as input to the Trust Sources, which then generate trustworthiness evidence consumed by the Local TAF Agent for local trust assessment. In this way, the CASTOR Service Orchestrator can receive per-TNDI trust reports via the TNDI-SP data channels and maintain a global, up-to-date view of the trust state of the network that is required for calculating and enforcing trusted paths.

Runtime tracing is essential to detect misconfigurations, indicators of compromise, and ongoing attacks that affect the security posture of a TNDI. Examples include configuration changes that violate the deployed trust policy, unexpected control-plane behaviour, or fine-grained indicators of exploitation attempts, such as anomalous system call activity or unexpected modifications of routing state. When such events are detectable in the collected traces and reflected in the trustworthiness evidence generated by the Trust Sources, the Local TAF Agent can lower the assessed trust level of the affected TNDI, and the Global TAF can update the global trust state accordingly based on the received trust reports (and, where applicable, shared evidence).

Within the TNDE, the *Tracing Hub* serves as the main coordinator for runtime trace collection and distribution. Configured by the TN-DSM during TNDI onboarding, and reconfigurable at runtime via TNDI-SP control channel messages, the Tracing Hub is responsible for instantiating and configuring one or more Trace Units for each TNDI, coordinating the authentication of trace data, and managing the sharing of traces with the Trust Sources and, where applicable, with the TN-DSM. Towards the Trust Sources, the Tracing Hub provides the traces required to generate configurational and behavioural evidence for the Local TAF Agent. Trace data that is made available to the TN-DSM (either directly from the Tracing Hub or indirectly via the Trust Sources) can, where configured, be further shared with the CASTOR DLT to support post-mortem analysis and auditing.

The Tracing Hub orchestrates one or multiple *Trace Units* per device, which act as the concrete tracing backends for the managed TNDIs. Each Trace Unit is set up and configured by the Tracing Hub on a per-TNDI basis and is responsible for collecting specific types of runtime traces for the TNDIs for which it is enabled, such as memory-inspection traces, OS-level traces over system calls and kernel data structures, or protocol-specific traces of control-plane and forwarding-plane behaviour. Whether a given Trace Unit instance serves multiple TNDIs or is instantiated separately per TNDI is an implementation detail; the architectural requirement is that the Tracing Hub must support per-TNDI tracing configurations using its managed Trace Units.

In the context of CASTOR, *multi-level tracing* refers to the ability of the Tracing Hub to dynamically select and configure combinations of Trace Units that cover the traces required for different types of evidence at different granularities. This includes, for example, using a strongly isolated out-of-TNDI Trace Unit as the primary secure tracing source and enabling additional in-TNDI Trace Units to obtain more fine-grained behavioural traces when needed. The dynamic reprogrammability of the Tracing Hub allows the CASTOR Service Orchestrator, via the TN-DSM, to adapt tracing configurations in response to events within the trust network, such as raising tracing granularity when indicators of compromise are observed for a TNDI, or updating the set of active Trace Units when trust policies or service-level agreements change.

Beyond selecting and enabling specific Trace Units, the Tracing Hub must also support configuration options that influence how traces are transported within the TNDE and towards the orchestration layer. Examples might include choosing between push- and pull-based trace delivery to Trust Sources or the Global TAF, adjusting batching sizes and flushing intervals for trace records, and configuring rate limits or sampling strategies to balance detection latency, resource usage, and bandwidth overhead. The CASTOR Service Orchestrator can request the TN-DSM to update these parameters using TNDI-SP control messages, allowing CASTOR deployments to tune the tradeoff between tracing granularity, timeliness of evidence, and operational cost (performance impact).

To implement these behaviours, the Tracing Hub must provide interfaces and capabilities for selecting and routing relevant traces to the appropriate Trust Sources for evidence generation. In particular, it needs to support flexible subscription and filtering mechanisms so that Trust Sources can obtain only those traces that are required for the propositions they evaluate, while avoiding unnecessary overhead. At the same time, the sharing of trace data between Trace Units and other TNDE components, especially the Tracing Hub, Trust Sources, and TN-DSM, must be secure and efficient. CASTOR will therefore explore efficient local control and data channels for trace sharing inside the TNDE, where control channels are used to configure and manage Trace Units and their connections, and data channels are used to transfer trace records with minimal overhead. Possible realization options for these local channels include local socket connections, shared-memory mechanisms, or other zero-copy schemes that preserve the isolation and authentication requirements of the device-side TCB while enabling high-throughput tracing.

8.1.1 Trace Encoding and Authentication

To enable multiple Trust Sources and analysis tools to process traces in a consistent way, CASTOR requires a common, efficient, and vendor-neutral representation for trace records. Trace formats must be compact enough to minimise bandwidth and storage overhead, yet sufficiently expressive to capture the semantics needed for trust assessment and post-mortem analysis. Where possible, CASTOR will investigate the use of existing encoding and data-modelling technologies, such as CBOR for binary encoding and, where appropriate, YANG-based models for describing trace record types and their fields. At the same time, CASTOR does not exclude the use of CASTOR-specific encodings where this is necessary to meet efficiency or deployment requirements.

Beyond efficient encoding, trace records must be authenticated to ensure that they have not been modified (integrity) and that they can be reliably linked to the TNDIs and TNDEs that produced them (provenance). Locally, Trust Sources must be able to verify that the traces they consume originate from the device-side TCB and have not been tampered with by components outside the TCB boundary. Depending on the concrete instantiation of the RoT and isolation layer (cf. Section 3.3.1 and Section 7.1.1), this may, for example, require protecting the integrity of traces emitted by in-TNDI Trace Units against a compromised TNDI environment, or ensuring that out-of-TNDI Trace Units and the Tracing Hub are isolated from the TNDIs they monitor. Remotely, parties that inspect traces stored in the CASTOR DLT for post-mortem analysis or auditing should ideally be able to verify that these traces were produced for a specific TNDI under the control of a specific TNDE and that they have not been modified since recording.

CASTOR will therefore investigate mechanisms for providing such verifiability for exported traces, while balancing overhead and deployment constraints.

As summarised in Section 3.3.2, CASTOR therefore introduces a per-TNDI *trace key*, which is created by the TN-DSM as part of the device-side key hierarchy and cryptographically bound to the corresponding TNDI (and thus to the TNDE). The TNDI trace key is defined as a symmetric key that is used to authenticate trace records associated with that TNDI, for example via message authentication codes, so that Trust Sources and the TN-DSM can efficiently verify trace integrity and provenance inside the TNDE. This symmetric-key approach keeps the per-record overhead low, which is important for fine-grained tracing. Depending on the concrete instantiation, the TNDI trace key may be used directly by the Tracing Hub to authenticate trace records on behalf of each TNDI, or it may be distributed to individual Trace Units so that they can authenticate the traces they generate. The former centralises key usage and can simplify key isolation within the TNDE, whereas the latter can improve scalability by allowing Trace Units to operate more independently and in parallel. CASTOR will analyse these design options and their tradeoffs when defining concrete trace-encoding and signing schemes for different deployment environments.

For remote verification by external parties that only have read access to traces via the CASTOR DLT, the symmetric TNDI trace key cannot be shared (externally). To address this, CASTOR will explore a layered approach in which the TN-DSM additionally signs traces selected for export (or aggregated trace artefacts) with the per-TNDI attestation key before sending them over the TNDI-SP data channels to the CASTOR DLT. In this model, the symmetric TNDI trace key is used for fast, local authentication of individual trace records between Trace Units, the Tracing Hub, Trust Sources, and the TN-DSM, while the TNDI attestation key provides an asymmetric, externally verifiable signature over the trace data as it leaves the TNDE.

8.1.2 Motivation: Runtime Observability for Trust Assessment

Modern, multi-stage attacks against network infrastructure rarely consist of a single exploit; they progress through reconnaissance, exploitation, privilege escalation, and persistence, and often manifest only through subtle runtime anomalies, such as irregular system-call patterns or statistically unusual control-plane or data-plane traffic. Many of these activities fall outside the scope of traditional perimeter defences and cannot be reliably detected by static boot-time checks alone.

In this sense, CASTOR adopts a *Below Zero-Trust* posture in the spirit of AR4SI [80]: trust in a TNDI is not treated as a one-time property established at boot, but as an evolving quantity that must be continuously justified by fresh runtime evidence. Boot-time mechanisms such as secure boot or static binary verification help ensure that a TNDI starts from a known-good state, yet they do not protect against transient, memory-only attacks, configuration drift, or behavioural deviations that occur long after initialization.

As outlined in Chapter 4, CASTOR's trust assessment relies on multiple Trust Sources that generate evidence about both the system-level state and the networking behaviour of each TNDI. CASTOR combines traces of firmware and software configuration state with behavioural traces of critical routing functions to support dynamic, evidence-based trust assessment. These two aspects correspond to CASTOR's two main Trust Sources (Attestation Source and FSM Source), which together allow the Local TAF Agent and Global TAF to react to changes in the assessed trust levels of TNDIs over time.

8.1.3 System-level and Networking Traces

As outlined in Section 8.1, CASTOR must observe both the execution environment of a TNDI and its routing behaviour. At a high level, the traces collected for each TNDI can be grouped into two domains:

- **System-level traces:** (N)OS-level signals about the execution environment of a TNDI, such as

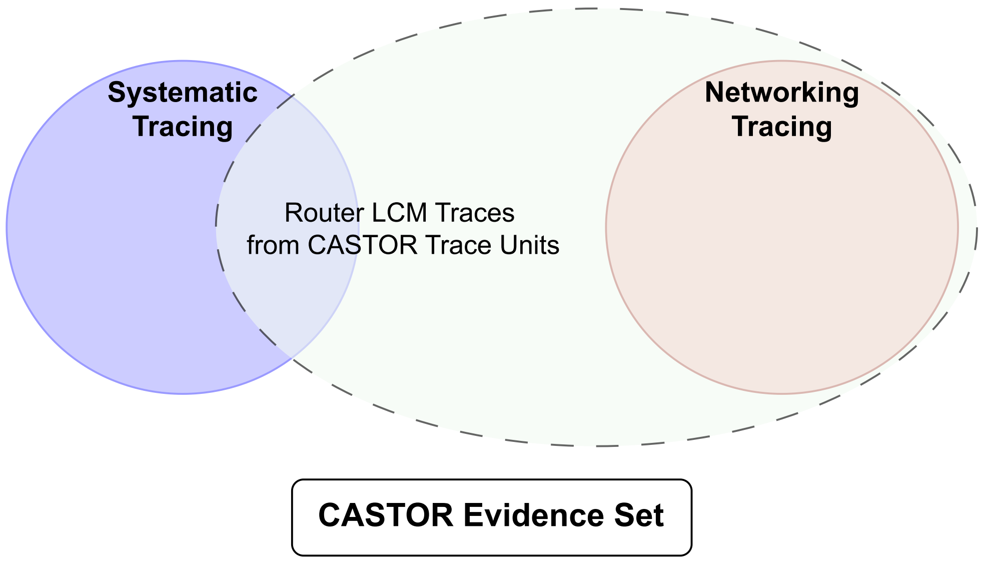


Figure 8.1: Difference between Systematic and Networking traces

system calls, process and kernel events, and selected configuration or memory state. These traces primarily support propositions about platform integrity, software and firmware configuration, and the overall operational posture of the TNDI.

- **Networking traces:** Traces related to the control and forwarding plane of the TNDI, such as routing-protocol events or packet-processing behaviour. These traces primarily support propositions about correct routing behaviour, policy enforcement, and control-plane correctness.

Both domains contribute to the trustworthiness evaluation of a TNDI, but they originate from different Trace Units and encode different semantics. As a result, they may require different pre- and post-processing steps in the Tracing Hub and/or Trust Sources before they can be consumed by the Local TAF Agent.

8.1.3.0.1 Tracing across the routing lifecycle To address the CASTOR threat model, tracing must cover both the routing plane and the underlying system behaviour of each TNDI throughout their operational lifecycle. Networking traces help detect adversaries targeting the communication fabric, for example by revealing anomalies associated with BGP prefix hijacking or OSPF Link State Advertisement falsification (see Threat Model analysis in Chapter 10). As shown in Figure 8.1, the focus on system-level traces is limited to monitoring activities that directly affect the router software stack, and in particular to detecting attempts to compromise the TNDI through malware, counterfeit firmware, or memory-only attacks such as return-oriented programming or code injection.

Under the envisioned threat model — at least for the first version of the CASTOR framework — we do not aim to systematically assess the general security posture (i.e., correctness) of the underlying infrastructure layer (e.g., virtualization layer or host OS in the context of vRouter instantiation). Instead, infrastructure-level traces are collected and analyzed only with respect to threats, attacks, or anomalous behaviours that have a direct impact on the TNDI and its execution. By combining systematic-and networking-level traces, CASTOR enables trust assessments that incorporate evidence of threats targeting both the routing plane and the execution environment in which it operates. Moreover, CASTOR can optionally leverage correlations across these domains to strengthen detection and attribution (e.g., for example, relating anomalous memory usage or system-call patterns to observed network behaviour).

8.1.4 Background: Software- and Hardware-based Tracing

Software-based tracing mechanisms have evolved from basic code- or configuration-attestation schemes towards richer runtime tracing. DIALED [27], for example, establishes a root of trust on constrained devices by leveraging ARM's Memory Protection Unit (MPU) to compartmentalize firmware into isolated regions and monitor memory accesses between them. By observing control transfers across MPU regions and applying access policies, DIALED can detect deviations from expected control flows without requiring complex hardware extensions. C-FLAT [2] advances this line of work by using a trusted execution environment to collect control-flow traces of applications at runtime. It instruments the program to emit control-flow events, which are securely recorded and aggregated inside the TEE to attest that execution follows an expected control-flow graph, extending assurance beyond boot-time checks. However, this reliance on program instrumentation and frequent transitions into the TEE introduces non-negligible overhead and increases intrusiveness.

GuaranTEE [58] refines this approach for environments where the target software itself runs inside an isolated enclave. It instruments program execution at the instruction and function-call levels using trampoline-based hooks and records a hash-chained sequence of execution events, which is communicated to a verifier enclave via shared buffers. The verifier enclave can then validate the reported execution path against a reference model, enabling detailed runtime attestation of enclave-resident code.

All of these software-based mechanisms share a common pattern: they do not rely on specific hardware tracing extensions and can, in principle, be deployed on a wide range of platforms, but they require access to and modification of the target software or firmware to insert tracing hooks. Furthermore, a detailed understanding of the target environment's semantics is required to interpret low-level events as meaningful, higher-level behaviour, which becomes particularly challenging in complex, multi-component network elements.

Hardware-assisted tracing aims to lower overhead and avoid intrusive code changes by extending processors with dedicated monitoring logic. LO-FAT [31], for instance, augments a processor pipeline with hardware extensions that capture control-flow events and compute cumulative hashes in parallel with program execution. This enables attestation of unmodified ("legacy") binaries, since control-flow information can be obtained without instrumenting the program itself. Hardware-based schemes can provide accurate, low-overhead monitoring of program execution and are less dependent on prior knowledge of the target software's internal structure. At the same time, they require non-trivial modifications to processor designs and are bound to specific hardware platforms and vendor support. In the context of CASTOR, deploying such hardware-assisted tracing (and thus attestation) primitives directly in network elements such as routers is further complicated by the typically closed nature of network equipment ecosystems, which limits access to internal hardware designs and restricts the ability to integrate custom monitoring extensions.

Together, these software- and hardware-based tracing mechanisms illustrate a trade-off space between intrusiveness, expressiveness, performance overhead, and deployment constraints. In the current version, CASTOR therefore emphasises flexible, software-based Trace Units that can be deployed on existing network equipment without requiring specific hardware changes, while still allowing the use of hardware-assisted tracing mechanisms where such capabilities are available on a given platform.

Table 8.1: Example technologies for realising Trace Units in CASTOR.

Category	**Technology**	**Granularity**	**Typical evidence focus**
Software	eBPF	Medium–high	System calls and networking hooks
Software	LibVMI	High	Selected in-memory state of TNDIs
Hardware	Intel PT	Instruction-level	Control-flow behaviour of binaries
Software	Kernel modules	High	Kernel-level events and data structures

In CASTOR, such technologies serve as building blocks for Trace Units (see Table 8.1). For example, LibVMI in combination with hypervisor support can underpin a memory-inspection Trace Unit, whereas eBPF or kernel modules can be used to realise OS-level Trace Units inside a TNDI. Hardware tracing features such as Intel PT can complement these software-based mechanisms where available, for instance by providing more fine-grained control-flow information for selected components. The concrete Trace Units considered in CASTOR, and their tradeoffs with respect to security, performance, and portability, are discussed in the next section.

8.2 CASTOR's Trace Units

Trace Units are the building blocks that realise CASTOR's multi-level tracing on each device. They implement concrete tracing mechanisms under the control of the Tracing Hub and expose their output in a uniform way to the TNDE, where it can be authenticated and bound to a TNDI-specific context by either the Trace Units themselves or by the Tracing Hub, depending on the chosen deployment (see Section 8.1.1). When required by CASTOR policies, the resulting traces can also be shared via the TN-DSM over the TNDI-SP data channel(s) with the CASTOR DLT to support post-mortem analysis by external verifiers.

8.2.0.0.1 Trace Unit Concept Trace Units are the components that perform the actual tracing of a TNDI's runtime behaviour on behalf of their managing Tracing Hub. Each Trace Unit encapsulates a specific low-level tracing mechanism (for example, hypervisor-based memory inspection, an eBPF program, a kernel module, or hardware-assisted tracing) and abstracts it behind a common interface so that higher layers of the TNDE do not depend on a particular implementation. In this way, the Tracing Hub can treat each Trace Unit as a logical tracing backend with well-defined configuration options and trace outputs, independent of the underlying software or hardware primitive.

Multiple Trace Units can be associated with a single TNDI at the same time, as illustrated in Figure 8.2. This allows CASTOR to combine different tracing mechanisms for the same TNDI, for example to obtain both system-level and networking traces, or to combine coarse-grained and fine-grained tracing for the same trust requirement as part of its multi-level tracing approach. Depending on the concrete Trace Unit implementation, a given Trace Unit may be instantiated once per TNDI or shared across several TNDIs, as long as isolation and correct attribution of trace data are preserved. While the current design assumes that each Trace Unit is backed by a single primary tracing mechanism, the abstraction does not preclude more complex implementations in which a Trace Unit internally orchestrates several complementary mechanisms, as long as they are presented as one logical tracing backend to the Tracing Hub.

Trace Units are configured on a per-TNDI basis by the Tracing Hub during TNDI onboarding. Their configuration can later be updated dynamically when the CASTOR Service Orchestrator sends tracing policy or trust model updates over the TNDI-SP control channel to the TN-DSM (see Section 7.3). The TN-DSM then deploys the resulting configuration changes to the Tracing Hub, which enforces them on its Trace Units. The concrete tracing configuration for a TNDI is derived from these policies and trust requirements, allowing the orchestrator to select which Trace Units should be active on a given TNDI and how they should be parameterised (for example, with respect to tracing granularity, performance–precision tradeoffs, and trace export behaviour).

8.2.0.0.2 Design Dimensions and Tradeoffs Different Trace Units can provide different tradeoffs and are intended to cover a spectrum of platform and deployment scenarios. In particular, they can differ along the following dimensions:

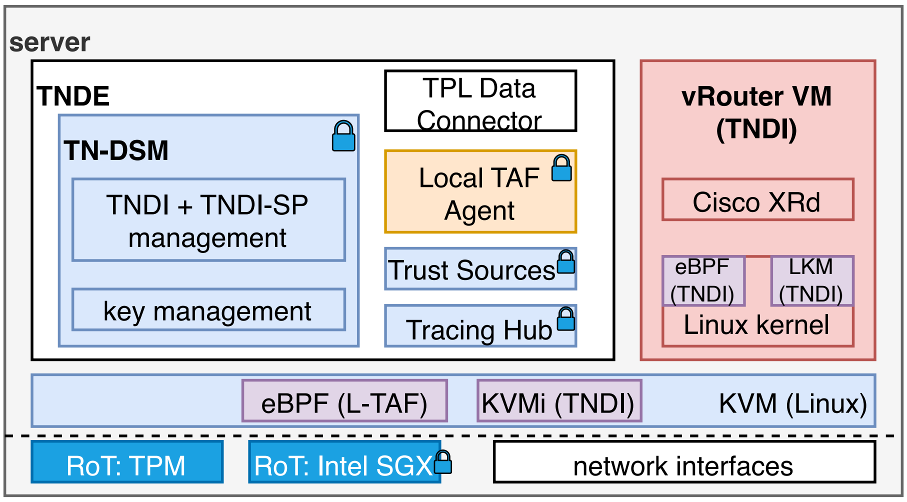


Figure 8.2: Placement of the four considered CASTOR Trace Units (purple colour). Three of them target TNDIs and one targets the Local TAF Agent (if considered not part of the TCB), as indicated in parentheses. Trace Units are considered to be part of the CASTOR device-side TCB and each of them provides different evidence, security, and performance tradeoffs.

- **Tracing primitives, capabilities, and evidence expressiveness:** Trace Units can observe different kinds of events and state, such as arbitrary memory contents, selected configuration structures, inline tracepoints in the control and data plane, or hardware-level execution traces. As a result, they support different types of evidence and propositions (for example, about firmware and configuration integrity versus routing behaviour) and operate at different granularities, which in turn determines how expressive the resulting traces can be for a given trust objective.
- **Placement and security properties:** Trace Units can be located outside the TNDI (for example, in a hypervisor, TEE, or other privileged execution context) or inside the TNDI (for example, inside the network OS or data-plane software). Their placement affects the level of isolation from a potentially compromised TNDI, the assumptions they make about the underlying RoT and isolation mechanisms, and the achievable security level.
- **In-TNDI changes and data-access difficulty:** Trace Units differ in how much explicit support they require inside the TNDI, ranging from no in-TNDI changes at all to additional instrumentation or tracing hooks (for example, kernel extensions, eBPF programs, or application-level probes). They also differ in how easily they can obtain the required data and events, from relatively straightforward access via well-defined APIs or tracepoints to more involved reconstruction from raw memory or low-level hardware traces.
- **Performance overhead:** Trace Units differ in how much runtime overhead they impose, depending on their tracing granularity, event rate, and where they execute. Highly expressive, fine-grained Trace Units can provide detailed traces but may incur higher CPU, memory, or I/O costs, whereas coarser-grained or sampled Trace Units may offer lower overhead at the cost of reduced visibility.
- **Portability and deployment constraints:** Depending on whether a Trace Unit relies on generic primitives (such as virtual machine introspection, simple memory reads, or widely available hardware tracing features) or on platform-specific extensions (such as a particular network OS API or vendor-specific instrumentation), its portability across device types and vendors can range from high to limited. This also captures differences in how easily Trace Units can be deployed and maintained in heterogeneous environments.

Across these dimensions, the traces that Trace Units provide can broadly be categorised into system-level traces (about the TNDI's execution environment and configuration) and networking traces (about the TNDI's control- and data-plane behaviour), as discussed in Section 8.1.3. Individual Trace Units may focus on one of these domains or cover both, depending on their implementation and placement.

8.2.0.0.3 Per-TNDI Configuration and Additional Targets By default, Trace Units collect traces on TNDIs, and their configuration is maintained per TNDI by the Tracing Hub. For each TNDI, the Tracing Hub decides which Trace Units to enable, how to parameterise them (for example, which events or memory regions to monitor, which sampling rates or batching sizes to use, and how to batch or stream traces towards the TNDE), and how to route their trace output to the appropriate Trust Sources, based on the policies and trust requirements obtained from the CASTOR Service Orchestrator.

While the primary focus is on tracing TNDIs, CASTOR also considers a variation of the device-side TCB in which the Local TAF Agent is excluded from the TCB to further minimise the TCB footprint, as described in Section 3.4.1. In such configurations, specialised Trace Units (instantiated with suitable tracing mechanisms for the platform) can be deployed to monitor the operation of the Local TAF Agent itself, providing traces about the local trust assessment and report-generation process that can be consumed by the Global TAF as additional information about the correctness and trustworthiness of the local trust computation.

8.2.0.0.4 System Model and Concrete Trace Units in CASTOR In the remainder of this section, we detail three (technically four) concrete Trace Units that we consider for CASTOR, which instantiate these design dimensions in different ways. In CASTOR version 1, especially in the context of the CASTOR use cases, we regard the following setting as our main system model on a server platform: As illustrated in Figure 8.2 (and described in Section 7.1.1), we consider a server hosting virtual machine-based vRouters, where each vRouter VM forms a dedicated TNDI that runs a Linux kernel as the network OS and a Cisco XRd container providing the routing stack. The server hosts the VMs using the KVM hypervisor of Linux. While only a single TNDI is depicted, the server might host several vRouters, each as a dedicated VM. The TNDE runs outside of all VMs, isolated by the hypervisor (forming an isolation layer, see Section 7.1.1).

We now consider the following Trace Units in this setting: First, an out-of-TNDI Trace Unit based on hypervisor-level memory inspection realises tracing from outside the TNDI's execution environment, offering stronger isolation at the cost of increased complexity and dependence on platform virtualisation features. Second, an in-TNDI Trace Unit based on eBPF leverages operating-system kernel tracing facilities to capture system-level and networking events inside the TNDI. Third, a kernel-module-based Trace Unit provides similar observability from within the TNDI OS, but with different deployment and integration characteristics. Figure 8.2 illustrates the placement of all three considered Trace Units and additionally shows a fourth Trace Unit targeting the Local TAF Agent instead of a TNDI (see previous paragraph). As the latter one is also based on eBPF and thus shares mechanisms with the second Trace Unit, we postpone a more detailed discussion of its specifics to D3.2.

As part of Section 8.3.2, we will briefly discuss implications on the Trace Units (especially the eBPF and kernel module ones) when considering a system model where TNDIs represent container-based vRouters (without VMs), as discussed in Section 3.2.2 and Section 7.1.1 (see Figure 3.2b).

8.2.1 Trace Unit 1: HV-based Memory Inspection

8.2.1.1 Background on Virtual Machine Introspection

Virtual machine introspection (VMI) is a well-established technique for observing the state and behaviour of a virtual machine from an external vantage point, typically the hypervisor, for purposes such as security monitoring, forensics, and debugging [47]. Existing systematization-of-knowledge works emphasise both the opportunities and challenges of VMI, in particular the semantic gap between low-level hardware views (for example, physical memory and CPU registers) and the high-level abstractions used by the guest operating system [47]. Prior work has explored a wide range of techniques to bridge this semantic gap and to apply VMI in security contexts, including agentless memory forensics, virtual-machine monitoring in cloud environments, and live tracing of system calls using non-intrusive memory sampling [61, 74]. This Trace Unit builds on these foundations but adapts VMI to the specific requirements and constraints of CASTOR's multi-level tracing, where continuous monitoring of performance-critical TNDIs is a central design goal.

8.2.1.2 Role, Placement, and Basic Design

The out-of-TNDI Trace Unit considered in our current CASTOR prototype is based on VMI from the hypervisor. Its main goal is to observe the runtime state of a TNDI without requiring any changes inside the TNDI itself and while remaining strongly isolated from a potentially compromised TNDI. Concretely, we assume a setting in which each TNDI runs as a virtual machine (VM) on a server-class hypervisor, or as a VM that hosts a partition of a physical router's network operating system. In this setting, we implement this Trace Unit using the KVMi VMI extensions for the KVM hypervisor on Linux [51]. The Trace Unit runs alongside the hypervisor in a privileged context that is part of the CASTOR device-side TCB and uses VMI to inspect the memory of the TNDI VMs.

8.2.1.3 Tracing Primitives and Extracted Information

The primary tracing primitive of this Trace Unit is non-intrusive, read-only memory inspection. It interfaces with KVMi to read selected memory regions of a TNDI VM without pausing the VM or modifying its execution [51]. In principle, modern VMI frameworks also support setting breakpoints and watchpoints, trapping selected events, or single-stepping the guest, which enables fine-grained control-flow tracing and rich debugging capabilities [47]. However, these features tend to introduce substantial runtime overhead and are therefore not suitable for continuous, live tracing of performance-critical systems such as network routers (i.e., TNDIs). This Trace Unit therefore focuses on read-only inspection and avoids heavy-weight traps or single-stepping in the common case.

Using memory inspection, the Trace Unit can extract system-level information and configuration state from a TNDI. For example, it can reconstruct process lists, kernel data structures, memory mappings, interface configuration, or routing tables by reading the corresponding memory pages and interpreting them with knowledge of the guest operating system (network OS) or network stack. Where the locations of relevant data structures can be identified reliably in memory, the Trace Unit can periodically poll them to monitor changes over time, thereby providing traces about configuration integrity, software inventory, and selected networking behaviour. For highly dynamic events, such as individual system calls, specialised VMI techniques have demonstrated that live tracing via periodic polling and careful reconstruction of register state is possible even without costly trapping mechanisms [61]. However, they rely on careful timing and high polling rates to not miss events and offer limited expressiveness (e.g., system call argument extraction) and therefore must be carefully evaluated in the context of CASTOR's continuous TNDI tracing.

8.2.1.4 Benefits, Limitations, and Alternative Designs

An out-of-TNDI design based on memory inspection offers several security and deployment benefits. Because the Trace Unit resides outside the TNDI and leverages the hypervisor and underlying virtualisation hardware as part of the trusted computing base, it is strongly isolated from an attacker who has compromised the TNDI's network operating system or routing stack. The Trace Unit also does not require any changes or additional components inside the TNDI, which simplifies deployment and avoids an in-TNDI

TCB. Furthermore, the ability to access the full virtual address space of the TNDI VM gives this Trace Unit broad visibility into the TNDI's state, at least in principle [47, 51].

At the same time, the Trace Unit faces well-known challenges of VMI. First, there is a semantic gap between the low-level memory view available at the hypervisor and the high-level abstractions used by the guest operating system and network stack; bridging this gap requires detailed and version-specific knowledge about in-memory data structures, and it must be maintained as software evolves [47]. Fortunately, many network operating systems nowadays are based on Linux and thus share well-known open-source kernel structures [73]. Second, live introspection without pausing the TNDI can lead to transient inconsistencies when data structures are being updated concurrently, which complicates the interpretation of extracted data and may require additional filtering or cross-checks to ensure that traces are meaningful. As a result, this Trace Unit is particularly well suited to monitoring relatively stable configuration and system state, or to providing periodic snapshots that complement more fine-grained in-TNDI tracing mechanisms, rather than to capturing every transient event in the TNDI in full detail.

Although our prototype instantiates this Trace Unit using a hypervisor-based VMI approach on KVM, similar out-of-TNDI memory-inspection Trace Units can also be realised on platforms that do not expose a conventional hypervisor interface. For example, recent work on DPU-based architectures shows how a physically isolated, DMA-capable SmartNIC can be used to perform efficient host and guest introspection, offloading VMI-like functionality from the main CPU and enabling introspection even on bare-metal machines [63]. In CASTOR, such DPU- or DMA-based designs would play a similar role to the hypervisor-based memory-inspection Trace Unit, by providing out-of-TNDI access to the TNDI's memory and state with strong isolation, but with different performance and deployment tradeoffs that depend on the underlying hardware platform.

8.2.2 Trace Unit 2: eBPF

The second Trace Unit that we consider uses eBPF programs to perform the trace collection from an in-kernel execution environment. As shown in Figure 8.2, in our server setting with per-TNDI virtual machines, the eBPF Trace Unit is located inside the TNDI VM when tracing a TNDI. That is, user programs within the TNDI will load eBPF programs into the network OS of the TNDI to perform the tracing. While native kernel extensions offer unrestricted access and computing (see third Trace Unit), eBPF introduces a virtualized, safety-first execution environment within a kernel that is perfectly aligned with the ”Below-Zero-Trust” requirement for high-integrity observability without compromising system stability. In the case where multiple (container-based) TNDIs are sharing the same host kernel, then eBPF monitoring hooks are instantiated at the kernel level and enable the monitoring from outside the TNDI with the same level of granularity.

eBPF introduces a radically different approach to in-kernel extensibility by interposing a virtualized execution environment between custom logic and native kernel execution. Rather than loading opaque native binaries, the kernel accepts programs expressed in a restricted bytecode language and subjects them to strict verification before execution. This model is often informally compared to sandboxed scripting environments, in the sense that it enables expressive logic to execute close to privileged resources without granting unrestricted authority over them. Unlike user-space sandboxes, however, eBPF operates entirely within the kernel context, enabling low-latency access to system events while preserving strong safety guarantees.

In the context of the CASTOR framework, the eBPF-based Trace Unit serves as a safety-first mechanism for extracting high-fidelity traces without the stability risks associated with native kernel extensions. Before an eBPF program is executed, the In-Kernel Verifier performs a symbolic analysis of all potential execution paths it. This process ensures memory safety by prohibiting out-of-bounds access and prevents system destabilization by ensuring the eBPF program is guaranteed to terminate. Once verified, the bytecode is Just-In-Time (JIT) compiled into native instructions, providing the performance required for real-time introspection while maintaining a secure isolation boundary.

For CASTOR, this Trace Unit is greatly positioned to collect both system-level and networking traces and it can be located either directly within the TNDI or outside at the host kernel level (depending on the vRouter setup). The Trace Unit can utilize, for example, kprobes and tracepoints to let its eBPF programs monitor vital network functions throughout the TNDI's lifecycle and capture fine-grained data on system calls and process transitions, characterizing the TNDI's overall state. This enables Trust Sources to analyze the collected traces and provide runtime information on potential violations affecting the control-plane logic of a TNDI, triggered by protocol-level anomalies such as BGP prefix hijacking or OSPF falsification. This information is then propagated to the Trust Assessment evidence-based theory, which reflects this behavior in the ATL calculations.

In principle, eBPF enables efficient sharing of raw traces (introspection results) via eBPF maps with user space components for processing (observability). eBPF maps (e.g., ring buffers) are efficient, shared-memory data structures that facilitate communication between the kernel and user space. That is, while raw introspection occurs within the kernel's eBPF runtime (the Trace Unit's eBPF programs), the resulting traces are passed to the Trace Units user space component and then securely forwarded to the TNDE for interpretation and processing (see channels in Section 8.1). This allows for the processing of the raw traces by the Trace Unit and Tracing Hub (e.g., encoding, authentication) and eventually the processing of structured, well-formatted traces by the Trust Sources.

8.2.2.1 Security-Performance Tradeoff

Positioning a Trace Unit within the TNDI constitutes a double-edged sword: while it enables increased visibility and tight integration with the TNDI for efficient and expressive inline tracing, it offers limited isolation compared to an out-of-TNDI Trace Unit. For instance, when deploying an eBPF-based Trace Unit, we inherently consider it part of the TCB and, therefore, outside the scope of the threat model. However, when adopting a weaker trust assumption in which the TNDI kernel is considered fully untrusted, any compromise of the TNDI environment may directly impact the integrity and availability of tracing capabilities for Trace Units deployed within it. If an attacker successfully compromises the TNDI and exploits a privilege escalation attack to gain control over its kernel, the output of the eBPF Trace Unit becomes insecure. This is a security-performance tradeoff compared to the previous VMI-based Trace Unit (Section 8.2.1) which provides strong isolation from TNDIs but at the cost of a semantic gap and live inspection challenges. Therefore, CASTOR follows its multi-level tracing approach, collecting traces on a TNDI's security state with multiple Trace Units and thus leveraging the benefits of each Trace Unit to mitigate weaknesses according to the envisioned threat model within a domain.

8.2.2.2 Background: The eBPF Virtual Machine

eBPF programs are typically written in a high-level language (e.g., Python, C) and then compiled into a deterministic bytecode set. This execution model is deliberately constrained to ensure that the monitoring logic has a predictable and minimal footprint: programs operate on a fixed set of eleven 64-bit registers ($r0$–$r10$) and a bounded stack of 512 bytes. From a trust perspective, these constraints are vital; they ensure that eBPF programs remain lightweight and restricted.

Just-In-Time Compilation: To maintain the performance required for high-speed routing functions (e.g., BGP or OSPF), the Trace Unit utilizes the kernel's JIT compiler to speed up the execution of its eBPF programs. Upon successful verification, eBPF bytecode is translated into native machine instructions for the host architecture ($x86 - 64$). This eliminates interpretation overhead and thus enables a tracing performance comparable to handwritten kernel code. This is a critical enabler for the Below-Zero-Trust model, as it permits real-time, high-frequency tracing of a TNDI without introducing latency (runtime overhead) that could disrupt the TNDI's operational throughput.

Register Mapping: The JIT compiler maps eBPF virtual registers directly onto physical CPU registers, minimizing memory accesses and register spills. In CASTOR, this property allows the Trace Unit (via its eBPF programs) to be effectively inlined into hot kernel execution paths — e.g., routing table updates, control-plane protocol handling (e.g., BGP or OSPF message processing) — with negligible overhead. This tight integration enables CASTOR to experiment with tracing router behavior as control-plane packets and state transitions are processed, potentially yielding additional traces that enhance the Trust Sources' ability to generate trustworthiness evidence and, consequently, improve the accuracy of the trust assessment process.

Control-Flow Analysis: eBPF programs are restricted in computation by the eBPF verifier to ensure the stability of the kernel. When an eBPF program gets loaded into the kernel, the eBPF verifier constructs a complete control-flow graph (CFG) of the program and performs a symbolic execution of all possible paths. By enforcing strict bounds on loops and instruction counts, the verifier guarantees termination for all executions of the eBPF program to prevent kernel freezes. That is, the eBPF programs of the Trace Unit will not cause hangs or a denial-of-service of the TNDI's network OS, maintaining its availability during the tracing process.

Memory Safety: Each register within eBPF programs is tracked with precise type information, distinguishing scalar values from pointers to kernel-managed objects such as packets, maps, or context structures. Pointer arithmetic is tightly constrained; for instance, the eBPF verifier prevents the creation of out-of-bounds offsets that could lead to unauthorized kernel memory reads. Any potential out-of-bounds access results in immediate rejection of the program. Within CASTOR, this spatial safety ensures that an eBPF Trace Unit remains strictly within its allocated "sandbox," preventing the leakage or corruption of sensitive routing tables or cryptographic keys during the introspection process. This is particularly relevant when an eBPF Trace Unit cannot be strictly contained within the TNDI (e.g., TNDI containers sharing the host kernel rather than per-TNDI VMs) or in CASTOR's alternative TCB model (see Section 3.4.1 and Section 8.2) where an eBPF Trace Unit might target the Local TAF Agent rather than a TNDI and is thus located close to the TNDE.

Uninitialized State Protection: To maintain the purity of the collected traces, the eBPF verifier ensures that all registers and stack slots are initialized before use. This prevents the inadvertent leakage of residual kernel stack contents into the trace stream. In the context of the trust assessment process, this property ensures that the traces reported by the Trace Unit to the TNDE's Tracing Hub are not tainted by "noise" or undefined kernel states, providing a clean and deterministic signal.

In the CASTOR framework, the practical utility of the eBPF-based Trace Unit is determined by its ability to intercept, contextualize, and safely process system events across the network and compute layers. This capability is realized through a well-defined hierarchy of attachment points, or hooks, which allow CASTOR to adjust the granularity of its tracing based on the provisioned trust policy. These hooks expose different layers of kernel and application behaviour, offering varying tradeoffs between semantic stability and execution context. Crucially, all data collection mechanisms operate under the strict governance of the eBPF verifier, which bounds their expressiveness to preserve the integrity of the "Below Zero-Trust" environment.

8.2.2.3 Hierarchy of eBPF Hooks

@Thanassis to check whether the long-term monitoring of critical network functions make sense here.

Tracepoints represent the most stable and robust source of longitudinal evidence for the CASTOR framework. Statically defined by kernel maintainers, tracepoints such as *sched_switch* or networking-specific events provide structured metadata about the system's state without requiring the Trace Unit to interact directly with volatile kernel internals. In the CASTOR context, these are utilized for long-term monitoring of critical network functions, as their stability ensures that trust measurements remain consistent across kernel upgrades. When a tracepoint is triggered, the Trace Unit receives a read-only context structure, allowing it to extract high-fidelity trace information with minimal measurement drift. This makes tracepoints ideal for documenting the steady-state behaviour of a TNDI and detecting long-term deviations from its certified execution model.

For more granular or exploratory introspection, CASTOR can leverage robust dynamic instrumentation mechanisms (e.g., kprobes) to observe and analyze kernel execution behaviour. This mechanism is particularly valuable for identifying sophisticated, networking-driven intrusions that target internal routing logic, such as the manipulation of BGP or OSPF state machines. By attaching to functions like *fib_lookup* or protocol-specific handlers, the Trace Unit can inspect function arguments and control flow at the moment of execution. While Kprobes provide deep visibility, they are inherently fragile due to their dependence on internal kernel symbols and calling conventions. Within the CASTOR lifecycle, Kprobes are typically activated during high-assurance phases or targeted incident responses where the Local TAF requires maximum-fidelity evidence to resolve a sudden drop in the ATL of a node or an entire path.

Uprobes can extend CASTOR's observability into the application layer, allowing the Trace Unit to observe the router's internal control- and forwarding-plane behavior without modifying its software stack (i.e., no binary instrumentation). This can help to detect abnormal execution paths that remain "invisible" to the kernel — such as unexpected changes in routing state, deviations in protocol state machines, or unauthorized modifications to forwarding entries that may be triggered by control-plane events that are originated from the network segment or topology. Nevertheless, Uprobes impose higher overhead, largely because each instrumentation event requires a (costly) transition between user and kernel space.

8.2.3 Trace Unit 3: kernel module

While eBPF provides a safe and flexible environment for tracing, the collection of certain traces can require unrestricted privileged access and operation that exceed the eBPF verifier's safety restrictions. To address these cases, CASTOR's prototype considers another in-TNDI Trace Unit that leverages a specialized loadable kernel module (LKM) within the TNDI's network OS for deep inspection.

This CASTOR Trace Unit will perform its tracing via a native kernel extension — typically packaged as Executable and Linkable Format (ELF) objects in Linux linked against the kernel's exported symbol table. Upon loading, the module is dynamically relocated and integrated into the kernel address space, becoming indistinguishable from the core kernel code from the processor's perspective. This architectural integration is necessary to provide the deep, low-latency introspection required for CASTOR's system-level trace generation.

On x86-64 systems, the processor enforces a hierarchical privilege model (Rings), where the kernel executes in Ring 0 and user applications are confined to Ring 3. As the Trace Unit's LKM executes directly in Ring 0, it inherits the full authority of the kernel. This privilege level permits the execution of sensitive instructions essential for high-fidelity tracing, including interrupt control and manipulation of control registers governing memory translation

8.2.3.0.1 Capabilities and Operational Responsibilities: As the Trace Unit's kernel module is executing directly as part of the network OS within the TNDI, the Trace Unit gains full access to the TNDI, but also must avoid introducing operational failures into the TNDI:

- **Direct Memory Access:** The Trace Unit can access any virtual address mapped in the kernel's page tables. This allows for the traversal and modification of security-critical data structures, such as process descriptor lists or system call dispatch tables, providing the raw signals for system-level traces.

- **Synchronization Responsibility:** Without an eBPF verifier, concurrency control of the kernel module lies entirely in the responsibility of the Trace Unit developer. The correct implementation of spinlocks, mutexes, and Read-Copy-Update (RCU) primitives is mandatory. Any deviation—such as sleeping while holding a spinlock—can lead to immediate kernel failure. In the CASTOR context, the Trace Unit must be engineered to ensure that the act of ”observing” (tracing) does not destabilize the TNDI or the underlying system it is attempting to protect.

8.2.3.1 Fault Isolation and Security Considerations

The kernel module of the Trace Unit executes directly as part of the TNDI's kernel. That is, the module shares the same stack, heap, and address space as the TNDI kernel. Consequently, any memory corruption originating within the Trace Unit's module propagates directly into unrelated kernel subsystems of the TNDI. In Linux, an invalid memory access within the module triggers a kernel panic, halting the system to prevent further corruption. That is, in contrast to the eBPF-based Trace Unit with its verifier-vetted eBPF programs, there is no fault isolation between the Trace Unit's kernel module and the TNDI network OS. The Trace Unit must be carefully implemented to not negatively impact the stability and security of the TNDI.

Furthermore, this in-TNDI Trace Unit has a similar security-performance tradeoff as the eBPF-based Trace Unit: performing tracing at the kernel-level inside the TNDI provides efficient tracing capabilities with great visibility for expressive traces, however, puts the Trace Unit at risk on a TNDI compromise. If an attacker gains full control over the TNDI's kernel, it can also control the Trace Unit's kernel module, rendering its traces insecure. That is, from a ”Below Zero-Trust” standpoint, this means: once the TNDI's kernel integrity has been compromised, this Trace Unit can no longer provide trustworthy traces. Therefore, as mentioned in the previous section (Section 8.2.2), CASTOR needs to leverage its multi-level tracing to detect anomalous behaviour indicating a potential compromise of a TNDI based on traces from multiple Trace Units with different security-performance tradeoffs. That way, even if one in-TNDI Trace Unit is under attack, the other out-of-TNDI (and in-TNDI) Trace Units can capture the TNDI's security state, potentially also revealing a kernel compromise affecting the in-TNDI Trace Units.

8.3 Evidence Richness: The Visibility Gap

In the CASTOR framework, the practical utility of the Trace Unit is defined by the balance between depth of visibility and system safety. While both native kernel extensions and eBPF offer powerful mechanisms for trace collection, there is a clear distinction on the threat model that is unlocked per solution. Especially, for the former, extending the kernel stack with native monitoring functions that can introspect each single memory structure access and/or the management of all registers, also exposes this (sensitive) information to adversaries, in the case of compromise. Thus, there is always the balance and trade-off between the richness of monitored evidence so as to be able to expose the minimal (yet enough) set of traces to allow for the runtime trust quantification of the host element but without risking its safety in case of such evidence can also become available to an adversary.

Furthermore, another important dimension is the **visibility or translation capability of the monitored evidence:** In the case of kernel extensions, it is the kernel that already knows how to interpret the mapping of which evidence capture the behaviour of what processed (possibly the TNDI). A native kernel extension executes with unrestricted access to the kernel's entire virtual address space. It may cast arbitrary addresses into structure pointers, dereference them freely, and traverse kernel memory without mediation. This capability enables powerful forensic techniques. A driver can manually walk kernel linked lists—such as the global process list —without relying on exported APIs or debug symbols.It can scan heap memory for signatures indicative of stealthy persistence mechanisms, such as unlinked process descriptors or tampered scheduler objects, even when these artifacts are deliberately concealed from standard kernel interfaces. If such processing is to be offloaded outside the kernel, then this information (symbol regime) needs to also be exposed by default increasing the risk level that we are allowing the system to operate - such forensics functionalities can be misused by an adversary.

In contrast, eBPF operate under a read-only by default model with respect to kernel memory. Direct pointer dereferencing is forbidden. Instead, all memory access must be mediated through dedicated helper functions which safely copy data while handling faults and enforcing bounds checks. The verifier further constrains access by requiring that all memory references be provably derived from a known and trusted context—such as a pointer provided by a tracepoint, kprobe, or LSM hook. As a result, eBPF programs cannot perform blind memory scans or heuristic pattern matching over the kernel heap. This restriction decisively prevents entire classes of attacks but also precludes certain forensic techniques. Hidden objects that are not reachable through legitimate kernel references remain invisible to eBPF. Consequently, eBPF favours integrity-preserving observation over exhaustive memory introspection, aligning with attestation goals that prioritize correctness and non-interference over completeness.

Although eBPF enables unprecedented levels of safe, in-kernel observability, it is not omnipotent. When contrasted with native kernel extensions, a clear visibility gap emerges—one in which safety, portability, and verifiability are traded against unrestricted access to kernel internals. In CASTOR, this gap is a deliberate architectural choice, reflecting a fundamental tension between evidence richness (completeness) and system survivability (integrity). Understanding this gap is essential for the Trust Assessment Framework, as it defines which classes of evidence can be collected to justify a TNDI's state and which remain inaccessible without expanding the trusted computing base.

8.3.1 Arbitrary Memory Access vs. Read-Only Integrity

Kernel Extensions (Native Trace Unit): A Trace Unit with a native kernel extension (Section 8.2.3) executes with unrestricted access to the TNDI kernel's entire virtual address space. It may cast arbitrary addresses into structure pointers, dereference them freely, and traverse kernel memory without mediation. Within CASTOR, this capability enables powerful forensic techniques for high-assurance trust assessment. A Trace Unit with a kernel extension/module can manually walk kernel linked lists—such as the global process list—without relying on exported APIs. It can scan heap memory for signatures indicative of stealthy persistence mechanisms, such as unlinked process descriptors or tampered scheduler objects, even when these artifacts are hidden from standard kernel interfaces.

Moreover, these extensions possess write capabilities, allowing them to modify kernel data structures or executable code at runtime. While this underpins mechanisms like live kernel patching, it also introduces the risk of Direct Kernel Object Manipulation (DKOM). From a purely evidentiary standpoint, this write access allows for the strongest possible introspection, but it simultaneously undermines the trustworthiness of any measurements produced since we are weakening the trust guarantees of the trace units binded to a custom kernel.

eBPF (Sandboxed Trace Unit): In clear contrast, the eBPF Trace Unit (Section 8.2.2) operates under a "read-only by default" model inside the TNDI's kernel. Direct pointer dereferencing is forbidden; instead, all memory access must be mediated through dedicated helper functions, such as *bpf_probe_read_kernel()*, which safely copy data while handling faults and enforcing bounds checks. The eBPF verifier further constrains access by requiring that all memory references be provably derived from a known and trusted context, such as a pointer provided by a tracepoint or kprobe.

In the context of the Below-Zero-Trust model, this restriction decisively prevents entire classes of attacks (like memory corruption by the monitor itself) but also precludes certain forensic techniques. Hidden objects unreachable through legitimate kernel references remain invisible to eBPF. Consequently, CASTOR favors the eBPF Trace Unit for continuous, integrity-preserving observation where correctness and non-interference are prioritized over exhaustive, high-risk memory scans.

Virtual Machine Introspection (Out-of-TNDI Trace Units) As opposed to the previous in-TNDI introspection techniques, hypervisor-based (Section 8.2.1) and out-of-band (e.g., using a DPU) VM introspection operates entirely outside the TNDI VM, with no reliance on the TNDI kernel. Memory access is performed indirectly via the hypervisor or dedicated monitoring hardware (e.g., a DMA-capable DPU), without modifying or pausing the target VM which forms (or hosts) the TNDI. This approach ensures that the TNDI cannot tamper with the monitoring process of the Trace Unit, providing strong isolation and trust guarantees even when the TNDI gets fully compromised. However, as explained in Section 8.2.1, the Trace Unit trades in the strong security guarantees for performance and precision challenges. VMI faces a semantic gap requiring the interpretation of the raw VM memory and the address identification of the relevant data structures [47], as well as live introspection challenges. When the VM continues execution while memory is being read, introspection can suffer from a "smearing effect" [63], where traced memory content may change during access. While hypervisor-based trapping mechanisms can eliminate these effects, they pause the VM and thus cause non-acceptable performance overhead. Therefore, to address this, CASTOR aims to explore different memory introspection techniques as part of the Trace Unit 1 design.

8.3.2 The Identity Gap: Daemon Processes vs. Container-Aware Primitives

8.3.2.0.1 Host vs. VM Introspection and Container Identification in the Case of VMI: In a setting where each TNDI forms a single VM, a VMI-based Trace Unit (Section 8.2.1) can easily map traces to TNDIs based on the VMs from which they were extracted. Each VM has its own VM control data structures and associated memory space, known to the hypervisor and thus directly usable by hypervisor-based Trace Units. Out-of-band VMI Trace Units, e.g., based on DPUs (not further discussed in Section 8.2.1), need to explicitly locate and parse these structures in the host memory.

In the server-based setup described in Section 8.2, where each TNDI VM runs a Linux OS and XRd container, the VMI Trace Units additionally benefits from the capability of mapping processes to the XRd container within the VM. That way, the Trace Unit can see which processes run in the context of the XRd routing and networking stack of the TNDI. CASTOR can explore VMI strategies to perform the process-to-container mapping based on the same kernel data structures and similar ideas, but using raw VM memory introspection to locate and parse them. In this context, the VMI Trace Units again face the previously discussed semantic gap and live inspection challenges.

8.3.2.0.2 Going beyond the Legacy Kernel Extension View: For a native kernel extension, a container is not a first-class object but rather an abstraction synthesized from a collection of standard kernel entities. From the perspective of the core kernel, a container is essentially a daemon process (or a tree of processes) isolated through namespaces. When a kernel module-based Trace Unit intercepts an event, it sees only a Process ID (PID) or a task structure. In the instantiation case where multiple TNDIs may share the same underlying kernel on top of a common infrastructure element, the attribution of such an event to a specific TNDI (i.e., virtualized router) requires to traverse through the process hierarchy and parse human-readable cgroup strings. This process is inherently fragile; because the kernel does not maintain a persistent "container ID" in its legacy task structures, any change in the orchestration layer or a minor kernel update to cgroup management can lead to misattribution.

8.3.2.0.3 The eBPF View: With eBPF, container identification is streamlined: built-in helpers allow direct retrieval of the relevant cgroup ID, simplifying the Trace Unit logic compared to implementing the same functionality in a native kernel extension. Instead of relying on ephemeral PIDs or manual path traversal, the eBPF-based Trace Unit utilizes atomic kernel helpers like *bpf_get_current_cgroup_id()*. This helper returns a stable, 64-bit identifier that uniquely maps to the container's resource boundary at the moment of execution. This allows CASTOR to bypass the "daemon process" abstraction and observe the system as a collection of isolated workloads. In the modern computing landscape of containerd 2.0 and high-scale Kubernetes clusters, this provides a decisive advantage: the Trace Unit can associate every syscall with a specific TNDI identity in a deterministic manner. This stable mapping ensures that the Actual Trust Level (ATL) is calculated for the appropriate TNDI, even if the underlying PIDs are recycled or the container is rescheduled across nodes.

8.3.2.0.4 Expressiveness and Performance: This architectural shift from ”parsing process hierarchies” to ”direct ID mapping” also removes a significant performance bottleneck. While a native extension must perform complex, lock-heavy traversals to identify a container (increasing the risk of race conditions), the eBPF Trace Unit performs a lock-free lookup. This allows CASTOR to maintain high-fidelity observability even in dense multi-tenant environments, where hundreds of TNDIs may be competing for the same kernel resources. By establishing this clear container-to-trace mapping (in the instantiation scenario where multiple TNDIs share a common kernel), eBPF enables CASTOR to fulfil the Below Zero-Trust requirement of non-repudiable attribution, ensuring that every piece of trace can be linked back to its specific source without the ambiguity inherent in legacy daemon-monitoring approaches.

Chapter 9

CASTOR Network Attestation for Secure Routing

Trusted Path Routing in the Compute Continuum requires trust decisions to be taken 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.

At the same time, CASTOR operates in multi-domain environments where routing elements cannot disclose raw trust values or detailed evidence to other domains. Nevertheless, routing decisions require the ability to compare trust properties across routers, such as verifying that all elements on a path satisfy a given trust threshold. This requires mechanisms that enable **ordering-based comparison of trust values without revealing their actual value**, motivating the use of Order-Revealing Encryption (ORE). Furthermore, composed trust assertions must preserve the **integrity, authenticity, and freshness** of evidence produced by each device-side TCB and must scale efficiently to multi-hop paths in the routing plane. Together, these requirements motivate the use of **composite attestation combined with ORE** as core building blocks of CASTOR's network attestation architecture, which are formalised in the following sections.

9.1 CASTOR Network Architecture

As described in Section 3.4.2, within the CASTOR architecture, evaluation of a trusted routing path requires the Service Orchestrator to monitor evidence from each TNDE along the path using composite attestation. Recalling the structure of composite attestation shown in Figure 3.5, from a cryptographic perspective, each TNDE may consist of one or more TNDIs. During the evaluation process, each TNDE generates and forwards Compound Evidence (CE) to the next TNDE, where it may be optionally verified and subsequently aggregated. This process continues until the final TNDE forms the Aggregate Attestation Result (AAR) and transmits it to the Service Orchestrator, which acts as a relying party and verifies the AAR to confirm both the number and the ordering of routers in the enforced path. In this process, if a router becomes compromised, it should be detected and revoked by the Service Orchestrator. However, achieving revocation within an aggregated signature is highly challenging, as further explained in the following section. Furthermore, when the trust model extends beyond a single segment, the attested claims are inherently designed to be verifiable by and linked to, those from neighbouring segments.

In inter-domain settings, a BGP node should be able to be aware of the minimum trust level that a path can exhibit to serve a target service, without requiring knowledge of all individual trust levels along the path. Specifically, to address this confidentiality requirement, particularly in scenarios involving composite attestation across multiple infrastructure elements or continuous state reporting to adjacent routers, CASTOR adopts Order-Revealing Encryption (ORE) primitive to achieve the order of all trust levels of routers in a domain, allowing the minimum trust level to be derived without disclosing the underlying values. In an ORE primitive, the order of plaintexts can be obtained by comparing ciphertexts without revealing plaintexts. In the following sections, we introduce the cryptographic structures and building blocks that satisfy the aforementioned CASTOR's requirements (e.g., confidentiality and ordering of trust values etc.).

9.2 Crypto Structures & Building Blocks

From a cryptographic perspective, two primitives are required. First, CASTOR needs **composite attestation**, a mechanism to produce an aggregated/combined proof of TNDIs arranged in sequence along an enforced routing path, which can be evaluated by the Service Orchestrator. To achieve this target, the possible solution is through an order-preserving multi-signature scheme. Second, a BGP node should be aware of the **minimum trust level**, without revealing the individual trust levels, that a candidate path can exhibit to serve a target service. A potential solution to this requirement is (Delegatable) Order-Revealing Encryption (DORE). The following sections, describe these two solutions in the context of CASTOR, discuss relevant state-of-the-art schemes, and define their syntax as well as functional and security requirements.

9.2.1 Composite attestation

Potential solutions to achieving composite attestation. To achieve composite attestation, several fundamental requirements must be satisfied first to ensure both security and practicality. These requirements are summarised below:

1. There should be multiple signers/verifiers, while the provers should be instantiated by routers.
2. Multiple proofs generated by different routers along an enforced path should be collected by the Service Orchestrator, enabling the evaluation of both the number and order of routers in the path.
3. Revocation of compromised or rogue routers is required by the global TAF during evidence collection.
4. A prover's identity should be cryptographically bound to secret keys used to generate proofs, in order to prevent impersonation attacks.

To achieve composite attestation, a natural first candidate is aggregatable signature primitive [41]. However, standard aggregatable signature cannot guarantee that all combined proofs follow a predefined order. To support ordered composition of proofs, *ordered multi-signatures* [13] or aggregatable signature in sequence [55] can be considered. Compared to conventional digital signatures, these schemes (e.g., ordered multi-signatures/sequential aggregatable signatures) supports both aggregation of proofs and the preservation of signer order within the aggregated proof. In addition, CASTOR requires support for revocation and for binding a router's secret signing key to its root identity key (i.e., the TNDE platform key).

Revocation is particularly challenging in aggregated signature settings. In general, revocation can be achieved through mechanisms such as verifier-local revocation (VLR), online certificate status protocol

(OCSP) and certificate revocation lists (CRLs). In VLR, the verifier maintains a locally stored revocation list (RL) and checks credentials without contacting an online authority. In OCSP, the verifier queries a trusted online service in real time to check credential validity. In CRLs, the issuer periodically publishes and signs a list of revoked certificates, which verifiers must consult during verification. Beyond these approaches, Direct Anonymous Attestation (DAA) [16], introduces linkability, which allows an entity's signing ability to be withdrawn via link tokens without revealing its identity.

While these revocation mechanisms are well understood for non-aggregated signature schemes, revocation becomes significantly more complex in aggregated signatures depending also on the scenario. Revoking a signer from a sequential group is not trivial, because the order of the remaining valid users and the aggregated proof excluding the revoked signer must still be guaranteed, which is more complicated and challenging. To date, the state-of-the-art schemes, revocable aggregated signature scheme [83] and ordered multi-signature scheme [8], can not support all aforementioned requirements.

The state-of-the-art schemes. The first ordered multi-signature scheme was proposed by Boldyreva *et al.* [13], allowing multiple signers to sequentially produce a compact, fixed-length signature simultaneously attesting to the message(s) they want to sign. However, this scheme is based on Type I pairings, which have been shown to have serious security issues [39]. Since then, several works have addressed ordered multi-signatures, [82] without random oracle used in proofs, [77] with public key aggregation and constant signature size, [8] removing the pairing operations to improve efficiency of verification. The state-of-the-art ordered multi-signature scheme is [8]. However, the scheme in [8], while aggregating multiple signatures, only supports signatures on the same message. In contrast, sequential aggregate signatures can support aggregation of signatures on multiple messages, a notion first proposed in [55]. Subsequent works on sequential aggregate signatures include [9, 13, 60]. In these schemes, signers compute the aggregated signature in order: the output of each signer is used as input to the next signer, who must verify the received aggregated signature before signing. This is referred to as the "general case of sequential aggregate signatures". In CASTOR, however, each router does not need to verify the previously aggregated signature; it simply receives, combines, and forwards signatures. This aligns with the notion of history-free sequential aggregate signatures, in which each participant can sign without verifying previously received signatures, and the aggregation is independent [17, 36, 40, 66]. Existing history-free schemes, however, do not support revocation or identity-signing key binding, which are essential for CASTOR. Therefore, CASTOR requires the design of a novel signature scheme for composite attestation to establish trusted routing paths.

Below, we provide, in order, a definition of a general digital signature followed by the definition of a sequential aggregate signature (ordered multi-signature), focusing on the history-free case, which aligns more closely with the requirements of the CASTOR project. These definitions are presented sequentially to help the reader better understand the concepts. We then list the functional requirements and security properties that the aggregated signature (ordered multi-signature) must satisfy. All notation is summarized in Table 9.1.

Definition of a basic digital signature. In a basic digital signature, there are two entities, i.e., a signer and a verifier. In CASTOR, a signer can be instantiated by a router and a verifier can be any entity other that the signer. A basic digital signature is denoted as **SIG** $= (\text{KeyGen}, \text{Sign}, \text{Verify})$, where the details of each algorithm are described as follows:

- **KeyGen**: The key generation algorithm takes the security parameter λ as input and outputs a pair of secret/public key, i.e., (sk, pk).
- **Sign**: Given a secret key sk and a message $M \in \{0,1\}^*$ to be signed, the signing algorithm generates a signature, σ.
- **Verify**: Given a public key pk, a message $M \in \{0,1\}^*$, and a signature σ, the verification algorithm verifies whether the signature σ is valid or not. If yes, the output is '1' for accepting this signature; otherwise, the output is '0' for rejecting this signature.

Notation	Description
λ	security parameter
sk	secret key for the signer in a basic signature scheme
pk	public key for the signer in a basic signature scheme
M	message to be signed in a basic signature scheme
σ	a signature/aggregated signature
i	index for a signer in composite attestation
n	the number of signers in composite attestation
sk_i	secret key for the signer i in composite attestation
pk	public key for the signer i in composite attestation
m_i	the message to be signed by the signer i
σ_i	a signature generated by the signer i
lk_i	link token for a signer i
t	a threshold value

Table 9.1: Notation summary for composite attestation

Definition of an ordered multi-signature. Composite attestation can can be realized by an aggregatable signature scheme, such as the ordered multi-signature primitive. Such a scheme involves several entities such as multiple signers, multiple verifiers, and a Service Orchestrator. Note that revocation must be considered in the context of aggregated signatures; here, we follow the approach used to achieve linkability in [16]. Furthermore, the key-binding property is required, therefore, we introduce an additional algorithm IdKeyGen to generate identity keys. The aggregatable signature scheme with order is then denoted as **ASIG** $=$ (Setup, IdKeyGen, aKeyGenaSign, aVerify) with the details described as follows:

- **Setup**: Takes a security parameter λ as input and outputs the public parameters pp and, if necessary, a master secret/public key pair (msk, mpk) for the key generation centre.
- **IdKeyGen**: Each signer i generates an identity key pair (isk_i, ipk_i).
- **aKeyGen**: Each signer i interacts with the key generation center providing its identity i and its identity key pair (isk_i, ipk_i) as inputs. The key generation center, which holds the secret/public key pair (msk, mpk), generates a signing secret/public key pair (sk_i, pk_i).
- **aSign**: This algorithm is used to aggregate multiple signatures by one entity. Given the secret/public key pair (sk_i, pk_i), a *so-far aggregated signature* σ_{i-1}, a message m_i to be signed, an entity generates a new aggregated signature σ_i, where the ordered public key list L is involved in the computation of σ_i.
- **aVerify**: Given an ordered public key list $L = \{pk_1, ..., pk_i\}$, a message m_i, and an aggregated signature σ_i, this aVerify verifies whether the aggregated signature σ_i is valid or not and whether there is one or more rogue signers. Apart from checking the validity of a signature, the verifier also needs to check whether the signer is revoked or not. If both checks are passed, the output is '1' for accepting this signature; otherwise, the output is '0' for rejecting this signature.

Based on the CASTOR project setting, several requirements must be satisfied. We categorize these requirements into two types: functional requirements and security requirements.

Functional requirements. We first list functional requirements as follows:

- **Revocation**: If a signer is compromised, it must be revoked. Here, we adopt the approach used in DAA to achieve revocation i.e. *link token*.

- **Performance**: The size of the proof should be as small as possible; ideally, it should be constant and independent of the number of aggregated signers. In addition, the running time for proof aggregation and verification should be as short as possible.

Security requirements. Similar to a general digital signature, the security requirements of the aggregated signature is *correctness* and *unforgeability*. Moreover, to prevent impersonation in composite attestation, we include the *identity-key binding* property.

- **Correctness**. Correctness means that a valid signature generated using a legitimate secret key must be successfully verified by the verification algorithm Verify, and a valid aggregated signature must be must be successfully verified by the aggregated verification algorithm aVerify.
- **Unforgeability**. Unforgeability means that an adversary that does not have a valid secret key can not forge an aggregated signature that can pass the verification algorithm aVerify.
- **Identity-key binding**. Any adversary, who does not know both valid identity key and secret key, cannot generate a valid signature that can pass the verification.

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

Threat model. An adversary $\mathcal{A}$ can launch the following queries and attacks:

- **Queries**: The adversary $\mathcal{A}$ can query key generation oracle, signing oracle and aggregated signing oracle to get valid keys, signature and aggregated signature except the target user to be forged. Details of these oracles will be provided later.
- **Attacks**:
  1. Attempt to forge a signature by getting a valid identity key isk without knowing the signing key sk.
  2. Attempt to forge a signature by getting a valid signing key sk without knowing the identity key isk.
  3. Attempt to forge a signature without knowing the identity key isk and the signing key sk.

Definition 1. (**Correctness**) An aggregatable signature scheme **ASIG** is correct, if Setup$(\lambda) \rightarrow (\mathsf{pp}, \mathsf{msk}, \mathsf{mpk})$, IdKeyGen$(\lambda, i) \rightarrow (isk_i, ipk_i)$, aKeyGen$(\lambda, \mathsf{msk}, \mathsf{mpk}, isk_i, ipk_i) \rightarrow (sk_i, pk_i)$, and aSign$(sk_i, \sigma_{i-1}, m_i) \rightarrow \sigma$ then it guarantees aVerify$(L, m_i, \sigma_i) = 1$.

Unforgeability. The unforgeability of **ASIG** is defined using an experiment $\mathrm{Exp}_{\textbf{ASIG},\mathcal{A}}^{EU-CMA}$, which is played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. It is shown in Figure 9.1, where $\mathcal{M}$ is the message space used by $\mathcal{A}$ to query to the signing oracle Sign and aggregated signing oracle aSign. Given public information, the key generation oracle KeyGen and signing/aggregated oracle Sign/aSign are defined as follows:

- aKeyGen$(\lambda, i, \mathsf{msk}, \mathsf{mpk}) \rightarrow (sk_i, pk_i)$: Given the security parameter λ, when an adversary $\mathcal{A}$ makes a query to the keyGen oracle for a signer i, this oracle returns two pairs of keys, i.e., the identity key pair (isk_i, ipk_i) and the signing key pair (sk_i, pk_i).
- aSign$(pk, m_i, \cdot) \rightarrow \sigma_i$: When an adversary $\mathcal{A}$ make a query by using a public key pk, this aggregated signing oracle returns an aggregated signature σ_i.

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

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

- (pp, msk, mpk) $\leftarrow$Setup(λ).
- $(L^*, lk_i, m_i^*, \sigma^*) \leftarrow A^{\text{aKeyGen}(\lambda,i),\text{aSign}(m,pk,\cdot)}$(pp, mpk).
- Return 1 iff
  - ✓ aVerify$(L^*, lk_i, m_i^*, \sigma^*) = 1 \wedge m_i^* \notin \mathcal{M}$, where $L^* = \{pk_1, ..., pk_i\}$;
  - ✓ $\exists pk_{i^*} \in L^*$ has not been queried before;
  - ✓ $L^* = \{pk_1, ..., pk_{i-1}\}$ has not been queried to aggregate signing oracle aSign before.

Figure 9.1: EU-CMA game for an aggregatable signature.

Experiment $\mathrm{Exp}^{key-bind}_{\textbf{ASIG},\mathcal{A}}(\lambda)$

- (pp, msk, mpk) $\leftarrow$Setup(λ).
- $(L^*, m_i^*, \sigma^*) \leftarrow A^{\text{IdKeyGen}(\lambda,i),\text{aKeyGen}(\lambda,i),\text{aSign}(pk,m_i,\cdot)}$(pp, mpk).
- Return 1 iff
  - ✓ aVerify$(L^*, lk_i, m_i^*, \sigma^*) = 1 \wedge m_i^* \notin \mathcal{M}$, where $L^* = \{pk_1, ..., pk_i\}$;
  - ✓ $\exists pk_{i^*} \in L^*$ has not been queried before;
  - ✓ $L^* = \{pk_1, ..., pk_{i-1}\}$ has not been queried to the aggregate signing oracle aSign before;
  - ✓ ipk_{i^*} associated with pk_{i^*} have not been queried before.

Figure 9.2: Key-Binding game for an aggregate scheme.

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

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

Key-Binding. The key-binding of **ASIG** is defined using an experiment $\mathrm{Exp}^{key-bind}_{\textbf{ASIG},\mathcal{A}}$, which is played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. It is shown in Figure 9.2, where $\mathcal{M}$ is the message space used by $\mathcal{A}$ to query to the signing oracle Sign and aggregated signing oracle aSign. Given public information, the identity key binding oracle IdKeyGen, the key generation oracle KeyGen and signing/aggregated oracle Sign/aSign are defined as follows:

- IdKeyGen$(\lambda, i) \rightarrow (isk_i, ipk_i)$: This algorithm takes λ and i as inputs, outputs the identity secret/public key pair (isk_i, ipk_i).
- aKeyGen$(\lambda, i, \text{msk}, \text{mpk}) \rightarrow (sk_i, pk_i)$: Given the security parameter λ, when an adversary $\mathcal{A}$ makes a query to the keyGen oracle for a signer i, this oracle returns two pairs of keys, i.e., the identity key pair (isk_i, ipk_i) and the signing key pair (sk_i, pk_i).
- aSign$(pk, m_i, \cdot) \rightarrow \sigma_i$: When an adversary $\mathcal{A}$ make a query by using a public key list pk, a message m_i to be signed, this aggregated signing oracle returns an aggregated signature σ_i.

We denote the advantage of $\mathcal{A}$ as $\text{adv}^{key-bind}_{\textbf{ASIG},\mathcal{A}}(\lambda)$, which is defined by the probability that $\mathcal{A}$ wins the unforgeability game.

Definition 3. (**Key-Binding**) An aggregatable signature scheme **ASIG** is said to provide key-binding property if for any polynomial time adversary $\mathcal{A}$, the following equation holds:

$$\mathsf{adv}^{key-bind}_{\mathbf{ASIG},\mathcal{A}}(\lambda) = \Pr\left[\mathrm{Exp}^{key-bind}_{\mathbf{ASIG},\mathcal{A}}(\lambda) = 1\right] \leq negl(\lambda)$$

9.2.2 Order-revealing Encryption

Motivation of Order-Revealing Encryption. The motivation behind using such a scheme is a two-fold. In the intra-domain case, we need to ensure that the ATLs of the routers along a path are above a certain threshold value (i.e., a minimum ATL). In cross-domain scenarios, different orchestrators need to exchange information about the ordering of ATLs without revealing their actual values.

From a cryptographic aspect, this requires a primitive that can compare the order of private information but without revealing the private information. To address this requirement, CASTOR employs Order-Revealing Encryption (ORE), which can determined the order of plaintexts by comparing their associated ciphertexts without revealing the plaintexts.

The state-of-the-art Order-Revealing Encryption schemes. In 2014, Boneh *et al.* [14] proposed the first Order-Revealing Encryption (ORE) scheme by leveraging multi-input functional encryption without obfuscation. ORE was motivated by the problem of answering queries over remote encrypted database [3, 12] that stores encrypted pairs of the form (name, salary). The data owner wishes to retrieve all records with a salary greater than a threshold t. If salaries are encrypted using ORE, the database can sort the encrypted records from lowest to highest salary without decrypting them. This sorting can be done even when records are inserted sequentially into the database (perhaps by multiple users who share the secret encryption key) and requires no interaction with the data owner(s). To issue the range query the data owner sends the encryption of t under the ORE key. In response, the database first uses binary search on the encrypted salaries to locate the smallest encrypted record R with a salary greater than t and then simply sends all records to the “right” of R back to the user. Thus, for a database of n records, the database’s work is $O(log\ n)$ and requires only one round of interaction with the client, as in the case of a cleartext database. The security guarantees of ORE ensure that the database learns nothing beyond the relative ordering of records and queries. These ORE schemes are limited that comparisons are among different messages for *a single user*. To overcome this limitation to extend the single-user setting to *multi-user setting*, delegatable ORE (DORE) was proposed [54, 65]. In DORE, every user can generate their tokens for comparisons by using Public Key Infrastructure (PKI). This property makes DORE a promising candidate for the CASTOR. However, existing DORE schemes [54, 65] do not cover secret-identity key binding. Therefore, we need to design a new DORE scheme with secret-identity key binding.

To do so, we follow the DORE scheme [65] and insert identity key into definitions of DORE to cover secret-identity key binding.

Definition of Delegatable Order-Revealing Encryption (DORE). We list all notations used in a DORE scheme in Table 9.2. A DORE scheme is denoted by $\Pi = (\mathsf{DORE.Setup}, \mathsf{DORE.KeyGen}, \mathsf{DORE.Encrypt}, \mathsf{DORE.TGen}, \mathsf{DORE.Compare}, \mathsf{DORE.Verify})$ defined over a well-ordered message space $\mathcal{M}$:

- **DORE.Setup**$(1^\lambda) \rightarrow$ pp: Given a security parameter, this algorithm returns the system public parameter pp.
- **DORE.KeyGen**$(\mathsf{pp}) \rightarrow ((\mathsf{pk}, \mathsf{sk}), (\mathsf{isk}, \mathsf{ipk}))$: On input a security parameter λ, the setup algorithm DORE.Setup outputs a secret/public key pair $(\mathsf{sk}, \mathsf{pk})$ and a identity key pair, $(\mathsf{isk}, \mathsf{ipk})$.
- **DORE.Encrypt**$(\mathsf{pp}, \mathsf{sk}, \mathsf{isk}, m) \rightarrow$ ct: On input the system public parameter pp, secret key sk, identity secret key isk and a message $m \in \mathcal{M}$, the encrypt algorithm DORE.Encrypt outputs a ciphertext ct.

Notation	Description
λ	security parameter
N	the number of users involved
pp	system public parameters
sk	secret key
pk	public key
i, j	indices for users
$\mathcal{D}$	message domain
m	plaintext to be encrypted
ct	ciphertext output by the encryption algorithm
$\mathsf{tk}_{i,j}$	authorization token generated by sk_j and pk_i
ct_1, ct_2	two different ciphertexts for comparison
b	different values of b stand for different comparison results

Table 9.2: Notation summary used in a DORE scheme

- **DORE.TGen**$(\mathsf{pp}, \mathsf{pk}_i, \mathsf{isk}_j, \mathsf{sk}_j) \rightarrow \mathsf{tk}_{i,j}$: On input the system public parameter pp, public key pk_i for a user "i", secret key sk_j for a user "j", the token generation algorithm DORE.TGen outputs an authorization token $\mathsf{tk}_{i->j}$.

- **DORE.Compare**$(\mathsf{pp}, \mathsf{ct}_i, \mathsf{ct}_j, \mathsf{tk}_{i,j}, \mathsf{tk}_{j,i}) \rightarrow b$: On input the system public parameter pp, two ciphertexts $\mathsf{ct}_i, \mathsf{ct}_j$ and two associated tokens $\mathsf{tk}_{i,j}$, $\mathsf{tk}_{j,i}$, the compare algorithm DORE.Compare outputs a bit $b \in \{-1, 0, 1\}$. The output values represent: "1" means $\mathsf{ct}_1 > \mathsf{ct}_2$; "0" means $\mathsf{ct}_1 = \mathsf{ct}_2$; "-1" means $\mathsf{ct}_1 < \mathsf{ct}_2$;

- **DORE.Verify**$(\mathsf{pp}, \mathsf{pk}_j, \mathsf{pk}_i, \mathsf{tk}_{i,j}, \mathsf{tk}_{j,i}) \rightarrow 0/1$: On input the system public parameter pp, two public keys $\mathsf{pk}_j, \mathsf{pk}_i$ associated with two authorization tokens $\mathsf{tk}_{i,j}, \mathsf{tk}_{j,i}$. If both tokens pass the verification, this algorithm returns "1"; otherwise returns "0".

Threat model. Before giving definitions of security properties, we model the power of an adversary $\mathcal{A}$ in security definition as follows:

- **Data Privacy violation**: The adversary may attempt to disclose the contents of encrypted data along with trying to obtain the ordering of the information and the index of the first differing bit between the two ciphertexts and ultimately recovering the data.

- **Token forgeability**: The adversary may attempt to forge tokens in order to access encrypted database.

Security requirements. The security requirements of an ORE scheme is *correctness*, *indistinguishability under an ordered chosen plaintext attack (IND-OCPA)* and token *unforgeability*. Note that the *key-binding* is implicitly embedded in the token unforgeability.

Correctness. Fix a security parameter λ, a DORE scheme is denoted as $\Pi = (\mathsf{DORE.Setup}, \mathsf{DORE.KeyGen},$ DORE.Encrypt, $\mathsf{DORE.TGen}, \mathsf{DORE.Compare})$ over a message space $\mathcal{M}$ is correct if for DORE.Setup $(1^\lambda) \rightarrow \mathsf{pp}$, $\mathsf{DORE.KeyGen(pp)} \rightarrow ((\mathsf{pk}_1, \mathsf{sk}_1), (\mathsf{isk}_1, \mathsf{ipk}_1))$, $\mathsf{DORE.KeyGen(pp)} \rightarrow ((\mathsf{pk}_2, \mathsf{sk}_2), (\mathsf{isk}_2, \mathsf{ipk}_2))$, and for all messages $m_1, m_2 \in \mathcal{M}$, the following two equations both holds

$$\Pr[\ \mathsf{DORE.Compare}\ \ (\mathsf{pp}, \mathsf{ct}_1, \mathsf{ct}_2, \mathsf{tk}_{1,2}, \mathsf{tk}_{2,1}) = \mathbf{1}\,(m_1 < m_2)] = 1 - \mathrm{negl}(\lambda)$$

$$\Pr\left[\mathsf{DORE.Verify}(\mathsf{pp}, \mathsf{pk}_j, \mathsf{pk}_i, \mathsf{tk}_{i,j}, \mathsf{tk}_{j,i}) \rightarrow 1\right] = 1 - \mathrm{negl}(\lambda)$$

where $\mathsf{ct}_1 \leftarrow$ DORE.Encrypt $(\mathsf{pp}, \mathsf{sk}_1, \mathsf{isk}_1, m_1)$, $\mathsf{ct}_2 \leftarrow$ DORE.Encrypt $(\mathsf{pp}, \mathsf{sk}_2, \mathsf{isk}_2, m_2)$, $\mathsf{tk}_{1,2} \leftarrow$DORE.TGen $(\mathsf{pp}, \mathsf{pk}_1, \mathsf{isk}_2, \mathsf{sk}_2)$, $\mathsf{tk}_{2,1} \leftarrow$ DORE.TGen$(\mathsf{pp}, \mathsf{pk}_2, \mathsf{isk}_1, \mathsf{sk}_1)$ and the probability is taken over the random coins.

IND-OCPA. We now give the simulation-based notion of security for a DORE scheme. The security definition is parameterized by a leakage function $\mathcal{L}$, which exactly specifies what is leaked by an ORE scheme.

Definition 4. (**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$(1^\lambda)$
  2. $((\mathsf{pk}, \mathsf{sk}), (\mathsf{isk}, \mathsf{ipk})) \leftarrow$ DORE.KeyGen $(1^\lambda)$
  3. $(m_1, \mathrm{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1 (1^\lambda)$
  4. $\mathsf{ct}_1 \leftarrow$ DORE.Encrypt $(\mathsf{sk}_1, \mathsf{isk}_1, m_1)$
  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) $\mathsf{ct}_i \leftarrow$ DORE.Encrypt$(\mathsf{sk}_i, \mathsf{isk}_i, m_i)$
  6. output $(\mathsf{ct}_1, \ldots, \mathsf{ct}_q)$ and $\mathsf{st}_{\mathcal{A}}$

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

  1. $\mathrm{st}_{\mathcal{S}} \leftarrow \mathcal{S}_0 (1^\lambda)$
  2. $(m_1, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1 (1^\lambda)$
  3. $(\mathsf{ct}_1, \mathrm{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_1 (\mathrm{st}_{\mathcal{S}}, \mathcal{L}(m_1))$
  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) $(\mathsf{ct}_i, \mathrm{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_i (\mathrm{st}_{\mathcal{S}}, \mathcal{L}(m_1, \ldots, m_i))$
  5. output $(\mathsf{ct}_1, \ldots, \mathsf{ct}_q)$ and $\mathsf{st}_{\mathcal{A}}$
```

We say that Π is a secure ORE 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 5. (**IND-OCPA Security**). Let $\mathcal{L}$ be the following leakage function:

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

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

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

- pp $\leftarrow$ DORE.Setup(1^λ)
- $(\mathsf{tk}^*_{i,j}, \mathsf{tk}^*_{j,i}) \leftarrow A^{\mathsf{DORE.KeyGen(pp)},\mathsf{DORE.TGen}(\mathsf{pk}_i,\mathsf{pk}_j,\cdot),\mathsf{Corr(ipk/pk)}}(\mathsf{pk}_i, \mathsf{pk}_j)$.
- Return 1 iff DORE.Verify$(\mathsf{pk}^*_i, \mathsf{pk}^*_j) = 1 \wedge (\mathsf{tk}^*_{i,j}, \mathsf{tk}^*_{j,i})$ and $(\mathsf{pk}^*_i, \mathsf{pk}^*_j)$ have not been queried before.

Figure 9.3: Token unforgeability game.

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 toke that passes DORE.Verify algorithm. The definition of token unforgeability is shown in Figure 9.3. The oracles DORE.KeyGen(pp), DORE.TGen$(pk, \cdot)$ and Corr(ipk/pk) involved in this game are defined as follows:

- **DORE.KeyGen**(pp) $\rightarrow$ ((pk, sk), (isk, ipk)): Given the system public parameter pp, when the adversary $\mathcal{A}$ makes queries about a user's key, this oracle returns the identity and secret key pairs ((pk, sk), (isk, ipk)).
- **DORE.TGen**$(\mathsf{pk}_i, \mathsf{pk}_j, \cdot) \rightarrow (\mathsf{tk}_{i,j}, \mathsf{tk}_{j,i})$: Given two public keys $\mathsf{pk}_i, \mathsf{pk}_j$, when the adversary makes queries about two user's authorization tokes, this oracle returns two associated authorization tokens $(\mathsf{tk}_{i,j}, \mathsf{tk}_{j,i})$.
- **Corr**(ipk/pk) $\rightarrow$ isk/sk: When the adversary makes queries using a user's identity public key ipk or public key pk, this oracle returns the corresponding identity secret key isk or 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 6. (**Token Unforgeability**) A DORE scheme is said to provide token unforgeabiliry if for any polynomial time adversary $\mathcal{A}$, the following equation holds:

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

Based on the two primitives described above, CASTOR can employ an aggregated signature (instantiated by ordered multi-signature scheme) to achieve composite attestation. This process verifies the sequence of routers in an enforced trusted routing path. In a cross-domain setting, however, a more granular security requirement often arises: ensuring that the ATL of each router in the path meets or exceeds a predefined minimum threshold ATL. Simply confirming the path order is insufficient; the integrity of the path must also be evaluated based on the security posture of each constituent router.

To satisfy this requirement while adhering to strict privacy constraints—such as preventing the disclosure of individual routers' exact ATL values, CASTOR utilizes a (Delegatable) Order-Revealing Encryption, (D)ORE, scheme. This scheme enables comparisons over encrypted values. First, the ATL of each router in the path is encrypted using (D)ORE. The system can then perform computations on these ciphertexts to determine their collective order relative to the encrypted ATL, all without ever decrypting the sensitive individual values. The final output is a privacy-preserving, cryptographic proof that yields a binary decision: whether all routers in the path satisfy the minimum ATL requirement (e.g., possess an ATL above the required minimum), thereby fulfilling both the security policy and the privacy requirements of the cross-domain operation.

Chapter 10

Detailed Threat Model and Considered Attacks

This chapter provides the **host-level threat model for the routing plane**, analysing key threats from D2.1 [22] deemed most relevant to the CASTOR environment. Each threat, identified in Table 5.2, is detailed in its own subchapter and organized into two parts: (i) a breakdown of its vectors (how), enablers (why) and impacts (what), and (ii)) a corresponding adversary model that outlines the end-to-end narrative attack path from start to finish.

10.1 BGP Prefix Hijacking (TM.3.1)

BGP Prefix Hijacking, also known as BGP hijacking, route hijacking, or IP hijacking, represents a critical and persistent threat to inter-domain routing, enabled by fundamental protocol design flaws and specific implementation vulnerabilities. More specifically, an attacker illegitimately manipulates the BGP routing protocol to reroute traffic through the attacker's preferred path. The attack materializes through several attack vectors. An **Exact Prefix Hijack (Type-0)** involves the attacker announcing an identical prefix (IP addresses) as the legitimate owner, resulting in a partial hijack for nearby routers [24, 84]. A **Sub-prefix Hijack (Type-1)** exploits BGP's ”longest prefix match” logic, where announcing a more specific prefix grants the attacker total control over that sub-prefix's traffic. Additionally, **AS-Path Manipulation Hijack** allows an attacker to falsify the AS-PATH to appear shorter or more legitimate, thus increasing its attractiveness to other routers. The feasibility of these vectors stems from **BGP's fundamental weakness** [70, 45], specifically the lack of a universally deployed, built-in mechanism to validate an AS's authority to announce a prefix. This weakness is exploited through various enabling threats, such as **Host Compromise** of a BGP-speaking router (via exploits [15], weak credentials [45] or orchestrator compromise [35]), **Insider Threats** abusing valid credentials [45] or simple **Configuration Errors** leading to ”route leaks” [7]. The resulting impacts on network operations are severe and diverse. These outcomes include **Denial of Service (Blackholing)** where traffic is dropped [24], active **Traffic Interception (MiTM)** for spying, **Traffic Impersonation (Spoofing)** to redirect users to malicious services and widespread **Network Instability** due to ”route flapping” that can overload router CPUs globally.

10.1.1 Adversary Model on BGP Prefix Hijacking

In this threat, the adversary is modelled as an AS-level actor that behaves opportunistically yet pragmatically. First, the adversary performs targeted reconnaissance (mapping victim prefixes, routing policies, monitors and likely upstreams) [19]. Then, they position themselves by acquiring or co-opting routing presence, such as by compromising a transit/peering element (via exploits [15], weak credentials [45], or by compromising a VM [35] or the orchestrator). From that foothold, the adversary announces crafted routes (exact/sub-prefix [24, 84] or AS-path manipulations) timed and scoped to maximize interception while minimizing detection. They actively leverage systemic weaknesses, e.g., RPKI/RIB validation downgrade or poisoning [53, 46], selective route leaks [7], and local preference/tie-breaking behaviour, to increase propagation and persistence. Finally, the attacker amplifies and conceals impact through short, targeted hijacks, AS-path obfuscation [24], or by exploiting limitations of monitoring/detection systems [19] to evade remediation while monetizing or exploiting intercepted traffic [24].

10.2 Active Wiretapping (TM.3.2)

Active Wiretapping is a specialized form of Man-in-the-Middle (MiTM) attack. Unlike general MiTM attacks which typically target end-user data streams (e.g., web traffic), this threat is aimed specifically at the network's **control plane**, involving the in-transit interception and potential modification of BGP protocol traffic itself [70, 45]. This threat remains relevant due to both legacy protocol characteristics and new vulnerabilities, such as the **CVE-2025-9961** [15]. Specific attack vectors to achieve this include **ARP Spoofing** [24] on a shared L2 network to position the adversary between peers, **TCP Session Hijacking** by predicting sequence numbers to inject forged packets or establishing a **MiTM via a Compromised Transit Node** to intercept traffic on TCP port 179 (the standard TCP port for BGP traffic [70]). This can even be achieved recursively through a preliminary **BGP Hijack** [24] to force peer connections to the attacker. The feasibility of these attacks stems from several enabling threats, primarily the **Lack of Encryption** in standard BGP [70] which transmits data in plaintext and **Weak Authentication** mechanisms like the legacy TCP-MD5 [45]. Furthermore, **Host Compromise** of a hypervisor, SDN microcontroller, or transit router via weak credentials allow an attacker to insert malicious VNFs or compromised software into the network [35]. Even from a physical security perspective, weaknesses in infrastructure protection may allow an attacker to deploy taps and carry out active wiretapping. The impact of a successful wiretap is severe, ranging from active **Route Tampering** (integrity loss) by modifying legitimate UPDATEs [70] and **Malicious Route Injection** [24], to **Session Denial of Service** via injected TCP Resets or BGP NOTIFICATIONs. Even a passive attack provides high-value **Reconnaissance**, allowing the attacker to steal the entire routing table and map network topology for future attacks [19].

10.2.1 Adversary Model on Active Wiretapping

An informed adversary first performs targeted reconnaissance [19] to discover vulnerable peers, exposed management interfaces, or candidate interception points where they can obtain peering with neighbouring autonomous systems (ASes) or place probes. Next, they position themselves on the control-plane path by one or a combination of means: establishing upstream peering or a malicious transit ASN (BGP hijack [24]), exploiting an L2 link (ARP spoofing), or a compromised transit host/hypervisor to intercept TCP port 179 traffic, or inducing transient routing that forces peers to re-establish sessions through attacker-controlled infrastructure [19]. Once positioned, the adversary exploits the (plaintext) weakly authenticated BGP sessions [45, 70] to eavesdrop [24], inject or modify UPDATEs (e.g., forged attributes, crafted UPDATE floods, TCP injection or resets) to effect route tampering or traffic redirection; if session protection (MD5/IPsec/RPKI) is absent, degraded, or temporarily disabled [45, 53], injection becomes trivial. After initial exploitation, they move to persistence and amplification, installing backdoors on compromised infrastructure [35]. Finally, they cover their tracks or monetize access by obfuscating AS paths [24], removing noisy indicators [19], or using hijacked traffic for reconnaissance/credential theft [24].

10.3 Attacks on BGP Routers (TM.3.3)

Attacks on BGP Routers shift the focus from manipulating protocol behaviour externally to achieving a full compromise of the routing hardware or software instance itself. Such attacks are often enabled by specific high-severity vulnerabilities, exemplified by **CVE-2023-38802** [44], which allows for OS command injection in PAN-OS. The vectors to achieve this compromise are diverse. An attacker might exploit a software flaw for **Remote Code Execution (RCE)** against a router's daemons or use **Authentication Bypass** and **Credential Theft** [45]. A privileged **Insider Attack** abusing valid credentials is also a primary vector. In virtualized environments, this threat extends to **Virtual Host Compromise** by targeting the underlying hypervisor, VM, or container [35], or it can be initiated through a **Compromised Supply Chain** attack via a malicious software update. The success of these vectors is predicated on enabling conditions such as **Unpatched Software** containing known vulnerabilities or **Weak or Default Configuration** of security settings [7]. The impact of such a compromise is catastrophic as the attacker gains full control. This allows for the injection of **Malicious BGP Attributes** to manipulate traffic [24], the injection of entirely **Malicious Routes** (hijacking) from a trusted source, or simply a **Router Denial of Service** by crashing the daemon, creating a network black hole (where traffic is dropped or lost instead of reaching its intended destination).

10.3.1 Adversary Model on Attacks on BGP Routers

An attacker who successfully compromises a router should be modelled as a powerful, trusted routing element [19]. In practice, the adversary first stages and positions themselves by acquiring co-located VMs/peering [35], exploiting reachable management endpoints, or using supply-chain footholds. Once code execution or admin control [45] is obtained, the adversary can directly manipulate BGP state (injecting or suppressing routes [24], altering Routing/Forwarding Information Base, fabricating attributes) and can craft actions that exploit systemic fallback behaviours (e.g., inducing RPKI downgrades or validation failures [53, 46]). From this foothold, they can amplify damage by triggering control-plane instability (crafted or high-volume updates [24]) that overwhelm processing and provoke wide convergence churn [34]. Finally, the adversary pursues persistence and stealth, installing backdoors, creating long-lived or transiently plausible announcements, and selectively reverting noisy operations to impede detection [19].

10.4 Compromise of Management Systems (TM.3.4)

This threat represents a high-impact compromise not of the central orchestrator, but of the distributed router management plane itself. It focuses on vulnerabilities that allow an attacker to bypass cryptographic and authentication controls on a trusted router. This is exemplified by **CVE-2024-0012** [64], an authentication bypass that lets unauthenticated attackers gain full administrator privileges via the equipment's management interface. This allows an attacker to bypass authentication on the management plane and gain administrative privileges. Specific attack vectors to achieve this compromise include exploiting such **Authentication Bypass** flaws, brute-forcing **weak credentials**, or exploiting **software vulnerabilities** in the router's exposed management services. These attacks are made possible by enabling factors such as **Unpatched Software** containing these critical bypasses, **Weak or Default Credentials** [45] and **Exposed Management Interfaces** [7] that allow the attacker to reach the vulnerable endpoint. The impact of this vulnerability can be detrimental (the vulnerability is assigned with an impact rating of 9.3/10 in the latest Common Vulnerability Scoring System 4.0 specification [64]), as the attacker gains full administrative control of an individual router. This allows for the **unauthorized spawning of malicious processes** (e.g., a rogue VNF or container [35]) on the router-host and the **direct manipulation of its routing table to hijack paths** [24]. The attacker can create **malicious routing graphs** by injecting false routes, leading to **traffic interception** or a **network DoS**, and can subsequently **erase all local logs** [19] to cover their tracks.

10.4.1 Adversary Model on Compromise of Management Systems

The adversary that is able to compromise critical aspects of the router software stack is modelled as a high-impact actor who - after targeted reconnaissance to discover exposed router management interfaces [7] - bypasses cryptographic-based authentication by exploiting a protocol flaw or a software vulnerability. Once administrative control [45] is gained, the adversary can spawn unauthorized processes (e.g., a packet sniffer) on the router-host and atomically falsify local routing policies and FIB/RIB entries [24]. This allows them to mislead peers and create malicious routing graphs, enacting targeted hijacking or denial-of-service.

10.5 RPKI Repository Attacks (TM.3.5)

Resource Public Key Infrastructure (RPKI) is a security framework designed to prevent BGP hijacking [53]. It functions as an out-of-band system where IP address owners issue cryptographically signed objects, known as Route Origin Authorizations (ROAs), to declare which AS is authorized to originate their IP prefixes. RPKI Repository Attacks, therefore, target this critical out-of-band infrastructure responsible for publishing and validating ROAs, aiming to corrupt this ground truth of BGP [71]. Implementation flaws in the RPKI ecosystem, exemplified by Improper Input Validation vulnerabilities (e.g., see CVE-2021-31375 [48]) that permit BGP origin-validation bypass, create critical entry points for attackers. Specific attack vectors include a direct **Repository Host Compromise** of the web/rsync server [45] followed by **Data Tampering** to inject malicious ROAs or delete legitimate ones [71] or a **Repository Transport (MiTM) Attack** against unencrypted rsync connections. The feasibility of these attacks increases due to **Weak Host Security** on unpatched repository servers and the use of **Insecure Transport Protocols** such as rsync instead of HTTPS/RRDP [71]. The RPKI's centralized trust model also presents a systemic risk: a compromise of a single repository (e.g., via TM.3.4) could have catastrophic consequences.

10.5.1 Adversary Model on RPKI Repository Attacks

The RPKI-repository adversary is modelled as an out-of-band attacker who can (a) compromise a repository host or its transport (e.g., rsync/RRDP MiTM [71]) to inject, delete, or alter ROAs (a form of data misconfiguration), or (b) craft "poison" RPKI objects that exploit validator implementation bugs to corrupt caches or crash validators. Such models assume the attacker can influence publication points, enabling either wrongful validation of malicious origins (making a hijack appear valid) or mass invalidation of legitimate prefixes. Practical measurements show repositories and RPs remain attackable (e.g., rsync/RRDP fallbacks, implementation flaws), so threat models treat this adversary as capable of causing both targeted RPKI-valid hijacks and wide validator DoS that force operational "fail-open/fail-closed" dilemmas [71].

10.6 Protocol DoS (TM.3.6)

Protocol DoS targets the stability and availability of the network control plane by exploiting the routing protocols themselves, rather than merely saturating data links [84]. Such attacks are often enabled by specific implementation vulnerabilities, as exemplified by **CVE-2024-39531** [49], where malformed BGP

UPDATE messages can crash the routing daemon. The specific threat vectors are varied and include **Protocol Fuzzing** (sending malformed packets to crash a parser), **Resource Exhaustion** (flooding a router with valid protocol messages from spoofed IPs to exhaust its CPU) [70], **State Exhaustion** (sending messages like IGMP Joins to exhaust router RAM) and even **L2 Topology Attacks** such as manipulating Spanning-Tree. The feasibility of these attacks stems from **Insecure-by-Default Protocols** that often lack strong, mandatory authentication, the prevalent use of **Weak Authentication** with static, shared keys, **Flat Network Architectures** lacking proper segmentation or access control lists and **Vulnerable Software Stacks** as the aforementioned vulnerability demonstrates. The impacts of such an attack are severe, ranging from a **Router Crash** to widespread **Routing Instability** and network-wide re-convergence events [84]. This instability often results in **Traffic Black-holing** as routes are withdrawn and, in some cases, can even lead to **Topology Poisoning** and MiTM attacks if the attacker successfully forms a malicious adjacency to inject fake routes [24].

10.6.1 Adversary Model on Protocol DoS

The Protocol DoS adversary is modelled as a scalable, protocol-aware attacker who first discovers vulnerable implementations and routable session endpoints [70]. Then, they leverage modest resources (e.g., multiple co-located VMs/ASNs or compromised peers) to generate either crafted malformed messages that trigger parser crashes or low-rate/targeted update floods that overwhelm route processing [70]. Such an adversary is assumed capable of controlling or spoofing multiple BGP sessions [45] and adapting payloads to evade simple rate-limits or signature-based detectors. Overall, threat models treat the attacker as able to combine implementation-specific crashes with volume-based exhaustion to create widespread control-plane outages [84].

10.7 Inter-Domain Border Security Exploitation (TM.3.7)

Inter-Domain Border Security Exploitation focuses on vulnerabilities arising at the ”seam” where two distinct administrative domains, such as partner ASes, interconnect. The specific attack vectors include **Cross-Domain Identity Spoofing** [24], **Security Policy Downgrade Attacks** to force weak security protocols [45] and targeted **Border Gateway Resource Exhaustion** to sever the link [70]. The feasibility of these attacks stems from significant enabling threats, such as the **Lack of a Common Trust Anchor** between the domains' respective trust systems [53], the use of **Insecure Inter-Domain Protocols** with weak or no encryption [45] and **Mismatched Security Policies** that default to the weakest link [19]. The impacts of such a border compromise are severe, allowing an attacker to inject **Malicious Traffic or Routes** that are accepted as legitimate, cause a total **Inter-Domain Denial of Service** by black-holing cross-domain traffic or facilitate **Sensitive Information Leakage** by passively sniffing the unencrypted link.

10.7.1 Adversary Model on Inter-Domain Border Security Exploitation

The inter-domain border adversary is modelled as an opportunistic actor who — after conducting reconnaissance to identify exposed management APIs, misaligned policies, or weakly protected peering sessions — exploits cross-domain weaknesses (e.g., BGP/DNS hijacking [24], API/web vulnerabilities, or link-layer MiTM [45]). By doing so, the attacker can impersonate or downgrade a peer, inject routes or traffic that will be accepted [24], or disrupt connectivity to induce black-holing [7] or data leakage [45].Prior measurement and survey work shows such seams are routinely the weakest link (mismatched policy, insecure transports [45], and exposed portals), so threat models treat this adversary as able to weaponize a single border compromise into accepted, widely-propagating routing or traffic failures.

10.8 Flex-Algo Manipulation (TM.3.8)

Flexible Algorithm (Flex-Algo) is a mechanism within IGPs (e.g., OSPF or IS-IS) that allows the computation of multiple, independent topologies in the same IGP domain [68]. Each Flex-Algo is identified by an algorithm identifier (typically 128–255) and can have its own calculation type, metric type, and constraints. This allows operators to define multiple parallel routing topologies within the domain for traffic engineering or policy purposes. In this context, Flex-Algo Manipulation, is a sophisticated attack that targets this traffic engineering process. A primary vector for this attack is the compromise of a network node. This can be achieved by exploiting vulnerabilities such as CVE-2023-28771 [85] — a remote code-execution flaw in Zyxel devices — to gain a foothold for manipulating, among other things, Flex-Algo configuration. As noted in the relevant IETF specification [68], Flex-Algo introduces two concrete attack vectors: (i) malicious injection of Flex-Algo Definitions (FADs) by a compromised router or controller, and (ii) Flex-Algo topology tampering, in which a rogue node falsely advertises support for a Flex-Algorithm that it does not actually implement. These attacks are made possible by a **Lack of Cryptographic Integrity** in the underlying IGP [53], the primary enabler of **Router/Controller Host Compromise** [24] or simple **Configuration Errors** [32]. The impacts are severe, leading to **Traffic Hijacking** by forcing a slice onto a malicious path [84], **Denial of Service** via black-holing or routing loops [70], **Security Policy Bypass** of security nodes [35] and a fundamental **Network Slicing Failure** that breaks isolation between slices [35].

10.8.1 Adversary Model on Flex-Algo Manipulation

This adversary is modelled as a node-capable and policy-aware attacker who, after gaining a foothold on an IGP participant, can inject or tamper with Flex-Algo Definitions (FADs) or IGP attributes [84] to steer slices onto attacker-chosen paths, trigger algorithm fallbacks, or introduce loops/black-holes [70]. Because Flex-Algo lets networks compute policy paths within the IGP, a compromised participant that advertises or alters FADs can quietly subvert slice isolation [35] or force traffic through chosen waypoints (bypassing policy) [35].

10.9 Interception of PCE requests or responses (TM.3.9)

The Path Computation Element (PCE) is a centralized network component that acts as a ”brain” for traffic engineering, responsible for computing complex, constrained-based paths for routers [79]. These routers, known as Path Computation Clients (PCCs), use the PCE Protocol (PCEP) to send path requests to the PCE and receive computed paths in response. Interception of PCE requests or responses targets this critical and centralized signalling. Vulnerabilities in specific implementations, such as **CVE-2024-20323** [25] which can lead to a denial of service in the PCEP component of Cisco IOS XR, highlight the fragility of these systems. Specific attack vectors include passive **PCEP Session Eavesdropping** for reconnaissance [24], active **PCE Response Tampering** to inject malicious nodes into a computed path [45], **PCC Request Tampering** to modify constraints and force a malicious path calculation, or full **PCE Impersonation** via methods like BGP hijacking [24]. The feasibility of these attacks stems from the **Insecure-by-Default** nature of plain PCEP sessions — which are often unencrypted and unauthenticated [79] — as well as from **Weak or Absent Authentication** in deployed environments [53]. The impacts are severe, leading to **Traffic Hijacking**, **Security Policy Bypass** of firewalls, **Denial of Service** via black-holes or routing loops and invaluable **Topology & Policy Reconnaissance** from passive sniffing.

10.9.1 Adversary Model on Interception of PCE requests or responses

The PCE-interception adversary is modelled as an on-path or impostor control-plane actor who can passively eavesdrop on PCEP sessions [24], tamper with requests or replies (modifying constraints or inserting malicious nodes [45]), or impersonate a PCE/PCC by hijacking/redirection [24] or exploiting PCEP/PCE implementation bugs. This enables targeted path hijacks [84], policy bypass [35], or large-scale disruption [70]. PCEP remains exposed to such threats unless protected (e.g., PCEPS/TLS or TCP-AO), so threat models typically assume that an attacker capable of observing or interposing on PCEP can carry out precise, low-cost path manipulation or reconnaissance.

10.10 Impersonation of PCE or PCC (TM.3.10)

While TM.3.9 involves intercepting PCEP traffic, impersonation of PCE or PCC involves an attacker actively spoofing one of the endpoints. This threat is made practical by implementation flaws, such as weaknesses that allow authentication safeguards in the PCEP daemon to be bypassed, thereby enabling unauthenticated actors to interact with the PCE by impersonating other PCC elements in the topology. Specific attack vectors include **PCE Impersonation via BGP/DNS Hijack** [24] to trick all PCCs into connecting to the attacker, **PCC Impersonation via IP Spoofing** to send malicious requests to the real PCE, or **Impersonation via Compromise** [84], which acts as a ”trusted impostor”. The primary enabler for these attacks is the **Lack of Strong, Mutual Authentication** [53] combined with the **Insecure-by-Default** nature of plain PCEP [79], or by **Insecure Addressing/Routing** [24]. The impacts fall into two categories: a successful **PCE Impersonation** grants the attacker **Total Traffic Hijacking** capabilities [84], while a **PCC Impersonation** can be used for **PCE Resource Exhaustion DoS**, **PCE State/Topology Poisoning** [45] or stealthy **Topology Reconnaissance** [24].

10.10.1 Adversary Model on Impersonation of PCE or PCC

The PCE/PCC-impersonation adversary is modelled as a protocol-level actor: a control-plane actor who, after reconnaissance, can present forged identities or hijacked addresses to PCCs or PCEs (via BGP/DNS redirection [24], IP spoofing, or by exploiting implementation bugs. They establish a PCEP session or inject crafted PCEP messages, and thereby issue erroneous path computations [84], exhaust PCE resources, or poison topology/state [45]. When it comes to the PCE adversary model, we need to consider where the PCE is actually deployed. On one hand, the PCE may be instantiated as a dedicated network element, running a lighter router software stack image (e.g., Cisco XRd control plane image) and listening to all network-related information flooding the network. In this context, PCE is susceptible to the aforementioned threat models that could hinder the traffic engineering process. On the other hand, it is also possible that the Network Controller exposes the PCE capabilities, taking care of all the PCEP interactions with the corresponding PCCs of the topology. As mentioned in D2.1 [22], the entire Orchestration Layer is considered to be trusted in the context of an administrative domain. Therefore, any PCE-related threats are considered out of scope from this instantiation. Details on all possible approaches revolving around the PCE functionalities in CASTOR are presented in D5.1 [21].

10.11 Falsification of TE information, policy information or capabilities (TM.3.11)

Falsification of TE information, policy information, or capabilities is a threat where a network element provides false data to its peers or to a central controller to manipulate network decisions. Once compromised, the rogue node can execute several attack vectors, including **TE Information Falsification** by lying about link attributes like latency [84], **Policy Information Falsification** by modifying BGP Community strings [70], **Node Capability Falsification** by falsely claiming protocol support [79], or **Topology Falsification** by advertising ”ghost links” [24]. The success of these vectors relies on a primary enabler of **Host Compromise** [84], a **Lack of Cryptographic Integrity** in the underlying protocols [53], or pre-existing **Configuration Errors** [32]. The impact of this falsified data is severe, leading to **Traffic Hijacking** [84], **Denial of Service** via black-holes or routing loops [70], **Security Policy Bypass** [35] and **Sub-optimal Routing** that causes subtle performance degradation [24].

10.11.1 Adversary Model on Falsification of TE information, policy information or capabilities

The adversary in this context is modelled as a node-capable actor who, after gaining a foothold on an IGP/BGP participant or controller, advertises crafted TE/policy/capability data (e.g., fake link metrics, forged BGP communities, ”ghost links”) [70]. This is done to affect path computation and policy enforcement. The adversary can then carefully time and scope these falsified announcements to steer traffic [84], create stealthy black holes or routing loops [70], or subvert monitoring and remediation mechanisms [24]. Overall, threat models treat this adversary as able to both quietly influence routing decisions at scale and to evade detection by making announcements appear operationally plausible such as a simple misconfiguration.

10.12 Denial-of-service attacks on PCE or PCE communication mechanisms (TM.3.12)

Denial-of-service attacks on PCE or PCE communication mechanisms aim to exhaust the resources of the centralized PCE or sever its communication links, crippling the network’s TE ”brain” [79]. The attack vectors range from simple **Volumetric Floods** [70] to more sophisticated **PCEP Protocol Floods** using computationally expensive requests [24], **PCEP State Exhaustion** to consume all RAM, **Malformed Packet Attacks** to crash the parser, or **Communication Link DoS** via TCP Resets [45]. These attacks are possible due to PCEP’s **Insecure-by-Default** nature [79], **Weak or Absent Authentication** [53], **Vulnerable Software Implementations** [15] and a **Lack of Control Plane Policing** [53]. The impacts are systemic, leading to **network-wide re-routing** as routers fall back to default paths, an **inability to provision new services**, **path flapping** and signalling storms, and ultimately a complete **loss of central control**, leaving the network effectively “headless”.

10.12.1 Adversary Model on Denial-of-service attacks on PCE or PCE communication mechanisms

The PCE-DoS adversary is modelled as a scalable, protocol-aware resource-exhauster who probes PCEP/PCE endpoints for vulnerable implementations or reachable session endpoints. They then flood them with high volume and/or computationally expensive PCEP requests [24], forge/spoof PCEP sessions [53], or send crafted/malformed PCEP packets [15] that trigger parser crashes [24]. They combine modest distributed resources with the exploitation of implementation bugs to render the PCE unavailable or too slow to serve path requests. In this threat model, the PCE is considered a critical target whose compromise can have severe consequences.

10.13 Router-reflection attack (TM.3.13)

Router-reflection attack exploits the absence of end-to-end replay suppression in many networking services to degrade the connectivity of a remote Internet region just by replaying legitimate packets [52]. This threat is made practical by the widespread absence of **End-to-End Replay Detection**. The attack applies to widely deployed services that lack replay detection, including DNS, NTP, SSDP, SIP, and other UDP-based or stateless control-plane protocols. To launch such an attack the adversary performed the following four steps: (a) chooses the sets of target area hosts *T* and a set of vantage point hosts *V*, (b) analyses the network to identify routing bottlenecks affecting the target area *T*, (c) selects suitable routers for compromise and (d) leverages a compromised router to replay packets belonging to specific traffic flows. The resulting impacts include **Service Disruption** and **Routing Bottlenecks** of the target region.

10.13.1 Adversary Model on Router-reflection attack

In this router-reflection attack, the adversary is modelled as an actor who first identifies a set of hosts representing the target area whose connectivity is to be degraded, along with a set of vantage-point hosts from which trace routes toward the target area are conducted to construct a detailed link map. The attacker then filters out unstable or transient routes from these maps to isolate persistent paths and thereby identify and overload the network's routing bottlenecks (i.e., critical links that serve the highest amount of traffic flows).

Chapter 11

Evidence Collection for Runtime Threat Detection

We have defined all components related to the TCB and the CASTOR Trace Hub, including the associated Trace Units. Tracing enables introspection, from which we can extract observations. The central question that follows is how these observations can be used to identify abnormal behaviour, either through attestation mechanisms or CASTOR's FSM. Trust, as already mentioned in Chapter 4, is a dynamic property that requires continuous introspection and observability at runtime. Introspection and observation represent the two complementary facets of router trustworthiness, or the two sides of the ”trusted-state coin”. On one side lies the type of evidence that can be collected through introspection; on the other side lies the interpretation of this evidence to determine whether the observed behaviour is benign or abnormal.

In CASTOR, all observations required to identify the considered attacks are derived through introspection over an initial design space of evidence, including, but not limited to, secure and measured boot states, integrity measurements, and runtime configuration of routing components (e.g., XRD container configuration). In particular, the systematic introspection of these evidence items allows the extraction of state transitions and behavioural signals (observations) that can unlock the detections of FSM. Recall that introspection refers to the extraction of raw runtime traces, enabling low-level examination of system execution, while observability denotes the interpretation of these traces into human-readable and actionable information that allows the system to infer its internal state.

As a first step in this process, we provide a comprehensive list of observations. The common denominator of these observations, as explained in Chapter 4, is that introspection enables tracing of all elements related to memory structures, system calls, and execution-phase characteristics of the infrastructure hosting the vRouters or the physical router itself. This initial observation list represents the set of evidence that CASTOR examines to determine whether it can be reliably extracted and interpreted. If so, CASTOR is positioned to detect the potential threats identified in our threat model.

This chapter establishes the critical link between abstract threats (TM.X.Y) and the concrete, measurable evidence that a deployed security solution must collect and analyse, as required by modern NFV and softwarized network security architectures [5, 4, 62]. Recall that CASTOR in order to encompass adherence to intended protocols and verifiable proofs of routing paths considers two streams of evidence the systematic and the network evidence. Thus, this analysis is segmented into two primary categories: **Systemic Evidence** (general host and environment state) and **Networking Evidence** (protocol-specific control plane and data plane activity), reflecting also the best practices identified in SDN/NFV security literature [32, 35].

11.1 Systematic Evidence: Detecting Host-Level Compromise

Systematic evidence refers to measurable state changes, deviations and anomalies within the underlying computing host (the Hypervisor, Host OS, VM, or Container) that indicate a successful compromise or the presence of malicious activity. This evidence is critical for detecting enablers like Host Compromise (HT3, VT1, VT7), Vulnerable Software (CT4) and Privilege Escalation (AUT1), which are consistently identified as root causes of NFV security failures [5, 62].

In the context of virtualized network functions (VNFs) and trusted computing, systematic evidence is derived from the runtime integrity of the host environment. This involves verifying that the software stack, from the kernel up to the router application, has not been tampered with and is behaving according to its intended configuration, a principle emphasized in cross-layer NFV verification models [62]. Detecting these threats requires analyzing low-level system artifacts to verify the **runtime integrity** of the VNF's execution environment. Three distinct streams of modern systematic evidence are identified, drawing on recent academic developments:

1. **Kernel-Level Process and Resource Monitoring (HT3, CT3):**
   - **Mechanism:** Rather than relying on easily tampered user-space logs, modern detection systems leverage **eBPF (extended Berkeley Packet Filter)**. eBPF allows the safe execution of custom monitoring programs directly within the Linux kernel, minimizing performance overhead while providing deep visibility into system calls, process execution and resource consumption [5, 62].
   - **Evidence:** An attacker attempting a Host DoS (HT3) on the VNF will cause significant deviation in the VNF's expected CPU/Memory/I/O Footprint compared to a known baseline. An attacker attempting to conceal activity or bypass logging (CT3) will trigger Unauthorized Syscalls or File Access outside the router's expected behaviour [4].

2. **Container Isolation and Namespace Monitoring (VT7):**
   - **Mechanism:** Container security relies on kernel-level isolation mechanisms, primarily **Linux Namespaces** and **Control Groups (cgroups)**. A Container Breakout (VT7) or a Privilege Escalation (AUT1) attack fundamentally aims to escape this isolation boundaries, a known weakness in softwarized infrastructures [32, 35].
   - **Evidence:** A successful escape attempt (e.g., runc vulnerability exploit) is evidenced by Unexpected Namespace Manipulation, such as a container process attempting to join the host's network namespace or accessing unauthorized files outside its virtualized filesystem root. This results in the detection of Unauthorized Syscalls that directly target host resources [4].

3. **Microarchitectural Behavior and Side-Channel Attacks (VT8, VT9):**
   - **Mechanism:** Attacks like Spectre and Meltdown (VT8) or Cache-Based Side-Channel Attacks (VT9) exploit shared microarchitectural resources, causing subtle, detectable anomalies. Detection involves using **Hardware Performance Counters (HPC)** to profile the execution or other vulnerability intelligence sources such as the NVD [15].
   - **Evidence:** Side-channel attacks manifest as anomalous resource utilization, such as sudden, statistically significant changes in CPU Miss Rates or Branch Prediction Behavior within a VNF's execution time, which constitutes Deviating CPU Footprint evidence. These deviations are distinct from normal VNF traffic load variations [62].

11.2 Networking Evidence: Detecting Control Plane Manipulation

Networking evidence focuses on the observable behavior of the routing protocols (BGP, OSPF, PCEP) and the traffic flow itself, providing direct indicators of a compromised Network and Protocol Layer. Unlike systematic evidence, which detects how the host was breached, networking evidence detects what the compromised VNF is doing to the network, particularly the falsification of crucial routing and policy data [75]. Detection requires analyzing anomalies across both the control plane and the data plane.

11.2.0.1 Control Plane Evidence: Route and Signaling Anomalies

The control plane is the primary vector for threats that involve route injection or modification. Evidence is generated by monitoring the content, volume and cryptographic validity of routing protocol messages. Recent academic consensus is shifting away from simple anomaly detection toward multi-dimensional, behavior-based models to reduce false positives [59].

- **Anomalous Route Announcements (NT1, NT3):** This is the most direct evidence of a routing attack, encompassing content and volume anomalies.
  - ✓ **Content Mismatch:** Detection relies on comparing received BGP UPDATEs against external trust anchors (like RPKI) [53, 46, 71] and historical behavior. Evidence includes prefixes announced by an unauthorized Autonomous System (Origin AS Mismatch) or an unusually short AS-Path attribute, which often indicates an attack attempting to gain preferential routing, or a forged AS-Path designed to evade RPKI filtering (e.g., Forged-Origin Hijacks) [75].
  - ✓ **Volumetric Anomaly:** An unexpected, high volume of BGP UPDATEs or NOTIFICATIONs is strong evidence of a Protocol DoS (TM.3.7) designed to exhaust a router's CPU (HT3) or cause network-wide instability [59] (route flapping, NT3).
- **Session State and Signaling Tampering (NT4, CT1):**
  - ✓ **Unprotected Communication:** The observation of plaintext BGP or PCEP sessions when secure extensions (like TCP-AO or TLS) are expected is evidence of a Protocol Downgrade (TM.3.10) or a misconfiguration (CT1) that enables Active Wiretapping (TM.3.2) [45, 76].
  - ✓ **PCEP Impersonation:** Maliciously crafted Path Computation Requests or tampered PCE Responses that attempt to insert malicious nodes or violate policy constraints constitute evidence of Interception of PCE requests (TM.3.12) or Impersonation of PCE or PCC (TM.3.13) [79]

11.2.0.2 Data Plane Evidence: The Consequence of Compromise

Data plane evidence is the ultimate measurable consequence of a successful control plane attack, indicating that packets are no longer following the correct or intended path.

- **Anomalous Forwarding (NT2):** Traffic is unexpectedly redirected, the classic signature of a successful Traffic Interception (MiTM, TM.3.1, TM.3.14). This is detected by analyzing flow records (NetFlow, sFlow) or using active probing techniques to observe packets taking Unexpected Hops or violating established Forwarding Policy [18].
- **Blackholing (NT3):** Traffic destined for a legitimate prefix is dropped, which is the direct result of a route withdrawal or a malicious router installing Malicious ACLs (Access Control Lists) in its forwarding table [7, 19].

11.2.0.3 Focus Area: BGP Prefix Hijacking and Flex-Algo Manipulation

The decision to focus on BGP Prefix Hijacking (TM.3.1) and Flex-Algo Manipulation (TM.3.11) is rooted in the core objectives and architecture of the CASTOR project. As CASTOR aims to establish secure routing paths, these two attacks represent the most direct and impactful methods to subvert its security goals. BGP Hijacking is the fundamental vulnerability of the global Internet [84, 24, 19], directly threatening the integrity of inter-domain paths which CASTOR must secure. Flex-Algo Manipulation, conversely, represents the state-of-the-art in policy-based, intra-domain routing (Segment Routing), making it a key component of modern 5G/6G environments [4, 35]. A successful attack here allows an adversary to bypass all security middleboxes and subvert networking slicing isolation (VT7), compelling CASTOR's detection system to simultaneously monitor both legacy global threats and complex cloud-native policy threats. The combination of evidence streams required to detect these two threats (passive BGP observation for the former and active policy/metric validation for the latter) provides the necessary comprehensive scope for the CASTOR solution.

1. **BGP Prefix Hijacking (TM.3.1):** This is typically a Control Plane attack, but its detection relies on multiple evidence streams:
   - **Network Evidence:** The primary indicator is the abrupt appearance of an Invalid Prefix/Origin AS Mismatch announcement (NT1) in the BGP update stream, often accompanied by Anomalous Forwarding (NT2)as victim traffic shifts to the attacker's AS [18].
   - **Systematic Evidence:** If the hijacking originates from a compromised CASTOR VNF, systematic evidence provides the root cause: an Integrity Check Failure (CT4 or HT2) on the BGP daemon host, or Unauthorized Processes (AUT1) that injected the malicious routes [62].
2. **Flex-Algo Manipulation (TM.3.11):** This advanced threat targets policy-based routing (Segment Routing) and specifically requires unique Control Plane evidence:
   - **Network Evidence:** The key evidence is the injection of a Malicious Flex-Algo Definition (FAD) (NT1) or the falsification of link-state metrics (e.g., advertising a high-latency link as low-latency). Crucially, the subsequent data plane analysis must show a Policy Violation: traffic belonging to a high-security slice is observed being Steered onto an Unauthorized Path (NT2) that bypasses security nodes (APT2), even though the overall path appears valid in the manipulated policy [4].
   - **Systematic Evidence:** Since Flex-Algo is often managed by a central controller or a specific VNF, the attack is likely preceded by a successful Controller Compromise (VT6) or an RCE (CT4) on the advertising router VNF, detectable via Deviating CPU Footprint (HT3) or Unauthorized Syscalls [62].

11.3 Detailed Evidence Listing

The evidence streams are structured to align with CASTOR's architectural assumptions. Systematic evidence, derived from kernel integrity checks (eBPF), process monitoring and hardware counters, forms the crucial first line of defense. CASTOR operates under the principle that if the host's systematic state is successfully attested and proven trustworthy, the initial attack vectors (RCEs, privilege escalation) have been mitigated. However, due to the inherent complexity and shared nature of modern virtualization platforms (Containers/VMs), the possibility of undetected zero-day exploits remains. Consequently, Networking evidence is prioritized as the primary detection mechanism. This approach focuses the security analysis on the external observable effects of a compromise (e.g., malicious route injection, traffic misdirection) that threaten the integrity of the routing service itself, regardless of whether the underlying host was successfully attested.

Table 11.1 consolidates the entire threat analysis by defining the specific data points essential for CASTOR's dual security objectives. The table maps the most critical networking threats to the raw protocol fields that must be continuously traced. Crucially, the Condition (The ANOMALY) column establishes the criteria for both detection systems: anomalies marked by comparison against an Attested Policy (ensuring configuration integrity) or comparison against an FSM-identified expected norm (ensuring behavioral integrity). This structure confirms that CASTOR can detect attacks ranging from passive hijacking attempts that pollute the RIB to active DoS attacks that attempt to overload the Control Plane. This listing forms the basis for CASTOR's detection and attestation mechanisms concerning route integrity and policy enforcement, especially against BGP Prefix attacks and Flex-Algo manipulation (yellow and green highlighted rows).

Table 11.1: Networking Evidence Listing on the Routing Plane

TYPES OF TRACED/ MONITORED ARTIFACT	**DATA POINT TO TRACE**	**CONDITION (The anomaly)**	**HOST TRACING MECHANISM**	**MAPPED THREATS (Ref)**
BGP Update Content (AS-Path Falsification)	Raw BGP UPDATE Message Fields: AS-PATH Attribute (containing the sequence of ASNs) and NLRI (Prefix + length), captured via eBPF from BGP process memory or intercepted TCP/179 stream	The FSM detects an illegal state transition in the AS-Path control flow. Specifically, an unexpected truncation or modification of the path length that bypasses attested logic, forcing the RIB to prefer a malicious route	eBPF Socket Filter *	NT1 (Spoofing)
BGP Update Content (Origin AS Mismatch) & RPKI Validation Chain	Decoded BGP UPDATE Message Fields: Origin Attribute (the originating ASN) and NLRI (prefix), correlated with ROA entries (ASN/Prefix/Max-Len) and RTR Protocol syscalls (cache-server queries/responses). This data is used for domain-to-domain validation.	The FSM detects a breakdown in the validation pipeline: either the local ROA state is tampered with (invalidating a legitimate prefix), or the RTR protocol fails to enforce the "Invalid" state, allowing an unverified origin to be committed to the RIB	RTR/ROA Syscall Tracing & Log Integrity Monitoring **	NT1 (Spoofing), CT1 (Misconfig)
BGP Update Content (Prefix Specificity)	Decoded BGP UPDATE Message Fields: NLRI (Prefix/Length field) and associated BGP Extended Community (used for Segment ID (SID) advertisement within the domain)	The VNF is advertising a more specific prefix (/24) and this prefix violates the attested prefix policy, overriding the legitimate route due to LPM	eBPF Prefix Filter *	NT1 (Spoofing)
PCEP Signaling/Messaging (Explicit Path Injection)	PCEP Explicit Route Object (ERO) (Sequence of SIDs/Hops) and Netlink syscalls (e.g., rt_msg_newroute). The ERO dictates the path and the syscall attempts to write it into the kernel's FIB	A compromised PCC (Path Computational Client) sends a PCReq containing an ERO that explicitly forces the path to use a quarantined Segment ID (SID) (e.g., SID: 90001) which violates the Attested Security Policy Database (SPD)	Netlink Syscall Tracing	NT1 (Spoofing), TM.3.12
PCEP Signaling/Messaging (Impersonation/Session Tampering)	TCP/IP Header Source IP Address (Network Layer), PCEP OPEN message source ID (PCEP Layer) and PCEP OPEN Authentication status	The Source IP does not match the Attested PCC Identity List or the session is initiated without Attested TCP-AO/TLS security, allowing unauthenticated injection	Kernel Socket Monitoring ***	TM.3.13
PCEP Signaling/Messaging (Attribute Override)	PCReq Metric Object field (containing the Maximum Latency or Maximum Hops constraint numerical value), captured from the PCEP input buffer and PCE process memory	The FSM identifies two failure states: (1) An attempt to originate a compromised/malformed PCReq at the source, or (2) A failure to converge on a path because the metric attributes were stripped or tampered with by a malicious process at the destination	eBPF Resource Hook & Process Trace ****	NT3 (DoS)
IGP Link-State Attributes (Metric Falsification)	OSPF/ISIS TE-LSA/LSP fields: Link Metric (delay or link-cost) numerical value, captured from the IGP daemon's Link-State Database (LSDB)	The Advertised Latency (0 ms) is significantly lower than the Attested Physical Baseline. This intentional deception forces Flex-Algo path miscalculation	eBPF LSDB Hooking ****	NT1 (Spoofing), TM.3.11
IGP Link-State Attributes (Policy/Tag Tampering)	OSPF/ISIS Extended Community Tags: Admin Group/Color fields (Affinity Bits), captured from the IGP daemon's LSDB and used for Flex-Algo calculation	The FSM detects a faulty enforcement of security coloring. A link state transition occurs where the required security affinity bits (e.g., Firewall tag) are omitted or altered, causing Flex-Algo to incorrectly calculate a path that bypasses mandatory security nodes	eBPF LSDB Hook / FSM Policy Trace	CT1 (Misconfig), TM.3.11

Table 11.1 – *Continued from previous page*

TYPES OF TRACED/ MONITORED ARTIFACT	DATA POINT TO TRACE	CONDITION (The anomaly)	HOST TRACING MECHANISM	MAPPED THREATS (Ref)
Network Flow Records (Next Hop Divergence)	NetFlow/sFlow fields: Destination Prefix, Next Hop IP Address and Output Interface Index, traced from the router's data plane and correlated against the local FIB	The Observed Next Hop (192.168.1.5) in the flow record does not match the Attested FIB (Forwarding Information Base) Entry (192.168.1.10) for that destination prefix. This proves forwarding state pollution	eBPF FIB Lookup Audit	NT2 (MiTM), NT1 (Spoofing)
Network Flow Records (Egress Interface Violation)	NetFlow/sFlow field: Output Interface ID (IfIndex) and associated Interface Description from the router's configuration, traced from the router's Data Plane egress	The flow is observed exiting an interface (e.g., eth0/admin) that violates the Attested Interface Policy (management-only port), indicating a Dataplane bypass	Kernel Netfilter Audit *	NT2 (MiTM)
BGP/PCEP Message Volume/Rate	Message Count per Second (msg/s) and Total TCP/Session Count, correlated with Active Service Metadata (Type/Quantity)	The Global FSM detects a message rate that is disproportionate to the number of active service instances. While individual rates may seem normal, the aggregate network-level volume reveals an anomaly relative to the service-type density, signaling a distributed or behavioral DoS	eBPF CoPP Counters & Global FSM Audit	NT3 (DoS)
Control-Plane Session State (Encryption, Authentication status)	TCP Header Options (Presence of TCP-AO or MD5 signatures), PCEP Open message Authentication flag status, TLS ClientHello/ServerHello records	Session established using None/MD5 when Attested Policy requires TCP-AO or TLS. This downgrade exposes the Control Plane	Kernel Packet Sniffing	CT1 (Misconfig), AUT2 (Weak Auth)
Session Token Usage (SSH, API Access)	Source IP, Token/Session ID, Round-Trip-Time (RTT) metrics, and Command/API Call history (e.g., RIB/Topology queries)	The FSM detects unauthorized or anomalous session activity. This includes token reuse from suspicious origins or ”impossible” locations, and attempts by unauthorized control services to exfiltrate routin topologies or manipulate the local RIB	API Gateway & SSH FSM Tracing	NT4 (Session Hijacking), NT1 (Spoofing)
Remove transient links from the link map	Remove transient links form the Link-State Database (LSDB) to overload links identified as prone to bottleneck	The FSM detects unauthorized removal of transient links activity.	eBPF LSDB Hook	NT3, TM.3.13

Note: BGP artifacts and validation logic are defined in RFCs (4271, 6811) [70, 57], and the RPKI framework in RFCs (6480, 6487) [53, 46]. PCEP signaling semantics and ERO manipulation are defined in RFC (5440) [79]. NFV, controller compromise, and cross-layer evidence correlation follow ETSI NFV-SEC and recent NFV security analyses [35, 4, 62]. Data-plane confirmation of routing attacks is supported by recent work on flow-based hijack detection [18].

**These mechanisms act as inline kernel gates that intercept traffic at the lowest level of the networking stack. They perform a real-time comparison between the observed packet data and the Attested Identity/Policy List to prevent unauthorized flows or prefix advertisements.*
***This leverages Measured Boot to hash every software component into the TPM hardware; because these values are cryptographically chained, the Measurement Log becomes immutable, ensuring the Verifier can detect any attempt to alter the records or the system state.*
****Metadata is stored in kernel-space socket structures (sock_common); eBPF is used to trace the PID and execution path of any actor attempting to access this memory, ensuring only the attested routing daemon can read the session state.*
*****The Resource Hook provides evidence of protocol depth to detect malformed or complex DoS payloads, while LSDB Hooking provides Augmented Remote (AR) Attestation evidence by verifying that the internal topology database matches the attested network state.*

The highest-priority networking artifacts were selected for their direct correlation to the integrity of the computed path (BGP/Flex-Algo) and the resulting data flow [4, 35].

11.3.0.0.1 BGP Update Content (Prefix, AS-Path, Origin) The BGP update stream is the foundational evidence source for inter-domain routing security [70]. Analysis focuses on comparing the received route attributes against cryptographic and historical records. A primary concern is the detection of Origin AS Mismatch and AS-Path manipulation (NT1) [84, 75]. Modern detection techniques, often leveraging machine learning and anomaly detection models, analyze temporal and topological deviations in BGP messages to detect forged routes (i.e., BGP Prefix Hijacking) [59]. Recent work demonstrates that tracking the specific changes in prefix announcements allows for timely isolation of malicious routes before they propagate globally [18].

11.3.0.0.2 PCEP Signaling/Messaging The Path Computation Element Protocol (PCEP) domain is strictly centralized, used for complex, cross-domain, or explicit path calculation, often interacting with the IGP's data [79]. The integrity of PCEP signaling must be ensured, as manipulation here allows an attacker to hijack traffic by tampering with the centralized control plane without altering the underlying IGP state [4]. This targets threats like Impersonation (TM.3.13) or Explicit Path Injection (TM.3.12). A compromised PCC (router) or an external adversary can tamper with PCReq (Path Computation Request) or PCRep (Path Computation Reply) messages to inject malicious Explicit Route Objects (EROs) or force policy violations [79, 62].

11.3.0.0.3 IGP Link-State Attributes (Latency, Cost, Policy Tags) The Interior Gateway Protocol (IGP) domain is entirely distributed, operating within a single Autonomous System. Security evidence here focuses on the trustworthiness of the internal topology and metrics [35, 32]. IGP mechanisms exchange Link-State Attributes (Latency, Cost, Policy Tags). This evidence targets the Falsification of Link-State Attributes (NT1,CT1) [5, 62]. An attacker compromised VNF may advertise a false low-latency metric for a congested link or maliciously inject a Flex-Algo Definition (TM.3.11) that violates policy constraints (e.g., mis-tagging a security-critical link) [4]. Detection requires continuous validation of these metrics against a trusted oracle or a physical measurement layer, exposing the underlying manipulation necessary to subvert slice isolation [5, 62].

11.3.0.0.4 Network Flow Records (NetFlow, sFlow-source, destination, next hop) While control plane evidence reveals attempts at manipulation, flow records provide irrefutable evidence of the consequence in the data plane [18]. Flow analysis monitors the actual path taken by user traffic, recording the sequence of hops (next-hop) and overall traffic volume. The detection of Network Flow Log Redirection (NT2) or blackholing is critical as it confirms that a control plane attack (like BGP Hijacking) has successfully translated into Traffic Interception [19, 7]. Advanced techniques compare observed flow paths against the theoretically correct path derived from the FIB (Forwarding Information Base), effectively proving a successful MiTM attack, even if the control plane announcements were subtle [18].

11.3.0.0.5 Remove transient links from the link map A compromised router may perform a reply attack in order to degrade or block the legitimate traffic flow to a remote target area (TM.3.13). Detection requires continuous observation of the link map to identify unauthorised removal of transient links.

Chapter 12

Summary and Conclusions

This deliverable has established the conceptual architecture of CASTOR Trusted Computing Base for enabling runtime trust assessment in the routing plane. D3.1 has shown how trusted computing principles can be systematically applied to routing infrastructures by defining a device-side Trusted Computing Base that is anchored in hardware Roots of Trust and integrated directly into routing elements.

The document defined the scope and structure of the CASTOR TCB, identified the key trust sources and introspection mechanisms required for runtime monitoring, and analysed how these components can securely generate and protect verifiable evidence about the configuration and behaviour of routing elements. By extending trusted computing concepts to the network domain through CASTOR-specific abstractions and interfaces, the deliverable enables routing elements to actively participate in trust assessment rather than being treated as implicitly trusted components.

In addition, D3.1 explored the threat landscape affecting the routing plane and mapped adversarial capabilities to the evidence required for detection and assessment. The presented architectural model supports both local and composite attestation, enabling trust assertions to be formed not only at the level of individual devices but also across links and end-to-end network paths. This provides a necessary foundation for Trusted Path Routing decisions that reflect the current operational state of the network.

Overall, this deliverable provides the architectural baseline and requirements for CASTOR's trust mechanisms, expressed through detailed component definitions and engineering stories. It deliberately does not address protocol instantiation or implementation details; instead, it supplies the design principles and constraints that will guide the protocol specifications and mechanisms to be developed in D3.2.

List of Abbreviations

Abbreviation	Translation
AAR	Aggregate Attestation Result
AK	Attestation Key
AS	Autonomous System
ATL	Actual Trust Level
CE	Compound Evidence
CRL	Certificate Revocation List
DAA	Direct Anonymous Attestation
DICE	Device Identifier Composition Engine
DORE	Delegatable Order-Revealing Encryption
DPE	DICE Protection Environment
EK	Endorsement Key
FSM	Finite State Machine
fTPM	Firmware Trusted Platform Module
OCSP	Online Certificate Status Protocol
ORE	Order-Revealing Encryption
PCR	Platform Configuration Register
PKI	Public Key Infrastructure
RL	Revocation List
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
SGX	Software Guard Extensions
SLO	Service-Level Objective
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
TDE	Trust Decision Engine
TEE	Trusted Execution Environment
TLEE	Trustworthiness Level Expression Engine
TM	Trust Model
TMM	Trust Model Manager
TN-DSM	Trust Network Device Security Monitor
TNDE	Trust Network Device Extensions
TNDI	Trust Network Device Interface
TNDI-SP	TNDI Security Protocol
TPM	Trusted Platform Module
TS	Trust Source
TSM	Trust Sources Manager
TVM	TEE Virtual Machine
VLR	Verifier-Local Revoation
vTPM	Virtualized Trusted Platform Module
ZKP	Zero-Knowledge Proofs

CASTOR Vocabulary Towards Trusted Traffic Engineering Process

ATL (Actual Trustworthiness Level) The ATL reflects the result of an evaluation of a specific (atomic or complex) proposition for a specific scope provided by the TLEE. It quantifies the extent to which a certain node or data can be considered trustworthy based on the available evidence.

ATO (Atomic Trust Opinion) An ATO is a subjective logic opinion created by the TSM when quantifying one specific type of a Trust Source based on trustworthiness evidence. An ATO is formed by a Trust Source in the context of a single trust relationship.

Attestation (I) The process of providing a digital signature for a set of measurements securely stored in hardware, and then having the requester validate the signature and the set of measurements. (II) The issue of a statement, based on a decision, that fulfillment of specified requirements has been demonstrated.

Component Diagram A component diagram is a graph whose vertices represent components necessary to realize a concrete system function, and whose edges represent communication links between these components. A component diagram is used to design function-specific trust models.

Constraint A restriction that governs the behaviour, relationships, or properties of trust objects, trust relationships, or trustworthiness claims. Constraints ensure assessments and attestations adhere to predefined security, integrity, and operational requirements.

Credential A data object that is a portable representation of the association between an identifier and a unit of authentication information, and that can be presented for use in verifying an identity claimed by an entity that attempts to access a system or that can be presented for use in verifying those authorizations for an entity that attempts such access.

Data-centric Trust Data-centric trust is evaluated in the context of a data-centric trust relationship. In this trust relationship, trust is evaluated from one node to data, where the trustor (the one who trusts) is a node and the trustee (the one who is trusted) is data.

Functional (direct) Trust Functional (also called direct) trust is evaluated in the context of a functional trust relationship. In this trust relationship, trust is evaluated between two trust objects having a direct relation, i. e., the trustor has a direct observation of the trustee and engages in a functional relationship with the trustee within the system model, e.g., receiving a data item from the trustee.

Inter- and Intra-domain orchestration The execution of the operational and functional processes involved in designing, creating, and delivering an end-to-end service. It uses network automation to provide services through the use of applications that drive the network. An orchestrator arranges and organises the various components involved in delivering a network service.

Network Exposure Function The Network Exposure Function (NEF) is a pivotal component in the 5G architecture. It functions as a gateway that interfaces between the 5G network and external applications. In essence, NEF exposes the network's capabilities to third-party developers and applications, allowing them to leverage these capabilities for various services. This exposure is done in a secure and controlled manner, ensuring that the network's integrity and efficiency are maintained. NEF provides a set of standardised APIs (Application Programming Interfaces) through which applications can request and interact with the 5G network. This enables a wide range of services, such as real-time analytics and automated responses, to operate seamlessly.

Network Telemetry Network telemetry is a technology for gaining network insights and facilitating efficient and automated network management. It encompasses various techniques and processes used to generate, export, collect, and consume that data for use by potentially automated management applications.

Node-centric Trust Node-centric trust is evaluated in the context of a node-centric trust relationship. In this trust relationship, trust is evaluated from one node to another node so that the trustor (the one who trusts) and the trustee (the one who is trusted) are both nodes.

Objective A user-defined goal that guides the design, implementation, and evaluation of trust assessment mechanisms. Objectives ensure relationships, trustworthiness claims, and attestation processes align with security, integrity, and reliability requirements.

Obligation Obligations describe mandatory actions that subjects needs to perform as a side effect of holding certain rights.

Policy (as RFC4949) 1a. (I) A plan or course of action that is stated for a system or organization and is intended to affect and direct the decisions and deeds of that entity's components or members. (See: security policy.) 1b. (O) A definite goal, course, or method of action to guide and determine present and future decisions, that is implemented or executed within a particular context, such as within a business unit. [R3198].

PROP (Proposition) A *proposition* is a logic statement about some phenomenon of interest whose level of trustworthiness we are interested in assessing. A proposition could be 1) atomic—a proposition whose truth or trustworthiness can be directly assessed, or 2) composite, comprising of multiple atomic propositions. The proposition describes the fulfillment of the properties in relation to *data* or *nodes*.

Referral Trust Referral trust is evaluated in the context of a referral trust relationship. In this trust relationship, trust is evaluated between two trust objects that do not have a direct relation but the trustor has a direct relationship with another intermediate node(s) that have a direct observation of the trustee.

RTL (Required Trustworthiness Level) The RTL reflects the amount of trustworthiness of a node or data that an application considers required in order to characterize this object as trusted and rely on its output during its execution.

Security Policy (as RFC4949) (I) A set of policy rules (or principles) that direct how a system (or an organization) provides security services to protect sensitive and critical system resources. (II) A set of rules to administer, manage, and control access to network resources. [R3060, R3198] (III) A set of rules laid down by an authority to govern the use and provision of security services and facilities.

TA (Trust Assessment) The TA is a component inside the TAF which orchestrates the overall process of trustworthiness level evaluation and trust decision taking.

TAF (Trust Assessment Framework) A software framework which, given a trust model for a specific function running inside a CCAM system, is able to evaluate Trust Sources for trustworthiness evidence and evaluate propositions within the trust model to obtain their ATLs. Optionally, also an RTL can be evaluated and trust decisions can be taken and communicated to the application.

TAF-API (TAF - Application Programming Interface) Application Programming Interface by which the TAF and its functionality can be accessed from an application.

TAF-DT (TAF - Digital Twin) The digital twin allows a vehicle to replicate its TAF including among others its TMs and TSs within a MEC. This allows a vehicle to outsource trust assessment to a MEC where the TAF-DT is expected to run inside a TEE so that confidentiality and integrity of its data and state can be protected from the MEC.

TDE (Trust Decision Engine) The TDE is a component inside the TAF which performs the last step before an output is provided to the application that requested trustworthiness assessment. The TDE either forwards the Actual Trustworthiness Level (ATL) calculated by the TLEE along to the application or outputs a Trust Decision (TD). A TD is created after comparing the ATL to the Required Trust Level (RTL) in a predetermined manner. Whether the output of the TAF is an ATL or a TD depends on the needs of the application requesting trustworthiness assessment.

TLEE (Trustworthiness Level Expression Engine) The TLEE is a component inside the TAF that calculates the level of trustworthiness for a concrete trust model and the proposition that needs to be evaluated. The TLEE uses the numerical values of the Atomic Trust Opinions computed based on the Trust Sources collected in the TSM. Based on these inputs, the TLEE calculates an ATL and provides it to the TA. The TLEE encapsulates most of the Subjective Logic formalism.

TM (Trust Model) Trust model is a graph-based model which is built on top of a system model which represents all components and data needed to perform a certain function. Components represented either create, transmit, process, relay, and receive the data used as input to a function. The vertices in a trust model correspond to an abstraction called trust objects, and the edges in a trust model correspond to trust relationships between a pair of trust objects. The trust model also encompasses a list of Trust Sources used to build up / quantify trust relationships by providing atomic trust opinions. The trust model is a main input to the TMM and the TLEE. Since trust is a directional relationship between two trust objects and it is always in relation to a concrete property or scope, then as part of the trust model, there can be multiple trust relationships between the same two trust objects, depending on different properties of the trust relationship, or the scope of the trust relationship.

TMM (Trust Model Manager) The TMM is a component inside the TAF responsible for storing trust models and making them accessible for TLEE and other purposes. In particular, it is able to provide TMs for specific functions running in a CCAM system, also considering different scopes that TMs may cover.

TN-DSM (Trust Network Device Security Monitor) The TN-DSM is the central security manager of the network device which manages one or multiple TNDIs. The TN-DSM implements the TNDI-SP and enables the orchestrator to onboard and configure the TNDIs and their runtime trust assessment, including the TNDI-SP data channels. The TN-DSM is responsible for configuring the Tracing Hub to collect relevant traces per TNDI based on the active trust profile/s (models with trust propositions for a specific security property) and sharing the traces with the attestation and FSM sources for generating attestation reports / trustworthiness evidence, required for the per-TNDI ATL calculations by the Local TAF Agent. In addition, the TN-DSM is responsible for sharing the resulting traces, evidence, and ATLs with the Global TAF and CASTOR DLT using the TNDI-SP data channels.

TNDE (Trust Network Device Extensions) CASTOR's trusted components instantiated for each CASTOR-enabled network device, e.g., physical router or vRouter. The TNDE includes the TN-DSM, Local

TAF Agent, FSM source and attestation source, Tracing Hub, and TPL data connector. The TNDE is responsible for onboarding a CASTOR-enabled network device unit—forming a TNDI—into the trusted CASTOR network, securely collecting runtime evidence for it, and dynamically assessing its local trust levels.

TNDI (Trust Network Device Interface) Unit joining a CASTOR trusted network domain to perform trusted path routing. For each TNDI a runtime trust assessment will be performed by leveraging the TNDE. The TNDE manages one or multiple TNDIs, depending on the platform. Examples for a TNDI: one CASTOR-enabled HW router, vRouter instance (on a server), or far-edge device.

TNDI Artifact Data or code associated with a TNDI that is monitored by the TN-DSM using the Tracing Hub to collect traces for the trustworthiness evidence generation, e.g., kernel data structures in the router OS, the list of running processes, syscalls sequences, etc.

TNDI-SP (TNDI Security Protocol) Protocol used to onboard a TNDI into the CASTOR network and configure its evidence collection and sharing with the Global TAF and CASTOR DLT. The TNDI-SP consists of a control channel and a data channel. The control channel enables the CASTOR orchestrator to take ownership of a TNDI, onboard it into the trusted network, and configure the runtime evidence collection and trust assessment. The TN-DSM forms the endpoint of the TNDI-SP control channel on the network device of the TNDI. The data channels are configured using the control channel and enable secure sharing of runtime traces, evidence, and ATLs with the upper layer CASTOR components (Global TAF, DLT).

TNDI-SP Control Channel The control channel is used by the CASTOR orchestrator to onboard and configure the TNDI via the TNDE.

TNDI-SP Data Channel Communication channels through which the TNDE of the network device shares traces, evidence, and ATLs of the TNDIs with the upper layer CASTOR components (Global TAF and DLT). The channel provides secure transport and provides ordering guarantees for the trusted data across the CASTOR TNDIs using a centrally controlled freshness mechanism (e.g., IETF epoch markers).

Tracing Hub Tracing manager at CASTOR-enabled network devices, responsible for collecting the traces based on which the trustworthiness evidence will be generated (by the attestation and FSM sources) as a basis for the local trust assessment of each TNDI. The Tracing Hub is part of the TNDE and configured / interfaced by the TN-DSM. The Tracing Hub can operate multiple different Tracing Units to perform the collection of TNDI configurational and/or behavioural traces.

Tracing Unit TNDE-external tracing mechanisms (e.g., VMI, eBPF) operated by the Tracing Hub to collect configurational and/or behavioural TNDI traces, i.e., TNDI Artifacts.

Trust Objects Trust objects are core building blocks of a trust model. They represent entities that assess trust or for which trust is assessed. The trust objects are identified 1) based on the *components* from the component diagram and 2) the *atomic propositions* (i.e., the properties about data or nodes for which trust assessment is conducted).

Trust Relationships A trust relationship is a directional relationship between two trust objects that are called trustor (i. e., the “thinking entity”, the assessor) and a trustee (one who is trusted). The trust relationship is always in relation to a concrete property and a certain scope.

Trusted Path Routing Trusted Path Routing refers to the process and technologies that protects sensitive flows as they transit a network by forwarding traffic to/from sensitive subnets across network devices recently appraised as trustworthy.

Trustworthiness Claims A Trustworthiness Claim (TC) is a form of node-centric ATO provided by a TS and contains a specific data quote used for conveying the information needed by the TAF to make a decision on the trust level of an object. The TC is usually produced (by the Attester) so as to provide trustworthiness evidence (cf. "Trust Source") that can be used for appraising the trustworthiness level of the Attester in a *measurable* and *verifiable* manner. Measurable reflects the ability of the TAF to assess an attribute of the Attester against a pre-defined metric while verifiability highlights the need for all claims to have integrity, freshness and to be provably & non-reputably bound to the identity of the original Attester. TCs might include (among other attributes) evidence on system properties, such as: (i) **integrity**, in the context that all transited devices (e.g., ECUs) have booted with known hardware and firmware; (ii) **safety**, meaning that all transited devices are from a set of vendors and are running certified software applications containing the latest patches; (iii) **communication integrity**.

Trustworthiness Tier Trustworthiness Tier is a categorization of the levels of trustworthiness which may be assigned by the TAF (or another Verifier that appraises the attestation results of a TC and communicates them to the TAF) to a specific Trustworthiness Claim.

TS (Trust Source) A TS manages one or multiple trustworthiness evidence inside the TAF. On request of the TA, it quantifies the trustworthiness of a trustee based on a specific type of evidence in form of an atomic trust opinion.

TSM (Trust Sources Manager) The TSM is a component inside the TAF responsible for handling all available TSes inside a TAF and to establish and integrate new TSes dynamically through a plugin interface.

Zero-Knowledge Proofs (ZKP) is a protocol in which one party (the prover) can convince another party (the verifier) that some given statement is true, without conveying to the verifier any information beyond the mere fact of that statement's truth.

Bibliography

[1] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[2] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574, 2004.

[4] Abdullah K. Alnaim. Securing 5g virtual networks: A critical analysis of sdn, nfv, and network slicing security. *International Journal of Information Security*, pages 1–21, 2024.

[5] Abdullah K. Alnaim, Abdulaziz M. Alwakeel, and Eduardo B. Fernandez. Towards a security reference architecture for nfv. *Sensors*, 22(10):3750, 2022.

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

[7] Alexander Azimov. Route leaks: Status update. RIPE NCC Documentation, 2017. Presented at RIPE 74.

[8] 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.

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

[10] Stefan Berger, Ramon Cáceres, et al. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, 2006. USENIX Association.

[11] H. Birkholz, E. Voit, C. Liu, D. Lopez, and M. Chen. Trusted Path Routing. https://www.ietf.org/archive/id/draft-voit-rats-trustworthy-path-routing-11.html, January 2025.

[12] 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.

[13] 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.

[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] Harold Booth, David Rike, and Gregory Witte. The national vulnerability database (nvd): Overview. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, 2013. Accessed December 15, 2025.

[16] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, 2004.

[17] Kyle Brogle, Sharon Goldberg, and Leonid Reyzin. Sequential aggregate signatures with lazy verification from trapdoor permutations. *Information and computation*, 239:356–376, 2014.

[18] Thomas Bühler, Antonios Milolidakis, Robin Jacob, Marco Chiesa, Stefano Vissicchio, and Laurent Vanbever. Oscilloscope: Detecting bgp hijacks in the data plane. *arXiv preprint arXiv:2301.12843*, 2023.

[19] Kevin Butler, Toni R Farley, Patrick McDaniel, and Jennifer Rexford. A survey of bgp security issues and solutions. *Proceedings of the IEEE*, 98(1):100–122, 2009.

[20] CASTOR. Architectural specification of castor continuum-wide trust assessment framework. Deliverable 4.1, The CASTOR Consortium, 12 2025.

[21] CASTOR. Architectural specification of dynamic enforcement of trust-/network-aware path establishments. Deliverable 5.1, The CASTOR Consortium, 12 2025.

[22] CASTOR. Operational landscape, requirements and reference architecture - initial version. Deliverable 2.1, The CASTOR Consortium, 12 2025.

[23] CASTOR. Device-level & continuum-wide trust extensions and security controls (first release). Deliverable 3.2, The CASTOR Consortium, 03 2026.

[24] Suvradip Chakraborty. Security in border gateway protocol (bgp). In *Encyclopedia of Information Science and Technology*. IGI Global, 2014.

[25] Cisco Systems, Inc. Cisco security advisory: Intelligent node software static key vulnerability (cve-2024-20323). `https://sec.cloudapps.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-inode-static-key-VUVCeynn`, 2024. Accessed: January 5, 2026.

[26] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical Report Cryptology ePrint Archive, Report 2016/086, MIT Computer Science and Artificial Intelligence Laboratory, 2016.

[27] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Dialed: Data integrity attestation for low-end embedded devices. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 313–318, 2021.

[28] Heini Bergsson Debes, Edlira Dushku, Thanassis Giannetsos, and Ali Marandi. Zekra: Zero-knowledge control-flow attestation. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, page 357–371, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Heini Bergsson Debes and Thanassis Giannetsos. Zekro: Zero-knowledge proof of integrity conformance. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Yogesh Deshpande, zhang jun, Houda Labiod, and Henk Birkholz. Remote Attestation with Multiple Verifiers. Internet-Draft draft-deshpande-rats-multi-verifier-03, Internet Engineering Task Force, October 2025. Work in Progress.

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

[32] M. A. Diouf, S. Ouya, J. Klein, and T. F. Bissyandé. Software security in software-defined networking: A systematic literature review. *arXiv preprint*, 2025.

[33] Nada El Kassem, Wouter Hellemans, Ioannis Siachos, Edlira Dushku, Stefanos Vasileiadis, Dimitrios Karas, Liqun Chen, Constantinos Patsakis, and Thanassis Giannetsos. PrivÉ: Towards privacy-preserving swarm attestation. pages 247–262, 01 2025.

[34] Ahmed Elmokashfi, Amund Kvalbein, and Constantine Dovrolis. Bgp churn evolution: a perspective from the core. *IEEE/ACM Transactions on Networking*, 20(2):571–584, April 2012.

[35] ETSI. Network functions virtualisation (nfv). Technical Report ETSI GS NFV-SEC 006, European Telecommunications Standards Institute, 2023.

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

[37] 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.

[38] Nikolaos Fotos, Stefanos Vasileiadis, and Thanassis Giannetsos. Trust or bust: Reinforcing trust-aware path establishment with implicit attestation capabilities. In *2025 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 867–874, 2025.

[39] Steven Galbraith. New discrete logarithm records, and the death of type 1 pairings. https://ellipticnews.wordpress.com/2014/02/01/new-discrete-logarithm-records-and-the-death-of-type-1-pairings/, 2014.

[40] Craig Gentry, Adam O'Neill, and Leonid Reyzin. A unified framework for trapdoor-permutation-based sequential aggregate signatures. In *IACR International Workshop on Public Key Cryptography*, pages 34–57. Springer, 2018.

[41] Craig Gentry and Zulfikar Ramzan. Identity-based aggregate signatures. In *PKC*, pages 257–273, 2006.

[42] 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.

[43] Trusted Computing Group. Trusted platform module library specification, family 2.0, level 00, revision 01.84. Technical report, Trusted Computing Group, March 2025.

[44] Jeffrey Haas. BGP Attribute Escape. Internet-Draft draft-haas-idr-bgp-attribute-escape-03, Internet Engineering Task Force, April 2025. Work in Progress.

[45] Andy Heffernan. Protection of bgp sessions via the tcp md5 signature option. RFC 2385, Internet Engineering Task Force (IETF), 1998.

[46] Geoff Huston, George Michaelson, and Rob Loomans. A profile for x.509 pkix resource certificates. RFC 6487, Internet Engineering Task Force (IETF), 2012.

[47] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. SoK: Introspections on Trust and the Semantic Gap. In *IEEE Symposium on Security and Privacy*, 2014.

[48] Juniper Networks. Security bulletin: Receipt of a specific bgp update may cause rpki policy-checks to be bypassed (CVE-2021-31375). https://supportportal.juniper.net/s/article/2021-10-Security-Bulletin-Junos-OS-Receipt-of-a-specific-BGP-update-may-cause-RPKI-pol
2021. Accessed: January 5, 2026.

[49] Juniper Networks. Security bulletin: Protocol-specific ddos configuration affects other protocols (CVE-2024-39531). https://supportportal.juniper.net/s/article/2024-07-Security-Bulletin-Junos-OS-Evolved-ACX7000-Series-Protocol-specific-DDoS-confi
2024. Accessed: January 5, 2026.

[50] Ioannis Krontiris, Thanassis Giannetsos, and Henk Birkholz. Runtime trusted platform reporting (tpr). Internet Draft draft-rats-runtime-tpr-00, IETF Network Working Group, July 2025. Work in Progress, Intended Status: Informational; expires 8 January 2026.

[51] KVM-VMI Project. KVM-VMI: Kvm-based virtual machine introspection. https://github.com/KVM-VMI/kvm-vmi. Accessed 2026-01-05.

[52] Taeho Lee, Christos Pappas, Adrian Perrig, Virgil Gligor, and Yih-Chun Hu. The case for in-network replay suppression. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, Apr 2017.

[53] Matt Lepinski and Stephen Kent. An infrastructure to support secure internet routing. RFC 6480, Internet Engineering Task Force (IETF), 2012.

[54] 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.

[55] 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.

[56] S. Malik and S. Bera. Security-as-a-function for ids/ips in softwarized network and applications to 5g network systems. *arXiv preprint*, 2025.

[57] Prodosh Mohapatra, John Scudder, David Ward, Randy Bush, and Rob Austein. BGP Prefix Origin Validation. RFC 6811, January 2013.

[58] Mathias Morbitzer, Benedikt Kopf, and Philipp Zieris. Guarantee: Introducing control-flow attestation for trusted execution environments. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 547–553. IEEE, 2023.

[59] Soroush Motaali, Jorge E. López de Vergara, and Luis De Pedro. Real-time anomaly detection in bgp: Challenges, ipv6 considerations, and machine learning opportunities. In *Proceedings of the IEEE 11th International Conference on Network Softwarization (NetSoft)*, pages 380–383. IEEE, June 2025.

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

[61] Thanh Nguyen, Meni Orenbach, and Ahmad Atamli. Live system call trace reconstruction on Linux. *Forensic Science International: Digital Investigation*, 2022. Proceedings of the Twenty-Second Annual DFRWS USA.

[62] A. Oqaily et al. Cross-level security verification for network functions virtualization (nfv). *IEEE Transactions on Dependable and Secure Computing*, 22(4):3240–3258, July–August 2025.

[63] Meni Orenbach, Rami Ailabouni, Nael Masalha, Thanh Nguyen, Amhad Saleh, Frank Block, Fritz Alder, Ofir Arkin, and Ahmad Atamli. BlueGuard: Accelerated Host and Guest Introspection Using DPUs. In *USENIX Security Symposium (USENIX Security 25)*. USENIX Association, 2025.

[64] Palo Alto Networks. CVE-2024-0012: Pan-os: Authentication bypass in the management web interface. https://security.paloaltonetworks.com/CVE-2024-0012, 2024. Accessed: 2025-01-05.

[65] 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 *ESORICS*, pages 321–340, 2025.

[66] Arinjita Paul, Sabyasachi Dutta, Kouichi Sakurai, and C Pandu Rangan. An efficient sequential aggregate signature scheme with lazy verification. *Cryptology ePrint Archive*, 2025.

[67] PCI-SIG. TEE Device Interface Security Protocol (TDISP). https://pcisig.com/tee-device-interface-security-protocol-tdisp, 2022. Accessed: January 5, 2026.

[68] Peter Psenak, Shraddha Hegde, Clarence Filsfils, Ketan Talaulikar, and Arkadiy Gulko. IGP Flexible Algorithm. RFC 9350, February 2023.

[69] Dominik Arne Rebro, Stanislav Chren, and Bruno Rossi. Source code metrics for software defects prediction. In *Proceedings of the 38th ACM/SIGAPP symposium on applied computing*, pages 1469–1472, 2023.

[70] Yakov Rekhter, Tony Li, and Susan Hares. A border gateway protocol 4 (bgp-4). RFC 4271, Internet Engineering Task Force (IETF), 2006.

[71] RIPE NCC Learning & Development. Bgp security webinars: Deploying rpki. RIPE NCC Documentation, 2023. Accessed: 2023.

[72] Evgenia-Niovi Sassalou, Stefanos Vasileiadis, Stylianos A. Kazazis, Georgia Protogerou, Nikos Varvitsiotis, Dimitrios S. Karas, Thanassis Giannetsos, and Symeon Tsintzos. A puf-based root-of-trust for resource-constrained iot devices. In *2025 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 824–831, 2025.

[73] Fabian Schwarz. TrustedGateway: TEE-Assisted Routing and Firewall Enforcement Using ARM TrustZone. In *25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22. ACM, 2022.

[74] Fabian Schwarz and Christian Rossow. 00SEVen – re-enabling virtual machine forensics: Introspecting confidential VMs using privileged in-VM agents. In *USENIX Security Symposium (USENIX Security 24)*. USENIX Association, August 2024.

[75] Brandon A. Scott, Michael N. Johnstone, and Piotr Szewczyk. A survey of advanced border gateway protocol attack detection techniques. *Sensors*, 24(19):6414, 2024.

[76] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. Summarizing known attacks on transport layer security (tls) and datagram tls (dtls). RFC 7457, Internet Engineering Task Force (IETF), 2015.

[77] 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.

[78] Trusted Computing Group. TCG Attestation Framework, Part 1: Terminology, Concepts, and Requirements. Specification Version 1.0, Trusted Computing Group, November 2025.

[79] Jean-Philippe Vasseur and Jean-Louis Le Roux. Path computation element (pce) communication protocol (pcep). RFC 5440, Internet Engineering Task Force (IETF), 2009.

[80] Eric Voit, Henk Birkholz, Thomas Hardjono, Thomas Fossati, and Vincent Scarlata. Attestation Results for Secure Interactions. Internet-Draft draft-ietf-rats-ar4si-09, Internet Engineering Task Force, August 2025. Work in Progress.

[81] Sergei Volpe, Juan Caballero, et al. fTPM: A software-only implementation of a TPM chip. In *Proceedings of the 25th USENIX Security Symposium*, Austin, TX, USA, 2016. USENIX Association.

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

[83] Wenjie Yang, Junzhe Fan, Futai Zhang, Anjia Yang, and Zhiquan Liu. An efficient revocable identity-based aggregate signature scheme with designated verifiers in healthcare wireless sensor networks. *IEEE Internet of Things Journal*, 2025.

[84] Zhenhai Zhang, Yin Zhang, Y. Charlie Hu, and Z. Morley Mao. Practical defenses against bgp prefix hijacking. In *Proceedings of the 2007 ACM CoNEXT Conference*, ACM SIGCOMM Computer Communication Review, pages 1–12. Association for Computing Machinery, December 2007.

[85] Zyxel Networks. Zyxel security advisory for os command injection vulnerability of firewalls (cve-2023-28771). https://www.zyxel.com/global/en/support/security-advisories/zyxel-security-advisory-for-remote-command-injection-vulnerability-of-firewalls, 2023. Accessed: January 5, 2026.