Net-Zero self-adaptive activation of distributed self-resilient augmented services

D3.5 Pre-Deployment Security per Construction Measures.r1

Lead beneficiary	TSS	**Lead author**	Vincent Lefebvre
Reviewers	Md Munjure Mowla, Robert Gdowski (ISRD), Gürkan Gür (ZHAW)		
Type	R	**Dissemination**	PU
Document version	V1.0	**Due date**	30/09/2025

Co-funded by the European Union

6GSNS

Project funded by

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs, Education and Research EAER
State Secretariat for Education, Research and Innovation SERI

UK Research and Innovation

Project information

Project title	Net-Zero self-adaptive activation of distributed self-resilient augmented services
Project acronym	NATWORK
Grant Agreement No	101139285
Type of action	HORIZON JU Research and Innovation Actions
Call	HORIZON-JU-SNS-2023
Topic	HORIZON-JU-SNS-2023-STREAM-B-01-04 Reliable Services and Smart Security
Start date	01/01/2024
Duration	36months

Document information

Associated WP	WP3
Associated task(s)	T3.4
Main Author(s)	Vincent Lefebvre (TSS)
Author(s)	Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Eleni Chamou, Stelios Mpatziakas, Vangelis V. Kopsacheilis, Alexandros Papadopoulos, Aristeidis Papadopoulos, Konstantinos Nikiforidis, Athanasios Korakis, Anastasios Drosou (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Sumeyya Birtane, Mays Al-Naday (UESSEX), Maria Safianowska (ISRD), Wissem Soussi (ZHAW)
Reviewers	Md Munjure Mowla, Robert Gdowski (ISRD), Gürkan Gür (ZHAW)
Type	R - Document, Report
Dissemination level	PU - Public
Due date	M21 (30/09/2025) – Extended to M22 (10/10/2025)
Submission date	10/10/2025

Document version history

Version	Date	Changes	Contributor (s)
v0.1	16/06/2025	Initial table of contents	Vincent Lefebvre (TSS)
v0.2	11/07/2025	Initial contributions in various sections	Vincent Lefebvre, Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Alexandros Papadopoulos, Aristeidis Papadopoulos, Konstantinos Nikiforidis, Athanasios Korakis, Anastasios Drosou (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Sumeyya Birtane, Mays Al-Naday (UESSEX), Maria Safianowska (ISRD)
v0.3	24/07/2025	Updated content in sections 4 and 5	Vincent Lefebvre, Mark Angoustures (TSS)
v0.4	25/08/2025	Revised content for sections 2 and 3	Vincent Lefebvre, Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Eleni Chamou, Stelios Mpatziakas, Athanasios Korakis, Anastasios Drosou (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Sumeyya Birtane, Mays Al-Naday (UESSEX), Maria Safianowska (ISRD)
v0.5	10/09/2025	Pre-final version for late inclusions	Vincent Lefebvre (TSS)
v0.6	16/09/2025	Last refinements in various sections	Vincent Lefebvre, Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Eleni Chamou, Stelios Mpatziakas, Vangelis V. Kopsacheilis (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Sumeyya Birtane, Mays Al-Naday (UESSEX), Maria Safianowska (ISRD), Wissem Soussi (ZHAW)
v0.7	19/09/2025	Draft ready for peer review	Vincent Lefebvre (TSS)
v0.7.5	23/09/2025	Reviewed version	Md Munjure Mowla, Robert Gdowski (ISRD), Gürkan Gür (ZHAW)
v0.8	26/09/2025	Version after addressing peer reviewers' recommendations	Vincent Lefebvre (TSS), Virgilios Passas (CERTH)
v0.8.5	29/09/2025	Quality review	Joachim Schmidt (HES-SO)
v0.9	08/10/2025	Final review and refinements	Vangelis V. Kopsacheilis, Antonios Lalas (CERTH)

Version	Date	Changes	Contributor (s)
v1.0	10/10/2025	Final version ready for submission	Antonios Lalas (CERTH)

Disclaimer

Copyright message

Contents

List of acronyms and abbreviations

Abbreviation	Description
AD	Anomaly Detection
AI	Artificial Intelligence
AI/ML	Artificial Intelligence / Machine Learning
API	Application Programming Interface
CC	Confidential Computing
CCA	ARM's TEE Confidential Compute Architecture technology
CIA	Confidentiality Integrity and Availability
CPU	Central Processing Unit
CNCF	Cloud Native Computing Foundation
CNF	Containerized Network Function
CTI	Cyber Threat Intelligence
CVM	Confidential VM
D-MUTRA	DLT-backed MUtual Remote Attestation, a solidshield framework
DDoS	Distributed Denial of Service
DEFM	Decision eXplainablity for MTD
DFE	Decentralized Feature Extraction
DLT	Distributed Ledger Technology (aka blockchain)
DoS	Denial of Service
DoSt	Denial of Sustainability
DPSF	Data Plane Security Function
ELF	Executable and Linkable Format
E2E	End to end
IoT	Internet of Things
ISD	Intrusion Detection System
ITS	Intelligent Transportation System
JIT	Just in Time (compilation)
K8s	Kubernetes (i.e., a container-based framework)
KPI	Key Performance Indicator
MANO	Management and Orchestratoin
MGEC	MTD Green Energy Consumption
ML/DL	Machine Learning / Deep Learning
MMT	MONT's network Monitoring (i.e., commercial name)
MMTC	Massive Machine Type Communication
MTD	Moving Target Defense
MTID	Mean Time to Implement Action
MTTD	Mean Time to Detect
NCC	Network and Computing Convergence
NIC	Network Interface Component
O-RAN	Open Radio Access Network

Abbreviation	Description
ONOS	Open Network Operating System
PDSCM	Pre-deployment Security per Construction measure
P4	Programming Protocol-Independent Packet Processors (P4)
QoE	Quality of Experience
QoS	Quality of Service
RBAC	Role Based Access Control
SDN	Software Defined Network
SECaaS	Security as a Service
SEV SNP	AMD's Secure Encrypted Virtualization-Secure Nested Paging TEE technology
SGX	Intel's Secure Guard Extension TEE technology
SotA	State of the Art
TCB	Trusted Computed Basis
TDX	Intel's Trust Domain Extension TEE technology
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TPM	Trusted Processing Module
UC	Use Case
UDP	User Datagram Protocol
UE	User Equipment
UE	User Equipment
UPF	User Plane Function
uRLLC	Ultra Reliable Low Latency Communication
VM	Virtual Machine
VNF	Virtual Network Function
WASM	Web Assembly (an interpreted language technology derived from JavaScript)
WG11	O-RAN Working Group 11 (security WG)
x86	Intel processor architecture
XAI	Explainable AI

List of figures

List of tables

Executive summary

This deliverable presents the Pre-Deployment Security per Construction Measures (PDSCM) developed in the NATWORK project. PDSCM refers to security mechanisms and actions applied prior to payload deployment, aiming to strengthen software artefacts against security threats from the outset. The work addresses three main types of software payloads executed across the computing continuum: native binaries, containerized applications, and WebAssembly (WASM) modules.

The deliverable first defines the PDSCM concept and presents a state-of-the-art (SotA) review of existing measures and associated threat models. The SotA is then continued by addressing the three-payload format separately for clarity.

The deliverable enumerates the actions carried out in NATWORK for each payload format. Notably, they elevate significantly the security of xAPP during execution in the near RT-RIC, WASM module during execution and exemplify how a security service (i.e., MMT anomaly detection) can be secured by two alternative techniques, i.e. TEE (Trusted Execution Environments) and D-MUTRA remote attestation.

Finally, the deliverable analyses the differences in PDSCM applicability across workload and system levels and discusses their implications for usability and operational efficiency. The results demonstrate how PDSCMs contribute to NATWORK's overall vision of secure, performant, and sustainable operations across the computing continuum.

.

1. Introduction

NATWORK aims to regulate performance and security at sustainable resource consumption using bio-inspired principles as do natural entities and immune systems. When projected to telecom networks, these bio-inspired mechanisms can be set as means to reconcile security and sustainability, security and performance, performance and sustainability.

PDSCMs are pre-deployment security measures applied on software payloads, reinforcing their immunity against various threats. Three payload formats are discussed (i.e., native, containers and WASM) with their specific facets.

When payloads are protected by construction before deployment, all attack vectors are less efficient and the whole system healthier and more sustainable. However, on the other hand, an important consideration is to assess the direct performance impact caused by the PDSCM, which opposes NATWORK's concept of higher-performance and more cost-efficient cybersecurity. An overreaction induced by security is potentially resource costly. Performance penalties must be measured and when possible be adjustable, as discussed in this deliverable.

1.1. Purpose and structure of the document

The purpose of the document is to assess and position how PDSCMs are beneficial to the NATWORK concept. This work includes a specific SotA and the description of specific PDSCM applied over the three treated formats.

Following the Introduction, which sets the stage for the document's purpose, audience, and its interconnections within the project's framework, the structure continues as follows:

Sections:

2. **Section 2 SotA:** Presents the PDSCM state of the art, including our definition of PDSCM
3. **Section 3 NATWORK PDSCMs on containerized payloads:** Describes the NATWORK specific research actions towards container-oriented PDSCMs
4. **Section 4 NATWORK PDSCMs on native payloads:** Describes the NATWORK specific research actions towards native-oriented PDSCMs
5. **Section 5 NATWORK PDSCMs on WASM:** Describes the NATWORK specific research actions towards WASM-oriented PDSCMs
6. **Section 6 Conclusions:** Wraps up the document, reflecting on the project's strategic orientation and establishing expectations for future milestones.

1.2. Intended Audience

The NATWORK D3.5 Deliverable Pre-Deployment Security per Construction Measures.r1 is for Public Dissemination. It is there devised for the internal and external use of the NATWORK consortium, comprising members, project partners, affiliated stakeholders and the public. This document mainly focuses on the pre-deployment security per construction measures of the project, thereby serving as a referential tool throughout the project's lifespan.

1.3. Interrelations

The NATWORK consortium integrates a multidisciplinary spectrum of competencies and resources from academia, industry, and research sectors, focusing on user-centric service development, robust economic and business models, cutting-edge cybersecurity, seamless interoperability, and comprehensive on-demand services. The project integrates a collaboration of fifteen partners from ten EU member states and associated countries (UK and CH), ensuring a broad representation for addressing security requirements of emerging 6G Smart Networks and Services in Europe and beyond.

NATWORK is categorized as a "Research Innovation Action - RIA" project and is methodically segmented into 7 WPs, further subdivided into tasks. With partners contributing to multiple activities across various WPs, the structure ensures clarity in responsibilities and optimizes communication amongst the consortium's partners, boards, and committees. The interrelation framework within NATWORK offers smooth operation and collaborative innovation across the consortium, ensuring the interconnection of the diverse expertise from the various entities (i.e., Research Institutes, Universities, SMEs, and large industries) enabling scientific, technological, and security advancements in the realm of 6G.

The current **D3.5 – "Pre-Deployment Security per Construction Measures"** deliverable addresses specific software payload hardening techniques, applied at deep and elementary level (i.e., software payloads). However, high level security services, applied at the different layers and technical domains of NATWORK (RAN, cloud, core, data plane, orchestration and management layer), described in Deliverable D2.1 – "**General State of the Art Security"** and in Deliverable D3.1 – "**Secure by design orchestration and Management"** are all based on different and distributed software payloads. By hardening these payloads, PDSCMs directly impact the security of security services. These inter-deliverable relations are detailed in Section 2.

D3.5 is also related to D2.2 – "**Use Case Scenarios and Requirements",** where use cases will implement PDSCM (e.g., native PDSCM applied on MMT probe takes part of Use Case 4.5 Enabling optimized explainable MTD). Finally, D3.5 will feed the integration and validation efforts within WP6, for evaluating and improving the assets presented in this deliverable.

2. State of the art analysis

2.1. Introduction

Our state-of-the-art analysis is focused on pre-deployment security per construction measures (PDSCM), specific actions on the workloads, to be taken prior to their deployment, and covering the workload formats of x86 executables, containers and WASM.

For clarity, this section starts with a definition of PDSCM, followed with exemplification of their usage inside high-level service security solutions developed in priorly submitted deliverables (i.e., D 2.1, D3.1), clarifying how PDSCMs contribute to 6G services security. Through this, we intend to clarify their usability and merits to cope with NATWORK security challenges, reconciliating networking security, performance and sustainability.

2.1.1. Definition and usability

2.1.1.1. Definition

As a simple definition, PDSCMs embrace the following criteria:

- Act at the software payload level
- Enhance the security of workload against a specific security threat model
- Implemented prior deployment, by activation of tools or security methodologies
- Modify the workloads, security-related parameters and their execution environments

Table 1 enumerates PDSCMs, as commonly used in the SotA. It is worth noting that the first three PDSCM dealing with CIA threats are cardinal on which all following PDSCMs depend. For instance, user right enforcement solution must be protected and are defended against CIA attacks.

Table 1. PDSCM enumeration

PDSCM	Threat Model	Employed techniques
Confidentiality preservation	-Static (i.e., on executable or module file) code analysis -Dynamic (i.e., on memory footprint) code analysis	-Placement in trusted execution environment -Code section encryption -Code obfuscation
Integrity preservation	-Static file tampering -Memory footprint tampering	-Placement in trusted execution environment -Authentication (i.e., delivering execution environment local assurance of code integrity at onboarding)

PDSCM	Threat Model	Employed techniques
		-Remote attestation (i.e., delivering remotely the assurance that code is integrated at on-boarding) -Runtime integrity verification
Availability preservation	Flood DoS, DDoS, deprivation of resource	-Resource isolation -Resource monitoring -Workload performance monitoring -System level network traffic limitation
Singularization	-IP theft -Cloning -Impersonation	-Placement in TEE + selective provisioning of a key needed for execution -Selective provisioning of a key needed for execution
Locality enforcement	-IP theft -Illicit placement outside a permitted perimeter	-Placement in TEE + selective provisioning of a key needed for execution
Interfaces hardening	-API abuse	-Hardening APIs with RBAC
User right enforcement (against IPRs)	-Licence violation	-Digital right management techniques such as: -Code-to-machine binding -Software activation method delivering activation tokens - Tokenization
Vulnerability curation	-Detect then exploit a vulnerability	-Safe coding methodology -Dependency vulnerability scanning -Code confidentiality, preventing discovery
Access control	-Violation of access policy to content or functionalities, through impersonation or privilege escalation	-Role Based Access Control (RBAC) enforcement

2.1.1.2. Workload migratability

The usability of PDSCM varies with the level where it is setup as follows:

- Technology-level (e.g., TEE enforcement, VM isolation), restricted workload deployment in duly equipped or specific platform)

- System-level parametrization (e.g., network flow limitation) requires the user possession of system administrative right. Thereafter, at deployment, the workload should be deployed in such tuned systems.
- Workload-level (e.g., binary rewriting, vulnerability safe programming) requires getting access to the code (and right to change it). No restriction applies during workload deployment.

Workload migratability over the continuum is a very interesting property, magnified by workload-level PDSCM. The workload is protected by itself with no dependence on the platform.

2.1.2. Interaction with service-level security solutions

Already submitted D2.1 and D3.1 are service-level security focused, hence located at a higher level than atomic software payloads as considered here. D2.1 produces a 360° service security survey while D3.1 is more specifically addressing secure orchestration and data plane computation offloading (i.e., which brings its own security considerations and needs). As a 6G service security depends on many, composed and chained software payloads spread along the traffic pathway either directly traversed (i.e., network functions) or indirectly (e.g., networking management and orchestration, security functions), software security appears as a common exigency to be fulfilled on each running software payloads, being security-related or not. Hence, PDSCM is applied to each software workload deployed by service-level security solutions to enhance the trust and security assigned to service-level security solutions.

2.1.3. Interaction with D 2.1 service level security services

D2.1 – "*SotA Analysis & Benchmark Assessment"* provides a detailed SotA analysis on prevalent security solutions implemented at the different technical domains traversed by a 6G service. Key attack vectors on the RAN, data plane, orchestration, and edge-core are detailed. It produces a list of commonly found security services and associated technologies at different network layers. For clarity, we define the interaction of PDSCM against each of these functions in Table 2.

Table 2. Potential PDSCMs leverage in D2.1 Security Services

D2.1 Service level Security	List of potentially activated PDSCM (at workload level)
Defence in Depth	-RBAC perimetric security (parametrization)
Workload MTD	-TEE enablement, for a secured migration of attacked workloads -Remote attestation, easing workload secure migration -Workload deep monitoring (early bird anomaly detection)
Workload isolation	-Runtime integrity (preventing tampering attack) -Performance monitoring (preventing resource attrition by container co-residents)

D2.1 Service level Security	List of potentially activated PDSCM (at workload level)
Trust management	-Integrity of trust metrics collectors and aggregating scoring algorithms -Confidentiality of trust metrics collectors and aggregating scoring algorithms -Singularisation of trust assessors
Attack detection and protection	-On the anomalous traffic detection workload -Integrity elevation techniques (e.g., TEE, remote attestation, runtime integrity verification) -Confidentiality elevation techniques (e.g., encryption, placement in TEE) -Locality enforcement, mitigating impersonation
Machine learning Frameworks for CTI analysis	Multi stakeholder collaborative CTI, with distributed CTI nodes consuming and producing CTI feeds. -CTI node remote attestation and continuous integrity verification -TEE placement of each CTI node producing and consuming feeds -CTI node singularization for multi factor identification -Cloud security, VM introspection threat model: -Continuous workload integrity verification -TEE-bastioned VM (e.g., TDX, SEV-NP, CCA TEE) -Workload singularization for identity checks.
Service accurate monitoring and traceability	-Monitoring metric producers integrity verification -Monitoring metric producers confidentiality preservation by TEE placement -Monitoring metric producers singularization for MFA

2.1.3.1. Matching D2.1 priority challenges

The list of priority challenges and how they are potentially addressed through PDSCM included in deliverable D2.1 is presented in the following table.

Table 3. Security challenges by PDSCM leverage as per D2.1

D 2.1 prioritized challenge	PDSCM match and high-level specification
Fostering software migratability	- Payload **self-contained hardening**, directly implies platform-agnosticism. Not all PDSCMs are self-contained (e.g., TEE-dependent) but a significant number are (e.g., modified payload for tampering resilience)
Security and Privacy (e.g., platform agnostic security,	- Devise novel, **low-noise continuous integrity verification** Data governance Policies:

D 2.1 prioritized challenge	PDSCM match and high-level specification
continuous security, data governance)	-Establish non-ambiguous identification and localization of data consumers and producers, based on novel PDSCM.
Energy efficiency and sustainability	On-demand security: - Develop PDSCM which trigger security verification **on-demand**, hence drastically reducing energy consumption and performance impact.

2.1.4. Interaction with D 3.1

Deliverable D3.1 – "*Secure-by-design orchestration and management & Data plane computation offloading"* discusses two interrelated matters. The document details service and function orchestration at various technical domains (i.e., far edge, CRAN, core, and, finally, the data plane). In a general perspective, orchestration and function placement decisions shall derive from the ingestion of trustworthy metrics reflecting the current load state at the targeted hosting platform and of course be trustworthy and secure by itself. As stated in Table 4, PDSCM can contribute to reach these security attributes. The deliverable highlights the relevance of MANO API security, the high heterogeneity of the different entities which take part in the management and orchestration, as well as the core requirement for ultra-fast decision taking. When energy efficiency is piled up on the list of high-level specifications, we reach a challenge or trade-off between orchestration responsiveness, reliability/security and energy. Last, AI will be an essential asset to make these multi-modal and complex decisions. Therefore, AI trustworthiness and security are pivotal.

Table 4. Potential PDSCMs leverage in D 3.1 MANO

D3.1 Orchestration domains	How PDSCM interfere
Orchestration at the Extreme Edge	All orchestration and management solutions and workloads are exposed to CIA threat models. PDSCMs shall be used to harden such deployed workloads. Typically, code tampering can have devastating influence on the network reliability or energy consumption.
Orchestration at the CRAN	
Orchestration at the Core	
Data plane function off-loading	

2.1.5. Summary

As illustrated by this matching work, PDSCMs directly contribute to fulfil NATWORK's security challenges as stressed in D 2.1 and D 3.1. In practice, their contribution can be essential and a condition to meet specific identified challenge. Typically, magnifying payload migratability along the continuum can only met if the payload security is self-contained, resulting from a PDSCM.

In a general perspective, all NATWORK security services exemplified in D2.1 and MANO services exemplified in D3.1 are built and dependent of several software payloads whose hardening by PDSCMs makes them more reliable and at lower resource consumption.

2.2. PDSCMs for containers

2.2.1. Introduction

Containerized environments have become the de facto standard for deploying and scaling cloud-native applications, but their flexibility and efficiency also introduce new security challenges. Pre-deployment hardening is therefore a critical phase, as it establishes the foundation for ensuring that workloads can be executed with strong guarantees of confidentiality, integrity, and availability (CIA). By addressing security early in the lifecycle, it is possible to reduce the attack surface, prevent misconfigurations, and ensure that orchestration frameworks operate on trusted components.

This section surveys the key aspects of pre-deployment container and microservice security, building upon novel Confidential Computing (CC) frameworks, DevSecOps practices, and Kubernetes-native hardening approaches. First, we provide an overview of container security concerns and their mapping to the CIA triad, highlighting how emerging Confidential Computing frameworks and secure orchestration mechanisms extend traditional models of protection (Section 2.2.2). We then examine pre-deployment container security by construction, analysing frameworks such as CNCF's CoCo. This survey evaluates their security merits, performance implications, and migratability, identifying trade-offs relevant to practical adoption (Section 2.2.2).

Next, we focus on microservice-level hardening, building on the secure-by-design orchestrator (sFORK) introduced in Deliverable D3.1. From a DevSecOps perspective, declarative modelling, pre-established secure inter-cluster channels, and strict role-based access controls are discussed as essential measures to ensure that orchestration begins from a secure baseline (Section 2.2.3).

We then turn to Kubernetes security analysis, since Kubernetes has emerged as the dominant orchestration platform. We describe its main security features—such as Network Policies, Pod Security Policies (and their successors, Pod Security Admission modes), RBAC, and security contexts—while evaluating their strengths and weaknesses with respect to CIA, performance, and usability (Section 2.2.4).

Finally, we extend the scope beyond conventional cloud-native environments by considering the O-RAN ecosystem, where containerized xAPPs are onboarded in the near Real-Time RIC. This

context highlights specific pre-deployment verification and attestation challenges, as defined by O-RAN WG11, and illustrates how container hardening principles must be adapted to telecom-grade distributed systems (Section. 2.2.5).

Together, these subsections provide a comprehensive view of container and microservice pre-deployment hardening techniques, emphasizing how different frameworks and orchestrators converge toward the goal of delivering trustworthy and resilient cloud-native applications.

2.2.2. Confidential containers frameworks

With the growing traction to large TCB trusted execution environments, their adoption for containers through confidential containerization is becoming a mature technology. We had produced a technical survey of four emblematic frameworks (i.e., CoCo, MarbleRun, Parma,TCX) with the lens of performance, sustainability and security.

2.2.2.1. Confidential Containers (CoCo)

Security

The Confidential Containers (CoCo) project is an open-source Cloud Native Computing Foundation (CNCF) initiative that integrates Trusted Execution Environments (TEEs) with Kubernetes to protect data in use at the pod level [1] CoCo builds on Kata Containers by running each pod inside a Confidential VM (CVM) – a lightweight VM with memory encryption – so that workloads are isolated not only from each other but even from the host and cloud administrator [2]. CoCo introduces a “Trustee” component (including a Key Broker Service and Attestation Service) to handle remote attestation and key management for these CVMs [3]. This means that each container’s image and startup state can be measured and verified, and encryption keys for secrets or images are only released if the pod is confirmed to be running in a genuine TEE-backed VM. Overall, CoCo significantly strengthens confidentiality and integrity: even if the host OS or hypervisor is compromised, the encrypted memory and attestation process protect the container’s code and data.

Performance

CoCo’s use of hardware-backed VMs (like Intel TDX [4] or AMD SEV [5] adds some overhead compared to standard containers, but this overhead is mostly attributable to the TEE mechanisms themselves. In practice, CoCo can achieve near-native performance for many workloads. For example, one study found that running a workload under CoCo on an SEV-enabled cluster incurred only about an 8% throughput overhead relative to a native Kubernetes pod (versus ~5% overhead using Kata alone without memory encryption [6]. Another evaluation noted that with proper tuning, Confidential VMs perform comparably to non-confidential VMs – the difference

was often within single-digit percentages for CPU and I/O-bound tasks. The “peer pods” mode (where pods are launched as confidential VMs via cloud provider APIs) can introduce higher startup latency and resource overhead (e.g. one vCPU reserved for TEE runtime) but yields strong isolation [3]. In summary, CoCo’s performance overhead is low enough that most applications see only modest slowdowns, making it feasible for production use.

Sustainability

CoCo leverages TEEs already available in cloud and on-premises hardware to securely extend workloads across hybrid cloud [7]. It enables secure cloud bursting – confidentially offloading overflow work from private datacenters to public cloud – without maintaining duplicate infrastructure [8]. By scaling confidential workloads on demand using existing TEE-enabled servers, CoCo maximizes hardware utilization and avoids idle, redundant machines. This efficient use of resources translates into better energy efficiency for confidential computing deployments.

2.2.2.2. MarbleRun

Security

MarbleRun [9] is an open-source platform that acts as a “service mesh for confidential computing”, particularly targeting Intel SGX enclaves. Instead of VMs, MarbleRun orchestrates process-level TEEs (enclaves) across a distributed application. It provides a deployment manifest that specifies the expected cryptographic identity and connections of each microservice enclave, and it will only consider the overall application trusted if all components match this manifest. This yields a powerful security guarantee: an entire pipeline of services can be remotely attested and verified as a unit, rather than just individual enclaves. MarbleRun also handles secure key management and inter-service encryption transparently. Upon startup, each enclave gets a certificate issued by MarbleRun’s CA and uses it to establish mutually authenticated TLS connections with other enclaves, ensuring data exchanged between services is encrypted and only goes to attested endpoints. Secrets (like decryption keys or credentials) can be sealed to the enclave identities and distributed via MarbleRun once the deployment is verified. In essence, MarbleRun extends zero-trust principles: even if the underlying Kubernetes nodes or networks are untrusted, the enclave network remains secure and verifiable at runtime.

Performance

Running containerized workloads inside SGX enclaves does introduce performance considerations. Intel SGX enclaves have hardware memory protections that can cause overhead on I/O and memory-intensive operations (e.g., due to enclave context switches and limited secure memory sizes). In general, SGX-based solutions tend to show higher overheads for heavy workloads compared to VM-based TEEs – one study notes that AMD SEV (VM encryption) had negligible performance impact, whereas Intel SGX could introduce significant slowdowns in

certain scenarios. However, MarbleRun's design allows enclaves to be selectively used for the sensitive parts of an application, and its overhead can be modest for typical microservice interactions. MarbleRun itself adds a small constant overhead for attestation and key exchange during startup, but after that, services communicate directly over TLS with negligible additional latency. Thus, while developers should expect some overhead from using SGX enclaves, MarbleRun demonstrates that a distributed enclave architecture can still meet performance requirements for many applications.

Sustainability

MarbleRun orchestrates Intel SGX enclaves as Kubernetes-managed microservices, allowing confidential workloads to scale up or down flexibly without static overprovisioning. Secure services are deployed at a fine-grained microservice level, so each enclave is lightweight and uses only the necessary CPU and memory. By integrating enclave workflows into Kubernetes, MarbleRun ensures resources are allocated on demand, supporting sustainability goals. The result is a confidential microservice architecture that provides strong security while minimizing a waste of computing and energy resources.

2.2.2.3. Parma

Security

Parma is the architecture underpinning Microsoft Azure's confidential container groups, designed to provide strong confidentiality without altering container images [10]. It leverages AMD SEV-SNP (a VM-level TEE) to run an entire container group inside a hardware-encrypted VM (also called a UVM – UltraViolet VM in Azure's terms). Parma's key innovation is the use of an attested execution policy that defines exactly what actions the cloud's container runtime is allowed to perform within the guest VM. At launch, the policy (covering permitted system calls, mount operations, network config, etc.) is cryptographically measured and included in the hardware attestation report. This means the attestation not only vouches for the VM's initial software (kernel, guest agent) but also locks down how containers inside can behave. If anything outside the policy is attempted (e.g., mounting an unexpected filesystem layer or executing a disallowed command), the guest agent will block it, and the deviation would make the attestation report invalid. In addition, Parma uses proven techniques to protect container data: container image layers are stored on an integrity-protected file system (using dm-verity), and any writable storage is encrypted so that plaintext data only ever appears inside the VM's secure memory. The result is a very strong security posture: the cloud provider's host OS and hypervisor are excluded from the trust boundary, and even a malicious or compromised infrastructure cannot inject code or inspect data without detection. Only the combination of the tenant's approved container images and the Parma guest agent (running under SEV-SNP's protection) are in the Trusted Computing Base.

Performance

Parma was built to impose nearly zero performance penalty beyond the cost of hardware encryption. Empirical evaluations on prototype Azure Container Instances showed that introducing Parma's policy enforcement had a negligible effect on throughput and latency compared to using SEV-SNP alone [10]. Parma doesn't heavily modify the runtime execution path – it primarily adds checks during container setup and relies on hardware to handle memory protection. Networking and disk I/O operations are still hardware-accelerated inside the VM, so throughput remains high. In summary, workloads under Parma run at virtually the same speed as they would in a normal confidential VM. This low overhead means users of Azure confidential containers can achieve strong security without sacrificing the performance or scalability of their applications.

Sustainability

Parma isolates container groups inside individual VM-based TEEs, combining virtual machine security with container agility. Its design introduces almost no performance overhead – around 1% additional overhead in tests – which means negligible extra energy consumption for confidentiality [10]. With such low CPU and memory overhead, Parma's confidential containers run nearly as efficiently as ordinary containers. This balance of strong isolation and performance ensures security is achieved with minimal impact on resource usage, promoting sustainable computing.

2.2.2.4. Trusted Container Extensions (TCX)

Security

Trusted Container Extensions (TCX) is a research prototype architecture that combines the agility of Docker containers with the protection of hardware TEEs [11]. In TCX, each container runs inside a lightweight VM called a Secure Container VM (SC-VM), which is backed by AMD SEV encryption to ensure the container's memory is always encrypted and cannot be read or tampered with by the host OS or hypervisor. A unique aspect of TCX is that it uses a single trusted VM per host to coordinate security for all the SC-VMs on that machine. By centralizing services, TCX can, for example, set up a secure channel between containers even if they are on different hosts – to the container it looks like normal networking, but in reality, all traffic is transparently encrypted and authenticated by the TCX layer. The container runtime (Docker/Kubernetes) is extended so that when you launch a container, it is provisioned in an SC-VM with all these protections enabled. Integrity and confidentiality are enforced at multiple levels: the VM's disk image and the container filesystem are measured and encrypted, and all interactions between secure containers go through encrypted tunnels that the host cannot spoof. Essentially, TCX ensures that even in a hostile cloud, containers can only run trusted code and their data stays

safe. The strong hardware-enforced isolation means the attack surface is much smaller than in standard container setups. This architecture achieves protections similar to confidential VMs but maintains a container-centric deployment model.

Performance

The TCX researchers demonstrated that this approach incurs minimal performance overhead. Their implementation (built on Kata Containers with AMD SEV-SNP) showed an average overhead of about 5.8% on CPU-intensive benchmarks (SPEC2017) compared to native execution [12]. Real-world server workloads were also tested: for instance, Nginx web server throughput under TCX was only slightly reduced, and a Redis in-memory database saw modest slowdowns primarily due to the underlying SEV memory encryption cost. The overhead introduced by the TCX layer itself (beyond what SEV encryption alone causes) was very low. This is because TCX still leverages hardware virtualization extensions for speed and optimizes its secure services. Networking overhead in TCX's secure channels was also kept low by using efficient in-kernel encryption for virtual network interfaces. The research concluded that TCX's performance is practical for production, as even high-throughput workloads and multi-container deployments scaled well with TCX's protections in place.

Sustainability

TCX combines the manageability and agility of standard containers with the strong protection guarantees of TEEs, promoting efficient resource utilization. It provides significant performance advantages, reducing the need for excessive computational resources and supports sustainable deployment practices by enabling secure, high-performance computing within containerized environments [12].

2.2.2.5. Conclusions

In summary and depicted in Table 5, each framework has its unique strengths, catering to different workload requirements, infrastructure capabilities, and organizational priorities. CoCo stands out for its hybrid cloud support, enabling secure cloud bursting and efficient resource utilization across environments. Marblerun excels in Kubernetes-native orchestration, providing minimal overhead and seamless integration for enclave-based workloads. Parma offers VM-based isolation with minimal performance overhead, making it ideal for high-performance, confidential computing tasks. TCX combines the agility of standard containers with robust security guarantees, achieving a balance between performance and sustainability. The table summarizing each framework's security, performance and sustainability levels is provided below.

Table 5 Confidential Containers Comparison

Framework	Security	Performance	Sustainability
CoCo	High	Variable	High
MarbleRun	High	Moderate	High
Parma	Very High	High	Moderate
TCX	High	High	High

2.2.3. Pre-deployment Microservice Security by construction

In Deliverable D3.1, we described how the secure-by-design orchestrator (sFORK) manages resources and security at runtime using dependency graphs, CTI-driven selective sharing, hygiene scores and AI-based workload prediction service. The deployment side of the framework includes the instantiation of microservices of a 6G service, which starts from a secure baseline that complements the cybersecurity-based service placement and scheduling. From a DevSecOps perspective, several actions should be applied before deployment to reduce the attack surface and provide reliable security guarantees:

- Declarative modelling of microservice dependencies: Service dependencies must be explicitly described (e.g., Kubernetes YAML or Helm charts). This avoids insecure couplings between services and provides the orchestrator with a complete view of allowed communications and resource bindings.
- Secure inter-cluster channel establishment: Secure communication channels between clusters should be pre-defined at the configuration level. Using CLI-based tooling, developers/operators can set up encrypted tunnels (e.g., via mTLS, IPsec, or Submariner, etc.) that guarantee authentication and confidentiality before services are deployed. This step prevents unprotected connections from being instantiated in production.
- Role-Based Access Control (RBAC) rules setting: Access rights need to be restricted from the start. Role-Based Access Control (RBAC) policies should be defined before deployment, giving each service and operator only the permissions they really need.

These measures establish a baseline of trust and security for microservice deployments that can be embedded into the DevSecOps pipeline. The orchestration layer (sFORK) later operates on already-hardened payloads, where dependencies, communications, and access rights are strictly controlled.

2.2.4. Kubernetes security analysis

2.2.4.1. Main security features

Security in Kubernetes has emerged as a fundamental field of research, as this platform has been established as the standard for container orchestration in cloud-native environments. The Kubernetes architecture integrates multi-layered security mechanisms aimed at protecting both applications and the underlying infrastructure. At the heart of these mechanisms are two key tools: Network Policies and Pod Security Policies (PSPs).

Network Policies are a crucial tool for controlling the flow of data between Pods and services. Through them, administrators can define detailed rules for allowing communication, achieving isolation between applications, and limiting exposure to lateral movement attacks. Study [13] highlights the dual nature of these policies, examining both their performance and security. The results show that eBPF-based solutions, such as Calico and Cilium, offer robust security with negligible performance impact, making Network Policies suitable for use even in resource-constrained environments. Additionally, paper [14] emphasizes that the choice of network infrastructure (overlay vs. underlay) in edge environments significantly affects both the performance and effectiveness of policies, highlighting the need for a balance between security and performance in constrained systems. This allows for a more realistic and comparative evaluation of Network Policies depending on their application environment.

Although PSPs have been deprecated in newer Kubernetes versions, they served as a foundation for applying restrictions to Pods. Through them, it was possible to control the execution of privileged containers, access to host resources, and the use of Linux kernel capabilities. The modern approach is now based on Pod Security Admission (PSA) modes, which incorporate the same logic through admission controllers and security profiles such as seccomp and AppArmor. These tools allow the application of security policies with granular control, enhancing the protection of workloads. As stated in the official Kubernetes documentation, while PSA/PSPs provide significant security, full protection depends on proper implementation and a combination with other measures such as RBAC and Network Policies, which highlights the strengths and limitations of these tools in real-world environments.

The work presented in [15], offers a structured reference framework for Kubernetes security, systematizing best practices into eleven fundamental commandments. This framework covers all aspects of security, from hardening the control plane to the secure management of secrets and the implementation of network restrictions. The article's contribution is pivotal as it bridges the gap between theory and practice, providing a strategic tool for implementing secure containerized applications. Furthermore, this analysis allows for a comparison between different practices, highlighting where they excel and where they may present weaknesses (e.g., the need for automation or a combination of other measures).

The main security features of Kubernetes—Network Policies and Pod Security Policies—form the foundation for building a secure runtime environment. Their evolution, combined with the systematization of knowledge and the application of best practices, shapes the modern state of the art in Kubernetes security. Research in this field continues to evolve, responding to the growing demands of cloud-native and edge computing environments, and highlighting the need for a balance between security, performance, and usability in real-world settings.

2.2.4.2. Kubernetes security strengths

It is important to evaluate how the Kubernetes features collectively contribute to the broader security objectives of a system. A well-established framework for this analysis is the CIA triad (Confidentiality, Integrity, and Availability), which defines the fundamental pillars of information security. Kubernetes supports confidentiality by combining isolation mechanisms with fine-grained access controls to protect sensitive data in shared environments. At the orchestration level, as study [16] highlights, container-based scheduling and namespaces create separation between tenants, reducing the risk of data exposure across workloads. Evaluations of different Container Network Interface (CNI) plugins confirm that strong segmentation can be enforced even in resource-constrained environments, such as edge clusters or IoT gateways, where computational and networking capacity are limited. Studies comparing different plugins show that enforcing network policies does not introduce major penalties in throughput or latency, even when the number of policies scales into the thousands [14]. This demonstrates that confidentiality through network isolation is achievable not only in large cloud data centres but also in smaller, distributed deployments. At the configuration level, confidentiality also depends on secure manifests: empirical studies reveal that many security incidents are due to configuration errors, such as exposed credentials or overly permissive settings [17]. However, Kubernetes provides primitives like Secrets, security contexts, and privilege restrictions that, when properly applied, help protect confidential information. Complementing these features, there can be found best-practice guidelines such as enforcing role-based access control (RBAC) and applying network and Pod security policies, that further strengthen Kubernetes security, as they ensure that both data access and communication paths are tightly controlled [15].

Kubernetes provides integrity maintenance via multiple complementary mechanisms that ensure workloads and communications remain consistent with intended policies. On the orchestration side, Kubernetes uses policy-driven scheduling, which means that the system places workloads on nodes according to clear rules set by administrators. This helps keep the cluster running consistently and avoids situations where a workload might be started in the wrong place or with the wrong resources [14]. This consistency protects against accidental misplacements and helps preserve the correctness of operations even in large, dynamic environments. Integrity is also protected at the network level through access controls such as Role-Based Access Control (RBAC), which ensure that only authorized users or services can change important settings [17], [15].

Integrity is reinforced by network policies, which apply the principle of least privilege to communication. This means allowing Pods to connect only to explicitly permitted peers, therefore reducing the chance of malicious tampering or data injection between services. In addition, network policies improve integrity by limiting how Pods can talk to each other, so that they only exchange the data needed for their tasks and are less exposed to malicious or accidental tampering [16]. Importantly, evaluations show that these policies can be scaled to thousands of rules with negligible performance impact, meaning they preserve secure communication without weakening the system's responsiveness. Studies of real Kubernetes configurations show that while errors are common, features like security contexts and privilege restrictions give administrators tools to keep workloads in a safe and correct state when they are applied properly.

Complementing the above, Kubernetes contributes to availability by ensuring that applications remain operational despite failures, resource shortage, or the addition of security controls. Scheduling policies and replication strategies keep workloads running smoothly by redistributing them when nodes or resources become unavailable, which helps ensure that the service remains accessible and usable by authorized users whenever it is needed [16]. Automatic scaling mechanisms also adapt resource use to match changing demand, reducing the risk of service interruptions during peak loads. In addition, study [13] demonstrates that even with thousands of network policies in place, latency and throughput performance remains stable, showing that security enforcement does not come at the cost of system responsiveness.

2.2.4.3. Kubernetes security weaknesses

While Kubernetes offers robust security features, the literature identifies several weaknesses and recurring challenges that can undermine its overall security posture. A key issue arises from insecure or improper configurations, such as weak authentication, poorly defined network policies, and excessive permissions. Sometimes administrators set up Kubernetes with weak or missing security settings. Examples of this include giving users, Pods, or services more permissions than they really need, exposed credentials issues, manifesting by leaving passwords, API keys, or tokens in plain text inside configuration files, and default root access, which means letting containers run as root inside Pods, which makes it easier for attackers to break into the host. These misconfigurations increase the risk of unauthorized access, data breaches, and privilege escalation, leaving clusters open to attack. Closely related is the problem of root access and privilege escalation, where containers often run with elevated privileges by default, enabling malicious actors to break out of their containers and compromise the underlying host system if not properly restricted [19]. The empirical study of Kubernetes manifests [17] finds widespread security misconfigurations, such as missing security context, excessive privileges and hard-coded credentials. These significantly increase the attack surface and confirm that misconfiguration is one of the most critical and common weaknesses.

Another major weakness is the limited strength of namespace isolation. Although namespaces are designed to separate workloads logically, they do not provide strong security isolation. Without additional controls such as strict network segmentation or Pod security restrictions, attackers who gain access to one namespace may be able to move laterally across the cluster. The paper [19] also highlights that improper access control, including weakly configured role-based access control (RBAC), can allow users or services to perform unauthorized actions, undermining both confidentiality and integrity of workloads.

In addition, the reliance on container images introduces risks tied to image vulnerabilities and supply chain security. If images are not scanned or hardened, they may carry exploitable software flaws into the cluster. If combined with inadequate runtime security, this creates opportunities for attackers to escalate privileges or inject malicious code. As multiple studies show, vulnerability management and patching practices are often inconsistent. Delays in applying patches to Kubernetes components and container images expose clusters to known exploits, while insufficient monitoring and logging reduce visibility into ongoing threats. Paper [15] concludes that while security practices are well documented, many organizations fail to apply them consistently. This gap between available practices and their actual adoption remains a central security challenge.

2.2.5. O-RAN xAPP security

In O-RAN project, xAPP are containerized payloads onboarded the near Real Time RIC. O-RAN security Working Group 11 (WG11) has been very active in defining the security exigencies related to O-RAN open architecture, as its desired openness generates novel security threats. As the architecture includes several API-defined interfaces between several units, the security of these APIs is the main concern. As O-RAN enables operators or tech vendors proprietary software workloads to be on-boarded in the Near and Non -Real Time RICs, hence sharing local resources, these workloads shall be verified before being on-boarded and executed.

WG11 has produced several documents to establish how these workloads can be authenticated and remotely attested. In this sub section, we take a deep dive to assess the maturity of the specifications or recommendations, assessing in which directions NATWORK can elevate security, notably leveraging one or several PDSCM. As part of NATWORK, we consider IS-RD's Liquid xAPP as a workload to protect. For clarity, the following survey does not claim to be fully comprehensive in terms of analysis of WG11 recommendations document. The security aspect is treated in different areas and angles (e.g., O-cloud, risks assessment). However, our survey highlights what shall be retained in view of defining NATWORK's offering.

Our survey delivers the following order of precedence in WG11 documentation:

- O-RAN ALLIANCE TS, “O-RAN Near-RT RIC Architecture [20]
- O-RAN Security Requirements and Controls Specifications [21]
- O-RAN Study on Security for Service Management and Orchestration (SMO) [22]
- O-RAN Study on Security for O-Cloud [23]
- O-RAN Study on Zero Trust Architecture for O-RAN 2.0 [24]
- O-RAN xAPP SDK [25]
- O-RAN Study on Security for Near Real Time RIC and xApps 5.0 [26]

2.2.5.1. xAPP authentication process

Two stage xAPP registration (i.e., SMO, near RT RIC): MOI generation

Figure 1 reflects O-RAN two stage xAPP registration sequence. The xAPP signature produced by the xAPP provider is an asymmetric encryption of the xAPP manifest. The manifest contains the digest of xAPP package and security policy elements.

The service provider checks the signature delivered by the provider, verifying its provenance (i.e., public key delivered with signature). From this step, the service provider signs again the xAPP signature and delivers it to the SMO.

The SMO, after checking the identity of the service provider (i.e., public key delivered with the signature), will have access to the manifest. SMO also extracts its data structures and produces an integrity verification of the package, using that digest. Once this integrity verification is made, all fields are supposed to be valid, and SMO produces an xAPP metadata store in its catalogue.

The SMO will then forward these elements to the near RT RIC and instruct the generation, signing and catalogue storage of the xAPP managed object instance (MOI). The MOI will then be used for all verifications by the near real time RIC, validating the xAPP authenticity and compliance with security requirements.

The MOI is the composite metadata of the xAPP, containing references to the xApp identity (e.g., name, version), packing elements (e.g., list of containers, command used to run the container), controls (i.e., internal to the xAPP), metrics, certificates, the image digest, deployment policies, network policies, security policies and many other xAPP descriptive fields.

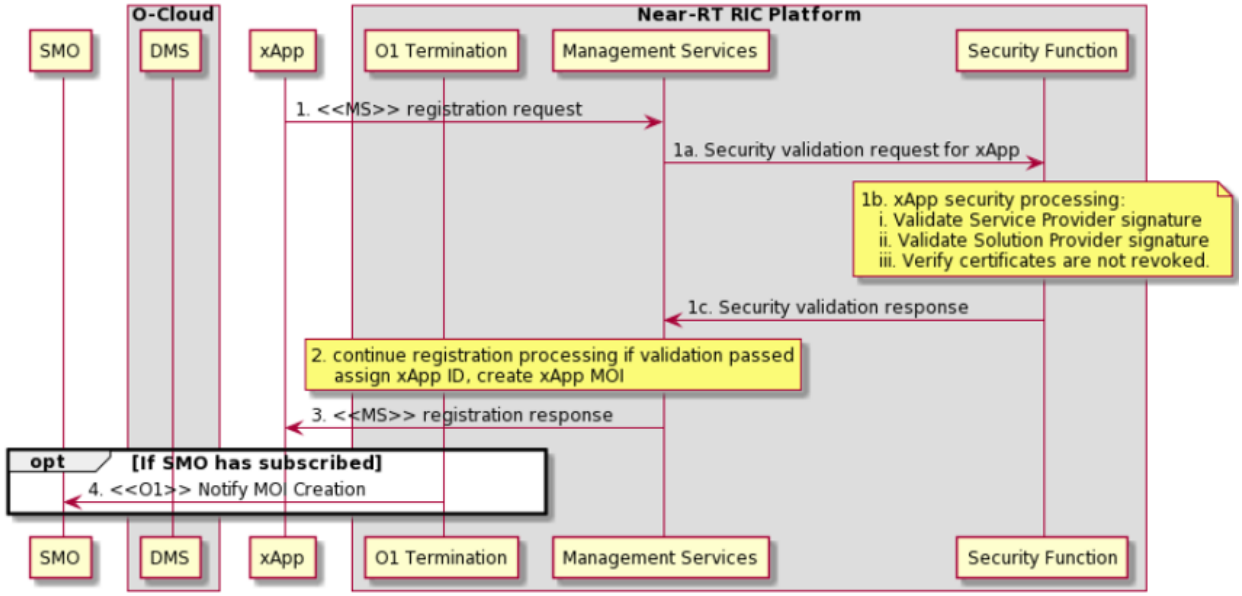

Figure 1. General xAPP authentication in near RT-RIC

xAPP onboarding verification workflow

The SMO initiates the onboarding workflow and triggers the near RT RIC. For that, the SMO provisions the xAPP metadata, enabling the reconciliation with the xAPP MOI.

The near RT RIC checks its operational status and validates the onboarding, checking that the deployment (i.e., affinity rule), networking and security exigencies as stated in the xAPP MOI, then transfers the onboarding status (i.e., possible, not possible) to the SMO. All authentication and authorization credentials used by the near RT RIC APIs are stored in the near RT RIC.

The near RT RIC does not produce an xAPP digest verification, which is done at the SMO level. If the xAPP deployment in the near RT RIC is possible, the SMO triggers for the deployment of the xAPP containers to the near RT RIC (where API authorization and authentication tokens are made ready).

Takeaways and identified security gaps

- The above-described workflow relates to the authentication process initiated with a registration in a catalogue and the verification of attributes in correspondence with the catalogue-stored artefact during on-boarding.
- The process is multi-stakeholder and complex. The xAPP integrity verification is done by comparing a signed digest stored at the SMO.
- The SMO acts as the security guard, making the peripheral checks before use. Once the xAPP integrity has been checked by the SMO, it is no longer checked thereafter. The SMO

is trusted to deliver correct onboarding triggers. A compromise SMO can produce corrupted xAPP onboarding demands luring the near RT-RIC.

- The near RTRIC consumes both and then validates the xAPP MOI, creating a trust anchor issue. A corrupted near RT RIC can vet unauthorized xAPP being trusted.
- The threat model considered by O-RAN excludes a corrupted near RT-RIC and a corrupted SMO.

2.2.5.2. WG11 statement on xAPP remote attestation

In O-RAN Security Requirements and Controls Specifications 11.0 [21], WG11 recommends a conceptually defined remote attestation service (AS) for providing additional benefits besides verifying the O-Cloud platform integrity by Chain of Trust. WG11 stipulates that the remote AS should be extended to include O-RAN Applications integrity as depicted in Figure 2.

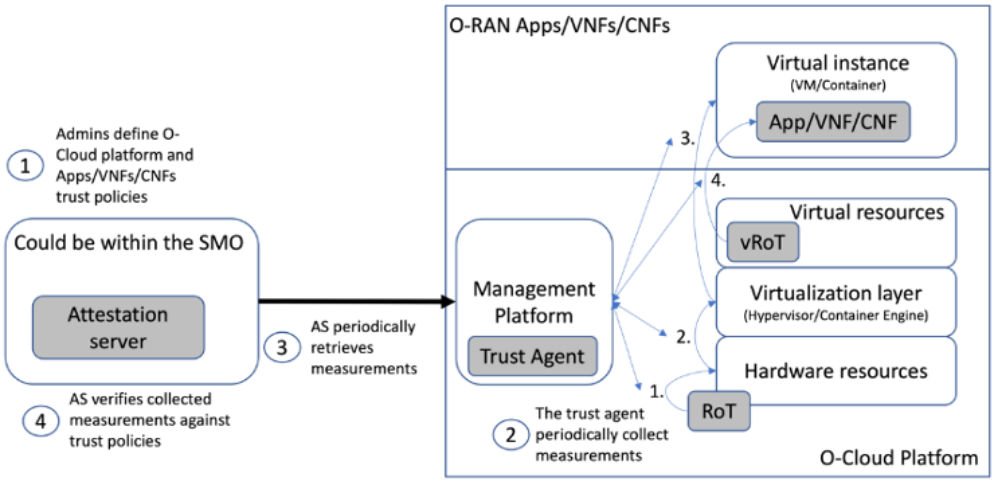

Figure 2. O-RAN theoretical full stack remote attestation framework

WG11 details the remote attestation of the O-CLOUD platform. The remote attestation is multi layered and full-stack, covering O-cloud hardware root of trust, the hardware resources, the virtualization layer, the virtual root of trust and finally the on-boarded applications (e.g., xAPP). Although this full stack remote attestation does not specify a TPM leverage but is inspired from it, as notions of root of trust (i.e., RoT in the picture) and virtual root of trust (i.e., vRoT) suggest it.

2.2.5.3. xAPP runtime integrity verification

xAPP tampering is identified as a security Key Issue (aka Op-2) by WG11, stressing the possibility for an attacker to “Negatively affect the O-RAN platform”. WG11 also states that “Detecting and preventing threats during application runtime is still an on-going research problem”. WG11 pinpoints three research issues of:

- Trust of the integrity monitoring solution which can be itself tweaked and corrupted,

- Verifying the integrity of a running application requires knowledge of the known good states of the application and what is not. Changing application data within memory may not necessarily indicate a tampered application, especially considering AI/ML application
- Performance impact of integrity monitoring for control loop execution times for xApps and rApps. The Near-RT RIC requires control loops from 10 milliseconds to 1s, and the non-RT RIC control loops are specified for more than 1 second. These control loop execution times must be considered, especially for xApps, when factoring in potential negative effects on performance by monitoring for integrity on running O-RAN applications.

These three elements are indeed to be considered when designing a runtime integrity verification.

2.2.5.4. xAPP SBOM management

WG11 stresses that one noticeable element is that the requirement or recommendation for xAPP SBOM processing does not translate into a “related security control”. WG11 recalls that SBOM verification is an activity practiced during development but checks of correct dependencies during runtime are hardly worked out.

2.2.5.5. Takeaways

Our survey has brought us the following vision and lessons learnt:

On xApp authentication

- WG11 has streamlined the design and workflow of xAPP authentication verification, articulated by the SMO (using the digest and supplier signature) first before the near RT RIC (using a locally produced security content-rich xAPP MOI grasping all security exigencies).
- The SMO is the central ledger which detains the catalogue of verified xAPPs and triggers the operations of the near RT-RIC at onboarding request.
- The maturity of the defined scheme is high. The accuracy of the different description reflects a strong understanding of all operational and security considerations for xAPP onboarding.
- The security relevance of this scheme imposes that both SMO and the near RT RIC are integrated.

On xApp remote attestation

- A general full stack remote attestation framework including all layers from the hardware anchor to the xAPP.

- No TPM requirement on the near RT-RIC is stated, although the terminology used evokes it.
- The schema defines the SMO as the central verification utility, comparing all measurements of different types and including xAPP digest.
- As mentioned for authentication, the SMO and the trust agent must be integrated.
- The maturity level of the recommendations can be assessed as preliminary and conceptual.

On xAPP runtime integrity verification

- WG 11 stresses the requirement for runtime integrity verification, breaking the threat model as employed in authentication (i.e., corrupted SMO, corrupted near RT-RIC)
- WG 11 positions runtime integrity as a research challenge, notably stressing the impact of performance induced by periodic integrity verification. In the NATWORK project, this can be viewed as a security gap to cover.

On xAPP SBOM management

- WG 11 stresses the requirement for continuous verification of used dependencies at runtime.

2.3. Binary pre-deployment hardening techniques

2.3.1. General

Native payloads (i.e., executables, libraries) have been PDSCM hardened against various threats for longbeen directly exposed to attackers, as their bare metal deployment limits the system protection virtualization or containerization can bring. For attackers who have acquired the code file, static analysis discovers the code and data structures, enabling both reverse engineering and tampering. For attackers with administrative rights on the platform that runs the payload, dynamic analysis is without limit, with the support of tracers, debuggers and decompilers, capable of mapping the memory allocated to the running process.

Against CIA attacks, a major shift came with the emergence of TEE from 2015 onwards. A 360° high-level survey of PDSCM techniques is given below, according to the different payload threat models.

2.3.2. Confidentiality preservation

The SotA integrates the following PDSCM techniques. In the last three decades, academic surveys [27][28][29][30] have covered the promises, efficiency and performance impact of the different

techniques employed in code security.

2.3.2.1. Code encryption

Encrypting the code section of the ELF formatted .exe or .so (i.e., for library) files, prevents static analysis but the protection breaks as soon as the code start executing (i.e., the encrypted code section is decrypted before execution). Code encryption protection depends on the encryption algorithm. AES 256 encryption is generally practiced and brings a plain security assurance (against static analysis)

Code encryption has no impact on performance as the decrypted code is identical to the original, and a short latency at start is caused by the decryption primitive plugged at the code entry point. Can be bridged or interfaced with a selective provisioning of the decryption key to a restricted perimeter of platforms. This does not prevent the integral collection of the decrypted code on a legit platform, which is migratable to any other platform.

Runtime code reconstruction uses basic code encryption on restricted snippet with a timely description just before the snippet execution. This method reduces the exposure window of the code snippet to a few CPU clocks around the snippet execution. Code encryption is a direct PDSCM as the code shall be modified before release (i.e., encrypted code section or snippets).

2.3.2.2. Code obfuscation

With the objective of elevating the level of efforts required to produce a reverse engineering through a dynamic analysis, code obfuscation over complexify the code structure. Obfuscation brings a relative security assurance but a hardly scaled resilience. Used in the video game industry, it aims at securing the publisher sales during first days after a game release date. Code obfuscation is generally associated with hidden anti-tampering traps impeding the progression of the attackers. Hidden traps are added snippets using elementary original code memory cells.

A large variety of code obfuscation techniques have been designed and used over time (e.g., control flow flattening, code virtualization, instruction substitution, opaque predicate/junk code insertion, data structure obfuscation, symbol and string encryption) [31][32][33]. Each of them brings in one specific context (i.e., when applied to one specific executable) a different efficiency and performance impact. The performance impact can be significant (i.e., in the range of 100% for complex code virtualization or control flow flattening), restricting their usage to security-sensitive code and skilled integration teams. Obfuscation requires a specific set up activity on each payload. AI-based tools simplify this workflow [34][35]. AI-based deobfuscation tools [36][37] de-obfuscate code, recognizing obfuscated code patterns and removing them. Code obfuscation is a direct PDSCM as the code shall be modified through obfuscation patterns.

2.3.2.3. Trusted Execution Environment

TEE is a processor vendor-supported technique, arising first in 2015 (i.e., with ARM's TrustZone), producing on-the-fly (encrypted) memory page decryption and integrity verification prior use. The pages are encrypted in DRAM, decrypted when used, and re-encrypted thereafter. The memory in the "enclave" or TEE-protected area (used by the processor) cannot be accessed for read or write by any external process. The Trusted Computing Basis (TCB) aggregates all memory pages protected therein.

Intel's SGX original design (i.e., 2019) restricted the TCB size to the bare minimum; notably, a vulnerable TCB can be equally exploited and totally covertly. Vulnerability scanners cannot access the TCB, hence are inoperant with a vulnerable TCB (aka evil TCB threat model). SGX is a shelter for security-sensitive executables and routines. No system calls are permitted from the TCB, generally implying code modifications. The new generation of VM-based ultra-large TCB (Intel's TDX, AMD's SEV-NP, and ARM's CCA) has emerged since 2020, following AMD's first SEV release for the cloud market. These TEEs drastically simplify the DevSecOps as untouched VM onboards the TCB, but they equally drastically augment the malicious TCB risk. The TEE impact on performance fluctuates from an average of 10-30% for SGX to an average of 5-10% for VM-based TEE. Placement inside SGX was a direct PDSCM, as code shall be modified and prepared to onboard SGX. With VM-based TEEs, PDSCM is an indirect action, consisting in selecting prior deployment the TEE equipped platforms.

PDSCM consist in either modifying the payload (i.e., by encryption or obfuscation) or ensuring its execution in a TEE.

2.3.3. Integrity preservation

The SotA integrates several techniques employed at various level (i.e., system level, application level through PDSCM).

System-level security: Software tampering protection is an epic battle, that one can date back to the 40s and 50s, at which Van Neumann's CPU architecture was preferred against Harvard's, which merges both data and code in the same memory space. As data shall be writable, there is native protection to prevent writing on code residing aside. It took a long time for operating systems to adopt *Write xor Execute (WxorE)* principle (i.e., in the 1990s onward) with declarative flags preventing code tampering. For attackers with administrative rights on the platform, *WxorE* flags can be removed and the code modified. Henceforth, *WxorE* protection does not prevent memory introspection and tampering.

2.3.3.1. Application-level security by PDSCM

The SotA contains the following integrity preservation measures:

- Authentication provides the recipient with the assurance that the static code file has not been tampered with after it was signed by the developer or operator and before it was onboarded on a platform. Both payload's provenance and integrity are checked together with the payload signature and the developer's public key delivered in the payload's manifest. For that, the code file hashing, then asymmetric encrypted deliver these two assurances.
- Remote attestation provides the payload operators with the assurance that their deployed payloads are integrated where they are deployed, leveraging similar basic techniques. Remote attestation needs a Prover where the code executes, producing the quote, and a remote verifier comparing the quote with a reference measurement.
- Granular and imbricated integrity verification: Software-based techniques have been designed to create a lattice of imbricated elementary memory state checks, which deliver probabilistic protection. The density of these buried traps, executed on the fly during execution, is correlated with the induced performance penalty.
- Trusted Execution Environment brings a de-facto integrity preservation to the TCB, notably through on-the-fly integrity check processed at memory page loads.

PDSCM consist in application-level security as stated above.

2.3.4. Availability preservation

For software, availability exclusively relates to the availability of the needed resources allocated by the execution environment. Techniques integrate system-level techniques, user-triggered resource reallocation, and application performance monitoring. The following exclusively relates to CPU sharing, while shared memory process allocation techniques similarly impact software availability. For simplicity, memory allocation is not listed below.

2.3.4.1. System-level native CPU allocation (to processes)

Natively, CPU regulates the resource allocation to the different processes in operations with time-sharing, priority scheduling or affinity pinning to distribute the processes execution over several cores. Similarly, VM hypervisors regulate the resource allocation between VMs, based on credit-based, fair scheduling, proportional share, and CPU quotas.

2.3.4.2. User-level arbitrary resource allocation

Systems deliver users sufficient administrative rights to adjust the resource allocated to a process (e.g., Linux's *cgroup*, *nice/renice* scheduling priority, *taskset* to bind processes and containers to

cores). Hypervisor administrators permit permissioned users to adjust the CPU resource allocation (e.g., allocation of vCPUs assigned to a VM, adjust CPU shares, limits, reservations (e.g., VMware, vSphere, Xen, KVM), and finally by pinning vCPUs to physical CPUs.

2.3.4.3. Performance monitoring

CPU performance monitoring is offered by many different tools natively including in the operating system or integrated applications

Operating system commands

- At CPU level (e.g., Linux commands *top, htop, uptime, ps*).
- At CPU core level (e.g., Linux commands *mpstat, pidstat, perf*)

Integrated performance application

A large set of Monitoring applications belongs in the SotA:

- **Nagios** [38], a widely used infrastructure monitoring (with plugins for CPU usage).
- **Zabbix** [39], a popular open-source monitoring system with CPU usage metrics.
- **Prometheus + Grafana** [40], a modern full stack for time-series CPU metrics and dashboards.
- **Datadog** [41], a cloud-based monitoring with strong CPU profiling and alerts.
- **New Relic** [42], a SaaS monitoring for applications and infrastructure, including CPU usage.

Self-contained performance monitoring

In [43], a disruptive approach intends to assess the allocated resource level by the payload itself (through payload rewriting) or better through an agent (i.e., sidecar container). PDSCMs consist of setting up monitoring applications in the execution environment, producing operating system commands, or modifying the code for self-probing.

2.3.4.4. IPR security

Several techniques are employed to prevent illicit use of software, infringing the licence rights. Techniques can be summarized as below:

Digital Right Management (DRM)

DRM techniques are present on the market in the following patterns:

- *Software activation* consists, through a remote server licence activation service, to (i) collect machine specific invariants, (ii) process accordingly an activation token by an activation server, once pending user right to install is verified, install the machine invariant specific activation token on the permissioned platform. At software start, the

machine invariants are collected a second time and reconcile with the activation token through a test, triggering the software start if positive.

- *Dongle binding* consists of checking the presence of a non-duplicable dongle on the machine, at the software start.

Both above techniques imply modification of the code, notably at its entry point to insert the DRM routine.

- *Machine binding* consists of checking the presence of a digital blob in the execution environment to start the software. Several flavours of blob anti-duplication and migration security are employed, preventing easy localization and copy. Ultimate security is achieved when the blob is resident in a TEE.
- Watermarking has no semantic and would not stop the software execution, but it is used to track each user licence, separately. In case of illicit replication, it is used to trace back the copy. In practice, watermarks come with a change on the code package.

PDSCM consist in modifying the code or placing a watermark inside the code package.

2.3.4.5. Vulnerability preservation

Vulnerabilities have several origins such as buffer overflow (i.e., exploitation permitted by *Van Neumann* architecture), memory management errors (e.g., use-after-free), Input validation and injection (e.g., SQL injection), race conditions, and higher-level origins (e.g., wrong authentication, misconfiguration, and cryptographic weaknesses).
PDSCM consists of identifying the vulnerability and curating the code (i.e., reprogramming as no curation exists at executable level). Vulnerability detection, however, can be practiced at all steps of the executable file generation (i.e., at programming, on binary executables).

2.4. Web Assembly security

2.4.1. WASM technology history and key design attributes

As stated in D2.1, WASM was defined by Internet browser giants [44], as a substitute to JavaScript highly portable interpreted language. The working group objective was to raise concurrently JavaScript's security, performance and sustainability, in alignment with NATWORK's vision. For that, WASM core asset is its lower-level instruction set, closer to machine atomic operations, faster to interpret and supposedly harder to reverse. Since its standardisation [45] in 2019, WASM has attracted several CPU intensive industries, domains and technologies (e.g., blockchains, FaaS, crypto mining, gaming engines). WASM module interpretation is faster and more sustainable than any containerization solutions and its inherited portability ideal for cloud

continuum payload migration in networking. The exact same WASM can be executed at very different platforms, equipped with their platform-specific WASM runtime (i.e., interpreter). Notably, by contrast to containers, WASM technology expands the continuum up to the User Equipment. Thanks to the high traction and work on WASM development activity, WASM compilers are today totally polyglot, able to compile programs written in any existing languages. Hereto, for networking, WASM is a high potential contender for instant and highly migratable 6G services over the continuum.

2.4.2. WASM security analysis

Table 6 provides a rapid view on the key pros and cons of WASM security, deriving from our own study and security surveys [46][47]. We have excluded the generally-cited low-level bytecode instruction set, supposedly making reverse engineering harder, a relative security guarantee (i.e., seasoned reversers are efficient at assembly level, a lower level than WASM instruction set) and AI-enhanced reverse engineering tools will bring instant, near-complete WASM module decompilation in several programming languages outputs [48][49][50].

Table 6. Strengths and weaknesses of WASM security

Strengths	Weaknesses
Sandbox execution environment, where: -Payloads can run but cannot access other process memory space. -Native enforcement of *Write xor Execute*, making code tampering impossible through data channel, without local memory introspection. This strong security assurance applies to native compiled WASM payloads (i.e., through JIT or AOT compilation) and the WASM runtime itself.	-**Code tampering**: Through remotely-spawn privilege escalation attacks or by direct memory introspection, the memory states can be modified. *Write xor Execute* memory protection cannot be applied WASM bytecodes (i.e., writable data structures).
-**Type-control** by WASM runtime producing buffer bound checking, making buffer overflow attack unexploitable.	-As an inheritance or common taken attack path to JavaScript, **JIT spraying** technique tweaks the JIT compilation to generate malicious code snippets activable as backdoors.

2.4.3. WASM module integrity.

In a general perspective, two techniques deal with workload integrity. *Trusted Execution Environments (TEE)* decrypt on-the-fly encrypted swapped memory pages, restricting access to the DRAM-stored ephemeral decrypted pages. Additional integrity checks are produced at each page load, on-the-fly. Confidentiality and Integrity are delivered de facto for any resident workloads. Authentication and remote attestation, produce a verification of integrity using

hashing and a comparison with a signed reference measurement (i.e., signed certificate attached with the workload). These processes ascertain both the provenance and integrity of the workload. While authentication operates at the workload location only, remote attestation operates on both ends (i.e., where the workload is deployed, at the remote location).

2.4.3.1. TEE-delivered integrity

By placing the WASM runtime into a TEE, both WASM runtime and module are preserved of confidentiality and integrity attacks. As detailed in [51], using TEE shall be considered and restricted to security-sensitive workloads as it implies higher memory and CPU consumption, leads to novel threat models (e.g., DoS by Raw hammering, evil TCB) and frustrates workload portability (i.e., heterogeneity). The workload-specific performance impact induced by TEE placement also varies with the type of TEE. VM-based optimized TEE (i.e., Intel's TDX, AMD's SEV and ARM's CCA) performance penalty range is generally below 10%. Executable-based TEE (i.e., ARM's TrustZone, Intel's SGX) impact is higher and bounded below 30%. In NATWORK, our vision is to consider TEE for these specific security sensitive workloads (e.g., network probe), which deployment can be managed with care and the extra resource consumption measured to be acceptable.

2.4.3.2. Distinction between authentication and remote attestation

Authentication solutions ascertain the provenance of the workload, leveraging asymmetric encryption and secondly the integrity or genuineness of the workload, leveraging a hashing on the workload artefact data. Authentication is a security service beneficial to the recipient of a workload (i.e., an infrastructure operator), taking for granted it is trustworthy to make this test. Remote attestation does not depend on the recipient's trustworthiness and is a security assurance delivered to the workload operator (i.e., service operator). For the service operator, there is a need to check that what is deployed remotely corresponds to what is expected (i.e., identity check). The verification is worked out remotely leveraging components on both side (a "prover" at the workload location, a "verifier" at another position).

2.4.3.3. SotA's Integrity techniques

Table 7 shows the current state of the art related to WASM identity and integrity verification, regrouping the usual techniques of WASM module authentication at onboarding, remote attestation at onboarding, TEE-based remote attestation, and WASM module runtime integrity.Table 7. WASM authentication and remote attestation techniques identifies, for each technique, the verified artefacts attributes, some noticeable operational considerations and how the current state of the corresponding SotA.

Table 7. WASM authentication and remote attestation techniques

Integrity technique	Location of the Prove and Verification routines	Artefact verified attributes	Other considerations	State of the art
Authentication	Both at payload's execution site	-Origin (i.e., Public key's owner) -Integrity (i.e., vs the signature generation time, prior deployment)	-No payload identity information collected -No signature management required	-Browsers ' SRI, -DIY, -Wasm-sign, -WABT, LUCET
Remote attestation	-Prove routine at payload execution site -Verify routine is remote	-Identity (i.e., artefact's ident) -Origin (i.e., by database signature public key) -Integrity (i.e., vs the signature generation time, prior deployment)	-Requires signature management (i.e., pristine and trustworthy signature database at the verifier site)	In NATWORK, by IMEC
TEE-based remote attestation (Intel'SGX sample)	3rd-party remote utility (i.e., Intel attestation service)	-SGX TEE genuiness Payload integrity (i.e., vs the SGX enclave generation time at build stage) -Implicit attributes of confidentiality and integrity assurances for the payload during runtime.	-Verified Attributes vary with TEE types (e.g., AMD's SEV-SNP) -No enclave identity delivered	-WaTZ, -Twine, -Enarx, -RA-WEB
Runtime integrity verification	-Prove routine is at execution site. -Verify routine can be either at execution site or remotely	Loaded memory pages footprint integrity (vs. a reference measurement made at first run or prior deployment)	-The memory pages footprint can also be used for remote attestation. Both remote attestation and runtime integrity checks can use the exact same maerial	**None**

Integrity technique	Location of the Prove and Verification routines	Artefact verified attributes	Other considerations	State of the art
			-Continuous attestation repeats periodically, integrity verification.	

2.4.3.4. WASM authentication techniques

Two different types of authentication techniques must be distinguished. The first being embedded by web browsers, and the second by runtimes.

Browser SubResource authentication

The .wasm file integrity verification is practiced through *Subresource* (SRI), an in-browser functionality, w/o checking the origin and taking for granted that the source is trusted. Typically, source trust can be derived by a variety of complementary techniques (e.g., OAuth, mutual TLS, session tokens) used to authenticate the source in the http/https handshake.

Runtime authentication

For WASM runtimes (e.g., WASMTIME, WASMER), programming a DIY (Do It Yourself) protocol leveraging a classical hashing routine is always possible as depicted in Figure 3. Moreover, several tools combine source authentication and payload integrity verification, leveraging a signed hash (i.e., signature) (e.g., Wasm-Sig, WABT, Lucet).

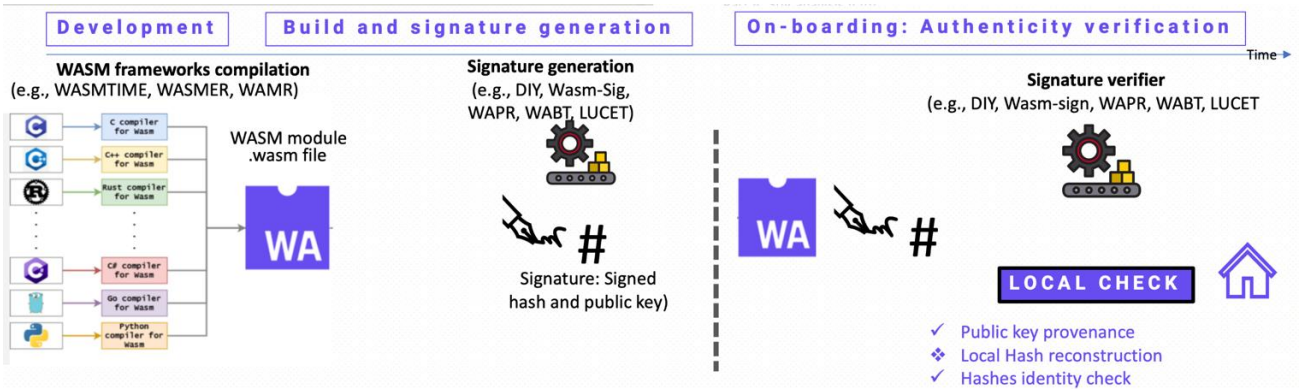

Figure 3: WASM runtime-orchestrated module authentication

2.4.3.5. Remote attestation

To the best of our knowledge, there is no existing remote attestation framework for WASM modules. In networking, WASM modules will be Network Functions and thus ETSI NFV security working group recommendations fully apply [52]. Module authentication suffices when payloads are executed in browsers, while it is not sufficient for networking, where remote attestation is required for service operators.

Theoretical WASM remote attestation implementation

Although we have not found any existing WASM module remote attestation framework, we believe that the first implementations will come soon. A WASM remote attestation framework can be constructed with the support of existing authentication tools, without major technical difficulties. For that, the authentication verifier can be turned into a prover, forwarding signed quotes to a distant verifier, which checks the validity of the prover's public key and the signature by comparison with the same payload's signature stored in its database. This theoretical implementation is illustrated in Figure 4.

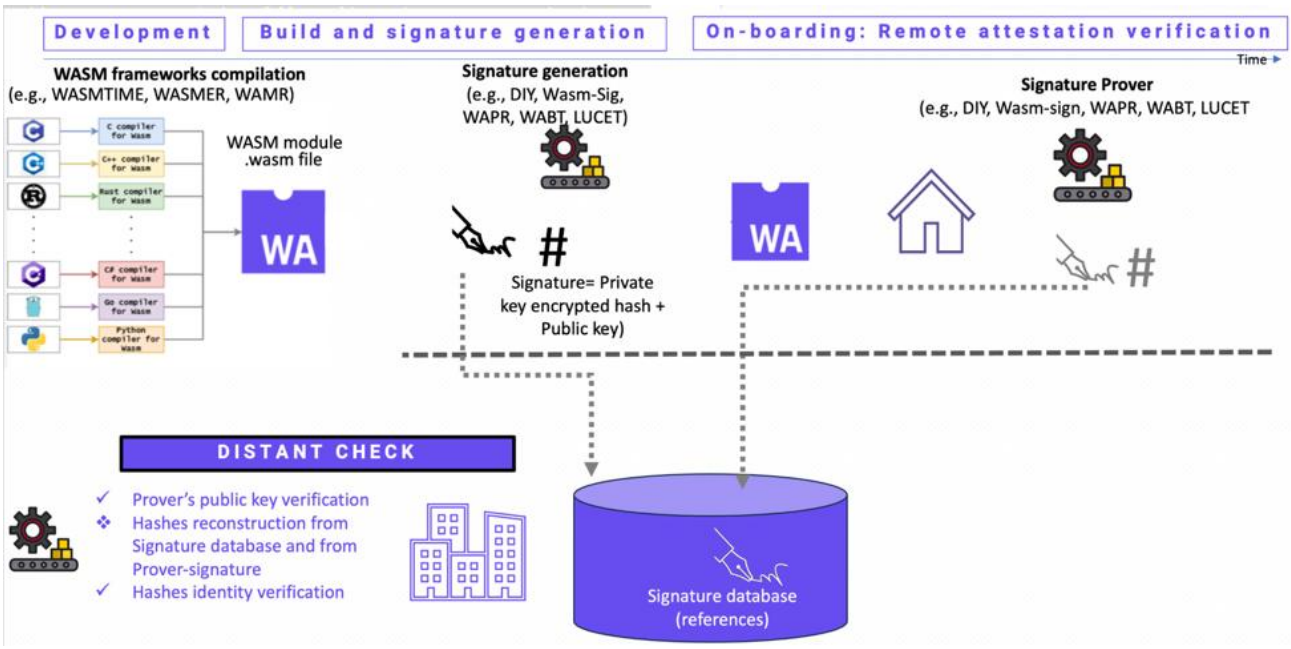

Figure 4: Theoretical WASM remote attestation framework

2.4.4. SotA Takeaways

We have reached the following conclusions:

- WASM authentication is a usual and well-established technique, delivering a security attribute for the workload recipient (i.e., the cloud infra operator) while remote attestation is required for service operators.

- All technical bricks used for authentication can be assembled to construct a remote attestation of the WASM module when they are onboarded.
- After the remote attestation onboarding test, the WASM module runtime integrity is lacking, and there is a security gap to fill.
- TEE remote attestation brings the assurance that the WASM module executes in a sheltered execution environment where confidentiality and integrity are near certain, but at the cost of deployment constraints, performance degradation, and excessive memory consumption. According to the TEE type and, more specifically, if the TEE is process- or VM-centric, the remote attestation respectively attests to whether this specific process is inside or not. VM-based remote attestation checks the complete VM in its initial state, without insight and accuracy down to the different processes inside the VM.

3. NATWORK's PDSCMs on containerized payloads

3.1. Kubernetes pre-deployment progress

Kubernetes provides several native mechanisms for security, including **Network Policies**, **Role-Based Access Control (RBAC)**, and **Pod Security Admission**. However, studies have shown that security incidents frequently arise not from the absence of these features but from their misconfiguration or misuse. Common problems include exposed credentials, overly broad RBAC assignments, containers running with elevated privileges, and network policies that are either missing or too permissive. These weaknesses are particularly dangerous in distributed environments such as edge clusters and multi-tenant deployments, where the attack surface is naturally wider. Alongside the identified weaknesses presented in Section 2.2.4.3, NATWORK addresses these challenges by emphasizing **pre-deployment hardening**, ensuring that workloads are validated and secured before they reach the production environment.

To achieve this, we propose the integration of CERTH's AI-based Intrusion Detection System (AI-IDS) into the Kubernetes pre-deployment pipeline. The AI-IDS would act as a policy gatekeeper within the CI/CD process and Admission Controllers, performing in-depth analysis of deployment manifests and configuration files. For example, in Figure 5: Pod manifest with root privileges and no resource limits, **Pod manifests and Helm charts** can be scanned to detect dangerous practices such as privilege escalation (e.g., containers defined with `runAsUser: 0`, running as root), missing resource limits (Pods deployed without `resources.limits`, able to consume unlimited CPU/memory), or the exposure of sensitive credentials (passwords hardcoded in environment variables instead of referencing Kubernetes Secrets).

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: insecure-pod
5  spec:
6    containers:
7    - name: app
8      image: myregistry/app:latest
9      securityContext:
10       runAsUser: 0        # <= Runs as root (privilege escalation risk)
11     env:
12       - name: DB_PASSWORD # <= Hardcoded secret
13         value: "supersecret123"
14     resources:            # <= Missing limits -> no control of CPU/Memory usage
15       requests:
16         cpu: "100m"
```

Figure 5: Pod manifest with root privileges and no resource limits

Another critical domain of analysis involves **Network Policies and RBAC rules**. Here, insecure configurations often allow unrestricted traffic or excessive permissions. For instance in Figure 6:

NetworkPolicy allowing unrestricted ingress traffic, a Network Policy with `ingress: { from: [] }` effectively permits all traffic, enabling lateral movement between Pods. Similarly, RBAC bindings that assign `cluster-admin` rights to a microservice service account provide unnecessary and dangerous access to the entire cluster. Even a misconfigured role granting write access to namespaces intended to be read-only can compromise cluster integrity.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
  - from: []  # <= Allows all sources to connect -> lateral movement risk

```

Figure 6: NetworkPolicy allowing unrestricted ingress traffic

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: insecure-binding
subjects:
- kind: ServiceAccount
  name: microservice-sa
  namespace: default
roleRef:
  kind: ClusterRole
  name: cluster-admin  # <= Grants full cluster-admin rights to a single service
  apiGroup: rbac.authorization.k8s.io

```

Figure 7: RBAC binding granting cluster-admin to a service account

Finally, as shown in Figure 7: RBAC binding granting cluster-admin to a service account**external interfaces and IP bindings** must be reviewed to prevent unintentional exposure of services to the internet. Examples as in Figure 8: Service of type LoadBalancer exposing an internal API include Services of type `LoadBalancer` created without source IP restrictions, exposing internal APIs publicly, Pods configured with `hostNetwork: true` that bypass the cluster network and bind directly to the host, or workloads mapping host ports such as `22` (SSH) onto every node, unintentionally opening attack vectors at the infrastructure level.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: public-service
spec:
  type: LoadBalancer     # <= Exposes service to the internet by default
  ports:
    - port: 8080
      targetPort: 80
  selector:
    app: internal-api    # <= An internal API unintentionally made public

```

Figure 8: Service of type LoadBalancer exposing an internal API

By catching these risks at the pre-deployment stage, insecure workloads are blocked before they ever enter production, ensuring that Kubernetes clusters start from a hardened and trustworthy baseline.

The benefits of this approach are multiple. By embedding **shift-left security** into NATWORK's DevSecOps workflow, developers and operators gain immediate feedback on security flaws during the build and deployment phases, long before the workloads run in production. This reduces the likelihood of misconfigurations reaching live clusters, thereby lowering the risk of privilege escalation, data leakage, or lateral movement attacks. Furthermore, the approach ensures that clusters start from a hardened security baseline, which improves resilience across both **cloud-native and edge/O-RAN environments**.

The proposed solution integrates **CERTH's AI-IDS** into the Kubernetes pre-deployment pipeline as a policy gatekeeper. By analysing manifests, Helm charts, RBAC rules, and Network Policies before workloads are deployed, it detects misconfigurations such as privilege escalation, exposed secrets, and overly permissive access. This **shift-left security** approach blocks insecure workloads early, provides immediate feedback to developers, and establishes a hardened baseline for deployment. As a result, NATWORK strengthens Kubernetes security against misconfiguration-driven risks, reducing the likelihood of privilege escalation, data leakage, and lateral movement attacks, while enhancing resilience across cloud-native and edge/O-RAN environments.

3.2. PDSCMs on microservices

The pre-deployment security measures outlined in *Section 2.2.3 - Pre-deployment Microservice Security by construction* are currently at varying stages of implementation within the NATWORK project. The declarative modelling of CNF (Containerised Network Function) dependencies and the specification of their cybersecurity requirements are under active development, with an initial mature version already integrated into the secure-by-design orchestrator (sFORK) for runtime management. These models, expressed through Kubernetes YAML manifests and Helm charts, provide an explicit description of service dependencies, allowed communications, and resource bindings. This allows sFORK to reason about CNF composition and to enforce secure scheduling decisions at runtime. The declarative approach is already being implemented in the NCL testbed to test inter-service communication and meeting security requirements of the 6G slices.

The project has successfully adopted and implemented Submariner to establish secure inter-cluster communication tunnels, providing encrypted connectivity between clusters. These tunnels provide confidentiality and authentication for cross-cluster service traffic and are integrated into the orchestration workflows. This step guarantees that communication links

between clusters are operational and secure at the service level. The use of IPsec has already been validated in initial testbed experiments, enabling secure multi-cluster orchestration.
The automatic (re)configuration of RBAC rules, informed by vulnerability assessment tools, is planned for a subsequent phase of the project to further harden the security baseline. While RBAC rules can already be defined declaratively, NATWORK is exploring automated (re)configuration based on vulnerability assessment results. Lightweight tools such as *Kubesec* can provide recommendations on patching runtimes and tightening access rights, which can then be translated into RBAC policies. This would allow DevSecOps pipelines to dynamically adapt permissions before deployment, aligning the access model with both security requirements and runtime risk assessment.

We created a secure baseline before deployment through dependency modelling and encrypted inter-cluster channels. The automatic RBAC policies are planned to be developed. On top of this baseline, runtime optimisation strategies, CTI-driven selective sharing of hygiene scores and AI-based workload prediction developed in D3.1 provide additional protection, optimisation and adaptability. This combination allows sFORK to make placement and scaling decisions with both pre-deployment hardening and live feedback in mind, linking DevSecOps practices with runtime orchestration.

3.3. O-RAN xAPP onboarding security analysis and progress

3.3.1. IS-RD Liquid xAPP threat model

IS-Wireless's Liquid RAN and Liquid near-Real-Time RIC (Radio Intelligent Controller) together form an open, cloud-native RAN system following O-RAN Alliance specifications. The near-RT RIC hosts xApps – microservice applications that ingest RAN data and issue control decisions – which interface with RAN components (e.g., O-DU/O-CU) over standardized O-RAN interfaces (such as the E2 interface). The entire system is deployed on a Kubernetes-based cloud-native platform, meaning RAN, RIC, and xApp components run in containers orchestrated by Kubernetes. This architecture introduces new security considerations due to microservice communication, multi-vendor plugin apps, and disaggregated network elements.

We applied the STRIDE methodology with its threat modeling categories – Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service (DoS), and Elevation of Privilege – to identify potential threats to the IS-Wireless xApp in four contexts:

- Kubernetes Cloud – The cloud-native infrastructure that orchestrates and secures containers running Liquid RAN, RIC, and xApps.
- near-RT RIC Platform – The control framework that hosts xApps and manages near real-time optimization of RAN functions via standardized O-RAN interfaces.

- 3rd party xApp – Another vendor's modular microservice deployed on the RIC that consumes RAN data and issues control actions for tasks like traffic steering or slice assurance.
- Liquid RAN Components – The disaggregated RAN building blocks (O-DU, O-CU, O-RU) that deliver radio access services and interact with the RIC through E2 and O1 interfaces.

Each STRIDE category is discussed with examples in these contexts in the following Table 8. Liquid near-RT RIC xApp STRIDE analysis

Table 8. Liquid near-RT RIC xApp STRIDE analysis

STRIDE Category	Kubernetes Cloud	near-RT RIC Platform (framework & services)	xApp (3rd-party plugin)	Liquid RAN Components (O-DU/O-CU and O-RU)
Spoofing	- Rogue pod impersonating a service due to lack of mTLS, allowing attacker to masquerade as xApp or API server. - Compromised credentials used to create malicious pods, appearing as legitimate components.	- Malicious xApp or process impersonates RIC internals via unsecured APIs, injecting commands as if from a trusted module. - Lack of mutual auth allows a fake RIC component to register as part of RIC and intercept traffic.	- Rogue xApp presents stolen or forged credentials during onboarding to pose as a trusted vendor's xApp (if onboarding process is weak). - One xApp pretends to be another via inter-xApp API if mutual auth isn't enforced.	- Fake base station (rogue O-DU) tries to connect to RIC's E2 interface, impersonating a legitimate RAN node to inject false data. - Spoofed gNodeB ID in E2 messages if authentication is missing, misleading the RIC about the sender.
Tampering	- Supply chain attacks inserting malicious code into container images. - Attacker modifies cluster config to alter network policies	- Manipulation of RIC message bus traffic (if not signed) . - Exploiting a vulnerability to modify RIC's state (e.g., change	- Malicious xApp tampering with control messages to disrupt service. - Altering data it receives (telemetry) before passing to other modules,	- Injection of false configuration via O1 or E2: attacker alters a parameter (like frequency or TX power) in transit, impacting RAN

STRIDE Category	Kubernetes Cloud	near-RT RIC Platform (framework & services)	xApp (3rd-party plugin)	Liquid RAN Components (O-DU/O-CU and O-RU)
	or disable security checks.	policy values in memory).	feeding false info into RIC decisions.	behavior. - Tampering with fronthaul or synchronization messages (if physical access gained) causing RAN faults.
Repudiation	- Inadequate audit logs let an attacker change settings and delete evidence, claiming innocence (e.g., deleting a rogue pod leaves no trace if logging is off). - No tracking of which admin or service account performed a critical action, enabling plausible denial.	- Poor logging of xApp actions means a rogue xApp can send a harmful command and later deny it was the source. - If RIC configuration changes aren't logged with who/when, an attacker could flip them undetected.	- If xApp actions aren't audited, a vendor can deny their xApp caused an incident. - XApp could manipulate its log outputs or use unsupported channels to perform actions, evading normal logs.	- A compromised RAN node could deny sending a critical alert if logs are not collected (e.g., it turned off an alarm and claims it never happened). - If RAN audit logs (of commands received from RIC) are absent, RAN vendor could repudiate that a detrimental command came from their equipment.
Information Disclosure	- Stolen service account token used to read all Kubernetes Secrets (e.g., RIC credentials). - Sniffing intra-cluster traffic if	- RIC's database or monitoring data exfiltrated via a debug interface left open, leaking cell performance or user metrics.	- xApp gains unauthorized read access to subscriber data or cell configs via a misused API, leaking sensitive info externally	- Unencrypted CUs/DUs control traffic could be sniffed, revealing subscriber traffic patterns or keys. - O1 interface data (config files,

STRIDE Category	Kubernetes Cloud	near-RT RIC Platform (framework & services)	xApp (3rd-party plugin)	Liquid RAN Components (O-DU/O-CU and O-RU)
	no encryption, revealing RAN telemetry or credentials.	- An unauthorized user in RIC could query data meant for SMO or operators (like network topology info).	(e.g., phone locations). - Supply-chain compromised xApp quietly sends confidential RAN data to attacker's server.	performance reports) intercepted by an attacker, leaking network configuration details.
Denial of Service	- Attackers spam the Kubernetes API to overwhelm the control plane (schedule countless pods) causing management outage. - A noisy neighbor container exhausts node CPU/memory, starving RIC components (if no limits).	- Crash of a core RIC service (E2 terminator, routing manager) triggered by malformed xApp message, halting near-RT control. - Multiple xApps issuing heavy compute tasks (like complex AI inferences) freeze the RIC's real-time processing.	- An xApp intentionally consumes excessive RIC resources (e.g., subscribes to every possible event at high frequency) to overwhelm the RIC or E2 node (preventing other xApps from timely processing). - Failure to handle backpressure: a slow or hung xApp causes queue buildup, blocking other xApps' messages (indirect DoS).	- Flooding the RIC with excessive measurement reports or fault indications (a hacked O-DU could spam E2 messages) to overwhelm RIC processing capacity. - Desynchronizing RAN: e.g., a timing sync attack making cells go out of sync, effectively a DoS on radio service.
Elevation of Privilege	- Compromised container escapes to host (if running	- A bug in RIC (or RMR library) allows code	- A compromised xApp exploits an API flaw to	- If an O-DU is compromised, it could potentially

STRIDE Category	Kubernetes Cloud	near-RT RIC Platform (framework & services)	xApp (3rd-party plugin)	Liquid RAN Components (O-DU/O-CU and O-RU)
	privileged), gaining root on host node. - Excessive RBAC privileges let a low-level service account modify cluster roles (becoming cluster-admin).	execution, letting an xApp gain control of the RIC host process. - An xApp without proper sandboxing calls privileged RIC APIs to change its permissions or access other xApps' data.	elevate its role (gaining permissions to control all cells instead of its assigned scope). - xApp escapes its container (if running with unnecessary privileges) and modifies host or RIC files, effectively becoming an admin on the system.	issue privileged Core network messages or alter its role (e.g., act as a 'master' node) beyond design. - Malware on a DU could use the O-RAN interfaces to pivot into the RIC's domain, escalating its reach into the control plane.

3.3.2. NATWORK work on xAPP security

3.3.2.1. General

To augment the traction of our security development, we shall first stand on WG11 integrity solutions showing a high maturity level, notably xAPP authentication at onboarding. Hence, our development will be steered to develop solutions in three areas where WG11 work is of lesser maturity level (i.e., remote attestation, runtime integrity and SBOM runtime enforcement). We will work with ISRD, acting as the aka Liquid xAPP provider and the integration of D-MUTRA blockchain-based remote attestation framework. The following work plan has been defined.

3.3.2.2. On remote attestation

The WG11 architecture places the SMO as the central and unique verification utility, which exposes it to DoS attacks caused by flooding attestation requests. The SMO is the utility which also validates remote attestation. Last and as stated above our work plan is twofold:

- Develop an alternative to the SMO centralized verification (and storage of xAPP catalogue), based on D-MUTRA decentralized framework. This alternative framework

shall no longer be directly dependent on the integrity of the verification and measuring entities.

- Develop a user-centric remote attestation where permissioned stakeholders can collect their workload remote attestation timestamped results.

3.3.2.3. On runtime integrity

We will progress on the three challenges defined by the WG11:

- Trustworthiness of the integrity monitoring: security analysis of our decentralized structure
- Applicability of remote attestation when applied to AI/ML: Consider how model parameters can tentatively integrate the range of the measured memory footprint
- Performance impact: Elaborate three possible schemes of:
  - *Spread-over-time hashing technique*, to reduce the resource consumption by the measuring thread
  - *Linux's cgroup CPU resource restriction*, applied to the measuring thread
  - *On-demand trigger* to limit to one measurement only (i.e., at user-defined timing).

3.3.2.4. On SBOM runtime verification

Taking advantage of the sidecar container as used for D-MUTRA, we will expand its functionality from integrity verification to dependency check, intercepting all called dependency at runtime. We will consider how an agent can share the file system and get a dynamic view of the called dependencies. According to the permission for sidecar mounting, we will define the appropriate implementation either using a sidecar or by binary rewriting.

3.3.2.5. PDSCM for xAPP security

The following PDSCMs will be applied on the Liquid xAPP:

- SECaaS processing for reference measurement: This pre-deployment step elaborates the reference measurement and stores it on the SECaaS. This reference measurement will be used by D-MUTRA for integrity verification.
- *Docker compose* for sidecar mounting: This operation consists of modifying the Docker orchestration, adding a script line referring to the sidecar container for its future collateral mounting aside the xAPP container.

3.3.2.6. xAPP migratability

xAPP migratability may be affected with the sidecar mounting, as this opposes some WG11 guidelines (i.e., restricting deployment to what SMO strictly knows). However, in some restricted conditions, sidecar mounting is permitted. Noticeably, in the *Security Near-RT RIC xApps* technical

report [26] , there is a specific mention of sidecar containers: “Optional side-car container for key and certificate management reduces the attack surface on the Applications.”

According to a deeper technical survey, we will opt for the sidecar mount or a direct SECaaS rewriting of the xAPP as is described in Figure 9.

Benefits of sidecar mounted SECaaS

In Figure 9, two different setup workflows are represented. The left-hand section shows a PDSCM based workflow with operations performed prior to deployment by a SECaaS. In the right-hand section, a diagram without SECaaS is shown, applying only to containerised workloads and enabling a Drop and Attest model, where the “dropped” container has not gone through pre-deployment change, therefore is deployed without modification or measurement. This simplified workflow is more scalable while limiting possible security functions to runtime integrity only. In fact, no prior-deployment code encryption, for code confidentiality preservation, can be delivered. Moreover, no prior-deployment reference measurement will be produced by the (inexistant) SECaaS but the reference measurement is collected at the first execution of the payload, by the sidecar container and serves for future integrity measurement. This schema does not bring remote attestation as the measure cannot be verified with a locally stored reference measurement but brings runtime integrity verification, which represents however the main security gap to fill.

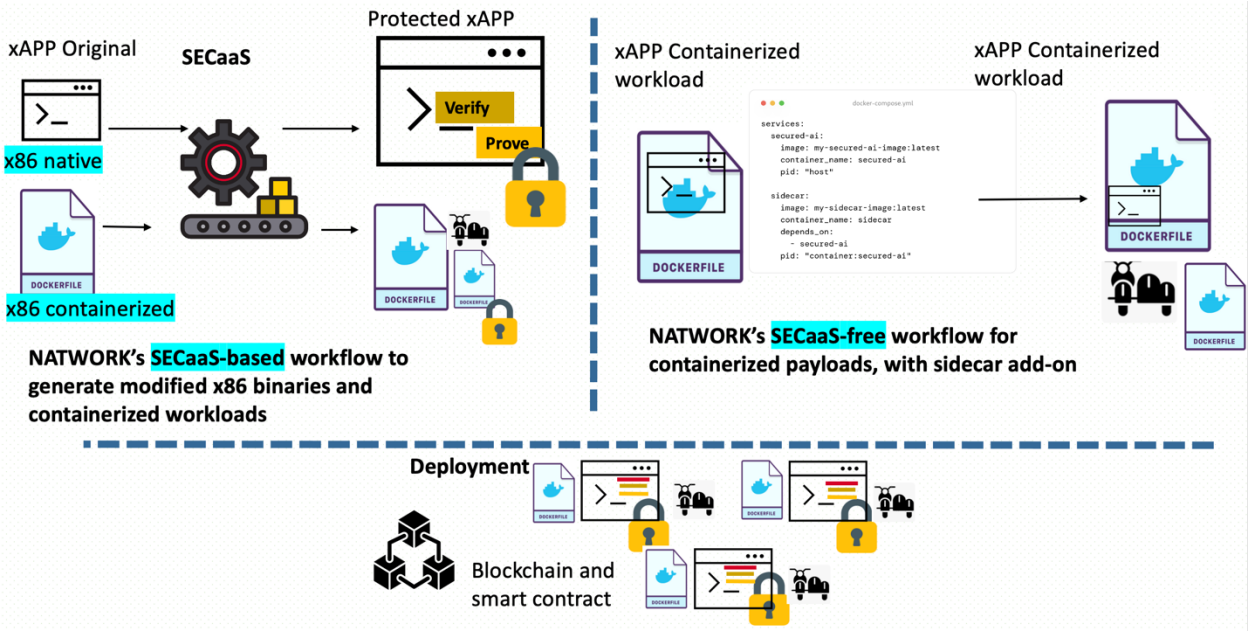

Figure 9. Two schemes for xAPP security.

3.3.2.7. xAPP performance and latency

The blockchain-based remote attestation induces a penalty of around 2-3 seconds, integrating the complete DLT cycle. We will investigate if this is acceptable with standard on-boarding (and RAN near real time loops (i.e., 10 ms -1s). A possible shift to *Attest-After-Starting* pattern will also be investigated as it drops latency to nil. The performance impact of three different runtime integrity verification methods will be measured.

4. NATWORK's PDSCMs on native workloads

4.1. General

Montimage's MMT (Montimage Monitoring Tool) network anomaly detection is a good candidate for PDSCM for the native deployment case. As evoked in Section 2, PDSCM can harden security services, by nature targets of choice exposed to various attacks. A dedicated effort has been initiated and is on-going in the NATWORK project.

4.2. MMT's threat model

The MMT Framework is a modular platform for network monitoring, traffic analysis, and security enforcement. It is designed for both research and operational environments, providing deep visibility into traffic patterns as well as the ability to enforce security rules in real time. The framework is composed of several key components:

- **MMT-Probe**: A high-performance packet capture and analysis engine based on Deep Packet Inspection technique that extracts traffic metadata, protocol details, and application-level insights.
- **MMT-Security**: A security enforcement module that applies detection rules (compiled into `.so` shared libraries) to traffic flows, identifying threats, anomalies, or policy violations.
- **MMT-Operator (optional)**: Interfaces for orchestration, visualization, and management of collected data and security alerts.

The MMT framework can be deployed in two main ways depending on the needs of the environment. One option is native installation, where users compile the binaries such as *MMT-probe*, *MMT-security*, and the associated `.so` modules, and then run them directly on Linux systems. This approach provides maximum flexibility for integration with custom setups and allows fine-grained control over configuration and optimization. Alternatively, a more modern and reproducible method is containerized deployment using pre-built Docker images. With this approach, all dependencies are packaged together, making it easier to install, test, and update the framework while ensuring consistency across environments. Containerized deployment also simplifies orchestration with tools such as Kubernetes or Docker Compose, which is particularly valuable in infrastructures that need to scale dynamically or enforce standardized deployment practices.

When deploying the MMT framework, whether through native binaries or containerized environments, the security of the software artefacts and associated rulesets is a critical concern.

A binary or shared library that has been tampered with, replaced, or misconfigured can undermine the entire monitoring and enforcement pipeline. To evaluate these risks systematically, we apply the STRIDE methodology [54], which helps us reason about potential threats across six key categories. We discuss in detail the threat model of deploying MMT that highlights multiple pre-deployment security considerations.

- **Spoofing** is a significant concern during the distribution and deployment of MMT. Without proper controls, an attacker could impersonate a legitimate developer or repository and trick operators into installing a maliciously crafted version of MMT-*probe*, *MMT-security*, or one of the `.so` detection modules. In practice, this can occur if binaries are downloaded from unofficial mirrors or if Docker images are pulled from unverified registries. Preventing spoofing requires strong authentication of both the source repository and the individuals who build and release MMT.
- **Tampering** focuses on the risk of modification to binaries or rulesets before they are deployed. Because MMT-Security relies on compiled `.so` rules to enforce detection logic, even a subtle modification in a library could result in rules being disabled, altered, or replaced with logic that intentionally bypasses threats. For example, a tampered library could silently allow specific malicious traffic through, creating a blind spot in monitoring. To counter this risk, deployments must incorporate artifact integrity validation, such as checksum verification, digital signatures, or trusted build pipelines.
- **Repudiation** arises when there is no clear accountability for changes made to binaries or configurations. In environments without proper logging and version control, it may be impossible to prove whether a binary was modified by a malicious actor or simply updated by a developer. This lack of traceability hinders incident response and weakens confidence in the security posture. Implementing auditable pipelines, logging all artifact changes, and enforcing commit signing are crucial to address repudiation threats.
- **Information disclosure** represents another serious category of risk. The detection rulesets themselves may encode proprietary intellectual property or sensitive patterns used for anomaly detection. If these `.so` libraries or associated configuration files are leaked, an adversary could gain insights into the organization's detection strategy, allowing them to craft evasive attacks. Moreover, improper containerization or configuration could inadvertently expose sensitive credentials used by MMT to external parties. Ensuring proper access control, encrypting secrets, and limiting container privileges are necessary steps to reduce exposure.
- **Denial of Service (DoS)** can result from the deployment of corrupted or malicious artifacts that cause instability in the monitoring stack. For example, a malformed ruleset could trigger a crash loop in *MMT-security*, rendering the detection system unavailable. Similarly, a binary modified to consume excessive resources could degrade the overall

monitoring environment. DoS threats highlight the importance of testing rulesets in staging environments before production rollout, as well as implementing resource isolation through container orchestration platforms like Kubernetes.

- **Elevation of Privilege** threats occur when compromised binaries or libraries exploit elevated system permissions to execute unauthorized actions. Since the MMT framework often runs in privileged environments where it has access to raw network traffic and sensitive telemetry, a backdoored version of *MMT-probe* or *MMT-security* could be used to exfiltrate data or manipulate monitoring outcomes. Preventing this requires strict adherence to the principle of least privilege in both IAM roles and container runtime configurations, ensuring that even if a component is compromised, its ability to escalate further within the environment is minimized.

4.3. PDSCM on MMT

With respect to detailed threat analysis exposed above, MMT integrity preservation appears to be the priority as the vast majority of threats are directly linked to a modification of the executable, being at the time of deployment (i.e., spoofing) or during its execution (i.e., tampering, repudiation, elevation of privileges and denial of service). Code tampering is the self-evident attack vector for all security-related software. NATWORK has considered two alternatives detailed below.

4.3.1. MMT remote attestation and continuous integrity verification

Covering both stages of deployment and runtime, the two techniques can be offered by D-MUTRA, a TSS's solution providing automatic remote attestation and using the same memory footprint measurement for both verifications. D-MUTRA is an outcome of DESIRE-6G SNS project [49] that aligns with NATWORK, fostering workload migratability by removing technical dependencies (e.g., TPM, Linux's IMA presetting). Its runtime integrity verification is designed for being penalty-free (with a cap of 1% performance penalty). Noticeably, D-MUTRA leveraging of Hyperledger blockchain shall not be perceived as a dependency as the blockchain can be setup anywhere, separately from the workload's execution environments.

In Use Case 4.6 dedicated to DoS prevention by self-monitoring, we will initiate our work with the implementation of D-MUTRA to assess the integrity of MMT. Penalty measurements will be worked out as well as the blockchain footprint inflation rate.

The PDSCM consists in modifying MMT executable to integrate different routines (i.e., Prove, Verify, DLT-com) for MMT to integrate D-MUTRA service, hereto be remotely attested and continuously integrity verified.

4.3.2. MMT in-TEE sheltering

As part of Use Case 4.5 dedicated to optimized and explainable MTD, ZHAW and Montimage are closely working on the dynamic placement of MMT in and out of a TEE enclave with AMD's SEV-SNP (Secure Encrypted Virtualization with Secure Nested Paging) [53], a VM-type TEE. The experiments are conducted on AMD EPYC v4 processors, which provide native support for AMD SEV-SNP [5] Noticeably, the experiment totally aligns with NATWORK's concept, consisting of a hot migration of MMT to a TEE sheltering, (only) at occurrence or presumption of a security threat. We believe that this policy totally makes sense, with the avoidance of a costly by default overprotection (i.e., if MMT were sheltered in TEE by default).

The planned PDSCM consists of inserting MMT inside a migratable SEV-SNP managed Virtual Machine. TEE sheltering overhead, although assessed to be relatively small with the VM-type form of SEV-SNP, is still to be measured. The impact in terms of memory consumption and CPU overhead will be measured in the use case as part of upcoming NATWORK efforts.

5. NATWORK's PDSCM on WASM workloads

NATWORK's work on WASM workload security has focused on module runtime integrity verification, a runtime method assessing the integrity of the workload memory footprint and comparing it with a pre-deployment reference measurement. This work aims at filling a security gap, deemed critical for WASM adoption in networking, where a WASM module can be directly modified since it is treated as a data structure where no *Write xor Execute* protection can be enforced. By doing so, NATWORK is taking a significant step. Referring to the list of possible PDSCMs as listed in Section 2.1 of this document, our work stands on integrity preservation, detecting tampering in the complete module lifetime.

The PDSCM consists of collecting a reference measurement prior to deployment (and used during the verification occurring on-boarding or module execution). It also consists in installing a workload identifier used for reconciling the workload and its reference measurement. The main difficulty of our development consists in collecting at runtime evidence that the workload has not been modified.

5.1. NATWORK runtime integrity technique

To collect integrity evidence at runtime, we first analysed a WASM payload memory map as shown in Figure 10.

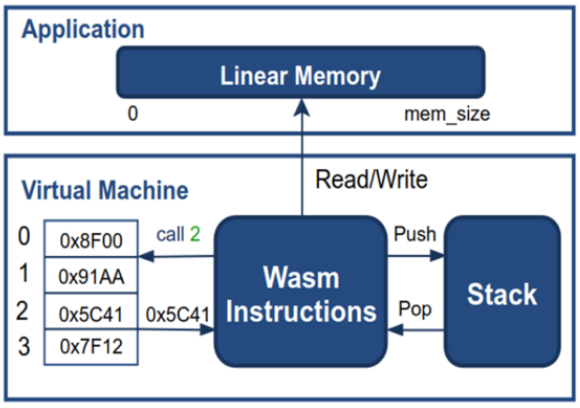

Figure 10. Memory map of WASM runtime (virtual machine) and module (application)

We discovered that WASMTIME created three distinct memory areas (i.e., stack, WASM instructions and linear memory). An integrity checker requires access to the WASM instructions during runtime which can only be accessed at the runtime level. As a matter of fact, linear memory only contains offsets to the WASM instructions, insufficient to assess integrity. Then, our reverse engineering of WASMTIME, we had then discovered that we could force a JIT compilation and a serialization of the executable binary blob. A serialized blob is a specific format

derived from ELF. This format contains the instructions in the classical ELF's text section. Noticeably, these x86 native assembly instructions result from the JIT compilation which ingests the WASM's module instruction. Any change to the WASM module results in a different set of native instructions. Our implementation is depicted in the flow graph of Figure 11.

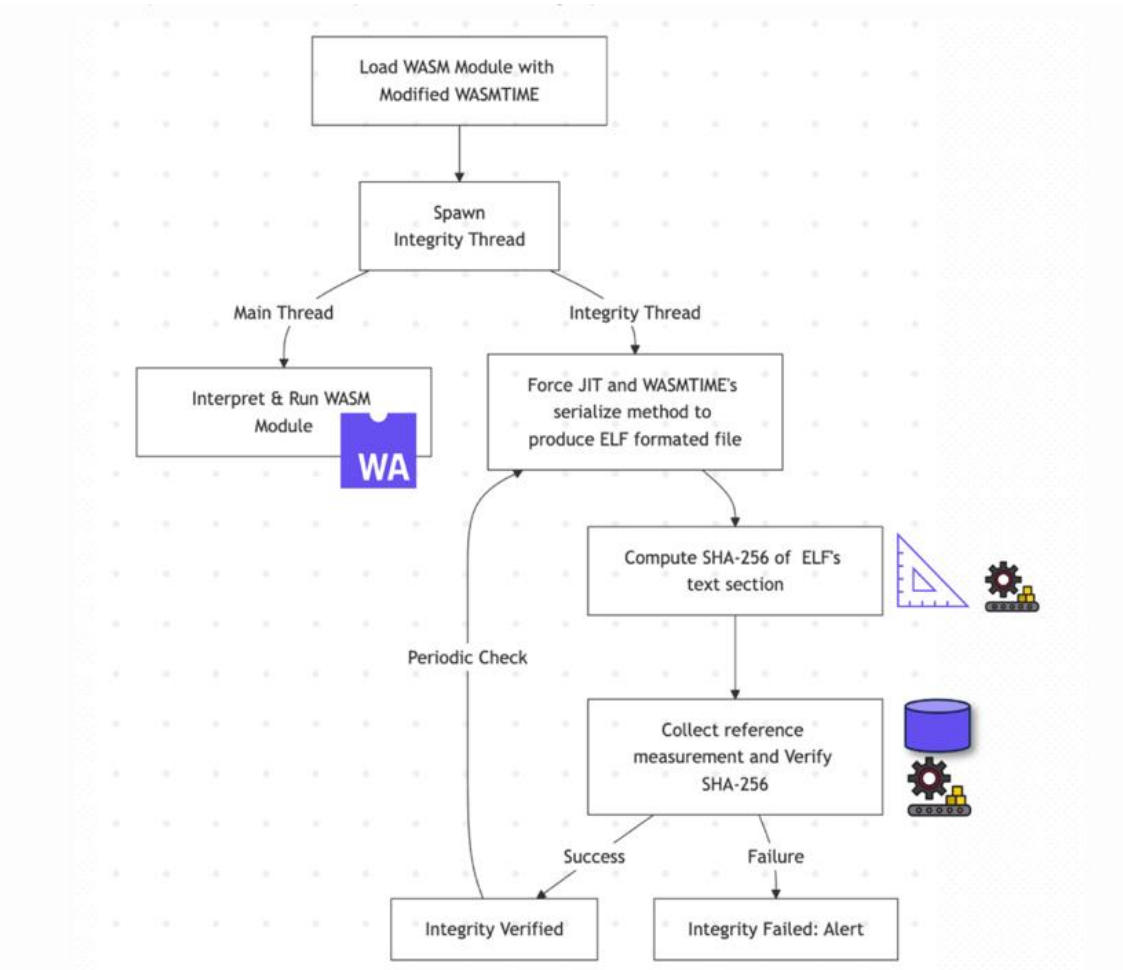

Figure 11. Flow diagram of NATWORK WASM module runtime integrity verification

We have created a second thread triggering JIT+ serialization and we produce a hash of the ELF's text section and compare it to a reference signature. The reference measurement derives from the same exact process in our SECaaS, using the exact same components. Two methods can be used to store the reference measurement. It can be appended directly on the WASM module, resident in our SECaaS or be stored after generation at the first loop iteration. It is then used for comparison.

Our WASMTIME interpreter has been appended with Prove and Verify routines able to generate the hash and verify it by comparison with the reference measurement.

5.2. Integration in D-MUTRA blockchain based mutual remote attestation

For the integration with D-MUTRA, a blockchain-based mutual remote attestation framework, the following steps shall be worked out.

Producing a modified WASTIME runtime

WASMTIME is open source, the integrity verification functions are programmed at source code level, and a novel compilation of the runtime is generated, as depicted in Figure 12.

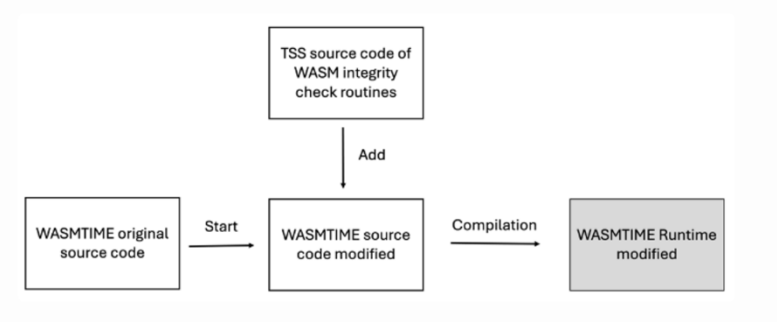

Figure 12. Modified WASMTIME runtime generation

Full stack remote attestation scheme

The modified WASM runtime is per-se a security-sensitive entity accessing the WASM payload at the first place and secondly producing its integrity verifications. It is a basic security provision to attest the runtime at the first place.

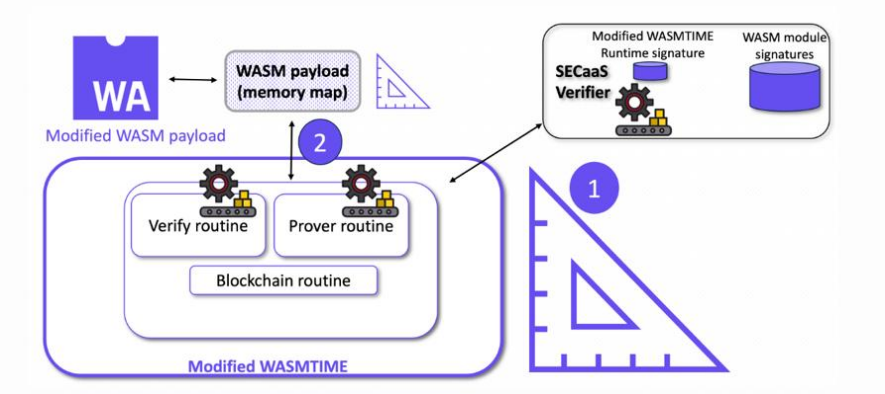

Figure 13. WASM full-stack remote attestation

Figure 13 shows a basic workflow where TSS modified WASMTIME interpreter is first checked (1) by comparison of a SECaaS reference measurement before the modified runtime delivers payload integrity verification (2).

Several implementations are considered for measuring the runtime. The agent can be appended with its own proving routine, or if the runtime is delivered as a container, a side car container will be added for the proving task. In both cases, the SECaaS will be used as a centralized verifier.

Mutual remote attestation scheme

D-MUTRA enacts a novel software-based chain of trust based on integrity freshness criteria, where the most recently verified payload is elected by consensus to make the next payload verification. In the context of WASM, D-MUTRA principle must be slightly deviated. WASM module cannot directly verify a peer (i.e., module), but WASM runtime can. The verifier election smart contract shall elect WASM runtimes, as reflected in Figure 14.

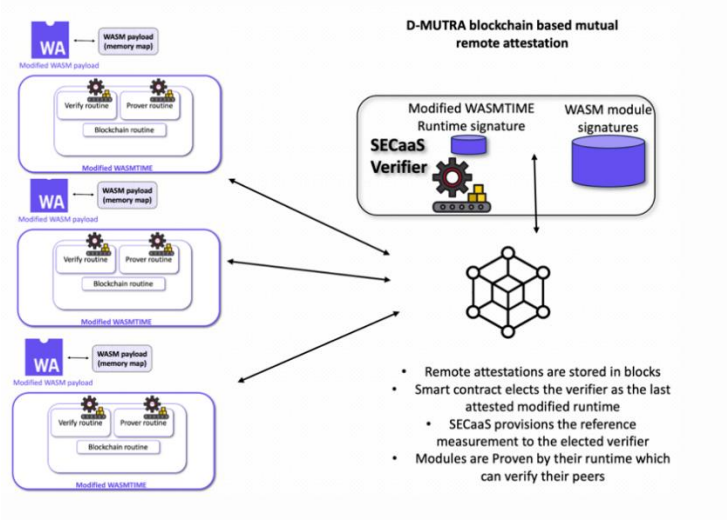

Figure 14. WASM mutual remote attestation by D-MUTRA

5.3. Alignment with NATWORK

5.3.1. Workload portability

Keeping in mind the stated priority of workload migration ability as defined in D2.1, it is worth stating that our solution restricts deployment to locations where a modified WASMTIME runtime is implemented, only where integrity verification is requested. Conversely, WASM modules can still be executed in a standard WASM runtime but without integrity being verified during their execution. In fact, WASM top notch migratability is conserved and untouched as WASM modules can still be executed on any platform duly equipped with a WASM runtime of any kind. In the context of the Telecom industry, workloads will be executed in either controlled or managed execution environments. The execution environments are either detained or managed by operators which deploy their workloads (i.e., telecom operators) or by their contractors (i.e.,

cloud vendors). In both cases, prior to deploying the WASM module, a managed deployment and verification leveraging either authentication or better remote attestation of the specific runtimes can be processed. In practice, telecom operators can set up a pre-deployment security per construction deployment of the runtimes to get a runtime integrity of their modules thereafter. This policy can be worked out because it implies a limited and controlled number of deployments at pre-identified locations. Moreover, the possible execution of the same WASM modules with unchanged runtimes, relaxes this runtime pre-deployment policy and justifies it, in the perspective of the telecom operators whose modules can still execute everywhere (up to uncontrollable end user endpoints) but will be integrity verified in a perimeter defined by the runtime pre-deployment policy.

5.3.2. Performance impact

As shown in [49], the performance impact induced by integrity verification can be capped to an average 1%, by use of two techniques:

- *Spread-over-time hashing*, where each step of the hashing process is paused with duration adjustable idle times. This technique resides in a specific development by modification of a hashing function. No administrative right is needed to employ this user-level technique. Noticeably, the performance impact is workload dependent.
- *Linux's cgroup resource limitation*, applied to the measuring thread, ensuring that the WASM module interpretation executed at the same time is not allocated with a reduced amount of CPU resources. Administrative rights must be delivered to leverage this system's utility.

These two techniques will be complemented with an on-demand activation pattern, lower bounding the impact irrespectively to the workload type and tentatively through an implementation without specific administrative right. The specific advantage of on-demand trigger is the limitation to one measure made sporadically, with no periodic repetition, hereto dropping drastically dropping the induced costs.

5.3.3. Sustainability

Sustainability shall be considered over resource consumption in terms of CPU processing and memory usage. With an adjustable CPU processing level limited to 1%, the solution can be considered sustainable.

The memory consumption of the measuring process is always lower than the memory footprint of the monitored process and generally smaller by several orders of magnitude. A short-lived buffer is used and released after each measurement cycle.

When integrated with the D-MUTRA blockchain-based remote attestation framework, blockchain inflation rate must be monitored and controlled by limiting the block creation

cadence. In a high-frequency scenario—one attestation every 2 seconds for 3 agents—the system produces approximately 47.3 million attestations per year, resulting in around 28.4 GB of blockchain data annually. However, D-MUTRA mitigates this by storing only tampering detections, which are rare events. This reduces the blockchain inflation rate by 6 to 12 orders of magnitude, depending on the frequency of detected anomalies. In this condition, one tampering detection inflates the blockchain by approximately 600 Bytes, which is totally negligible. In addition to tampering detection, each onboarding induces a remote attestation at the same cost.

5.4. Future work in the NATWORK project and beyond.

5.4.1. D-MUTRA integration

As explained, the full stack remote attestation and the integration into D-MUTRA will be carried out during the project.

5.4.2. Towards 0-latency at start

An implementation of a novel Attest-After-Starting measuring sequence will be established, enabling workload to instantly start. To cover the associated integrity blind window (i.e., between the workload start and its measurement), our design will consider a bridge with the workload authentication at on-boarding. From the authentication step onward, our attestation takes runtime measurements.

5.4.3. Lower bounding the performance impact in all situations with an on-demand trigger

An implementation of an on-demand activation of the runtime verification will be established, dropping the performance impact in all situations, irrespective of the workload size and detention of platform administrative right.

5.4.4. WASM module confidentiality preservation

We will devise and implement confidentiality preservation, through another set of modifications on WASMTIME runtime and a SECaaS processing. The latter will encrypt the WASM module while the former will decrypt the encrypted module before execution.

5.4.5. During or beyond NATWORK. Mitigating JIT spraying

During NATWORK, we will study the possibility to detect JIT spraying, leveraging our integrity verification second thread as described in Figure 11. Flow diagram of NATWORK WASM module runtime integrity verification JIT spraying defense was not part of our original plan, according to our feasibility study, the implementation of JIT spraying defense will be tentatively worked out during or following NATWORK.

6. Conclusion

In NATWORK, PDSCMs is elaborated on the three payload formats, **elevating security substantially in each domain:**

1. **Containerized xAPP** security is elevated, beyond O-RAN authentication specifications and fulfilling O-RAN WG11 identified persisting security threat (i.e., runtime tampering), applying runtime integrity verification. The PDSCM consists of a modified Docker's orchestration layer, to bridge a sidecar container. If this schema cannot be applied, the PDSCM will consist of modifying the xAPP to inject the required routines for the continuous integrity verification.
2. Highly migratable **WASM modules** security is significantly improved by a PDSCM consisting of modifying the WASM runtime, not the WASM module itself. This is a significant security improvement, done without touching the payload, **making WASM technology safer and usable for networking.**
3. Our work on **native payloads** integrates use cases illustrating how PDSCM hardens a security service. Two PDSCMs are implemented, consisting of (i) placing MMT probe (i.e., a network anomaly detection probe) into SEV-NP TEE and (ii) modifying it to be attestable and runtime verified. This work will be continued and exemplified in Use Cases 4.5 and 4.6, respectively. This will notably show how MMT integrity preservation can be offered by two opposing techniques (i.e., TEE and remote attestation) and the intricacies and impacts of each in terms of performance and sustainability.

PDSCMs contribute to NATWORK's reconciliation principle, reducing the costs of security by applying security at each elementary component. They also contribute to NATWORK's security challenges, **hardening the security code itself, for more reliable security services**. They differ in nature and are enacted at different levels, impacting differently workload migratability as recalled in Section.2.1.1.2.

As each use case and context differs (e.g., no workload migration considered, possession of platform administrative rights, severity of the security threat, and access to technology), the appropriate PDSCM can be implemented to better match the requirements and offered possibilities.

6.1. Next steps

The next phase will focus on validating and integrating the proposed Pre-Deployment Security per Construction Measures (PDSCM) across native, container, and WASM payloads. Efforts will extend towards adaptive, performance-aware security regulation, ensuring optimal trade-offs

between protection, efficiency, and sustainability. Validation through representative cloud–edge use cases will demonstrate secure and platform-agnostic payload mobility. Continued collaboration with confidential computing and WASM ecosystem initiatives will ensure alignment with state-of-the-art developments, reinforcing NATWORK's objective to deliver secure, interoperable, and energy-efficient computing continuum operations. These efforts will be reported in the deliverable D3.6 – "Pre-Deployment Security per Construction Measures.r2" due to M30.

References

[1] Intel. (n.d.). Confidential containers made easy. Retrieved September 21, 2025, from https://www.intel.com/content/www/us/en/developer/articles/technical/confidential-containers-made-easy.html

[2] RedHat. (n.d.). Zero trust starts here: Validated patterns for confidential container deployment. Retrieved September 21, 2025, from https://www.redhat.com/en/blog/validated-patterns-confidential-container-deployment

[3] Falcao, E., Silva, F., Pamplona, C., Melo, A., Asadujjaman, A. S. M., & Brito, A. (2025). Confidential Kubernetes deployment models: Architecture, security, and performance trade-offs. Applied Sciences, 15(18), 10160. https://doi.org/10.3390/app151810160

[4] Intel. (n.d.). TDX documentation. Retrieved September 21, 2025, from https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html

[5] AMD. (n.d.). SEV-SNP documentation. Retrieved from https://www.amd.com/fr/developer/sev.html

[6] Ye, M. (2024). Enabling Performant and Secure EDA as a Service in Public Clouds Using Confidential Containers. Retrieved September 21, 2025, from https://arxiv.org/pdf/2407.06040v1

[7] Confidential Containers. (2024). Introduction to Confidential Containers (CoCo). Retrieved September 21, 2025, from https://confidentialcontainers.org/blog/2024/02/16/introduction-to-confidential-containers-coco/

[8] Pronteff. (n.d.). Openshift confidential containers now on Microsoft Azure. Retrieved September 21, 2025, from https://pronteff.com/openshift-confidential-containers-now-on-microsoft-azure/

[9] Edgeless Systems. (n.d.). Marblerun. Retrieved from https://www.edgeless.systems/products/marblerun

[10] Johnson, M. A., et al. (2024). Confidential Container Groups: Implementing confidential computing on Azure container instances. ACM Queue 22(2). https://spawn-queue.acm.org/doi/pdf/10.1145/3664293

[11] Sanctuary. (n.d.). Trusted container extensions for container-based confidential computing. Retrieved September 21, 2025, from https://sanctuary.dev/en/blog/container-based-confidential-computing/

[12] Brasser, F. (2022). Trusted Container Extensions for Container-based Confidential Computing. https://arxiv.org/pdf/2205.05747

[13] Budigiri, G., Baumann, C., Mühlberg, J. T., Truyen, E., & Joosen, W. (2021). Network policies in Kubernetes: Performance evaluation and security analysis. In 2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit).

[14] Koukis, G., Skaperas, S., Kapetanidou, I. A., Mamatas, L., & Tsaoussidis, V. (2024). Performance evaluation of Kubernetes networking approaches across constrained edge environments. In 2024 IEEE Symposium on Computers and Communications (ISCC).

[15] Shamim, M. S. I., Bhuiyan, F. A., & Rahman, A. (2020). Xi commandments of Kubernetes security: A systematization of knowledge related to Kubernetes security practices. In 2020 IEEE Secure Development (SecDev) (pp. 58–64).

[16] Carrión, C. (2022). Kubernetes scheduling: Taxonomy, ongoing issues and challenges. ACM Computing Surveys, 55, 1–37.

[17] Rahman, A., Shamim, S. I., Bose, D. B., & Pandita, R. (2023). Security misconfigurations in open source Kubernetes manifests: An empirical study. ACM Transactions on Software Engineering and Methodology, 32, 1–36.

[18] Wlodarczak, P. (2017). Cyber immunity: A bio-inspired cyber defense system. In Bioinformatics and Biomedical Engineering: IWBBIO 2017, Proceedings, Part II (Vol. 5). Springer.

[19] Kampa, S. (2024). Navigating the landscape of Kubernetes security threats and challenges. Journal of Knowledge Learning and Science Technology, 3, 274–281.

[20] O-RAN Alliance. (n.d.). Near-RT RIC architecture 7.0. Retrieved from https://specifications.o-ran.org/specifications

[21] O-RAN Alliance. (n.d.). Security requirements and controls specifications 12.0. Retrieved from https://specifications.o-ran.org/specifications

[22] O-RAN Alliance. (n.d.). Study on security for Service Management and Orchestration (SMO) 6.0. Retrieved from https://specifications.o-ran.org/specifications

[23] O-RAN Alliance. (n.d.). Study on security for O-Cloud 7.0. Retrieved from https://specifications.o-ran.org/specifications

[24] O-RAN Alliance. (n.d.). Study on Zero Trust Architecture for Secure O-RAN. Retrieved from https://specifications.o-ran.org/specifications

[25] O-RAN Software Community. (n.d.). O-RAN xAPP SDK. Retrieved from https://lf-o-ran-sc.atlassian.net/wiki/spaces/ORANSDK/pages/14516830/xAppFramework

[26] O-RAN Alliance. (n.d.). Study on security for Near Real Time RIC and xApps 5.0. Retrieved from https://specifications.o-ran.org/specifications

[27] Collberg, C., Thomborson, C., & Low, D. (1997). A taxonomy of obfuscating transformations.

[28] Barak, B. (2016). Hopes, fears, and software obfuscation (survey / review).

[29] Xu, H., Zhou, Y., Kang, Y., & Lyu, M. R. (2017). On Secure and Usable Program Obfuscation: A Survey. arXiv:1710.01139.

[30] De Sutter, B., et al. (2024). Evaluation Methodologies in Software Protection Research.

[31] ARXAN Technologies. (n.d.). Application security. Retrieved from https://digital.ai/products/application-security/
[32] VMProtect. (n.d.). VMProtect. Retrieved from https://vmpsoft.com/
[33] SOLIDSHIELD. (n.d.). SOLIDSHIELD. Retrieved from https://www.solidshield.com/
[34] AppDome. (n.d.). AppDome. Retrieved from https://www.appdome.com
[35] Obfuscator-ai. (n.d.). Obfuscator-ai. Retrieved from https://pypi.org/project/obfuscator-ai
[36] Li, R., et al. (2024). PowerPeeler : A Precise and General Dynamic Deobfuscation Method for PowerShell Scripts.
[37] Garba, P., & Favaro, M. (2019). Saturn: Software deobfuscation framework based on LLVM
[38] Nagios. (n.d.). Nagios monitoring tool. Retrieved from https://www.nagios.org/
[39] Zabbix. (n.d.). Zabbix monitoring tool. Retrieved from https://www.zabbix.com/documentation/current/en/
[40] Prometheus. (n.d.). Prometheus + Grafana. Retrieved from https://prometheus.io/
[41] Datadog. (n.d.). Datadog monitoring tool. Retrieved from https://www.datadoghq.com/
[42] New Relic. (n.d.). New Relic monitoring tool. Retrieved from https://newrelic.com/
[43] DESIRE-6G Project. (n.d.). Deliverable D3.3. Retrieved from https://zenodo.org/records/17077365
[44] W3C. (n.d.). WASM W3 working group. Retrieved from https://www.w3.org/groups/wg/wasm
[45] W3C. (n.d.). WASM specifications. Retrieved from https://www.w3.org/TR/wasm-core-2/WASM
[46] Perrone, G., & Romano, S. P. (2024, July). WebAssembly and security: A review.
[47] Michaud, Q., et al. (2024, October). Securing stack smashing protection in WebAssembly applications
[48] She, X., et al. (2024) WaDec: Decompiling WebAssembly Using Large Language Model
[49] Werner, B. (n.d.). WasmRev. Retrieved from https://github.com/benediktwerner/rewasm
[50] Fang, W., et al. (2024). StackSight: Unveiling WebAssembly through large language models and neurosymbolic chain-of-thought decompilation.
[51] Lacoste, M., & Lefebvre, V. (2023). Trusted execution environments for telecoms: Strengths, weaknesses, opportunities, and threats. IEEE Privacy and Security Journal.
[52] ETSI. (2019). GR NFV-SEC 018 V1.1.1: Network functions virtualisation (NFV); Security; Report on NFV remote attestation architecture.
[53] AMD. (n.d.). SEV-NP TEE user documentation. Retrieved from https://www.amd.com/fr/developer/sev.html
[54] OWASP. (n.d.). STRIDE methodology for threat modeling. Retrieved from https://owasp.org/www-community/Threat_Modeling_Process#stride