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

D4.1 Payload security per runtime, intelligent runtime selection and attestation.r1

Lead beneficiary	IMEC	**Lead author**	Merlijn Sebrechts
Reviewers	Nasim Nezhadsistani (UZH), Joaquin Escudero (GRAD)		
Type	R	**Dissemination**	PU
Document version	V1.0	**Due date**	31/03/2025

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	WP4
Associated task(s)	T4.2
Main Author(s)	Merlijn Sebrechts
Author(s)	Vincent Lefebvre, Mark Angoustures (TSS), Tom Goethals (IMEC), Nasim Nezhadsistani (UZH), Jordi Thijsman (IMEC), Shankha Gupta, Mays Al-Naday, Sumeyya Birtane (UEssex), Ioanna Kapetanidou, Sarantis Kalafatidis, Antonios Lalas, Anastasios Drosou (CERTH), Joaquin Escudero (GRAD), Joachim Schmidt, Leonardo Padial, Eryk Schiller (HES-SO)
Reviewers	Nasim Nezhadsistani (UZH), Joaquin Escudero (GRAD)
Type	R – Document, Report
Dissemination level	PU – Public
Due date	M15 (31/03/2025)
Submission date	31/03/2025

Document version history

Version	Date	Changes	Contributor (s)
v0.1	6/11/2024	Template ready	Merlijn Sebrechts (IMEC)
v0.2	20/11/2024	Initial table of contents	Merlijn Sebrechts (IMEC)
v0.3	19/02/2025	Details for early sections, elaborated content structure. 3.1/3.2/4.1 content. Basis for 2.2. Raw content of 3.3/3.4.	Tom Goethals, Jordi Thijsman (IMEC)
v0.4	04/03/2025	Raw content of 2.3	Ioanna Kapetanidou, Sarantis Kalafatidis, Antonios Lalas, Anastasios Drosou (CERTH)
v0.5	05/03/2025	Structuring supplied content, intro sections	Tom Goethals (IMEC) Vincent Lefebvre, Mark Angoustures (TSS)
v0.6	10/03/2025	Integration before review	Tom Goethals (IMEC)
v0.7	18/03/2025	Review and comments added	Nasim Nezhadsistani (UZH), Joaquin Escudero (GRAD)
v0.8	26/03/2025	Integration of changes after review	Tom Goethals (IMEC)
v0.9	28/03/2025	Final Quality Assurance	Joachim Schmidt. Leonardo Padial, Eryk Schiller (HES-SO)
v0.95	30/03/2025	Final review and refinements	Antonios Lalas (CERTH) and all authors
v1.0	31/03/2025	Final version for submission	Antonios Lalas (CERTH)

Disclaimer

Copyright message

Contents

List of acronyms and abbreviations

Abbreviation	Description
AI	Artificial Intelligence
AMF	Mobility Management Function
ASLR	Address Space Layout Randomization
CLI	Command Line Interface
CNI	Container Network Interface
CRTM	Core Root of Trust for Measurement
CTI	Cyber Threat Intelligence
DCAP	Data Center Attestation Primitives
DEP	Data Execution Prevention
DoS	Denial of Service
D-MUTRA	DLT-based Mutual Remote Attestation solution
eBPF	extended Berkeley Packet Filter
EDA	Exploratory Data Analysis
HSM	Hardware Security Module
KVM	Kernel-based Virtual Machine
LSTM	Long Short-Term Memory
OAM	Open Application Modeling
OCI	Open Container Initiative
PCR	Platform Configuration Registers
RBAC	Role Based Access Control
REST	Representational State Transfer
SCTP	Stream Control Transmission Protocol
SDN	Software Defined Network(ing)
SECaaS	Security as a Service
SEV	Secure Encrypted Virtualization
SEV-SNP	Secure Encrypted Virtualization Secure Nested Paging
SGX	Software Guard Extensions
TCB	Trusted Computing Base
TDX	Intel Trust Domain Extensions
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
UDP	User Datagram Protocol
UE	User Equipment
UPF	User Plane Function
VM	Virtual Machine
vNICs	Virtual Network Interface Controller
WASI	WebAssembly System Interface
WASM	WebAssembly
vEth	Virtual Ethernet Device

List of figures

Executive summary

The deliverable D4.1 “Payload security per runtime, intelligent runtime selection and attestation.r1” provides a comprehensive overview of the NATWORK aspects enabling secure, flexible-runtime workloads. This is actually a report on the payload security per runtime, the intelligent runtime selection as well as the remote attestation, that derive from NATWORK innovations. The current deliverable (first version) derives from the work performed under the Task 4.2 “AIaaSecS for software payload”.

While containerized software and virtual machines have been employed for years in software orchestration, many security aspects of containers and security optimization through orchestration have been neglected in research and frameworks. On the one hand, containers are merely processes that are slightly isolated but not intrinsically secure. Alternative runtimes such as microVMs and WebAssembly WASM System Interface (WASI) have become popular, promising superior workload security and optimal performance. Still, their exact contributions are unclear, and integrating them into container-based software orchestration is non-trivial. The optimal runtime type should also be chosen for each payload based on security (and other) requirements.

On the other hand, several hardware-based approaches, such as Trusted Execution Environments (TEEs) and Trusted Platform Modules (TPMs), may be leveraged to secure deployed software through remote attestation by certifying nodes to be secure and reliable for the execution of sensitive workloads.

This document covers three aspects of enabling secure, flexible-runtime workloads. First, an overview of the most promising execution formats and runtimes, including unikernels and WASM, is provided. The second, intelligent runtime selection, involves selecting a suitable runtime for a specific workload based on particular parameters and workload properties and unifying network and orchestration aspects required for transparent, runtime-agnostic operation. Finally, the third aspect provides the building blocks for a feasible, flexible remote attestation framework. WASM payload security is discussed similarly to remote attestation and beyond during the payload execution. We discuss the core merits of security technology and emphasize bytecode tampering at runtime. The project covers this major weakness with a modified WASM runtime and a whole stack integrated two-layer remote attestation schema covering both the runtime and the payload.

It should be noted that this is the first version of the deliverable which will continue to evolve in the second year of the project, and will be described in its final version by the D4.2 “Payload security per runtime, intelligent runtime selection and attestation.r2” due to M24 and in accordance to the project’s description of work.

1. Introduction

Containerized software and virtual machines have been employed for years in software orchestration in the cloud and edge. However, research and frameworks have neglected many security aspects of both containers and security optimization through orchestration. This is a significant shortcoming for 6G frameworks, as edge networks are more vulnerable to attack vectors than physically and digitally secured data centers, and 6G inherently spans the entire continuum from the cloud to the edge.

In security terms, containers are merely processes slightly isolated from the rest of the operating system but not inherently secure. However, containers are not the only option for executing workloads (i.e., "runtime"). Alternative runtimes such as microVMs and WebAssembly (WASM) System Interface (WASI) have become popular, promising superior workload security and optimal performance. Still, their exact contributions are unclear, and integrating them into container-based software orchestration is non-trivial, while one has to define the optimal runtime for each payload based on security (and other) requirements.

On the other hand, several hardware-based approaches, such as Trusted Execution Environments (TEEs) and Trusted Platform Modules (TPMs), may be leveraged to secure deployed software through remote attestation by certifying nodes to be secure and reliable for the execution of sensitive workloads.

1.1. Purpose and structure of the document

This document covers the main aspects researched in T4.2 to enable secure, flexible-runtime workloads. In summary, these are:

- An overview of the most promising execution formats and runtimes, including unikernels and WASM. This includes state-of-the-art capabilities, resource use, and essential security aspects.
- Intelligent runtime selection, which is split into "enabling runtime selection" and "intelligent selection". The former details how various runtimes can be integrated into a uniform structure/API based on the most popular orchestration frameworks (e.g., Kubernetes) in preparation for intelligent intent-based selection. The latter considers workload and runtime properties, determining the optimal workload for a specific task and matching it to a suitable execution node.
- Building a remote attestation framework that ensures secure execution of the selected workloads by leveraging node hardware capabilities such as TEE and TPM.

The rest of the document is structured according to these three main aspects:

- **Section 2** provides state-of-the-art security aspects of the most promising execution formats and runtimes for deployable workloads. This includes execution and image payload security for Virtual Machines (VMs), containers, microVMs, and WASM. The inception of WASM technology is recalled in the context of security.
- **Section 3** covers the intelligent, intent-based selection of a suitable runtime for a specific workload based on particular parameters and workload properties, in addition to standardization and unification efforts in terms of networking and orchestration APIs. Finally, this section covers ML-based workload modeling in the context of resource optimization.
- **Section 4** describes TEE and TPM-based attestation methods as the building blocks for a scalable, flexible attestation framework. Moreover, WASM payload remote attestation is discussed, exploiting a novel runtime integrity verification scheme through a modified WASM runtime.

1.2. Intended Audience

NATWORK's D4.1 deliverable (Payload security per runtime, intelligent runtime selection, and attestation) is devised for the internal use of the NATWORK consortium, comprising members, project partners, and affiliated stakeholders. This document mainly focuses on the fundamental security aspects of software runtimes and hardware attestation required for the project, thereby serving as a referential tool throughout the project's lifespan.

This document contains sensitive context and is restricted exclusively to the consortium's collective entities and European Commission (EC) representatives. Also, the document highlights the project's strategic blueprint and collective vision, ensuring that all collaborative efforts are harmonized and directed toward fulfilling the project's ambitions.

Dissemination or disclosure of the contents herein is limited to the internal circles of the NATWORK consortium and the EC to maintain confidentiality and project integrity.

1.3. Interrelations

The NATWORK consortium integrates a multidisciplinary spectrum of competencies and resources from academia, industry, and research, 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 D4.1 deliverable addresses all activities of the NATWORK project related to T4.2 directly and as supporting activities for T3.1 (optimizing selection and embedding of AI security services, i.c.w. UESSEX, CERTH, and ZHAW as main stakeholders) and UC1 (Sustainability and reliability of 6G Slices and services i.c.w. UESSEX, ISRD and TSS).

2. Runtime overview & security

This section covers a state-of-the-art overview of critical aspects of various runtimes related to T4.2: security properties, relative resource use, ease of use, and compatibility with existing systems and software.

2.1. Virtual Machines

In today's digital landscape, ensuring robust runtime payload security in virtual machine (VM) environments is critical. The increasing adoption of virtualization technologies across cloud, edge, and data center infrastructures has introduced significant challenges in maintaining the confidentiality, integrity, and availability of sensitive workloads. Modern security strategies rely on an integrated approach that combines advanced hardware features, TEEs, and comprehensive software-based defenses. This discussion covers the latest developments in VM security, focusing on hardware-assisted virtualization, TEEs, network and storage protections, real-time monitoring, and emerging trends such as the unikernel technology. Hardware-assisted virtualization technologies like Intel Virtualization Technology for x86 (VT-x) and AMD Virtualization (AMD-V) are at the foundation of VM security, which provide essential isolation between guest VMs and the host system [1]. These technologies leverage hardware-level features to enforce strict separation, ensuring that operations within one VM cannot interfere with or compromise another. Recent advancements, such as the incorporation of nested page tables and extended page table protections, have further strengthened these measures by limiting the ability of malicious entities to exploit memory access flaws or execute cross-VM attacks [2]. Such enhancements improve the fundamental security posture and serve as critical enablers for implementing higher-level security mechanisms.

TEEs have emerged as a crucial technology for protecting sensitive payloads during runtime. Technologies like Intel Software Guard Extensions (SGX), Intel Trust Domain Extensions (TDX), and AMD Secure Encrypted Virtualization (SEV) create secure enclaves within the processor, allowing data and code to be executed with enhanced confidentiality and integrity even in scenarios where the broader operating environment might be compromised [3]. By enforcing memory encryption and implementing rigorous attestation protocols, TEEs verify the integrity of the executed code and protect against sophisticated threats, including side-channel attacks and memory tampering. Although these techniques have proven effective, they introduce performance overheads and implementation challenges that require careful balancing during system design and deployment.

Securing network communications inside virtualized environments is another critical aspect of runtime payload security. Software-Defined Networking (SDN) technologies enable administrators to implement fine-grained control over VM communication patterns, ensuring that only authorized traffic crosses virtual boundaries [4]. In addition, virtual network interface controllers (vNICs) equipped with hardware-assisted packet processing bolster security and performance. Techniques such as micro-segmentation and dynamic access control policies further limit the lateral movement of potential attackers, effectively containing breaches and enhancing overall network security. Storage security for VM payloads has evolved in response to emerging threats and compliance requirements. When combined with robust key management systems, full disk encryption technologies ensure that sensitive data remains confidential even if physical media are compromised [5]. Modern security practices increasingly integrate hardware security modules (HSMs) to safeguard encryption keys and other critical configuration data, helping to secure VM migration, backup operations, and adherence to industry regulations. These measures are particularly vital in environments where data confidentiality and integrity are critical, such as in regulated or sensitive information sectors.

Modern VM environments incorporate sophisticated runtime monitoring and threat detection mechanisms to bolster security further. Advanced intrusion detection systems now combine traditional signature-based methods with machine learning (ML)-driven behavioral analysis to identify known and novel threats in real time [6]. The development of automated response mechanisms has streamlined threat mitigation, enabling rapid system responses to potential security incidents. Nonetheless, challenges remain in minimizing false positives while maintaining high detection accuracy and overall system performance. Recent research into unikernel technology and lightweight virtualization has introduced a promising new paradigm for enhancing runtime payload security. Unikernels, which combine the essential functions of the application and operating system into a single executable, significantly reduce the footprint of code executed in privileged modes, thereby minimizing the attack surface. This streamlined approach offers substantial security benefits, particularly in cloud-native environments, though the operational complexity and compatibility challenges associated with unikernels continue to be active research areas [7].

Integrating security features through APIs allows different VM software services, containers, and other infrastructure components to dynamically interact and share threat intelligence. As isolated security devices detect potential threats, such as Distributed Denial of Service (DDoS) attacks, they can communicate this information to services running on virtual machines or across containers via APIs. This integration enables the system to automatically isolate compromised VMs, migrate critical services to secure environments, or trigger other countermeasures. Additionally, the feedback generated from these defensive measures provides continuous data

to the entire network, allowing the system to refine its security protocols. This feedback loop ensures that VM software and services align with the most up-to-date defense strategies, enhancing the network's resilience and allowing faster, more effective responses to emerging threats (cf. Section 3.3).

2.2. MicroVMs

MicroVMs are a lightweight form of VM designed to run individual processes. As VMs, these can leverage all the security benefits described in Section 2.1. As such, the rest of this section details the specific properties of microVMs. Figure 1 illustrates the architectural differences between various types of (micro)VMs and containers.

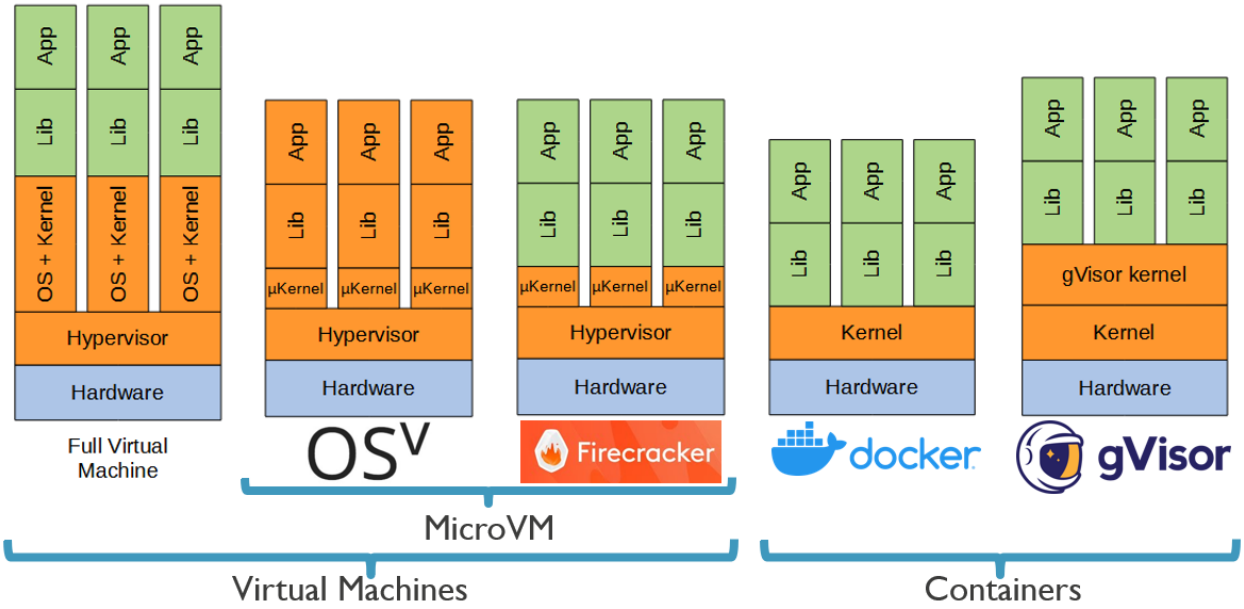

Figure 1: conceptual comparison of several runtime options. Userspace is marked in green. While gVisor is not discussed in the rest of the text due to lackluster performance, it presents an interesting architecture using a custom system interface.

Several technologies enable the creation of microVMs, among which unikernels are a varied group with excellent security and performance features [8],[9]. Specifically, unikernels are specialized operating system libraries that compile an application with only the necessary system components into a single executable, which runs inside a virtual machine. This reduces both the image size and the attack surface.

Apart from the kernel- versus userspace designs, microVMs can be roughly classified into two types: POSIX-compatible (Portable Operating System Interface [10]) ones that focus on existing software, and those based on non-POSIX system interfaces which sacrifice compatibility for smaller images and lower resource requirements. OSv [11] is a POSIX-compatible [10] unikernel

platform with wide compatibility for existing programs and programming language runtimes. Although microVMs generally support a wide variety of hypervisors for their execution, QEMU[12] with KVM (Kernel-based Virtual Machine [12], [13]) acceleration is a widely endorsed option due to its Type I hypervisor capabilities and integration with Linux.

Given the diversity in microVM architectures and hypervisor support, evaluating their performance and trade-offs across different virtualization technologies becomes crucial. Therefore, various classes of virtualization technologies, including microVMs, have been extensively compared and benchmarked [14], and their performance has been examined at the kernel level [15].

2.2.1. Non-POSIX

This type of microVM is designed from the ground up around a single programming language, generally a C dialect, providing a custom (not POSIX compatible) system interface. Unikernels most often use this approach: at compile time, all the libraries and system calls the program uses are compiled, together with the program itself, creating a single kernel that starts on boot. Platforms of this type are generally incompatible with existing source code, often requiring developers to rewrite applications entirely using platform-specific APIs. Such rewrites can introduce security risks, especially if critical security features are omitted or reimplemented incorrectly. These issues are particularly pronounced on ARM-based edge devices, where limitations like missing system calls, lack of 64-bit support, or ELF format incompatibilities further complicate development. Considering the still-evolving nature of existing platforms of this type, it also often means dropping functionality that is not yet implementable [23]. On the other hand, this type of microVM results in lower CPU and memory use and far smaller images than POSIX-compatible ones [17]. Direct security benefits are usually limited to “security through obscurity” and a reduced attack surface by modifying the kernel. IncludeOS[1] [22] and MirageOS[2] are good examples of this approach.

2.2.2. POSIX

2.2.2.1. Runtimes

MicroVM runtimes, such as Firecracker[3], are small hypervisors designed to minimize the footprint and enhance the security of the virtualization layer. Unikernels reduce both the attack surface

[1] https://www.includeos.org/ - IncludeOS - Run your application with zero overhead
[2] https://mirage.io/ - A programming framework for building type-safe, modular systems
[3] https://firecracker-microvm.github.io/ - Secure and fast microVMs for serverless computing

and the overhead of virtual machines by implementing only minimal virtual device drivers and omitting all non-essential functionality. They can achieve performance close to that of native processes, such as high Representational State Transfer (REST) API throughput under CPU load, by leveraging para-virtualized devices like virtio [18] and hardware-assisted virtualization technologies such as Intel VT-x and AMD-V [19] [73].

Guest operating systems often require minor changes to run in a minimal environment. The smallest, most optimized unikernel platforms, such as UniK[4] and Unikraft, provide custom system interfaces and dedicated programming languages. The latter produces unikernels of only around 1 MB for essential web functions such as Nginx, SQLite, or Redis [8].

2.2.2.2. Image formats & platforms

In addition to runtimes, several specialized VM guests and image formats have emerged. These fall into two categories: the first category runs existing software on a highly minified Linux kernel, such as the default Firecracker kernel [20][20]. While this approach ensures near-perfect compatibility with most Linux software and system calls, the need to support a broad set of Linux APIs and architectures—combined with the typically straightforward "transpiling" of existing libraries—limits how small these kernels can become. However, using custom kernels with reduced API support is often possible, which can further minimize both the image size and the attack surface. Unikernels represent the second category, in which the application is compiled together with only the necessary operating system components into a monolithic binary that runs entirely in kernel mode. This improves performance by removing context switches, providing smaller images, and reducing the attack surface [21]. POSIX-compatible unikernels, such as OSv[5], can integrate existing POSIX executables, such as Linux software, or compile software in various programming languages into a unikernel from scratch without modification (e.g., Rust, Go, C). However, compatibility and potential security issues depend highly on the active community maintaining such kernels, and most initiatives still focus on compatibility rather than specific security improvements. OSv, for example, has several significant issues that supersede security concerns [14], many of which are shared by other platforms:

- Multithreading: multithreaded REST services, specifically configured with four handler threads, result in up to 20% performance loss compared to single-threaded unikernels rather than an expected (up to 300%) increase. While this issue has been known for years, it remains unresolved.

[4] https://github.com/solo-io/unik - A platform for automating unikernel & MicroVM compilation and deployment
[5] https://github.com/cloudius-systems/osv - OSv, a new operating system for the cloud

- Multicore operation: like multithreading, allowing a unikernel to use multiple cores results in a significant performance hit rather than an improvement, indicating that it is not merely a thread scheduling issue on single cores.
- Reliability: when performing essential REST server/client evaluations, some requests get "stuck" while processing, resulting in extreme lag, which is orders of magnitude beyond that of running the same services in containers. This is the case even with single-threaded operations.
- Stability: After handling a certain (random) number of REST requests, unikernels often crash, independent of the actual REST service they provide. This is inconvenient in evaluation scenarios but unacceptable in production environments.
- Hardware platforms: ARM support may be seemingly random, especially for languages such as Golang, which tend to change the specific syscalls they use on any platform, depending on the language version.
- Resource use: memory and CPU requirements vary wildly depending on the scenario; a Minecraft server with a large world uses 25% less memory than its containerized equivalent but uses up to 30% more CPU. For REST services, this is inverted, with higher request throughput at the cost of the virtualization layer using an order of magnitude more memory than a containerized version.

2.2.3. Image & payload security

Apart from often quoted security benefits due to a reduced kernel attack surface and VM-based process isolation, little research has investigated other security aspects of microVMs. However, SEVeriFast [16] proposes a solution to run microVMs using AMD SEV with 86-93% performance improvements, in terms of boot times, over the state-of-the-art. On the other hand, recent work shows that to preserve kernel size, unikernels may skip important yet fundamental security features such as Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and Non-eXecutable (NX) bits [23].

2.3. Containers

Containerized services are prevalent in 5G networks and are considered a key enabler in 6G networks. Although they offer significant benefits such as portability, enhanced resource usage, and scalability, they also introduce inherent vulnerabilities that raise new security issues [24].

Several security strategies to mitigate risks have been explored, including isolation mechanisms, static image analysis and hardening, and network security policies. Even though these methods enhance security in different ways, they each have inherent limitations.

More specifically, isolation mechanisms, including namespaces and control groups (cgroups[6]), ensure that each container operates in its restricted environment, minimizing the risk of cross-container interference [25]. However, although effective isolation can provide security to some extent, it is not sufficient to prevent DoS attacks that are executed by consuming excessive CPU or memory [26].

In static image analysis, relevant tools inspect the container images to detect any potentially exploitable misconfigurations, while in image hardening, unnecessary packages are removed to strengthen the containers' security posture. Nonetheless, static analysis methods are ineffective against runtime nor zero-day attacks, whereas image hardening is subject to causing implications on the service's normal execution. Finally, network security policies restrict attackers' access to the service by enforcing access control rules, though they introduce additional execution overhead. [27] provides a more thorough review of related works that explore various container defense mechanisms.

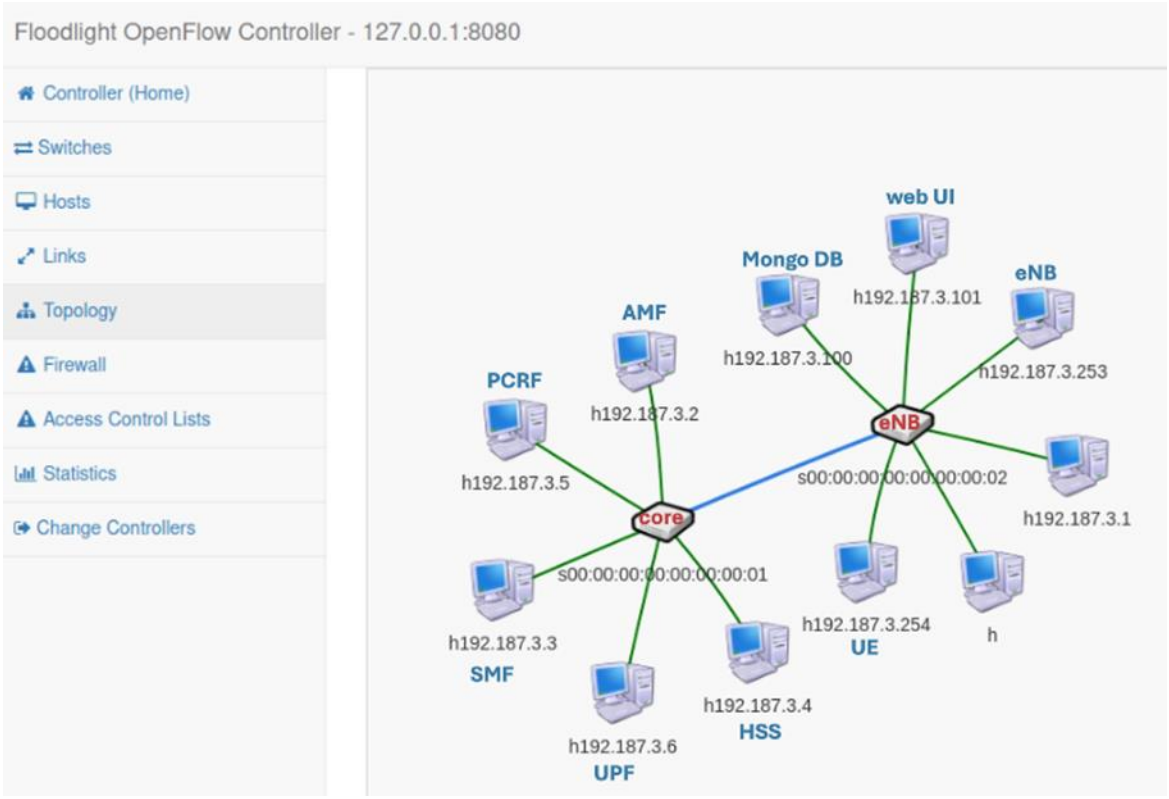

Figure 2: Containerized 5G topology

A recent approach introduces AI-based runtime monitoring, anomaly detection, and mitigation. This technique is more advanced and is suitable for mitigating container security threats at runtime. Motivated by this, NATWORK is developing an AI-powered monitoring and mitigation

[6] https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html

tool. This tool is built upon and extends previous works [28] to enable continuous analysis of system behavior, detecting anomalies, and responding in real time to contain any potential attacks efficiently and promptly. A 5G network topology is deployed for experimentation and validation using open-source implementations such as free5GC[7] and Open5GS[8], which aim to provide standards-compliant 5G core networks. To support standalone (SA) 5G core functions—such as the Access and Mobility Management Function (AMF) and the User Plane Function (UPF)—the projects mentioned above offer a microservices-based architecture, where each network function operates in its container.

SDN controllers and switches—such as Floodlight[9] or Container Network Interfaces (CNIs)[10]—facilitate communication between the containers and simulated 5G User Equipment (UEs). Figure 2 illustrates one example of such a topology. The computing and network resources consumed by each container are monitored in real-time, as shown in Figure 3.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
697b48c1baf2	att	0.00%	2.523MiB / 3.82GiB	0.06%	0B / 0B	6.1MB / 0B	1
1aadbfc941bf	ue	0.00%	87.94MiB / 3.82GiB	2.25%	0B / 0B	42.9MB / 12.3kB	8
223b756fd452	enb	0.01%	353.4MiB / 3.82GiB	9.03%	0B / 0B	29.7MB / 0B	13
e54fd3baec9a	webui	0.00%	126.5MiB / 3.82GiB	3.23%	336kB / 33.7kB	54.3MB / 16.4kB	23
19926e13a6d1	pcrf	0.01%	26.77MiB / 3.82GiB	0.68%	0B / 0B	5.12MB / 55.2MB	29
5e63ef5b39dc	hss	0.01%	21.68MiB / 3.82GiB	0.55%	0B / 0B	4.71MB / 55.2MB	29
4a50ca153d19	smf	7.30%	74.01MiB / 3.82GiB	1.89%	0B / 0B	14.5MB / 55.2MB	36
b3e05e3a49f5	upf	5.81%	97.14MiB / 3.82GiB	2.48%	0B / 0B	107MB / 55.2MB	4
cc954cbc82b4	amf	8.39%	74.77MiB / 3.82GiB	1.91%	0B / 0B	225MB / 52MB	37
308d710cd3a2	mongodb-svc	0.11%	47.41MiB / 3.82GiB	1.21%	42.6kB / 210kB	45.5MB / 807kB	33

Figure 3: Resource consumption statistics per container

This containerized architecture enables us to evaluate the tool's performance against various types of Denial of Service (DoS) attacks, including User Datagram Protocol (UDP) Flooding attacks on the UPF and Stream Control Transmission Protocol (SCTP) Flooding attacks on the AMF. Such attacks aim to exhaust the target network resources. Considering the above, the AI-powered tool operates as follows:

First, it utilizes packet sniffers (cf. Figure 4), such as tcpdump[11] and Wireshark[12]To capture network traffic of container interfaces in real-time across several transport layer protocols. UDP and SCTP were selected because of their widespread use in 5G/B5G networks.

It also monitors all active flows and their traffic in the network setup. Different traffic patterns are simulated using tools such as Apache JMeter[13], to create realistic attack scenarios.

[7] https://free5gc.org/
[8] https://github.com/open5gs/open5gs
[9] https://github.com/floodlight/floodlight
[10] https://github.com/containernetworking/cni
[11] https://www.tcpdump.org/
[12] https://www.wireshark.org/
[13] https://jmeter.apache.org/

```
tcpdump: listening on core1-eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
1000 packets captured
1226 packets received by filter
0 packets dropped by kernel
tcpdump: listening on core1-eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
1000 packets captured
1140 packets received by filter
0 packets dropped by kernel
tcpdump: listening on core1-eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
1000 packets captured
1426 packets received by filter
0 packets dropped by kernel
```

Figure 4: Packet sniffing using tcpdump

Based on the information collected on network traffic and flows, the tool performs online anomaly detection using a moving average algorithm. An attack is detected when an unexpected surge in traffic is generated by a particular flow, indicating the potential activity of a flooder. This case is shown in Figure 5.

```
No. active flows:  9
No. active flows:  11
No. active flows:  15
No. active flows:  15
No. active flows:  25
No. active flows:  25
No. active flows:  30
No. active flows:  168
No. active flows:  163
No. active flows:  151
No. active flows:  75
No. active flows:  24
No. active flows:  3
No. active flows:  3
```

Figure 5: Number of active flows captured at runtime

Upon detection, the tool immediately identifies the IP of the attacking flow. Subsequently, a new control rule is created to restrain incoming packets from the suspicious flow, thus effectively mitigating the attack, as illustrated in Figure 6.

```
alarm=True
2025-03-21 11:02:27.137
*******AN ATTACK HAS BEEN DETECTED*******
Details: The attacker's IP is: 192.187.3.200
*******ADDING NEW FLOW CONTROL RULE FOR ATTACK MITIGATION*******
Restricting flows from IP: ''192.187.3.200''
{"src-ip":"192.187.3.200/32","action":"deny"}
{"status" : "Success! New rule added."}
```

Figure 6: Online attack detection & mitigation

2.4. WebAssembly

2.4.1. Origin and core design objectives

WASM[14] was created as a collaboration between major browser vendors, including Mozilla, Google, Microsoft, and Apple, under the World Wide Web Consortium (W3C) WebAssembly Working Group[15]. It was first announced in 2015 and became an official W3C standard in 2019[16]. The core reasons behind WASM's creation were:

I. **Performance**: JavaScript[17] technology, while powerful, has inherent performance limitations, particularly for compute-intensive tasks such as gaming, video processing, and data visualization. WASM was designed to enable near-native execution speed by using a binary format that is more efficient to parse and execute. This binary format is structured on low-level instructions, which are closer to native instructions and, therefore, faster to interpret.

II. **Security**: WASM was designed with a sandboxed execution model, ensuring that WebAssembly code runs within the same security constraints as JavaScript in the browser. This makes it harder for WASM code to escape and compromise a system. Security derives from several memory designs:

   - Contiguous (i.e., linear) memory allocation where each WASM payload is allocated with its access-restricted memory map, preventing overwrite beyond this area (e.g., overflow for escalation)
   - No direct memory pointers (as enabled by native assembly), preventing direct jump to read and write anywhere
   - Control-flow integrity, enforced by the interpreters, prevents unvalidated indirect calls and mitigates control-flow-based attacks such as Return-Oriented Programming (ROP), self-modifying code, or attacks that exploit Just-In-Time (JIT) compilation to inject malicious code dynamically.

III. **Portability**: WASM is not tied to JavaScript—it is a low-level bytecode format that can be compiled from multiple languages (e.g., C, C++, Rust, Go), enabling developers to run high-performance code on the web without relying on JavaScript. From the outset, WASM was designed to be a polyglot platform. It can serve as a common

[14] https://github.com/webassembly
[15] https://www.w3.org/groups/wg/wasm/
[16] https://www.w3.org/TR/wasm-core-2/
[17] https://github.com/topics/javascript

compilation target for many programming languages rather than being limited to a single language or ecosystem.

IV. **Interoperability**: WASM was designed to work alongside JavaScript, not replace it. This allows developers to use WASM modules within JavaScript applications, leveraging its speed for performance-critical parts.

Although initially developed by browser vendors, WASM technology has expanded into other domains, including telecom, where it is used in Function-as-a-Service (FaaS) platforms like CloudFlare Workers[18]. These platforms enable serverless execution of code in WASM, allowing lightweight, high-performance functions to run on demand in the cloud. WASM is also viewed as a serious competitor against containers. WASM is also considered in gaming (e.g., Unity[19]), blockchain (Ethereum 2.0 WASM smart contracts[20]), and AI (e.g., Pyodide[21]). Browser vendors faced JavaScript's limitations in both security and performance. JavaScript high abstraction level made it slow to run, easy to reverse, and tamper. As importantly, its high-speed engine (i.e., the Just-In-Time compiler) was exploited, as demonstrated in a real-world attack [76]. The core prototype inheritance structure (i.e., the backbone of JavaScript object-oriented programming) was also an attack pathway, as illustrated in another real-world attack [77]. JIT spraying and prototype pollution are the two main attack pathways taken by attackers on JavaScript and are difficult to apprehend and prevent for developers.

While keeping the same level of portability as offered by JavaScript (i.e., through virtualization/interpretation), the WASM conceptor team has developed a lower-level interpreted language, more complex to reverse and faster to execute, also almost closing the door to JIT compilation through a preferred ahead of time compilation mode, notably by browser vendors, reluctant to use JIT for security reasons. JIT is, however, possibly activated (e.g., WASTIME runtime).

2.4.2. WASM security aspects

Several rich surveys of WASM security have flourished since 2020. In [29], the authors have sifted through 121 works covering different security aspects, including vulnerabilities and interactions with the operating system. In essence, primary emphasis is put on the opportunities and threats of the sandbox.

[18] https://github.com/cloudflare/workerd
[19] https://unity.com/
[20] https://ewasm.readthedocs.io/en/mkdocs/
[21] https://pyodide.org/en/stable/

The sandboxed payload isolation, a key promise of WASM, can be eroded by memory-unsafe native language programs (before being compiled as a WASM payload), which, of course, advocates eliminating C, C++, or other memory-unsafe language programs (e.g., Rust programming). Side-channel attacks are potentially robust sandbox data extraction techniques, as they operate within the domain of a trusted execution environment or Linux domain space. On both fronts, side-channel attacks require technology vendors (e.g., CPU manufacturers for TEEs) and software developers (e.g., those working on WASM or Linux distributions) to engage in constant and proactive engineering to mitigate vulnerabilities and strengthen isolation guarantees. While perfect security is unattainable, systems should be hardened to the point where only highly skilled attackers can pose a threat. In practice, reverse engineering and tampering remain common attack vectors, prompting developers to obfuscate most WASM payloads—often at the cost of performance—to protect confidentiality. However, performance is less relevant for attackers aiming to evade detection. In such cases, obfuscated WASM payloads are frequently deployed for illicit use, particularly on the dark web and in crypto mining operations where WASM thrives [30].

The strict memory management of WASM confers a high-security profile in contrast to native programming, as do other interpreted languages (e.g., Java), but in a more advanced manner. The relatively low abstraction level of WASM instructions only raises the bar for good reversers but without attaining any level of certainty. With a significant impact on performance, obfuscation is practiced preventing malware detection rather than intellectual property preservation.

3. Intelligent runtime selection

This section discusses some of the enablers of intelligent runtime selection. The most critical aspects are runtime unification, which ensures that the selectable runtimes are easily interchangeable; network unification, which provides a uniform network environment regardless of the underlying runtime; and standardization, which extends existing standards, e.g., Open Container Initiative (OCI), to other runtimes or applying more generic standards to a unified platform. Finally, workload modeling and resource optimization are considered. Although this work is currently limited to workload modeling, it can be easily extended to runtime-dependent properties, as the same workload over different runtimes is likely to have varying resource usage and optimization parameters.

3.1. Runtime unification

The current state of software orchestration in the network edge is fairly container-centric. Most standards, networking solutions, and plugins are designed for container orchestration, particularly platforms such as Kubernetes (K8s)[22], K3s[23], etc.

Several alternatives have been explored. For instance, one study leverages KubeVirt to deploy and evaluate (micro)VM-based workloads on K8s clusters [31]. While KubeVirt [24] supports virtualization within K8s, significant modifications to the cluster, including custom resources and daemon sets, are required. Other solutions exist, such as Firecracker[25] through Kata Containers[26] or WASM/WASI[27] workloads [32]. However, these approaches are typically designed to support specific workload types rather than offering a unified integration model across diverse execution environments. Throughout T3.1/T4.2, Feather was explicitly developed as a multi-runtime agent for edge computing designed to replace the Kubelet[28] in K8s clusters without requiring any modifications to the cluster. Feather utilizes a Virtual Kubelet as its foundation, receiving commands from Kubernetes through the REST interface. Workload platforms (e.g., container, unikernel) are referred to as “backends,” which may be supported by different runtimes as described in 2.2.2.1 (e.g., QEMU, VirtualBox). A high-level overview of the Feather architecture is presented in Figure 7, where newly developed components are highlighted in dashed red

[22] https://kubernetes.io/
[23] https://k3s.io/
[24] https://kubevirt.io/ - Building a virtualization API for Kubernetes
[25] https://firecracker-microvm.github.io/
[26] https://katacontainers.io/
[27] https://www.spinkube.dev/ - Hyper-efficient serverless on Kubernetes, powered by WebAssembly.
[28] https://virtual-kubelet.io/ - An open-source Kubernetes kubelet implementation that masquerades as a kubelet.

rectangles. The Virtual Kubelet runs a REST API, which receives deployments from K8s; these are forwarded to any registered providers to handle deployments at the K8s pod level. To separate node and pod logic from atomic workloads (i.e., individual containers), Feather implements a provider which takes care of all pod-level logic, leaving the workload-level (or instance) logic to backends (e.g., containerd, OSv). This approach simplifies the complexity of additional backend implementations. It enables mixing different runtimes in a single pod if security requirements require, for example, when trusted execution is necessary for the primary payload but not for a logger and web API clients in the same pod. The provider also interacts with a basic Resource Monitor to report node status and determine if it has enough resources to execute a deployment. For advanced monitoring outside the default K8s dashboard, a Prometheus Golang Exporter[29] may be enabled in addition to the default process metrics. The OSv backend is implemented as a proof of concept for unikernel microVMs, which are run on KVM-QEMU by default.

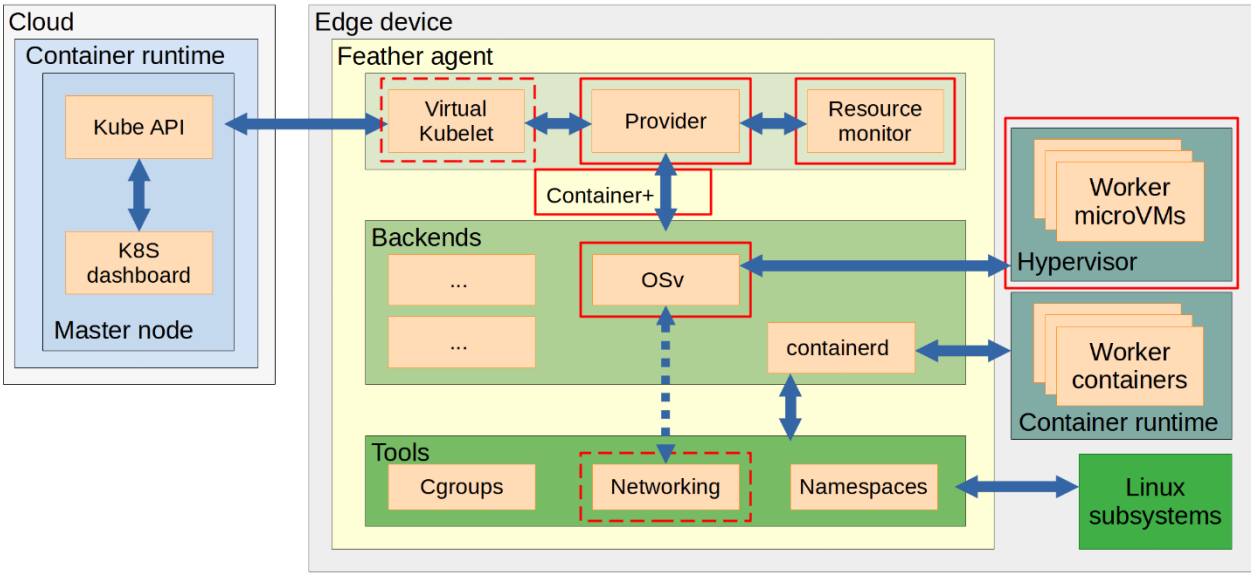

Figure 7: General architecture of the Feather multi-runtime platform.

Initial developments focused explicitly on enabling OSv unikernels within K8s, on edge devices, through a standard Open Container Initiative (OCI) image[30]. To achieve this goal, a second tool called Flint was developed, which can store microVM images inside an OCI image as a single layer, adding metadata information to indicate the type of backend and runtime to be called by Feather. Storing images unaltered inside an OCI image conserves any security mechanisms in place (e.g., encryption), which backends may validate. It enables additional image layers required for payload security protocols. Figure 8 shows the deployment flows for containers and unikernel

[29] https://github.com/prometheus/exporter-toolkit
[30] https://github.com/opencontainers/image-spec

images packed as OCI images within Feather. Key differences are how images are stored locally and loaded into their respective runtimes after unpacking:

- Container images are forwarded to the container runtime (e.g., containerd[31]), which stores the layers on disk and loads them as required when starting an instance.
- Feather extracts OSv images from the OCI image, stored in the OSv cache, and KVM start commands are constructed from deployment parameters to ensure expected functionality (e.g., mounts, networking, environment variables).

There are a few limitations to the current implementation. More importantly, although it can interact with most basic K8s resources (e.g., ConfigMaps[32], Secrets[33]), due to virtio-fs[34] and OSv limitations, the current version of Feather only supports read-only mounts of local directories.

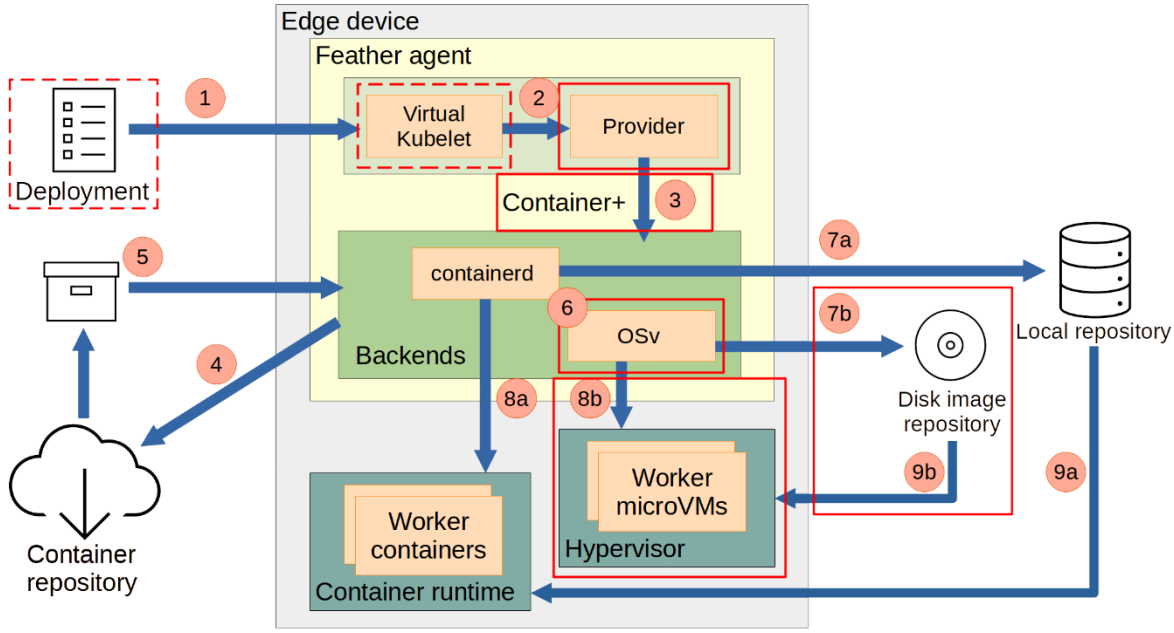

Figure 8: Example of deployment flows for different runtimes in Feather, in this case containers and OSv unikernels.

In addition to runtime security considerations, evaluations reveal notable hardware implications for future alternatives. While OSv unikernels use significantly less memory than containers, combining OSv images with QEMU/KVM introduces substantial processing overhead (cf. Figure 9).

[31] https://containerd.io/
[32] https://kubernetes.io/docs/concepts/configuration/configmap/
[33] https://kubernetes.io/docs/concepts/configuration/secret/
[34] https://virtio-fs.gitlab.io/

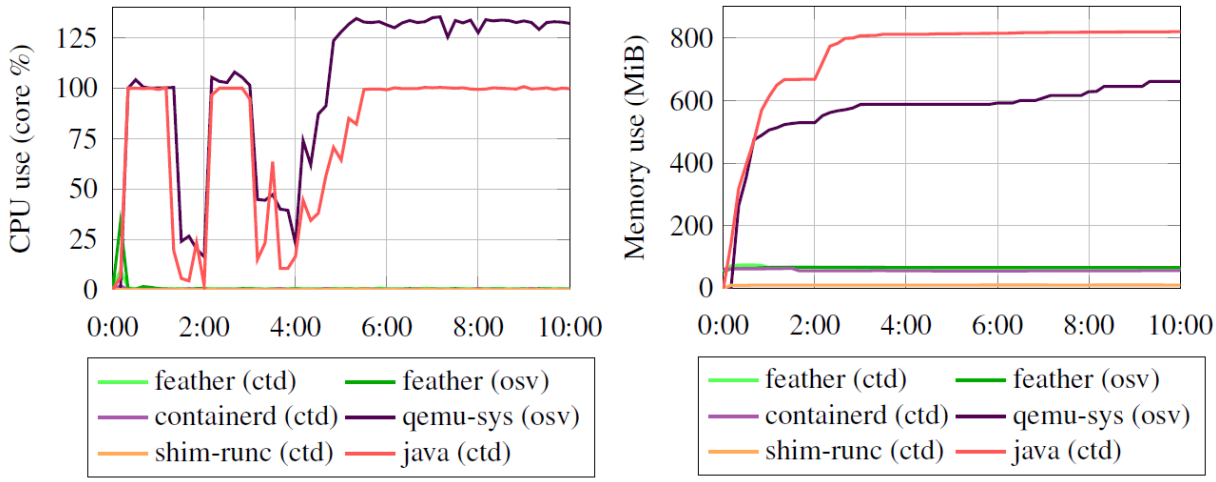

Figure 9: Tradeoff between hardware requirements of containers vs OSv unikernels, to consider along with security aspects.

3.2. Network Unification

3.2.1. Container Networking

Integrating various runtimes into a single platform is non-trivial because orchestration software's practices and standardization efforts have historically focused on containers[35], which are typically combined into pods consisting of multiple containers that form a logical unit of services. A single pod's container normally shares a network namespace, enabling local communication. Standardized pod networks handle network traffic between pods or Container Network Interface (CNI) plugins [33] in the case of Kubernetes (K8s), with implementations focusing on various aspects such as security, performance, and flexibility. Plugin design varies but is usually based on Layer 3 Internet Protocol (IP) routing and/or extended Berkeley Packet Filter (eBPF) for performance, with several alternatives evaluated by Koukis et al. [34] in the context of edge computing. Non-container runtimes can be integrated using various methods; for example, shim implementations such as the WebAssembly OCI-compliant runtime shim (CWASI) [35] may be able to interface directly with a CNI plugin. Kata Containers [36] are implemented using a custom runtime that embeds the containers of an entire pod within a microVM, and the microVM's network interface is attached to the pod's network. MicroVM workloads may integrate with a CNI to some degree. However, evaluations [37] (OSv, Firecracker, etc.) and guides (Nabla[36]) never deploy more than a single workload per pod, nor do they explicitly discuss the (pod) networking architecture. As a microVM uses a single IP address per machine, it is doubtful whether these

[35] https://opencontainers.org/ - Open Container Initiative - an open governance structure for the express purpose of creating open industry standards around container formats and runtimes
[36] https://nabla-containers.github.io/2018/11/05/nabla-k8s/ - Nabla on Kubernetes

solutions support more than a single workload per pod from a networking perspective. Furthermore, many microVM options are designed for FaaS operation in the cloud (e.g., OSv with Firecracker[37]), which, by design, does not require pod networking.

To that end, a runtime-agnostic networking solution was devised to provide mixed workload pods with all the standard behavioral features of pod networking. The core functionality of the proposed solution is sub-pod networking, which sets up small, independent subnets for each pod, rather than providing them with a single IP address. It is based on eBPF programs for efficient and secure traffic processing. The networking solution is integrated into Feather as a configuration option, in addition to IPv4/6 networking schemes and an additional REST API for stand-alone operation (i.e., without the necessity for a Kubernetes cluster).

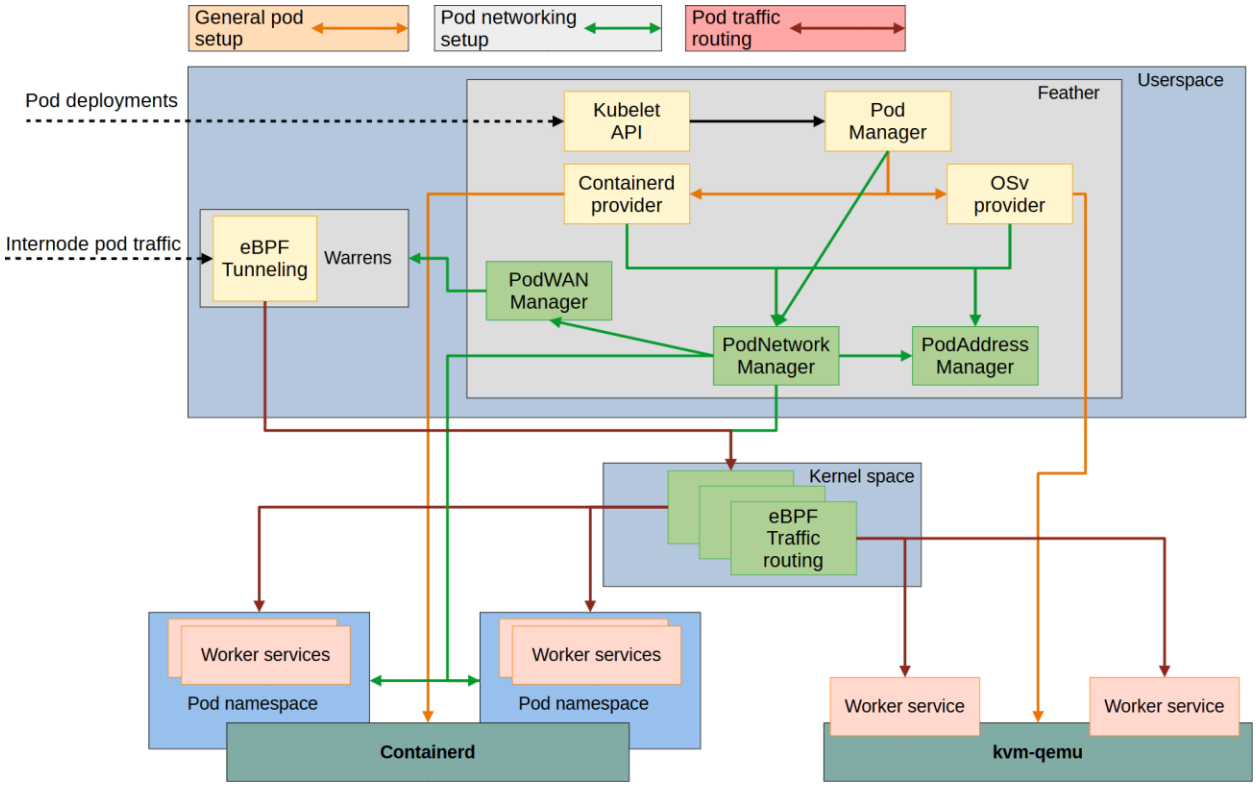

Figure 10: Architecture overview of Feather's multi-runtime networking solution.

Figure 10 shows the high-level component overview of the proposed architecture integrated into Feather. Existing components are indicated in yellow, while novel components are green. The Pod Manager performs General pod management, which receives commands through the Kubelet API. The Pod Manager examines workloads in every pod deployment to determine the runtime providers that should execute them; the Containerd and OSv provider components

[37] https://firecracker-microvm.github.io/

indicate these. However, these components only handle workload image management, resource restrictions (e.g., cgroups), and basic runtime parameters.

Mixed-runtime pod networking is enabled at the pod level by interaction of the Pod Manager with the PodNetwork Manager, which creates a dedicated bridge interface for each pod using the pod IP address and a dedicated pod network namespace connected to the bridge through a veth (Virtual Ethernet Device) pair [38]. Different implementations of this component exist, depending on Feather configuration options. The pod network namespace supports any workload process that can directly access the cni0 network interface inside it (e.g., containers) and can be optionally disabled for performance reasons if no workloads within a pod require it. Any pod workload that requires its interface, e.g., a TAP[39] device for a QEMU VM, is attached to the bridge instead of moving it into the pod network namespace. To support cross-runtime networking, providers must call either a PodNetwork Manager function, which assigns a workload to the pod network namespace, or a function that creates a sub-pod ready TAP device for a workload to use; in Feather the Containerd provider adds containers to the pod network namespace, while the OSv provider requests TAP devices. Feather provides a flexible interface to integrate additional runtimes as providers (e.g., WASM/WASI).

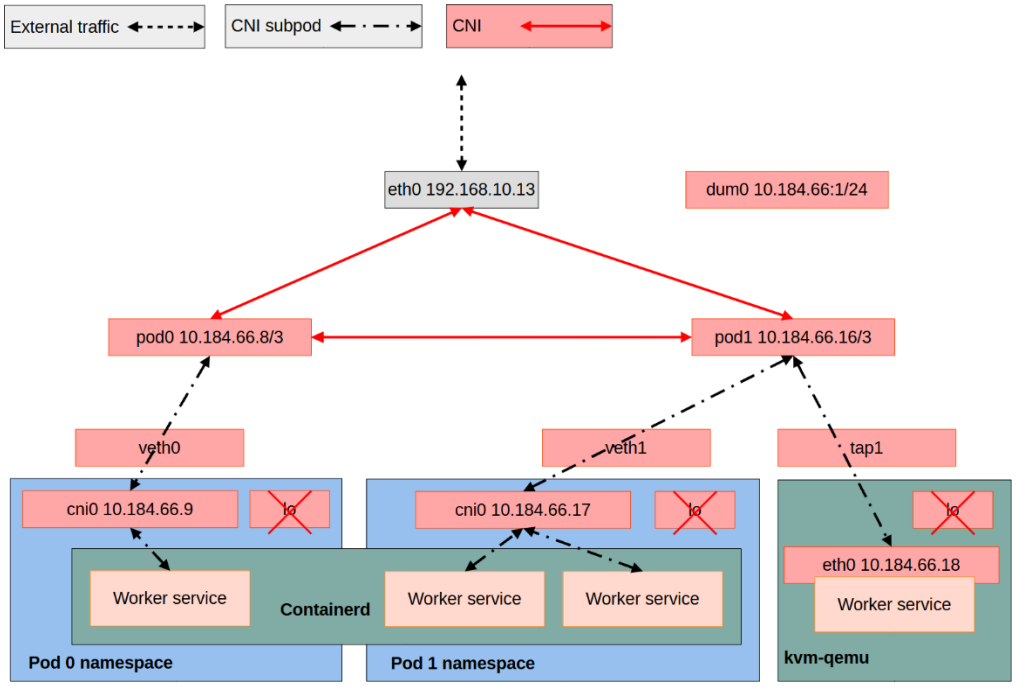

Figure 11: Example of network devices and connections within and between multi-runtime pods.

As the PodNetwork Manager is workload agnostic, any network configuration required by the runtimes is instead passed to them by Feather Providers. This allows for flexible network

[38] https://man7.org/linux/man-pages/man4/veth.4.html
[39] https://docs.kernel.org/networking/tuntap.html - Universal TUN/TAP device driver

implementations but has no measurable impact on performance compared to containers, i.e., a small setup cost per workload to create a network interface and attach an eBPF program. In addition to pod and workload level network initialization, the PodNetwork manager is responsible for sub-pod networking. This concept uses custom eBPF traffic routing programs to ensure optimal performance and disguise sub-pod networking from the point of view of other pods and K8s nodes. The lifecycles and routing tables of these programs are handled by the PodNetwork manager based on the interaction between the Feather Provider and the PodAddress manager. Finally, the PodNetwork manager interacts with the PodWAN Manager to route pod traffic to other nodes. In the example architecture, Warrens [38] is used for this purpose. Still, configuration options also allow the PodWAN Manager to directly route traffic to a suitable network interface for compatibility with Kubernetes clusters, expecting default CNI behavior.

A concrete example of the networking components created by this architecture is shown in Figure 11, in which two pods are deployed on a Feather-managed node; pod0 consists of a single container, while pod1 consists of two containers and an OSv unikernel. Three types of network traffic must be managed in this example; ``localhost", interpod, and internode traffic. eBPF programs attached to the pod0/pod1 interfaces and eth0 and TAP devices manipulate pod traffic at the packet level to achieve secure, high-performance pod-level networking between runtimes.

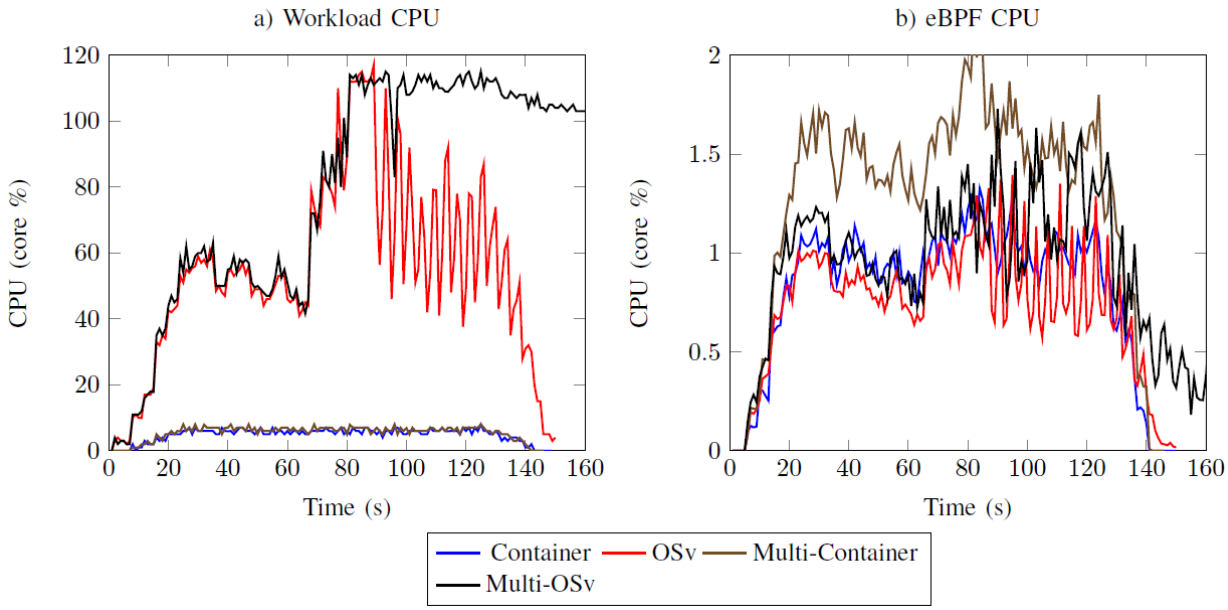

Figure 12: Workload vs networking CPU impact for a video streaming application.

Figure 12 illustrates the multi-runtime performance of a basic video streaming application serving multiple clients from static files stored on disk, achieving a throughput between 3 Gbps and 4 Gbps. The different series denote how the streaming server is executed. The “multi-container”

and “multi-OSv” scenarios refer to pod compositions where the streaming server runs as a container or microVM within a multi-workload pod. While performance scaling depends on the composition of both source and target pods, in most cases, the overhead will be approximately 1% of a single CPU core (x86 @ ~3 GHz), and a latency of 10 µs to 20 µs is added compared to standard processing. The erratic behavior of OSv-based pods is related to stability issues described in Section 2.2.2.2.

3.3. Network Resilience

In parallel with runtime and network unification, ensuring a network’s resilience to external threats is critical. The key objective is to identify attacks as close to their source. By deploying multiple devices across the entry points of an extensive network, the system can establish a unified strategy to block malicious traffic effectively. This collaborative approach has been shown to enhance detection accuracy using techniques such as entropy and Deep Learning (DL) [70]. This approach enables a coordinated defense mechanism that works across different network segments, mitigating potential threats early in the attack life cycle.

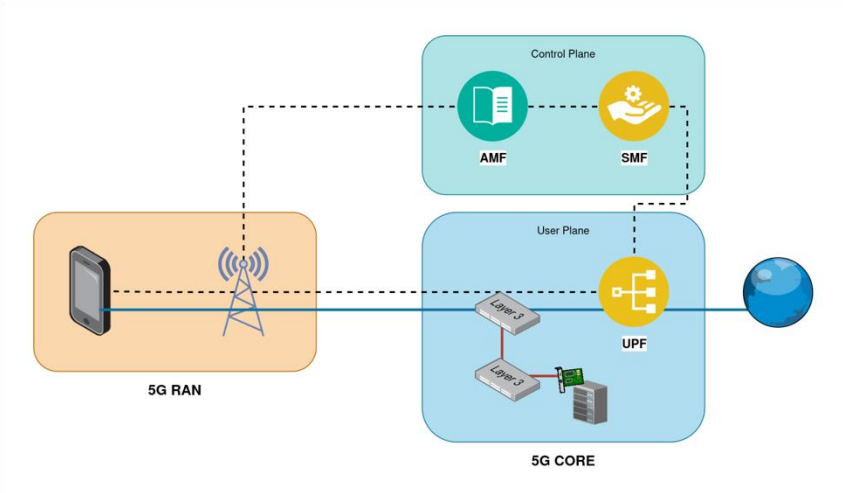

Figure 13: 5G infrastructure with DDoS attack detection mechanism.

The NATWORK solution (cf. Figure 13) is built to be easily integrated into existing infrastructure, enhancing the overall security posture of the network without requiring a complete overhaul of the system. To improve the security and resilience of 5G infrastructures, particularly in the context of emerging threats targeting 6G networks, the NATWORK network resilience framework has been defined. NATWORK introduces intelligent monitoring components throughout the network to observe and analyze traffic behavior. Specifically, network probes are strategically deployed within the RAN and the 5G Core, especially near the UPF, to capture traffic characteristics from both control and user planes. These probes feed data into specialized

detection models designed to identify anomalous behaviors indicative of cyberattacks, such as flooding, signaling storms, or protocol abuse. The models leverage machine learning techniques to differentiate between benign and malicious patterns, including the ability to detect previously unknown attack vectors. The classification of network traffic into “healthy” and “infected” categories will be performed in real time and can be adapted to specific services or protocols under analysis. In addition to anomaly detection, the framework will ensure data integrity through traffic anonymization, format normalization, and source consistency checks. This comprehensive approach allows the system to operate effectively at scale while preserving data quality and privacy. Ultimately, integrating the NATWORK framework with the 5G architecture supports a proactive security posture by enabling the early detection and mitigation of network threats. It offers a scalable and adaptive solution aligned with the needs of future mobile network generations. Studies have demonstrated that a distributed defense mechanism at the Internet scale offers scalability and resilience across diverse network topologies [70], [71]. To improve adaptability and flexibility, isolated security devices—such as DDoS detection systems—can be integrated via REST APIs to share real-time threat data with the broader infrastructure, contributing to a more comprehensive threat intelligence ecosystem (cf. Figure 14).

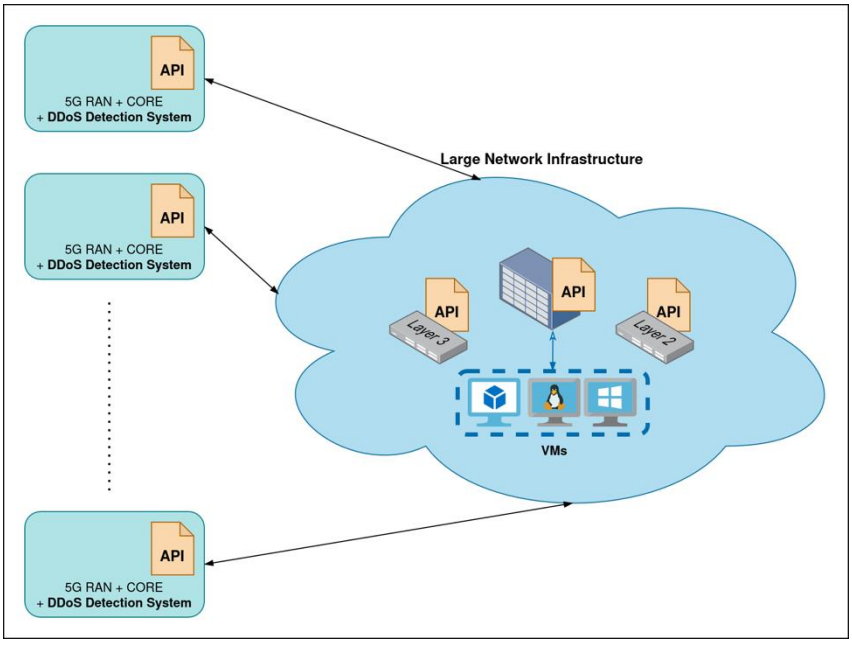

Figure 14: Integration through an API

When an attack is detected, the NATWORK system can trigger responses across the network, such as isolating contaminated equipment or shifting services from compromised virtual machines (VMs) to healthier, more secure locations. By leveraging API integration, the system ensures that threat data is seamlessly shared between devices and services, enabling faster and more coordinated responses to attacks. This integration also allows the network to dynamically

adjust by orchestrating services based on their health and security status. For example, VMs showing signs of compromise can be automatically quarantined, while healthy services are migrated to different network parts with minimal disruption. The result is a more resilient and responsive network, where the defense mechanisms work in real time to isolate and mitigate the impact of malicious activities.

Ultimately, the goal is to improve detection accuracy and response times, making the network more resilient to attacks, e.g., DDoS. Through this distributed and proactive method, the system ensures a more efficient defense strategy, reducing the impact of malicious flows on the network's resources and maintaining operational continuity. For example, recent work has also highlighted how ensemble-based packet processing and bandwidth optimization significantly improve DDoS attack detection and overall network resilience [71], [72].

3.4. Standardization

Standardization of the developments in T4.2 is key to long-term compatibility and extensibility. To that end, efforts have been made to ensure Feather's compatibility with Kubernetes and Open Container Initiative (OCI) standards. At the same time, the Open Application Model (OAM) is being adopted as an independent model for intent-based application modeling.

3.4.1. Kubernetes & Open Container Initiative

Feather supports Kubernetes API data structures as required for Pod deployment, i.e., Pods, Deployments, Containers, etc., as defined in the K8s v1 specification[40]. The API itself is compliant with several OCI guidelines and standards, including container and image formats.

Non-container images are supported within the OCI image format specification by leveraging freely defined metadata as described in Section 3.1; non-Feather Kubernetes nodes will ignore this metadata and (attempt to) deploy images in a Deployment as a container.

Networking options (multi-runtime or otherwise) are directly integrated into Feather and do not adhere to the principle of a CNI. However, they comply with all expected behavior of a container network from the point of view of other nodes and the control plane (barring Services and certain DNS features).

[40] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.30/#workloads-apis

3.4.2. Open Application Model

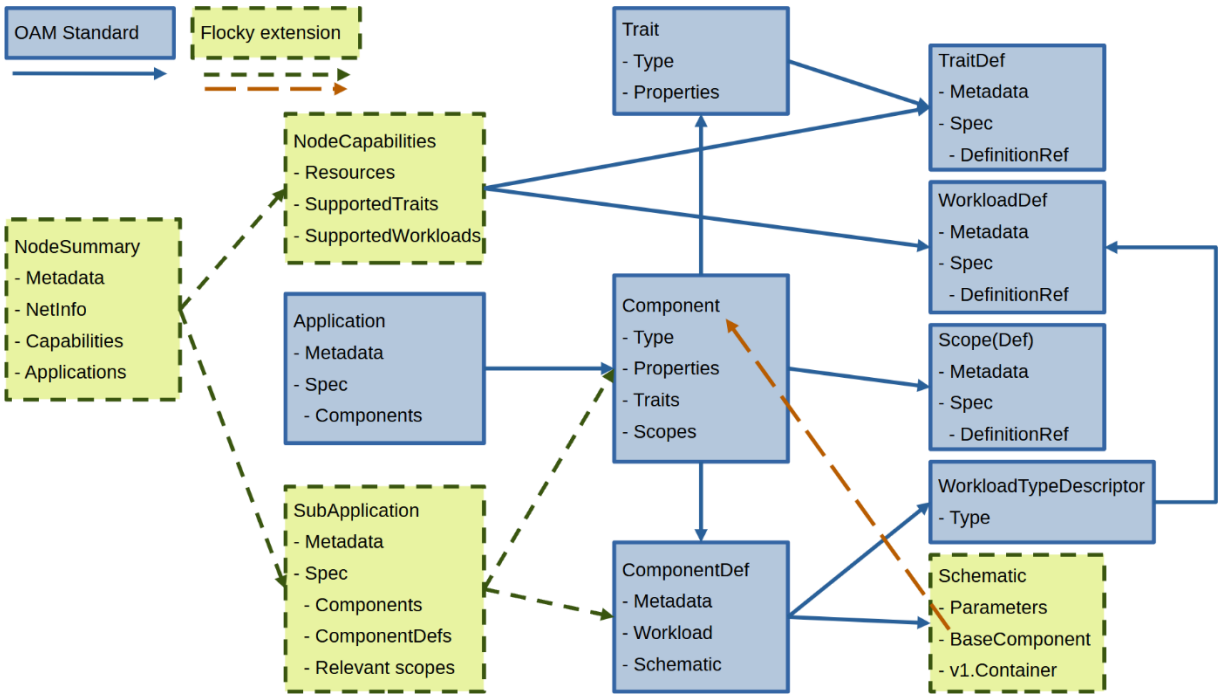

Figure 15: Overview of OAM and modifications required to provide intent-based orchestration

Open Application Model (OAM)[41] adopts multi-runtime intent-based deployment in a larger framework, Flocky, which is designed for decentralized intent-based node discovery and orchestrator-agnostic workload deployment. Technically, the framework allows for translating deployments to any platform, including K8s, although Feather was chosen for Flocky due to its low resource requirements and mixed-runtime capabilities. The modifications to OAM are shown in Figure 15, marked in light green. To summarize, the basis of OAM is the Application, which primarily consists of metadata and several Components. Components are used during deployment, referencing ComponentDefinition and Traits and Scopes. The ComponentDefinition contains the entire schema and all required details for its deployment, including a reference to its WorkloadType. Traits refer to TraitDefinitions, which are intended to apply specific behavior or restrictions to Components. Scopes refer to ScopeDefinitions and are primarily used to link components that need logical access to shared resources. Note that Figure 15 shortens these concepts to Component-, Trait- and ScopeDefs.

The modifications made to OAM are for internal use (decentralized orchestration) only, and consist of:

- The ComponentDefinition Schematic is the Flocky implementation of an unstructured field in the OAM standard, containing a Kubernetes API Container description and

[41] https://oam.dev

parameter data. Importantly, Flocky is workload runtime-agnostic; v1.Container[42] is merely used for convenience. ``BaseComponent'' allows several ComponentDefinition to implement the same base component under a single reference.

- WorkloadType is reinterpreted to serve as an indication of which runtime a ComponentDefinition requires. Basic workload types for OSv unikernels and Docker containers are implemented.
- Several example Traits are defined, including GreenEnergy, SecureRuntime, SoftDistanceLimit, Attestation, and NetworkEncryption. Some traits apply to WorkloadType selection (e.g., a unikernel workload uses SecureRuntime), while others apply to nodes (e.g., limiting QoE impact with SoftDistanceLimit, requiring a node with attestation) or node properties (e.g., NetworkEncryption, GreenEnergy).
- NodeSummary and NodeCapabilities, respectively, are used to build a Metadata Repository; these structures allow the Repository service to request the (available) hardware resources, Traits, supported WorkloadTypes and running (Sub)Applications of another node.
- To break down Applications into smaller deployable units for several nodes, SubApplication is used to deploy subsets of an Application while retaining important information such as specific ComponentDefinitions to be deployed for each component and any Scopes that should be applied to its particular collection of Components.

3.5. Intent-based selection

Flocky was developed as a framework for decentralized intent-based metadata gathering and orchestration. Its main goal is to allow any device to deploy an Application consisting of multiple Components to multiple discovered target devices, depending on the requirements of each Component and the target device's capabilities. Flocky is entirely decentralized, as indicated by the distribution of identical Flocky services among different nodes, as illustrated in Figure 16.

The Discovery service (green rectangles and interactions) is responsible for discovering nearby nodes running the Flocky framework, based on earlier work in SoSwirly [39]. The Discovery API uses a “ping” operation, which periodically checks the existence of a remote node and its network latency, and an operation, which requests the list of nodes known by a remote node. Through these, the Discovery service maintains a cache of nodes within a configurable maximum network latency by recursively contacting nodes and requesting their node lists. Network latency is chosen as a basic metric at this level as the Discovery service is only interested in exploring the actual network topology; advanced metrics are reserved for higher-level metadata and decisions. The

[42] https://github.com/kubernetes-client/python/blob/master/kubernetes/docs/V1Container.md

Discovery API also provides operations that allow subscribers to receive periodic updates on node topology changes.

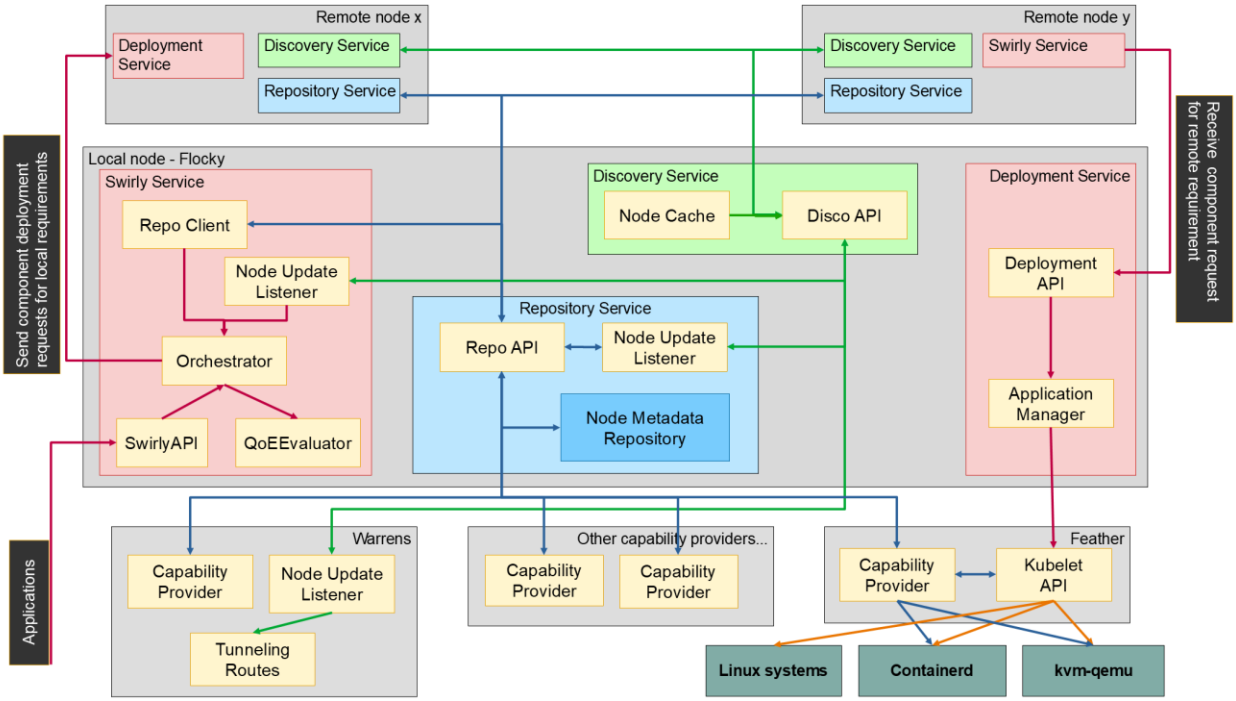

Figure 16: Overview of Flocky services for decentralized, intent-based orchestration

The Repository Service maintains the Metadata Repository (blue rectangles and interactions in Figure 16), which subscribes to the Discovery service for node updates. The repository is not updated on each node update; the service integrates all updates and only periodically contacts known nodes for additional metadata through the Repository API.

Remote nodes provide one part of this metadata, such as a NodeSummary, which contains node metadata, network interface information, capabilities, and any running applications deployed through Flocky. Every node maintains its own NodeSummary through the Repository API, which allows Capability Providers to register with it, as illustrated at the bottom of Figure 16. Registered Capability Providers are periodically queried for any NodeCapabilities and NodeApplications they provide, representing partial content of the ``Capabilities'' and ``Applications'' fields of the NodeSummary. As indicated, providers may include a container engine, i.e., Feather, which provides hardware resources, Flocky-deployed Applications, and supported WorkloadTypes based on detected runtimes. Other providers may be container network-based, e.g., Warrens or the presence of a VPN, enabling Traits such as NetworkEncryption. Other Traits may be similarly provided through vendor-specific adapters that detect the presence of green energy or an attestation agent.

Apart from a NodeSummary, remote nodes are also queried for any ComponentDefinitions in their repository, which are then merged with the local store, allowing new ComponentDefinitions to propagate throughout an entire topology from just a few nodes.

The orchestration component of Flocky is split into two services, indicated by red components in Figure 16: the Swirly service and the Deployment service. The Swirly service subscribes to the Discovery and Repository services, receiving updates on newly discovered nodes and metadata changes. When software on the local nodes requires the deployment of an OAM Application through the Swirly API, the application is split into its Components, and ComponentDefinitions fulfilling the required Traits for each Component are requested from the Repository API.

For each Component, the algorithm matches known nodes with the WorkloadTypes and resource requirements of suitable ComponentDefinitions, along with Component Traits, resulting in a list of candidate nodes. The first step of the matching algorithm considers nodes on which it already has a suitable ComponentDefinition deployed; if found, these are contacted through the Deployment API to determine whether they support another client. Suppose no existing deployment (or available node) is found. In that case, the algorithm iterates the remaining nodes to determine which is currently capable of deploying a suitable ComponentDefinition based on resources and metadata. A QoE Evaluator then ranks all eligible nodes for a specific Component based on hardware resources, Traits, WorkloadTypes, latency, and other relevant metadata. Current implementations include the Legacy evaluator, which ranks by latency, and the Scored evaluator, which extensively uses metadata using a static calculation. Other implementations may include online learning evaluators and matching components and nodes based on elusive user preferences, which are difficult to capture in a static model. By combining the flexibility of Trait providers, Trait implementation logic, and the freedom of custom Evaluators to customize Trait effects, Flocky provides potential support for a host of functional and non-functional intents, user-driven or otherwise.

After ranking, each node is contacted through its Deployment API, i.e., the most desirable ones, to determine whether it can currently deploy the required Component (specifically, a concrete implementation or ComponentDefinition). If the list is exhausted before the ComponentDefinition is deployed, the entire Application deployment fails and returns an error.

Due to the dependency of Components on Traits to ensure specific behavior or properties, a single node (i.e., Swirly service) may be requested to deploy several instances of the same Component with different non-compatible Traits, e.g., a database sidecar for a standard application, and a highly secured version of that same sidecar for a critical application which requires node attestation. In all cases, the algorithm will first attempt to reuse suitable deployments, even if they operate under strictly needed tracks. If no existing Component instance with the required Traits is found, another one will be deployed on a remote node.

3.6. Data collection methodology for ML Services

The NATWORK framework aims to strengthen the resilience of upcoming 6G networks against evolving attacks. Therefore, careful attention must be given to data collection, source reliability, and quality. In such a way, NATWORK will support accurate problem analysis and the development of well-defined models in various services as defined in D2.3. As new attacks emerge continuously, the focus must go beyond existing threats to enable the detection and mitigation of novel ones. Therefore, data will be gathered on a per-service basis. Given the volume of data involved, considerations such as anonymization, format normalization, and source consistency are essential.

The NATWORK project considers the two traditional sources of data:

- External databases. NATWORK uses an existing database that contains data traces.
- Internal databases. We generate the database from testbeds prepared in different NATWORK research centers.

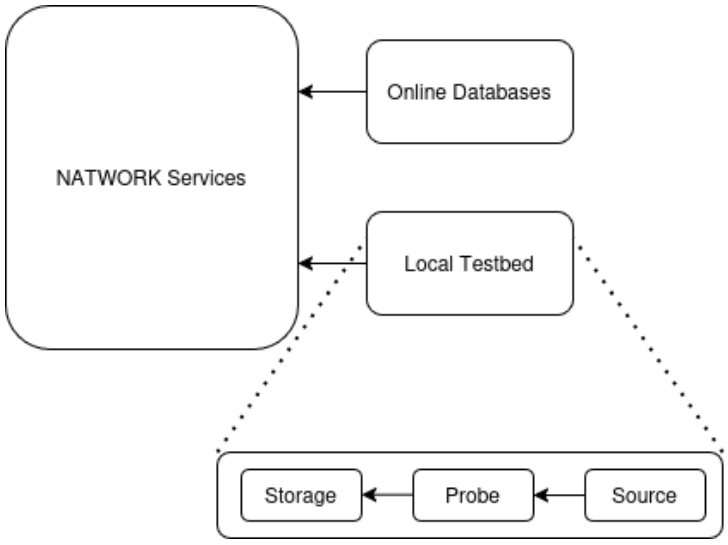

Figure 17: Data gathering & data sources for NATWORK

Figure 17 illustrates the data collection architecture within the NATWORK framework, integrating external and internal data sources. Two primary components support data acquisition: Online Databases and a Local Testbed. The online databases serve as repositories of pre-existing network data, potentially encompassing known attack traces, standard network patterns, and labeled datasets useful for training and validating anomaly detection models. In parallel, the local testbed enables controlled, real-time data generation and collection. Within this testbed, data originates from a source, is observed and analyzed by a probe, and then stored for further processing. This modular chain (Source → Probe → Storage) facilitates the acquisition of rich, context-aware traffic data under customizable conditions. The online and locally generated data

streams are funneled into the NATWORK services, the core processing and intelligence layer. This architectural design supports comprehensive data fusion, enabling robust model development, testing, and continuous learning in a dynamic network environment. Finally, model validation requires data that was not used during training. A common practice is to use 70% of the collected data for training and reserve the remaining 30% for validation, ensuring the model is tested on previously unseen data.

3.7. ML-based workload modeling & resource optimization

Machine learning modeling is crucial in resource optimization by enabling intelligent workload prediction and resource optimization. We are conducting benchmark analyses of machine learning models to predict workload patterns and incorporate intelligence into resource allocation strategies. Currently, we are working with Google workload traces [61]. This methodology will be extended to synthetic datasets generated from attack simulations against Cloud native functions (CNFs) and Fog-native deployments. Different traditional and gradient-boosting ML models will be used to test and benchmark analysis of their performance, ensuring the most effective models are selected for workload prediction. Federated learning (FL) is integrated across the edge-to-cloud continuum to enhance workload modeling and resource optimization further. This enables decentralized model training, aiming for data privacy while reducing communication overhead. By allowing edge nodes to train models locally, FL supports dynamic workload adaptation, proactive risk assessment, and efficient orchestration across the infrastructure.

3.7.1. Data Engineering and Preprocessing

Data engineering and preprocessing are key steps in structuring workload traces for analysis. Initially, the traces are processed to extract critical features for resource allocation. The data will be structured chronologically, ensuring suitability for time-series modeling, and undergo feature engineering, aggregation, and categorization to derive meaningful workload patterns. Our approach adapts to evolving workload datasets, limiting both under- and over-provisioning resources in cloud workload management. Before feeding the data into machine learning models, exploratory data analysis (EDA) is performed to understand workload characteristics and distributions. EDA helps identify patterns, detect anomalies, and assess correlations between workload attributes.

3.7.2. Model Building and Algorithm Selection

Various time-series forecasting models, including Long Short-Term Memory (LSTM) networks [63] and statistical models like ARIMA 64], are implemented to capture workload behavior. Additionally, gradient-boosting models such as XGBoost [65] are incorporated to enhance prediction accuracy. The selected algorithms will be determined through benchmarking various machine learning and statistical models to analyze and predict dynamic, non-linear data patterns. This evaluation will compare the performance of traditional time-series models against more advanced approaches, identifying the most effective methods for workload prediction. The chosen model will be trained on historical Google workload traces and newly generated attack datasets. The models will be validated against unseen data to ensure robustness and predictive accuracy, assessing its generalization to future workload behaviors and security threats.

3.7.3. Federated Learning

Federated learning (FL) is introduced as the next step. FL enables decentralized learning by allowing edge nodes to train models locally on their data. Each trained model is sent to an aggregation point within its cluster, merging with other locally trained models to form a more generalized aggregate model, as depicted in Figure 18.

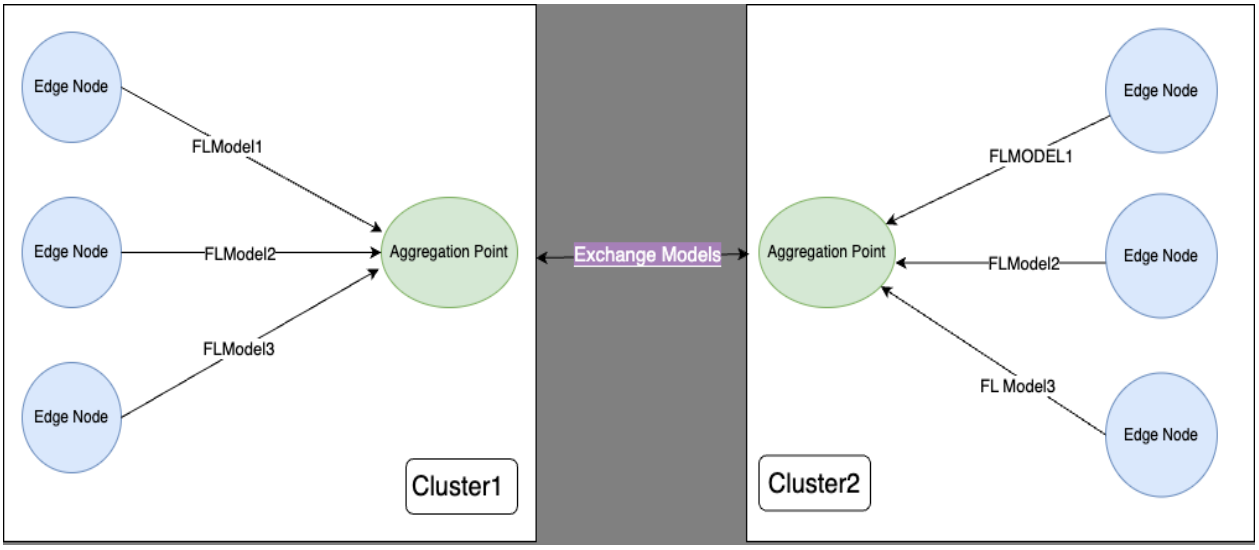

Figure 18: Federated learning over edge-cloud continuum

This aggregated model is then exchanged between clusters, further refining with insights from different network slices and workloads. The updated global model is redistributed to the edge nodes, which continue to be refined in iterative learning cycles until convergence [74]. Attack datasets generated from CNF deployments are utilized in this step to improve anomaly detection.

Historical data is leveraged to forecast potential security risks in the form of denial-of-sustainability (DoSt) attacks [75]. FL agents train models on attack-induced anomalies and exchange learned patterns with other nodes through aggregation. This ensures that models capture variations in network behavior across different infrastructure components, allowing for continuous adaptation.

3.7.4. Hyperparameter Tuning

In this step, the hyperparameters of the chosen algorithms are optimized using Bayesian optimization [66], grid search [67], or other systematic techniques to explore the parameter space and determine the best configurations efficiently. We differentiate our specific orchestration needs from the outcomes of Bayesian optimization to ensure the resulting configurations align with real-world requirements and constraints. The objective is to minimize prediction errors and ensure that the models perform well across different workload scenarios, including sudden spikes or drops in resource usage.

4. Remote Attestation

4.1. TPM-based attestation

A Trusted Platform Module is a secure cryptographic coprocessor. One of its most important functionalities is its ability to measure digests into so-called platform configuration registers (PCRs) [40]. The core idea of these PCRs is built around two basic operations: extend and reset. The extend operation performs an XOR operation with the current PCR state and the new digest as operands and saves the hash of the result as the new PCR value. The reset operations set a PCR to zero.

Proving that a device is running trusted software is a very complex process, stretching all the way from the lowest levels of the boot sequence to the kernel, the operating system, and the applications it runs. These components form a chain, relying on a lower level to tie the next to a trusted state by measuring the next binary into a TPM platform configuration register in a measured boot process. The lowest level of the chain is the so-called core root of trust for measurement (CRTM) [41], which anchors the entire chain to a piece of immutable CPU code. If a system lacks proper measurement for any part of this chain, it could compromise its ability to protect workloads, as many container isolation mechanisms rely on security features provided by the Linux kernel (such as cgroups and namespaces).

TPMs are often used in edge research to provide hardware-based trust. It is, however, essential to acknowledge a critical oversight regularly present in such implementations. These designs are often implemented on devices lacking a CRTM (e.g., Raspberry Pi). Without a CRTM, the chain of trust, crucial for ensuring the system's integrity, lacks a solid anchor [42]. Consequently, reliance on a potentially compromised kernel to transmit measurements to the TPM can render the entire attestation process unreliable. Simply plugging a TPM into an edge device and trusting it is not enough. This oversight highlights the importance of ensuring that the foundational elements, such as a complete boot attestation anchored in the CRTM, are present.

An issue with TPM-based attestation is its complexity. Interacting with a TPM requires an excellent understanding of the complex architecture [43] and is generally done by transmitting low-level binary commands [44] to the device. In recent years, the Trusted Computing Group (TCG) [45] has put a lot of effort into developing and standardizing higher-level APIs [46] to interact with the TPM, resulting in a toolset called *tpm2-tools*[43]. These tools allow for higher-level interactions with the TPM using the CLI. While the low-level code complexity has been reduced significantly, a deep knowledge of a TPM's inner workings and mechanisms is still required to use

[43] https://github.com/tpm2-software/tpm2-tools

it actively in an application. Research projects like Keylime[44] could help alleviate this complexity issue.

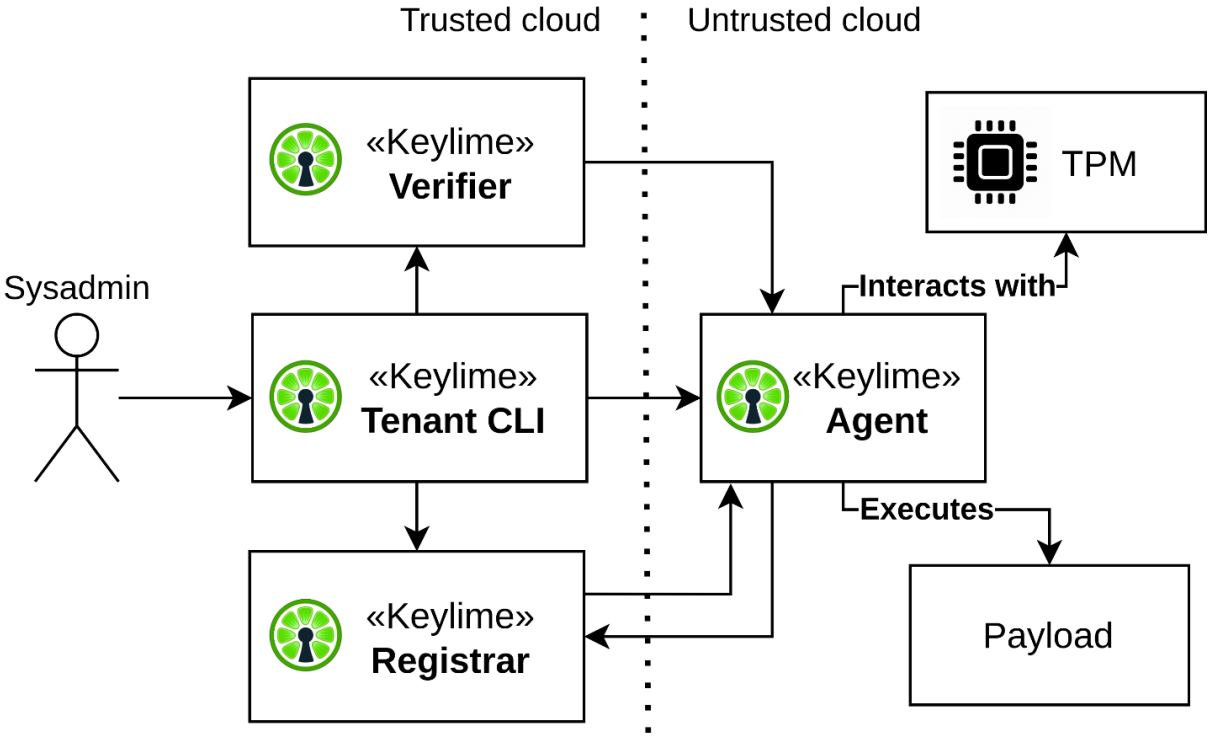

Figure 19: Architecture of the Keylime, abstracting away to complexity of TPM operations.

Keylime is an open-source CNCF project initially developed by MIT's Lincoln Laboratory [47]. It has since seen increased adoption with the support of RedHat, which is actively developing it for RHEL and OpenShift. Keylime provides an additional abstraction layer for TPM attestation on top of the existing tpm2-tools, allowing developers to integrate boot and runtime attestation into their architectures easily. Keylime consists of cloud components written in Python (verifier, registrar, and tenant) and a Rust agent running on the machine to be attested.

The Keylime architecture, as depicted in Figure 19, is controlled by the tenant CLI application, which enables a system administrator to enroll a device and configure it for TPM-based boot and runtime attestation using specific golden values for each device. These values contain known good states of the system and serve as a reference for future attestation. The tenant interacts with the other cloud components to deliver a secure payload to an agent. This agent is the device's connection point to the Keylime cloud components. It provides an API over HTTPS, allowing an abstract interaction with the device's TPM.

[44] https://keylime.dev/

Efforts are underway to develop a cloud operator to manage TPM-enabled nodes in a K8s-compatible environment [48]. This operator is designed to simplify the deployment of Keylime in cloud Kubernetes environments, providing node attestation.

In T4.2, TrustEdge was developed [49], and its architecture is shown in Figure 20. TrustEdge uses Keylime to automate the enrollment of trusted TPM-enabled edge devices into a heterogeneous Kubernetes cluster. It ties into K8s' Role-Based Access Control (RBAC) and dynamically adjusts a node's permissions based on its trust status. A sysadmin can register an edge device as a custom resource (CR), which the attestation controller monitors. As soon as the edge device comes online, it contacts the registrar, which updates the CR. The controller interacts with the K8s API to generate an identity and permissions for the edge device, which are distributed through the tenant and verifier to the edge device after successful attestation. The verifier monitors the attestation state of the edge device and updates the CR as necessary, potentially triggering a controller response that adjusts permissions for the edge device's identity.

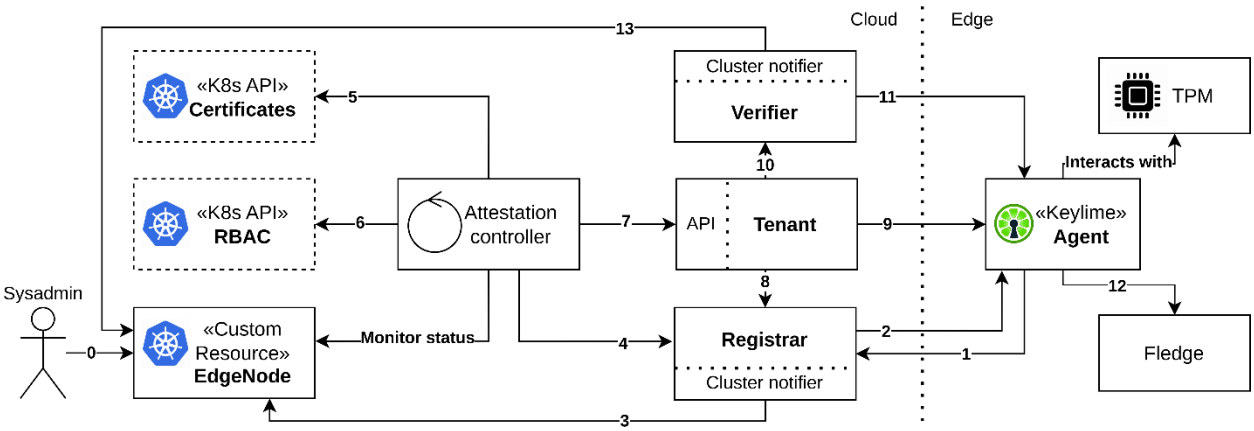

Figure 20: Architecture of the TrustEdge, with the centrol controller managing the trust state of edge devices.

4.2. TEE-based attestation

Remote attestation is a crucial mechanism in trusted execution environments (TEEs) that allows external parties to verify whether a system's hardware and software components are genuine and not tampered with [52]. This functionality is especially relevant in confidential computing scenarios, as organizations rely on TEEs in cloud or multi-tenant settings to protect sensitive data from potentially malicious privileged entities, including system administrators and hypervisors. Three prominent implementations of TEEs that incorporate attestation are Intel Software Guard Extensions (SGX) [51], AMD Secure Encrypted Virtualization Secure Nested Paging (SEV-SNP) [50], and Intel Trust Domain Extensions (TDX) [55]. There are also implementations for RISC-V [56] and ARM CPUs [57], which are less mature.

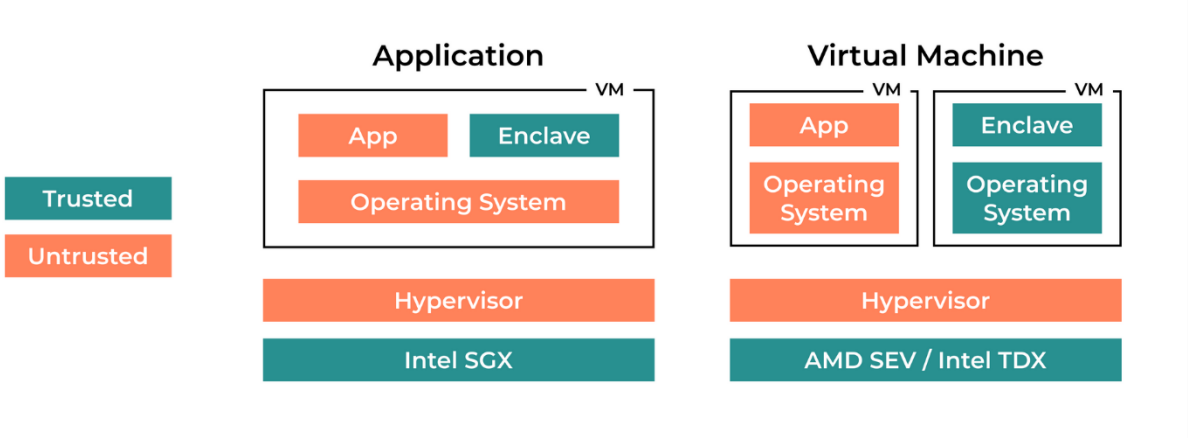

Figure 21: Trusted Computing Base comparison of process and VM based TEEs. Source: https://www.decentriq.com/article/swiss-cheese-to-cheddar-securing-amd-sev-snp-early-boot

SGX operates at the process level, creating small, isolated enclaves that protect code and data from access by the operating system, hypervisor, or other processes. Because SGX enclaves sit within a host process, they have a relatively tight trust boundary that excludes most system software. At the same time, enclaves must manage a restricted memory footprint, and applications that rely on SGX require substantial refactoring to partition sensitive code within enclaves. Remote attestation in SGX is facilitated by a hardware-based key provisioned on each SGX-enabled processor. An enclave produces a measurement reflecting its initial code and data, signs this measurement using a platform-specific key, and then delivers the signed quote to a relying party for verification. In practice, this verification commonly involves a trusted Intel attestation service [57] that checks the signature and the enclave's identity, confirming the enclave is running on genuine Intel hardware and its contents have not been modified. Recently, Intel announced the retirement of the Attestation Service [53] and the move to DCAP [54]. DCAP is more focused on cloud providers and allows them to cache Intel's cryptographic material, making verification of attestation reports possible without sending the reports to Intel.

AMD SEV-SNP differs from SGX in that it focuses on virtualization-based isolation. Instead of enclaves, SEV-SNP protects entire virtual machines, making it more straightforward to integrate with existing software stacks that are already virtualized. A VM under SEV-SNP runs with its memory encrypted and protected by AMD's secure processor, effectively preventing both passive attacks, for example, unauthorized snooping or reading of memory contents, and active attacks, such as inserting malicious code or modifying data in memory, which can originate from a compromised hypervisor or host firmware. Remote attestation for SEV-SNP entails the VM obtaining a hardware-signed report that includes information about its initial state, the version and policy configuration of the secure processor, and a cryptographic measurement of its memory. By default, SEV-SNP only measures the first component it loads during boot (for instance, the firmware). Additional measures must be taken to ensure the integrity of the full

system, including the static boot chain and userspace. These components must also be measured and verified, adding complexity to the attestation process. AMD offers an online service [58] that allows a remote party to fetch the required cryptographic material to independently verify the integrity of the attestation report, like Intel's DCAP.

Intel TDX is similar in principle to SEV-SNP, but it was developed for Intel platforms and designed to support confidential VMs (termed Trust Domains). Like SEV-SNP, TDX encrypts VM memory to ensure that neither the hypervisor nor other system software can read or modify the guest's data. Remote attestation in TDX is achieved by measuring the trust domain's contents and configuration, which is then signed by Intel's hardware-rooted attestation key. The verifying party can use Intel's infrastructure to confirm the authenticity and verify the trust domain's integrity or use DCAP similarly to SGX.

A significant distinction between SGX, SEV-SNP, and TDX is the size and composition of the Trusted Computing Base (TCB) as indicated in Figure 21. SGX enclaves reside within a process boundary and place minimal trust in operating system and hypervisor layers. This effectively restricts the TCB to the enclave code and the CPU microcode handling enclaves. This narrow TCB can be advantageous for reducing the potential attack surface. However, SGX enclaves demand more specialized development practices to split code between trusted enclave sections and untrusted host sections. By contrast, VM-based TEEs like SEV-SNP and TDX rely on a larger TCB that includes the VM components. In return, SEV-SNP and TDX typically require less invasive application changes since most workloads can run inside a protected VM without extensive changes.

Additionally, the increased TCB size of a VM-based TEE results in a much more complex attestation procedure. While both process and VM-based solutions must evaluate the platform (genuine CPU, genuine CPU microcode), the user-defined part is much larger than VM-based ones. SGX only requires measuring a single statically compiled binary file, while SEV-SNP requires evaluating the full boot stack (virtual firmware, kernel, OS, applications).

Like TPM-based attestations, TEE ones can be very complicated. While each platform technically achieves the same goal, the verification steps vary widely, making it difficult for developers. An active research domain is the development of abstraction layers for these technologies. Software like Enarx [59] and Trust Monitor [60] attempt to provide a generalized interface for attestation across multiple technologies.

4.3. WASM remote attestation

4.3.1. General. Development stages.

What is described below is the current state of progress of NATWORK's WASM runtime integrity verifier and its integration into D-MUTRA. At the current stage, we have gone through the stages of:

- WASM payload runtime integrity verification: Feasibility study, blueprint design, implementation of a proof of concept.
- WASM payload runtime integrity verification integration into D-MUTRA (blockchain based mutual remote attestation): Feasibility study, blueprint design.
- Full stack WASM remote attestation (i.e., including the runtime and payload): Feasibility study, blueprint design.

4.3.2. WASM authentication and remote attestation merits.

The state of the art provides all means to authenticate a WASM payload, whether used client-side (i.e., interpreted by web browser) or server-side (i.e., interpreted by runtimes such as WASMTIME[45], WASMER [46], WAMR[47]). A plethora of implementations (e.g., OAUTH[48], mutual TLS[49], JWT, session tokens) validate the origin and integrity of the WASM payload by turning on the classical signed hash technique. As a reminder, authentication enables a local verification of a payload and requires that the public key (i.e., identifying the signer) be recognized/accepted by the recipient. This security assurance is needed for those who control or own the execution environment. Typically, authentication is used to check before installing a piece of code, and this check-in is made at the code recipient site. Based on the same core maths (i.e., cryptographic hash function (CHF) and the Rivest–Shamir–Adleman (RSA) encryption), authentication delivers security assurance different from that of remote attestation.

In networking, payloads are operated with low control in off-premises execution environments. In these conditions, there is a need to verify that what is deployed remotely is integrated, and this goes through remote attestation. The ETSI network function security working group[50] highly recommends remote attestation as a strong foundation of networking service security[78]. However, as stated, remote attestation comes with heavy management and workflow

[45] https://github.com/bytecodealliance/wasmtime
[46] WASMER, available at: https://wasmer.io/
[47] WAMR (Web Assembly Micro Runtime), available at : https://github.com/bytecodealliance/wasm-micro-runtime
[48] https://oauth.net/2/
[49] https://en.wikipedia.org/wiki/Mutual_authentication
[50] https://docbox.etsi.org/ISG/NFV/Open/other/Tutorials/201805-Tutorials-NFV_World_Congress_San_Jose/NFV%20Security%20Layer123%20april%202018%20v3.pdf

considerations on signature management (creation, sharing, and revocation) and payload deployment dependencies (e.g., TPM). Ideally, novel remote attestation solutions should eliminate these two significant drawbacks.

4.3.3. State of the art

The state of the art does not offer today a remote attestation of WASM module per se, but close-by security attributes or existing bricks:

- **TEEs-enabled remote attestation** can attest that the WASM interpreter (i.e., piece of native code) runs inside a verified trusted environment and, hence, is integrated. The WASM payload (i.e., a piece of data as bytecode) is not verified remotely. Several TEE-enabled frameworks offer this security assurance (e.g., Occulm[51], Enarx[52], Gramine[53]). To the best of our knowledge, these frameworks do not check the authenticity of a WASM payload before running it through the TEE-sheltered (i.e., integrated) interpreter. It is worth noting that these solutions imply a dependency on the deployment of the interpreter, being the presence of such a type of TEE on the host. It is also worth noting that without authentication produced inside the TEE, the WASM payload, though interpreted inside a TEE, can be tampered with before entering the TEE.
- The WASM-sign[54] or equivalent tools (e.g., WebAssembly Binary Tool WABT[55], LUCET[56]) tool produces signatures of WASM modules and, hence, could be used as a "prover" of a remote attestation implementation if installed on the host. It is worth noting that what is measured here is the WASM module before execution, hence using the hash of the WASM module file (i.e., carrying the .wasm extension). To our knowledge, a remote attestation schema integrating WASM-sign with RSA encryption, which is necessary for signing and authentication verification, does not exist. As the prevalence of WASM in networking increases, with no specific technical difficulties encumbering it, a WASM remote attestation framework will emerge, utilizing direct .wasm file hashing and signing.

4.3.4. Continuous attestation

Continuous attestation is the novel trend the EU Agency for Cybersecurity (ENISA) recommends [68] and calls for permanent and constant service monitoring. Continuous attestation enables

[51] https://occlum.io/
[52] https://enarx.dev/
[53] https://gramineproject.io/
[54] https://github.com/frehberg/wasm-sign
[55] https://github.com/WebAssembly/wabt
[56] https://github.com/bytecodealliance/lucet

execution metrics or integrity verification when the service is executed. Static measurements cannot grasp the evolution of the threat level. Where integrity is concerned, it simply means that the verification shall be performed with a running payload without interruption. Technically, it relies on the runtime integrity verification technique, the first enabler we would need for WASM.

4.3.5. WASM runtime integrity verification

Runtime integrity verification is the technique that enables the integrity of software to be attested when it executes. Combined with the means for remote verification, this property enables runtime remote attestation.

WASM's primary security pitfall, often perceived as its downside, is the vulnerability to module tampering, which can manipulate the data structure during interpretation. Easy tampering can be viewed as the counterpart of WASM's easy portability. As for all interpreted languages, the processor treats WASM bytecode as a data frame. Bytecode tampering is a common rule for all interpreted languages (e.g., no runtime integrity verification for Java payloads). Currently, WASM module integrity can only be proven when the module resides in a TEE, leveraging TEE-based remote attestation. However, this comes with the workflow and operational drawbacks associated with workflow considerations for TEE. As stated in [62], TEEs incur performance and memory consumption penalties, as well as new TEE-associated security threats (e.g., DoS attacks through raw hammering and covertly spawned malicious payloads). Another significant side effect of TEE is its heterogeneity, which restricts deployment to specific processors—a major workflow issue in cloud or hybrid service deployment. Typically, this conflicts with the portability and mobility of WASM payloads, and NATWORK's contribution is to seamlessly bring WASM payload runtime integrity. This is achieved through runtime integrity, which detects tampering (i.e., when the attacker fails to reset the memory states to their original state at the time of measurement).

Setting integrity verification requires (i) **getting a good understanding of the WASM code structure** and how the different segments are processed during the module interpretation, (ii) **identifying the invariant sections,** and (iii) **devising a routine that processes a measurement (i.e., hash) and signature on these sections.**

4.3.5.1. Understanding the WASM module structure at runtime

Before its loading and interpretation, a .wasm file contains a header and several sections referring to imports, exports, global variables, data, and code. The latter includes the WASM bytecode instructions. Figure 22 maps the different data sections of the .WASM module.

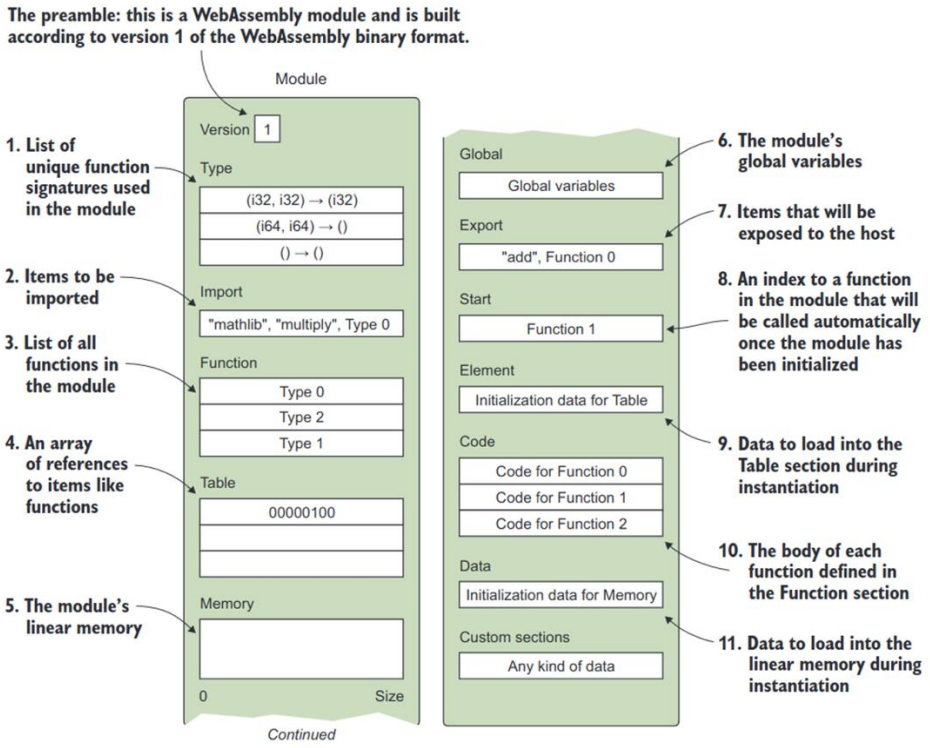

Figure 22. WASM module data structure

4.3.5.2. Understanding the created memory maps at runtime

To better understand the parsing and memory structure of the module at execution, we have studied the WASMTIME interpreter (i.e., an open-source interpreter)[57]. WASTIME interpreter parses the different sections to construct three core memory areas needed for the interpretation, as shown in Figure 23. These areas are:

- The instantiated stack for the module (which dynamically changes)
- Linear Memory, which contains data and offsets (static)
- The WASM instructions (static bytecode)

[57] https://github.com/bytecodealliance/wasmtime

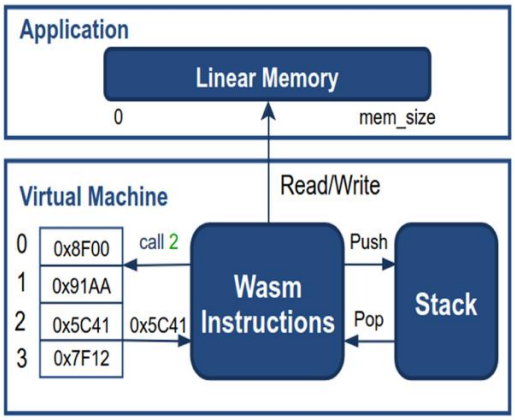

Figure 23. The memory structure of a loaded WASM application (and virtual machine)

4.3.5.3. Getting access to the WASM instruction memory mapping

Once the payload is executed (i.e., interpreted), the memory space of the WASM application is restricted to the Linear Memory, with no access to the WASM instructions, which are only accessible inside the interpreter, referred to as the "Virtual Machine" in Figure 23. The Linear Memory shall be viewed as a script of memory addresses (i.e., offset) to functions loaded inside the VM and residing in the WASM instruction area. Obtaining these offsets is insufficient for elaborating a memory map integrity verification, as one cannot map the physical memory from these offsets. Only the interpreter sitting in the memory map of the functions and WASM instructions can scrutinize the memory area of WASM instructions. The interpreter "sees" linear memory, but the reverse is untrue (i.e., the linear memory cannot map the interpreter). Last, for the search for exhaustivity, a measurement shall cover both WASM instructions and the Linear memory to guarantee the integrity of the WASM payload.

4.3.5.4. WASMTIME payload integrity verification, through the ELF generation.

WASMTIME offers different execution paths (i.e., interpretation, compilation) after parsing and unfolding the contents of the WASM file sections. Our analysis of this interpreter leads us to create a second thread, executed separately from the interpreter's main thread, enabling us to build an ELF-formatted payload through the WASMTIME "serialize" function. From this step onward, we calculate a hash, e.g., Secure Hash Algorithm (SHA)-256, of the ELF's text section, which contains the instructions. The new partial interpreter block diagram, integrating the added second thread, is shown in Figure 24.

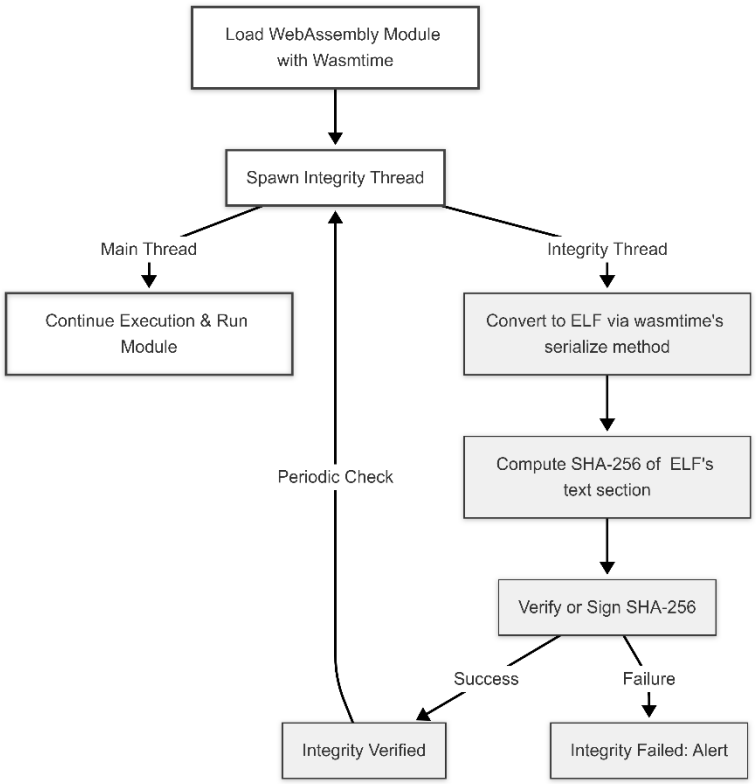

Figure 24. NATWORK's WASMTIME added second thread

4.3.5.5. Unavoidable and trustworthy interpreter change

For the above reasons, measurements shall be made at the interpreter level. **There is no alternative** to produce changes on the interpreter to produce the payload measurements as the latter, once loaded, has no insight into the memory map allocated for its unrolling.

Hence, there is a need for changes to be produced on the interpreter (i.e., as is currently carried out) or for a future plug-in embedding of the functionality. These changes result in creating a quote of the ELF-compiled payload, leveraging an existing Measure routine that can (i) map the text section once duly loaded by the system in memory and (ii) produce a hash with its content. These functions can be implemented as part of a plug-in, as shown in Figure 25, which resides aside the original WASMTIME interpreter, or by adding these functions directly inside the interpreter's source code, as shown in Figure 26. Noticeably, our current progression has gone down the second path, but a closer look shall be put on the plug-in alternative, which may stand as less intrusive and lead to higher scalability. Additionally, and as shown in Figure 25, in addition to the Measure function, Verification and Distributed Ledger Technology (DLT) routines can also be added, enabling the blockchain-enabled mutual attestation mode as offered by D-MUTRA, as discussed in Sect. 4.4.1.1.

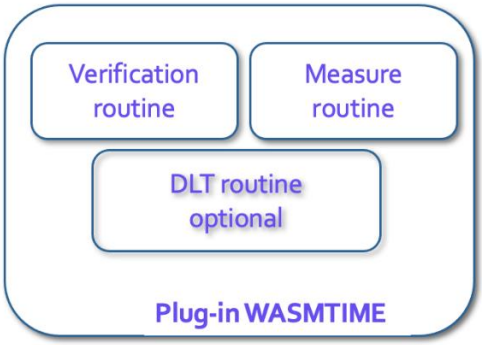

Figure 25. Simple representation of the NATWORK's WASM plug-in

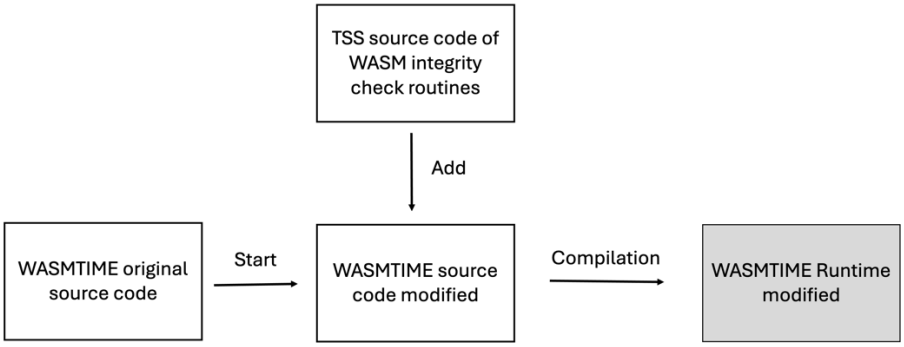

Figure 26. Workflow for changing WASMTIME interpreter

Trustworthy interpreter change: Supported by the above-mentioned "runtimes" attestation discussed in Section 4, it is engineered and progressed in NATWORK, ensuring the runtime is genuine and integrated. Hence, we can modify the runtime (i.e., the WASM interpreter) or add a plug-in and then run the "runtime" attestation, checking the set made of the novel potentially modified interpreter and the plug-in. In other words, the security foundation, as stated above, from the deep-rooted CRTM to the "runtime," ensures that the complete software stack is valid, and this includes our modified or plugged-in WASM interpreter. Compared to the TPM, which makes the memory measurement through direct memory access in an impenetrable environment, the modified or plugged interpreter is the binary code exposed to malicious introspection and tampering, which calls for regular remote attestation.

4.4. WASM runtime remote attestation

4.4.1. NATWORK's WASM runtime remote attestation

Leveraging our ability to verify the integrity of a WASM payload during its execution, as stated in Section 4.3 above, we can construct an integrated runtime remote attestation orchestrated in three different steps:

- Step 1: **SECaaS reference measurement generation**. NATWORK implements a SECaaS server, producing the reference measurement (i.e., a hash of the payload once instantiated by WASMTIME interpreter) used timely at the verification stage. The SECaaS consumes wasm payloads and hosts a WASMTIME interpreter, our developed ad hoc measurement plug-in. With these elements in hand, the SECaaS constructs the ELF file artefact and produces its hash. Concurrently, the SECaaS appends an RSA key pair inside the original .wasm module, enabling its future unambiguous identification of the payload using the public key and the authentication verification of the quote (i.e., RSA private key encrypted hash) by the verifier, using the private key.
- Step 2: **Prover quote generation**. On deployed instances with the same instance of WASMTIME interpreter and plug-in, the same measurement operation can be worked out by the prove module in a 1:1 replicated fashion as in step 1. This measurement is signed using the RSA private key to form the quote. The quote is transmitted with the public key.
- Step 3: **Quote verification**. Quote from step 2 is transmitted to a verifier, provisioned additionally by the reference measurement of Step 1. With both blobs (i.e., large integers) in hands and the public key, the verifier can:
    - Verify the **origin** of the quote using the public key
    - Decrypt the quote and compare it to the reference measurement. A positive check means that the WASM payload is integrated once instantiated, loaded, and running**.**

4.4.1.1. Future work

Design: The current modus operandi in modifying the WASMTIME interpreter can be perceived as intrusive. First, We will challenge that a more easily accepted plug-in can be designed alternatively.

Performance: The core consideration of NATWORK to **reconcile security and performance** (i.e., eternal rivals) is typically valid for runtime integrity verification, which shall be carefully conceived to preclude heavy performance penalties. For that sake, we will consider different techniques that turn on restricted resource allocation to the measuring function (e.g., Linux's

cgroups[58], Docker's CPU shares[59]), multi-threading typically using SharedArrayBuffer, and finally, idle time technique applied on the measuring thread. All these techniques must be analyzed in the context of WASM payloads and may not be practicable in this specific context, though a final penalty-friendly solution shall emerge.

Integration: The integration of **WASM runtime integrity verification to D-MUTRA** is the main progress to be made. A strong consideration of NATWORK is to develop platform-agnostic security for easy payload migration. D-MUTRA (i.e., DLT-based Mutual Remote Attestation) solution does not depend on hardware or kernel level routine. Moreover, the solution relaxes a strong blocker to remote attestation signature management. Being zero-touch, the signatures are automatically generated and provisioned where they are used (i.e., at the smart contract-elected verifier). Leveraging the power of blockchain and magnifying the concept of distributed security, D-MUTRA removes a single point of attack on a single verifier implementation, exposed to DoS by flooding sockets and distributing the verification to any nodes. D-MUTRA distributes the Measure and Verification function at each software node, able to prove themselves and verify peers through the blockchain. It also innovates with a pure software-based root of trust, which consists of electing the "freshest" node as the verifier for the next remote attestation job. The most recently verified node is the next verifier of D-MUTRA distributed remote attestation schema. It is worth noting that D-MUTRA relaxes both potent operational blockers of complex signature management and deployment dependencies. The signatures are automatically generated by the SECaaS and provisioned at the right end, being the elected verifier.

Confidentiality preservation: In addition to runtime integrity, we will investigate the relevance and practicality of confidentiality preservation by modifying the plug-in. We are not inclined to consider obfuscation for the associated performance penalty as discussed in 2.4.2, but instead consider encryption, preventing the reverse engineering of the WASM module. In this respect, we will analyze the inner mechanisms enforced in [69] but without leveraging TEE as considered by the authors.

4.5. NATWORK full stack remote attestation schema for WASM technology

NATWORK's work on remote attestation by IMEC and Solidshield provides a full stack integrity guarantee. As shown in Figure 27, IMEC first works (i.e., step 1) on the system and runtimes integrity, ensuring that the execution environment is correct, including the WASM's runtime (i.e.,

[58] https://en.wikipedia.org/wiki/Cgroups
[59] https://docs.docker.com/engine/containers/resource_constraints/

interpreter), while Solidshield deals with the WASM payload itself in step 2. This association is nothing but needed and operates at two different time scales. At first, IMEC's WASM runtime verification integrates the interpreter and Solidshield's plug-in, guaranteeing that the plug-in is integrated and providing correct measures. Thereafter, the WASM module remote attestation using the runtime plug-in can take place, delivering reliable attestations.

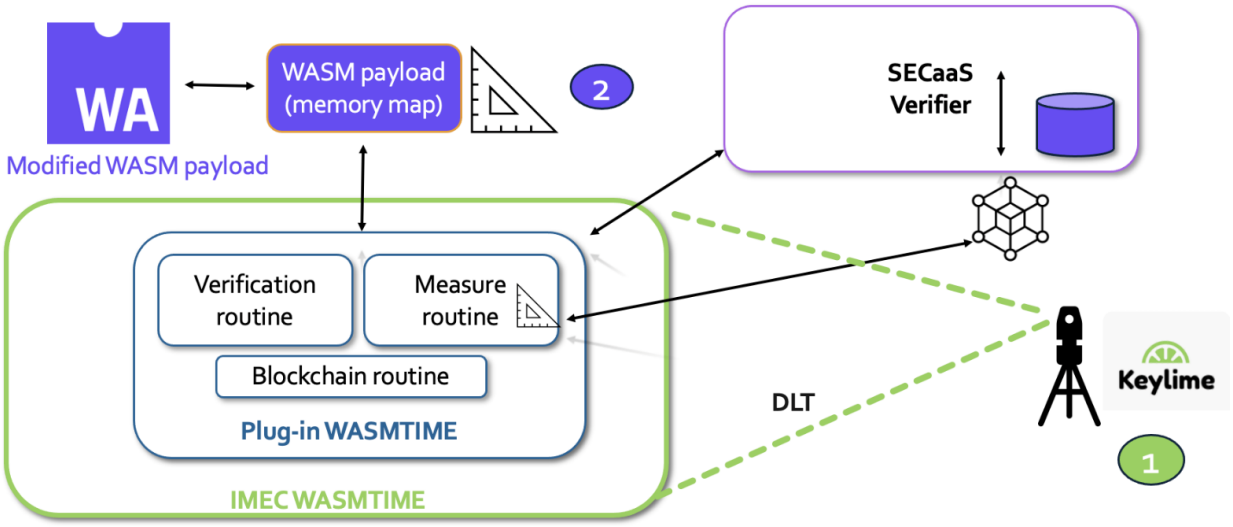

Figure 27. NATWORK's full stack remote attestation

5. Conclusions

The deliverable D4.1 “Payload security per runtime, intelligent runtime selection and attestation.r1” presented a comprehensive overview of the NATWORK aspects enabling secure, flexible-runtime workloads. This is actually a report on the payload security per runtime, the intelligent runtime selection as well as the remote attestation, that derive from NATWORK innovations.

While containerized software and virtual machines have been employed for years in software orchestration, many security aspects of containers and security optimization through orchestration have been neglected in research and frameworks. This document highlights the neglected factors, summarizing the work performed in each of them.

From the security perspective, relevant runtimes are presented, including VMs, microVMs, unikernels, containers, and WebAssembly. Each of these is examined in detail, providing an overview of essential security advantages and disadvantages compared to native processes and/or containers. Additional security features and measures are presented through relevant state-of-the-art research and activities performed within T4.2 “AIaaSecS for software payload”. This overview provides the groundwork for the rest of T4.2 and various other tasks within NATWORK concerning runtime choices.

Network and runtime API standardization provides a uniform way of selecting and deploying workloads independently of the runtime executing it. This paves the way for intelligent runtime selection, consisting of intent-based metadata gathering and deployment, which can be combined with an in-progress ML-based approach for workload modeling for optimal orchestration. These efforts feed into the “optimizing selection” part of T3.1 “Secure-by-design federated slice orchestration and management”, along with UC1, for sustainable and reliable 6G services.

The progress concerning remote attestation is presented based on TPM and TEE, which are capable of securing workloads at runtime and may be integrated as a feature within intelligent runtime/node selection at deployment time, aiding with the reliability aspect of UC1.

Finally, the second and final version of the NATWORK payload security per runtime, intelligent runtime selection as well as remote attestation aspects will be described by the D4.2 “Payload security per runtime, intelligent runtime selection and attestation.r2” due to M24 along with detailed capability maps, algorithms and effectiveness of the derived solutions.

References

[1] Singh, A., & Shrivastava, M. (2019). "Security in Hardware Assisted Virtualization for Cloud Computing - State of the Art Issues and Challenges." International Journal of Computer Sciences and Engineering, 7(1), 1-8.

[2] Blenk, A., Basta, A., Reisslein, M., & Kellerer, W. (2015). "Survey on network virtualization hypervisors for software-defined networking." IEEE Communications Surveys & Tutorials, 18(1), 655-685.

[3] Costan, V., & Devadas, S. (2016). "Intel SGX Explained." IACR Cryptology ePrint Archive, 2016(086), 1-118.

[4] NIST Special Publication 800-125B. (2019). "Security Recommendations for Server-based Hypervisor Platforms." National Institute of Standards and Technology. Retrieved from https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-125B.pdf

[5] Sabt, M., Achemlal, M., & Bouabdallah, A. (2015). "Trusted execution environment: What it is, and what it is not." In 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 1, pp. 57-64. IEEE.

[6] Choi, J. S., Renom, L. G., Yun, K., Casellas, R., Martínez, R., Vilalta, R., & Munoz, R. (2024). "Microsegmentation of a Microservice-Based Transport Control Plane for Multitenant Optical Virtual Networks." IEEE Network.

[7] Rehman, A., Alqahtani, S., Altameem, A., & Saba, T. (2014). "Virtual machine security challenges: case studies." International Journal of Machine Learning and Cybernetics, 5, 729-742.

[8] Kuenzer, S., Bădoiu, V.-A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., … Huici, F. (2021, April). Unikraft. *Proceedings of the Sixteenth European Conference on Computer Systems*. doi:10.1145/3447786.3456248

[9] Abeni, L. (2023). Real-Time Unikernels: A First Look. In *Lecture Notes in Computer Science* (pp. 121–133). doi:10.1007/978-3-031-40843-4_10

[10] Walli, S. R. (1995). The POSIX family of standards. *StandardView*, *3*(1), 11–17.

[11] Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., & Zolotarov, V. (2014). OSv—Optimizing the Operating System for Virtual Machines. *2014 Usenix Annual Technical Conference (Usenix Atc 14)*, 61–72.

[12] Bellard, F. (2005). QEMU, a fast and portable dynamic translator. *USENIX Annual Technical Conference, FREENIX Track*, *41*, 46. Califor-nia, USA.

[13] Habib, I. (2008). Virtualization with KVM. *Linux Journal*, *2008*(166), 8.

[14] Goethals, T., Sebrechts, M., Al-Naday, M., Volckaert, B., & Turck, F. D. (2022, July). A Functional and Performance Benchmark of Lightweight Virtualization Platforms for Edge Computing. *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*. doi:10.1109/edge55608.2022.00020

[15] Anjali, Caraza-Harter, T., & Swift, M. M. (2020, March). Blending containers and virtual machines. *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. doi:10.1145/3381052.3381315

[16] Holmes, B., Waterman, J., & Williams, D. (2024). Severifast: Minimizing the root of trust for fast startup of sev microvms. *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 1045–1060.

[17] Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D., … Lesli.e., I. (2015, May). Jitsu: Just-In-Time Summoning of Unikernels. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 559–573.

[18] Russell, R. (2008). virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, *42*(5), 95–103.

[19] Lee, H. (2014). Virtualization basics: Understanding techniques and fundamentals. *School of Informatics and Computing Indiana University*, *815*.

[20] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., & Popa, D.-M. (2020). Firecracker: Lightweight virtualization for serverless applications. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 419–434.

[21] Talbot, J., Pikula, P., Sweetmore, C., Rowe, S., Hindy, H., Tachtatzis, C., … Bellekens, X. (2020). A security perspective on Unikernels. *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, 1–7. IEEE.

[22] Bratterud, A., Walla, A.-A., Haugerud, H., Engelstad, P. E., & Begnum, K. (2015). IncludeOS: A minimal, resource efficient unikernel for cloud services. 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (Cloudcom), 250–257. IEEE.

[23] Wollman, A., & Hastings, J. (2024, October). A Survey of Unikernel Security: Insights and Trends from a Quantitative Analysis. In 2024 Cyber Awareness and Research Symposium (CARS) (pp. 1-9). IEEE.

[24] Jarkas, Omar, et al. "A Container Security Survey: Exploits, Attacks, and Defenses." ACM Computing Surveys (2025).

[25] Sultan, S., Ahmad, I., & Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead. IEEE access, 7, 52976-52996. Jarkas, Omar, et al. "A Container Security Survey: Exploits, Attacks, and Defenses." ACM Computing Surveys (2025).

[26] Chen, J., Feng, Z., Wen, J. Y., Liu, B., & Sha, L. (2019, March). A container-based DoS attack-resilient control framework for real-time UAV systems. In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE) (pp. 1222-1227). IEEE.

[27] Haq, M. S., Nguyen, T. D., Tosun, A. Ş., Vollmer, F., Korkmaz, T., & Sadeghi, A. R. (2024, May). SoK: A comprehensive analysis and evaluation of docker container attack and defense mechanisms. In 2024 IEEE Symposium on Security and Privacy (SP) (pp. 4573-4590). IEEE.

[28] Kalafatidis, Sarantis, et al. "Experiments with Digital Security Processes over SDN-Based Cloud-Native 5G Core Networks." 2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN). IEEE, 2024.

[29] Perrone, G., & Romano, S. P. (2025). WebAssembly and Security: a review. Computer Science Review, 56, 100728.

[30] Bhansali, S., Aris, A., Acar, A., Oz, H., & Uluagac, A. S. (2022, May). A first look at code obfuscation for webassembly. In Proceedings of the 15th ACM conference on security and privacy in wireless and mobile networks (pp. 140-145).

[31] Mavridis, I., & Karatza, H. (2021). Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. Concurrency and Computation: Practice and Experience, 35(11). doi:10.1002/cpe.6365

[32] Kjorveziroski, V., & Filiposka, S. (2023). Webassembly orchestration in the context of serverless computing. Journal of Network and Systems Management, 31(3), 62.

[33] Qi, S., Kulkarni, S. G., & Ramakrishnan, K. K. (2020). Understanding container network interface plugins: design considerations and performance. 2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN, 1–6. IEEE.

[34] Koukis, G., Skaperas, S., Kapetanidou, I. A., Mamatas, L., & Tsaoussidis, V. (2024). Performance Evaluation of Kubernetes Networking Approaches across Constraint Edge Environments. arXiv [Cs.NI]. doi:10.48550/ARXIV.2401.07674

[35] Marcelino, C., & Nastic, S. (2023). CWASI: A WebAssembly Runtime Shim for Inter-Function Communication in the Serverless Edge-Cloud Continuum. 2023 IEEE/ACM Symposium on Edge Computing (SEC), 158–170. IEEE.

[36] Randazzo, A., & Tinnirello, I. (2019). Kata containers: An emerging architecture for enabling mec services in fast and secure way. 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), 209–214. IEEE.

[37] Mavridis, I., & Karatza, H. (2023). Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. Concurrency and Computation: Practice and Experience, 35(11), e6365.

[38] Goethals, T., Al-Naday, M., Volckaert, B., & De Turck, F. (2024). Warrens: Decentralized Connectionless Tunnels for Edge Container Networks. IEEE Transactions on Network and Service Management, 21(4), 4282–4296. doi:10.1109/tnsm.2024.3417703

[39] Goethals, T., De Turck, F., & Volckaert, B. (2021). Self-organizing Fog Support Services for Responsive Edge Computing. Journal of Network and Systems Management, 29(2). doi:10.1007/s10922-020-09581-6

[40] Kinney, S. (2006). 6 - Platform Configuration Registers. Trusted Platform Module Basics, 53–64. Newnes. DOI: 10.1016/B978-075067960-2/50007-5.

[41] Cooper, D., Polk, W., Regenscheid, A., & Souppaya, M. (2011). Special Publication 800-147: BIOS Protection Guidelines. NIST. Gaithersburg.

[42] Arthur, W., Challener, D., & Goldman, K. (2015). A Practical Guide to TPM 2.0. Apress. DOI: 10.1007/978-1-4302-6584-9.

[43] Trusted Computing Group. (2019). Trusted Platform Module Library Part 1: Architecture.

[44] Trusted Computing Group. (2019). Trusted Platform Module Library Part 3: Commands.

[45] Trusted Computing Group. (n.d.). Trusted Computing Group. Retrieved from https://trustedcomputinggroup.org/.

[46] TPM2-Software. (n.d.). Developer community for those implementing APIs and infrastructure from the TCG TSS2 specifications. Retrieved from https://github.com/tpm2-software.

[47] Schear, N., Cable, P. T., Moyer, T. M., Richard, B., & Rudd, R. (2016). Bootstrapping and maintaining trust in the cloud. Proceedings of the 32nd Annual Conference on Computer Security Applications, 65–77. ACM. DOI: 10.1145/2991079.2991104.

[48] Keylime Project. (n.d.). Keylime/Attestation-Operator: Keylime easily deployable on Kubernetes/OpenShift. Retrieved from https://github.com/keylime/attestation-operator.

[49] Thijsman, J., Sebrechts, M., De Turck, F., & Volckaert, B. (2024). Trusting the Cloud-Native Edge: Remotely Attested Kubernetes Workers. *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*, 1–6. https://doi.org/10.1109/ICCCN61486.2024.10637515

[50] AMD. (2020). *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. AMD. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[51] Anati, I., Gueron, S., Johnson, S., & Scarlata, V. (2013). *Innovative Technology for CPU Based Attestation and Sealing*. https://www.semanticscholar.org/paper/Innovative-Technology-for-CPU-Based-Attestation-and-Anati-Gueron/708a3c03556b5bc20b5bd8e58ef2f47f6a9fc7d2

[52] Birkholz, H., Thaler, D., Richardson, M., Smith, N., & Pan, W. (2023). *Remote ATtestation procedureS (RATS) Architecture* (RFC No. 9334). Internet Engineering Task Force (IETF). https://www.rfc-editor.org/rfc/rfc9334

[53] *IAS End of Life Announcement*. (2023, November 20). https://community.intel.com/t5/Intel-Software-Guard-Extensions/IAS-End-of-Life-Announcement/m-p/1545831#M6018

[54] Intel Corporation. (2023a). *Intel® Trust Domain Extensions Data Center Attestation Primitives (Intel® TDX DCAP): Quote Generation Library and Quote Verification Library* (No. 0.9). Intel Corporation. https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/Intel_TDX_DCAP_Quoting_Library_API.pdf

[55] Intel Corporation. (2023b). *Intel® Trust Domain Extensions (Intel® TDX)*. Intel Corporation. https://cdrdv2.intel.com/v1/dl/getContent/690419

[56] *Keystone-enclave/keystone*. (2025). [C]. Keystone Enclave. https://github.com/keystone-enclave/keystone (Original work published 2018)

[57] Intel Corporation. (2025). *Attestation Service for Intel® Software Guard Extensions: API Documentation* (Technical Report No. 7.2). Intel Corporation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/sgx-ias-using-epid-eol-timeline.html

[58] Advanced Micro Devices, Inc. (2025). *Versioned Chip Endorsement Key (VCEK) Certificate and KDS Interface Specification* (No. 57230; Version 1.00). Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/57230.pdf

[59] *Enarx/enarx*. (2025). [Rust]. Enarx. https://github.com/enarx/enarx (Original work published 2019)

[60] Bravi, E., Berbecaru, D. G., & Lioy, A. (2023). A Flexible Trust Manager for Remote Attestation in Heterogeneous Critical Infrastructures. *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 91–98. https://doi.org/10.1109/CloudCom59040.2023.00027

[61] Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., ... & Wilkes, J. (2020, April). Borg: the next generation. In Proceedings of the fifteenth European conference on computer systems (pp. 1-14).

[62] M.Lacoste, V.Lefebvre, Trusted Execution Environments for Telecoms: Strengths, Weaknesses, Opportunities, and Threats, IEEE security and privacy Journal, 2023.

[63] Abbasimehr, H., & Paki, R. (2022). Improving time series forecasting using LSTM and attention models. Journal of Ambient Intelligence and Humanized Computing, 13(1), 673-691.

[64] Shumway, R. H., Stoffer, D. S., Shumway, R. H., & Stoffer, D. S. (2017). ARIMA models. Time series analysis and its applications: with R examples, 75-163.

[65] Zhang, L., Bian, W., Qu, W., Tuo, L., & Wang, Y. (2021, April). Time series forecast of sales volume based on XGBoost. In Journal of Physics: Conference Series (Vol. 1873, No. 1, p. 012067). IOP Publishing.

[66] Wu, J., Chen, X. Y., Zhang, H., Xiong, L. D., Lei, H., & Deng, S. H. (2019). Hyperparameter optimization for machine learning models based on Bayesian optimization. Journal of Electronic Science and Technology, 17(1), 26-40.

[67] Belete, D. M., & Huchaiah, M. D. (2022). Grid search in hyperparameter optimization of machine learning models for prediction of HIV/AIDS test results. International Journal of Computers and Applications, 44(9), 875-886.

[68] ENISA: Cyber risk management implementating guidance (Oct 2023): https://www.enisa.europa.eu/sites/default/files/2024-11/Implementation%20guidance%20on%20security%20measures_FOR%20PUBLIC%20CONSULTATION.pdf

[69] Sun, J. et al SELWasm: A Code Protection Mechanism for WebAssembly. December 2019 Computer Science IEEE Intl Conference

[70] Bai, Y., Li, B., & Zhang, W. (2020). A collaborative approach to detecting DDoS attacks in SDN using entropy and deep learning. *Journal of Telecommunications and Information Technology*.

[71] Gao, C., Wang, Z., & Yu, H. (2020). Distributed DDoS defense: A collaborative approach at Internet scale. *IEEE Xplore*.

[72] Zhang, X., Zhao, C., & Chen, Y. (2020). Enhancing DDoS attack detection and network resilience through ensemble-based packet processing and bandwidth optimization. *ResearchGate*.

[73] Briggs, I., Day, M., Guo, Y., Marheine, P., & Eide, E. (2014). A performance evaluation of unikernels. In *Technical Report*.

[74] Li, L., Fan, Y., Tse, M., & Lin, K. Y. (2020). A review of applications in federated learning. Computers & Industrial Engineering, 149, 106854.

[75] Aldhyani, T. H., & Alkahtani, H. (2022). Artificial intelligence algorithm-based economic denial of sustainability attack detection systems: Cloud computing environments. Sensors, 22(13), 4685.

[76] National Vulnerability Database - CVE-2014-0515 Detail - https://nvd.nist.gov/vuln/detail/CVE-2014-0515

[77] National Vulnerability Database - CVE-2021-20087 Detail - https://nvd.nist.gov/vuln/detail/CVE-2021-20087

[78] ETSI NFV Security WG: ETSI GR NFV-SEC 007 V1.2.1 (2024-11), report of Attestation Technologies and Practices (2024-11)