



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

D3.6 Pre-Deployment Security per Construction Measures.r2

Lead beneficiary	TSS	Lead author	Vincent Lefebvre
Reviewers	Roberto Gonzalez (NEC), Maxime Vinh Hoa La, Edgardo Montes de Oca (MONT)		
Type	R	Dissemination	PU
Document version	V1.0	Due date	30/06/2026



Co-funded by
the European Union

6G SNS

Project funded by



Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

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



UK Research
and Innovation

Project information

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

Document information

Associated WP	WP3
Associated task(s)	T3.4
Main Author(s)	Vincent Lefebvre (TSS)
Author(s)	Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Alexandros Papadopoulos, Maria Anna Pistela, Donatos Stavropoulos, Nikolaos Makris, Eleni Chamou, Stelios Mpatziakas, Vangelis V. Kopsacheilis, Dimitrios Manolakis, Aristeidis Papadopoulos, Nikolaos Vakakis, Konstantinos Nikiforidis, Athanasios Korakis, Anastasios Drosou (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Gupta Shankha, Mays Al-Naday (UESSEX), Maria Safianowska (ISRD), Wissem Soussi (ZHAW)
Reviewers	Roberto Gonzalez (NEC), Maxime Vinh Hoa La, Edgardo Montes de Oca (MONT)
Type	R - Document, Report
Dissemination level	PU - Public
Due date	M30 (30/06/2026) – Extended to M31 (07/07/2026)
Submission date	07/07/2026

Document version history

Version	Date	Changes	Contributor (s)
r1.v0.1	16/06/2025	Initial table of contents	Vincent Lefebvre (TSS)
r1.v0.2	11/07/2025	Initial contributions in various sections	Vincent Lefebvre, Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Alexandros Papadopoulos, Aristeidis Papadopoulos, Konstantinos Nikiforidis, Athanasios Korakis, Anastasios Drosou (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Sumeyya Birtane, Mays Al-Naday (UESSEX), Maria Safianowska (ISRD)
r1.v0.3	24/07/2025	Updated content in sections 4 and 5	Vincent Lefebvre, Mark Angoustures (TSS)
r1.v0.4	25/08/2025	Revised content for sections 2 and 3	Vincent Lefebvre, Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Eleni Chamou, Stelios Mpatziakas, Athanasios Korakis, Anastasios Drosou (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Sumeyya Birtane, Mays Al-Naday (UESSEX), Maria Safianowska (ISRD)
r1.v0.5	10/09/2025	Pre-final version for late inclusions	Vincent Lefebvre (TSS)
r1.v0.6	16/09/2025	Last refinements in various sections	Vincent Lefebvre, Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Eleni Chamou, Stelios Mpatziakas, Vangelis V. Kopsacheilis (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Sumeyya Birtane, Mays Al-Naday (UESSEX), Maria Safianowska (ISRD), Wissem Soussi (ZHAW)
r1.v0.7	19/09/2025	Draft ready for peer review	Vincent Lefebvre (TSS)
r1.v0.7.5	23/09/2025	Reviewed version	Md Munjure Mowla, Robert Gdowski (ISRD), Gürkan Gür (ZHAW)
r1.v0.8	26/09/2025	Version after addressing peer reviewers' recommendations	Vincent Lefebvre (TSS), Virgilios Passas (CERTH)
r1.v0.8.5	29/09/2025	Quality review	Joachim Schmidt (HES-SO)
r1.v0.9	08/10/2025	Final review and refinements	Vangelis V. Kopsacheilis, Antonios Lalas (CERTH)

Version	Date	Changes	Contributor (s)
r1.v1.0	10/10/2025	Final version ready for submission	Antonios Lalas (CERTH)
r2.v0.1	11/03/2026	Initial table of content	Vincent Lefebvre (TSS)
r2.v0.2	28/03/2026	Content-integrated version first edition	TSS team
r2.v0.3	30/04/2026	Initial contributions in various sections	Vincent Lefebvre, Mark Angoustures (TSS), Antonios Lalas, Virgilios Passas, Donatos Stavropoulos, Nikolaos Makris, Alexandros Papadopoulos, Aristeidis Papadopoulos, Anastasios Drosou (CERTH), Vinh La, Edgardo Montes de Oca, Manh Nguyen (MONT), Sumeyya Birtane, Mays Al-Naday (UESSEX), Maria Safianowska (ISR)
r2.v0.4	29/05/2026	Contributions for all Sections	All authors
r2.v0.5	05/06/2026	Version ready for review	Vincent Lefebvre (TSS), Virgilios Passas (CERTH)
r2.v0.6	12/06/2026	Review complete and feedback to co-authors	Roberto Gonzalez (NEC), Maxime Vinh Hoa La, Edgardo Montes de Oca (MONT)
r2.v0.7	20/06/2026	Refinement based on review comments	All authors
r2.v0.8	24/06/2026	Quality review	Joachim Schmidt (HES-SO)
r2.v0.9	30/06/2026	Final review and refinements	Virgilios Passas (CERTH) and CERTH team
r2.v1.0	07/07/2026	Final version for submission	Antonios Lalas (CERTH)

Disclaimer

Co-funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or 6G-SNS. Neither the European Union nor the granting authority can be held responsible for them. The European Commission is not responsible for any use that may be made of the information it contains.

While the information contained in the documents is believed to be accurate, the authors(s) or any other participant in the NATWORK consortium make no warranty of any kind with regard to this material including, but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Neither the NATWORK Consortium nor any of its members, their officers, employees, or agents shall be responsible or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein.

Without derogating from the generality of the foregoing neither the NATWORK Consortium nor any of its members, their officers, employees, or agents shall be liable for any direct or indirect or consequential loss or damage caused by or arising from any information advice or inaccuracy or omission herein.

Copyright message

© NATWORK Consortium. This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation, or both. Reproduction is authorised provided the source is acknowledged.

Contents

Contents	6
List of acronyms and abbreviations.....	14
List of figures.....	17
List of tables.....	19
Executive summary	21
1. Introduction.....	22
1.1. <i>Purpose and structure of the document</i>	22
1.2. <i>Intended Audience</i>	23
1.3. <i>Interrelations</i>	23
1.4. <i>Advancements with respect to deliverable D3.5</i>	24
2. State of the art analysis.....	26
2.1. <i>Introduction</i>	26
2.1.1. Definition and usability	26
2.1.2. Interaction with service-level security solutions	28
2.1.3. Interaction with D2.1 service level security services.....	28
2.1.4. Interaction with D3.1	30
2.1.5. Summary	30
2.2. <i>PDSCMs for containers</i>	31
2.2.1. Introduction	31
2.2.2. Confidential containers frameworks	32
2.2.3. Pre-deployment Microservice Security by construction	37
2.2.4. Kubernetes security analysis.....	38
2.2.5. O-RAN xAPP security	41
2.2.6. Evolution of xApp security needs and NATWORK perspective	42
2.2.7. Container availability.	47
2.3. <i>Binary pre-deployment hardening techniques</i>	50

2.3.1.	General.....	50
2.3.2.	Confidentiality preservation	51
2.3.3.	Integrity preservation	52
2.3.4.	Availability preservation	53
2.4.	<i>Web Assembly security</i>	<i>56</i>
2.4.1.	WASM technology history and key design attributes	56
2.4.2.	WASM security analysis	57
2.4.3.	WASM module integrity.	57
2.4.4.	SotA Takeaways	61
2.4.5.	WASM confidentiality preservation.....	62
2.4.6.	WASM availability preservation.....	64
3.	NATWORK's PDSCMs on containerized payloads	65
3.1.	<i>General.....</i>	<i>65</i>
3.2.	<i>Hardened K8s orchestration, through helm charts sanity scans</i>	<i>65</i>
3.2.1.	General.....	65
3.2.2.	Progress beyond the state of the art	68
3.2.3.	Progress towards D3.5	69
3.3.	<i>PDSCMs on microservices</i>	<i>72</i>
3.3.1.	Beyond the state of the art.....	73
3.4.	<i>O-RAN xAPP security</i>	<i>73</i>
3.4.1.	IS-RD Liquid xAPP threat model.....	73
3.4.2.	Runtime verification	77
3.4.3.	JSON Runtime Checker	79
3.4.4.	FALCO-based Runtime Behavioural Monitor.....	79
3.4.5.	Beyond the state of the art.....	80
3.5.	<i>Kubernetes-powered automatized remote attestation</i>	<i>83</i>
3.5.1.	Domain and problem statement	83
3.5.2.	Leveraging Kubernetes for xApp security enforcement.....	84
3.5.3.	D-MUTRA modified workflow	86

3.5.4.	Progress from D 3.5	87
3.6.	<i>Work on container availability.....</i>	<i>87</i>
3.6.1.	General.....	87
3.6.2.	Restricted container visibility by the deployment architecture	87
3.6.3.	Solution: xNOPs CPU-bound sidecar.....	88
3.6.4.	Results	92
3.6.5.	Penalty/Measurement overhead	93
3.6.6.	Beyond the state of the art.....	93
3.6.7.	Progress versus D3.5	94
4.	NATWORK's PDSCMs on native workloads	95
4.1.	<i>TEE-based confidentiality and integrity for MMT probe</i>	<i>95</i>
4.1.1.	MMT threat model.....	95
4.1.2.	Beyond the state of the art.....	99
4.1.3.	Progress from D3.5	100
4.2.	<i>Flexible and cost-effective native workload performance monitoring</i>	<i>103</i>
4.2.1.	Domain.....	103
4.2.2.	Problem statement	103
4.2.3.	Solution	104
4.2.4.	Detection model	104
4.2.5.	Implementation	105
4.2.6.	Per-basic-block elementary cost.....	107
4.2.7.	Validation methodology	107
4.2.8.	Test infrastructure	108
4.2.9.	Measures.....	109
4.2.10.	Interpretation	113
4.2.11.	Conclusion	113
4.2.12.	Beyond the state of the art.....	114
4.2.13.	Progress towards D3.5	114

4.3. Dynamic User Association and Resource Allocation in Modern Heterogeneous Cellular Networks	114
4.3.1. Domain	114
4.3.2. Problem Statement	115
4.3.3. Solution	116
4.3.4. Distributed game-theoretic decision model	116
4.3.5. Reinforcement-learning orchestration scheme	117
4.3.6. Evaluation Methodology	118
4.3.7. Results	118
4.3.8. Conclusion	120
4.3.9. Beyond the state of the art	120
4.3.10. Progress from D3.5	121
4.4. Native xApp remote attestation and runtime integrity verification	121
4.4.1. Domain and problem statement	121
4.4.2. NATWORK solution. D-MUTRA for xApp runtime integrity verification	123
4.4.3. Implementation and workflow	123
4.4.4. Dashboard	126
4.4.5. KPI compliance verification	127
4.4.6. Beyond the state of the art	129
4.4.7. Progress beyond D3.5	130
5. NATWORK's PDSCM on WASM workloads	131
5.1. WASM integrity	131
5.1.1. General	131
5.1.2. Progress versus the state of the art	131
5.1.3. Progress versus D3.5	131
5.2. WASM confidentiality	131
5.2.1. Introduction	131
5.2.2. Problem statement	132
5.2.3. Solution	133

5.2.4.	Implementation	133
5.2.5.	Validation Methodology	135
5.2.6.	Progress Beyond the State of the Art	137
5.2.7.	Conclusion	138
5.2.8.	Evolutions from D3.5	139
5.3.	<i>WASM availability</i>	139
5.3.1.	Domain and problem statement	139
5.3.2.	Solution	139
5.3.3.	First implementation and results	141
5.3.4.	Consistency validation across payloads	143
5.3.5.	Challenging our proof of concept. Tests	144
5.3.6.	Measured Penalty	146
5.3.7.	Beyond the state of the art	147
5.3.8.	Progress from D3.5	147
5.3.9.	Conclusion	147
6.	Conclusion	149
	References	152
	Annexes	161
	<i>Annex A List of DoS Attacks (non-exhaustive)</i>	<i>161</i>
	<i>Annex B Feasibility study. Using control flow probes to detect DoS attacks</i>	<i>162</i>
B.1	Introduction. Use case 4.6 description	162
B.1.1	General	162
B.1.2	Principle and core ideas	162
B.1.3	KPIs	162
B.1.4	Identified risks	163
B.1.5	Control flow timestamping	163
B.1.5.1	General	163
B.1.5.2	Principle of performance monitoring by control flow probing	163
B.1.5.3	Need for selective placement for sustainability	164

B.1.5.4	Control flow graph undecidability, breaking accuracy	164
B.1.5.5	Principle of code block tracing for anomalous behavior detection	165
B.2	Scoping phase	165
B.2.1	General.....	165
B.2.2	Scoping the objective.....	165
B.2.3	Discussed technical statements.....	166
B.2.4	Conclusions	166
B.3	Time-domain analysis	167
B.3.1	Succinct reminder of the Use Case 4.6 objective	167
B.3.2	Enumeration of time domain metrics.....	167
B.3.3	Control Graph inserted performance probes	168
B.3.3.1	Definition and scope	168
B.3.3.2	Per process isolation	168
B.3.3.3	Performance penalty occasioned by the monitoring	168
B.3.3.4	Workload preparation, if required	169
B.3.3.5	Required platform administrative right, if required	169
B.3.3.6	Usability	169
B.3.3.7	Metric discriminative strength	169
B.3.4	Conclusion.....	171
B.4	Hardware Performance Counters (HPC).....	171
B.4.1	Definition and scope	171
B.4.2	Per process isolation	172
B.4.3	Performance penalty occasioned by the monitoring	172
B.4.4	Workload preparation, if required	173
B.4.5	Required platform administrative right, if required	173
B.4.6	Usability	173
B.4.6.1	HTTP GET Flood (Layer 7).....	173
B.4.6.2	Typical Characteristics	175
B.4.6.3	Detection by Classical IDS	175

B.4.6.4	Relevance of HPC detection.....	175
B.4.6.5	Algorithmic Complexity Attacks (ACA).....	177
B.4.6.6	Detection by Classical IDS	177
B.4.6.7	Relevance of HPC detection.....	178
B.5	Process CPU use rate	180
B.5.1	Definition and scope	180
B.5.2	Performance penalty occasioned by the monitoring	180
B.5.3	Workload preparation, if required	180
B.5.4	Required platform administrative	180
B.5.5	Usability	180
B.6	Intel Processor Trace (IPT)	181
B.6.1	Definition and scope	181
B.6.2	Per process isolation	181
B.6.3	Performance penalty occasioned by the monitoring	181
B.6.4	Workload preparation, if required	182
B.6.5	Required platform administrative	182
B.6.6	Usability	182
B.6.6.1	IPT for Control Flow Integrity.....	182
B.6.6.2	IPT For DoS detection	183
B.7	Frequency-domain analysis	184
B.7.1	General.....	184
B.7.2	Frequency-Domain Analysis for Early DoS Detection	185
B.7.3	Graph-Based Detection coupled with frequency analysis.....	185
B.7.4	Benefits and costs of conversion to the frequency domain	185
B.7.4.1	Benefits	185
B.7.4.2	Costs.....	186
B.7.4.3	Relevance of frequency-domain analysis for early DoS detection	186
B.7.4.4	FFT on control flow probe metrics.....	186
B.7.4.5	FFT on HTTP GET flood Attacks by HPC	187

B.7.4.6	Performance Costs	188
B.7.4.7	Algorithmic Complexity attacks	188
B.8	Conclusion of the feasibility study	189
B.8.1	Lesson learnt and discoveries	189
B.8.2	Feasibility study conclusion	190

List of acronyms and abbreviations

Abbreviation	Description
3GPP	3rd Generation Partnership Project
ACA	Adaptive Chosen-Cyphertext Attack
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AFL	American Fuzzy Lop (fuzzer)
API	Application Programming Interface
B5G	Beyond 5G
BB	Basic Block
BS	Base Station
BPF	Berkeley Packet Filter (eBPF: extended BPF)
CDF	Cumulative Distribution Function
CF	Call Frequency
CIA	Confidentiality Integrity Availability (security triad)
CLI	Command-Line Interface
CPU	Central Processing Unit
CRE	Cell Range Expansion
CSI	Channel State Information
CUSUM	Cumulative Sum (control chart)
DPDK	Data Plane Development Kit
DPI	Deep Packet Inspection
DRL	Deep Reinforcement Learning
D-MUTRA	DLT-based Mutual Remote Attestation framework
ELF	Executable and Linkable Format
ET	Execution Time
ES-GT	Exhaustive Search-Orchestrated Game
EWMA	Exponentially-Weighted Moving Average
FFT	Fast Fourier Transform
GCM	Galois/Counter Mode
GCC	GNU Compiler Collection
GDB	GNU Debugger (used to inspect the execution of a program)
HetNet	Heterogeneous Cellular Network
HKDF	HMAC-based Key Derivation Function
HPC	Hardware Performance Counter
HTTP	HyperText Transfer Protocol
IND-CPA	Indistinguishability under Chosen-Plaintext Attack
IPT	Intel Processor Trace
IV	Initialization Vector

Abbreviation	Description
JOP	Jump Oriented Programming (attack)
KDF	Key Derivation Function
KPI	Key Performance Indicator
LBR	Last Branch Records
LLVM	LLVM compiler infrastructure
MARL	Multi-Agent Reinforcement Learning
MMT	Montimage Monitoring Tools
MQTT	Message Queuing Telemetry Transport
NIC	Network Interface Card
NIST	National Institute of Standards and Technology
NT-RIC	Near Real-Time Radio RAN intelligent Controller
NOP	No Operation (instruction)
O-CU	Open Central Unit
O-RAN	Open Radio Access Network (industry organization)
OCI	Open Container Initiative
OS	Operating System
PCAP	Packet Capture
PDSCM	Pre Deployment Security per Construction Measure
PGO	Profile-Guided Optimization
PMC	Performance Monitoring Counter
PSNE	Pure Strategy Nash Equilibrium
QoS	Quality of Service
ROP	Return Oriented Programming (attack)
SCOUT	Confidentiality-compliant Cyber Hygiene Observability of Vulnerability Threats
SEV-SNP	Secure Encrypted Virtualization — Secure Nested Paging (AMD)
SIEM	Security Information and Event Management
SINR	Signal-to-Interference-plus-Noise Ratio
SGX	Software Guard Extensions (Intel)
SMP	Symmetric Multi-Processing
SMT	Simultaneous Multitreading
SYN	Synchronized (SYN flood attack breaks handshake synchronization)
TDX	Trust Domain Extensions (Intel)
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
UARA	User Association and Resource Allocation
UE	User Equipment
UDP	User Datagram Protocol
WAF	Web Application Firewall
WASI	WebAssembly System Interface

Abbreviation	Description
Wasm	WebAssembly
xNOP	Several consecutive NOP instructions

List of figures

Figure 1. General xAPP authentication in near RT-RIC	43
Figure 2. O-RAN theoretical full stack remote attestation framework	45
Figure 3. CPU pressure saturation	49
Figure 4. WASM runtime-orchestrated module authentication	60
Figure 5. Theoretical WASM remote attestation framework.....	61
Figure 6. Pod manifest with root privileges and no resource limits.....	66
Figure 7. NetworkPolicy allowing unrestricted ingress traffic.....	66
Figure 8. RBAC binding granting cluster-admin to a service account.....	67
Figure 9. Service of type LoadBalancer exposing an internal API.....	67
Figure 10. LiquidRAN Kubernetes Runtime Monitoring Architecture	78
Figure 11. Three modes of workload preparation for D-MUTRA security from native to K8s orchestration.....	85
Figure 12 . Two common container deployment architectures	88
Figure 13. Illustration of performance evolution during CPU contention.....	89
Figure 14.Illustration of calculating total CPU shares in an unknown environment.....	91
Figure 15. Comparison of xNOPs execution time against the formula-based model	92
Figure 16. TEEMig multi-cluster architecture	99
Figure 17. Detailed compilation pipeline showing the placement of the BBProbeV2 pass.	105
Figure 18. Detailed runtime pipeline including EWMA computation and reporting output channels	106
Figure 19. ELF memory layout produced by the BBProbeV2 pass	107
Figure 20 Algorithmic flow of the proposed DQN orchestration scheme	117
Figure 21. Native xApp for Anomaly Traffic Dection and its interaction with the near RT RIC elements	122
Figure 22. D-MUTRA layout for xApp integrity verification.....	124
Figure 23. General workflow for module encryption (Build and Runtime).....	134
Figure 24. Binary format for an encrypted WASM module.....	134
Figure 25. WASMTIME two execution modes	140

Figure 26. Probe insertion into WASMTIME runtime decoding loop	141
Figure 27. Number of function calls versus Number of primes in Prime(x)	142
Figure 28. Patched block frequency vs. Prime1e3 execution time	145
Figure 29. Mean inter-block duration vs. Prime1e3 execution time.....	145
Figure 30. Control flow probe insertion	163

List of tables

Table 1. PDSCM enumeration	26
Table 2. Potential PDSCMs leverage in D2.1 Security Services	28
Table 3. Security challenges by PDSCM leverage as per D2.1	29
Table 4. Potential PDSCMs leverage in D3.1 MANO	30
Table 5 Confidential Containers Comparison	37
Table 6. CPU contention visibility delivered to containers.....	47
Table 7. Native code performance monitoring techniques.....	55
Table 8. Strengths and weaknesses of WASM security	57
Table 9. WASM authentication and remote attestation techniques	59
Table 10. WASM confidentiality techniques.....	62
Table 11. Final evaluation dataset and manifest-level decisions	70
Table 12: Final detection metrics.....	71
Table 13. Liquid near-RT RIC xApp STRIDE analysis	74
Table 14. Illustration of Kubernetes leverage merits	86
Table 15. Penalty induced by xNOPs on Prime (10^5)	93
Table 16. Per-rate-step throughput comparison: Standard VM vs AMD SEV-SNP VM. All values in packets per second. P5–P95 range after ramp-up trimming (first 5 s per step discarded). n = 50 steady-state samples per step per environment.....	101
Table 17. MMT probe self-monitoring used testbed specifications.....	109
Table 18. Five different compiled MMT probes for testing.....	110
Table 19. CPU expense of MMT probe self monitoring.....	111
Table 20. Penalty induced by the monitoring on MMT functions.....	112
Table 21: System Metrics Relative to the Balanced Policy for 20 UEs	118
Table 22: System Metrics Relative to the Balanced Policy for 40 UEs	119
Table 23: System Metrics Relative to the Balanced Policy for 80 UEs	119
Table 24: System Metrics of the Coverage-Enhancing Policy for 40 and 80 UEs	119
Table 25: System Metrics of the Power-Aware Policy for 20 and 40 UEs	120

Table 26. KPI validation for native xAPP remote attestation and runtime integrity verification	128
Table 27. KPI 1.3.1 compliance for xApp remote attestation.....	128
Table 28. KPI 1.3.3 compliance for xApp remote attestation.....	129
Table 29. KPI 1.3.4 compliance for xApp remote attestation.....	129
Table 30. WASM module decryption time measurements	136
Table 31. WASM confidentiality performance penalty measurements	136
Table 32. Confidentiality-protected WASM HTTP server penalty assessment	137
Table 33. Function calls versus x (for PRIME(x) wasm module	142
Table 34. WASM monitoring validation runs.....	143
Table 35. Measured penalty induced by WASM monitoring	146
Table 36. Summary of D3.6 outcomes versus D3.5	149
Table 37. Performance metric by inserted probes discriminative power towards all types of DoS attacks.	170
Table 38. Relevance of HPC metrics for DoS detection	174
Table 39. Methods for detecting HTTP GET flood DoS attacks by classical IDS	176
Table 40. Comparison of HTTP GET flood DoS detection abilities between classical IDS and HPC-based method	176
Table 41. Application Complexity DoS attacks detection by classical IDS	177
Table 42. Comparison of Algorithmic Complexity DoS detection abilities between classical IDS and HPC-based method	178
Table 43. Comparison of HTTP GET flood and Algorithmic complexity attacks detection by classical IDS	179
Table 44. Use of Intel Processing Trace for Control Flow Integrity verification	182
Table 45. Influence on the control flow by DoS types.....	183
Table 46. Usefulness of converting control flow probe time series in the frequency domain ..	187
Table 47. FFT relevance for HTTP GET flood attacks	187

Executive summary

This deliverable D3.6 “Pre-Deployment Security per Construction Measures.r2” presents the Pre-Deployment Security per Construction Measures (PDSCM) developed in the NATWORK project, in its second version, following D3.5 “Pre-Deployment Security per Construction Measures.r1” submitted in October 2025.

The PDSCM term refers to security mechanisms and actions applied prior to payload deployment, aiming to strengthen software artefacts against security threats from the outset. The work addresses three main types of software payloads executed across the computing continuum: native binaries, containerized applications, and WebAssembly (WASM) modules.

This document is therefore an evolution of D3.5, stressing the progress made by the project since then, with relevant sub-sections inserted in the different sections covering the three different types of workloads. From D3.5 edition, novel project outcomes are described here, covering the three types of software workloads and related to their confidentiality, integrity and availability. D3.6 therefore covers the full mapping and coverage, filling uncovered areas of D3.5.

This deliverable stresses where and how NATWORK consortium has put efforts over the different workload formats, resulting in promising researches and collaborative developments. The deliverable highlights how these developments align well with NATWORK performance to security reconciliation principle and in which extent these developments bring a step on the state of the art in their domain. The results demonstrate how PDSCMs contribute to NATWORK’s overall vision of secure, performant, and sustainable operations across the computing continuum.

The document also integrates a feasibility study, referring to use case 4.6 which leverages predevelopment probe insertion, a PDSCM technique per-se, making it relevant with this document. This feasibility study is the first milestone of UC 4.6.

1. Introduction

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

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

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

1.1. Purpose and structure of the document

The purpose of the document is to assess and position how PDSCMs are aligned with the NATWORK concept and beneficial to 6G deployments.

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

Sections:

2. **Section 2 - SotA:** Presents an evolution towards the state of the art as delivered in D3.5
3. **Section 3 - NATWORK PDSCMs on containerized payloads:** Describes the NATWORK specific research actions towards container-oriented PDSCMs
4. **Section 4 - NATWORK PDSCMs on native payloads:** Describes the NATWORK specific research actions towards native-oriented PDSCMs.
5. **Section 5 - NATWORK PDSCMs on WASM:** Describes the NATWORK specific research actions towards WASM-oriented PDSCMs
6. **Section 6 - Conclusions:** Wraps up the document, reflecting on the project's strategic orientation and establishing expectations for future milestones.
7. **Annex A and B - DoS Attacks and Feasibility study:** This section provides a list of DoS attacks and integrates a feasibility study related to the detection of DoS attacks through network function control flow probing. This section is an extra content non existing in D3.5, closely related to PDSCMs and use case 4.6.

1.2. Intended Audience

The NATWORK D3.6 Deliverable Pre-Deployment Security per Construction Measures.r2 is for Public Dissemination. It is there devised for the internal and external use of the NATWORK consortium, comprising members, project partners, affiliated stakeholders and the public. This document mainly focuses on the pre-deployment security per construction measures of the project, thereby serving as a referential tool throughout the project's lifespan.

1.3. Interrelations

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

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

The current **D3.6 – “Pre-Deployment Security per Construction Measures.r2”** deliverable addresses specific software payload hardening techniques, applied at deep and elementary level (i.e., software payloads), reflecting the work of task T3.4. However, high level security services, applied at the different layers and technical domains of NATWORK (RAN, cloud, core, data plane, orchestration and management layer), described in Deliverable D2.1 – **“General State of the Art Security”** and in Deliverable D3.1 – **“Secure by design orchestration and Management”** are all based on different and distributed software payloads. By hardening these payloads, PDSCMs directly impact the security of security services.

D3.6 is also related to D2.2 – **“Use Case Scenarios and Requirements”**, where use cases will implement PDSCM (e.g., native PDSCM applied on MMT probe takes part of Use Case 4.5 Enabling optimized explainable MTD). Finally, D3.6 will feed the integration and validation efforts

within WP6, for evaluating and improving the assets presented in this deliverable. Use case UC 4.6 is notably discussed in Annex B integrating its feasibility study.

1.4. Advancements with respect to deliverable D3.5

This deliverable introduces significant technical advancements compared to D3.5, either by increasing the maturity and comprehensiveness of previously presented concepts and implementations through refinements, complementary developments, and additional details up to the validation stage, or by introducing novel subsections that have also been developed and validated.

Indeed, at the time of delivery of D3.5, some technical solutions were not sufficiently mature, had not progressed enough, or had only been conceived and sketched, preventing their inclusion in the deliverable.

D3.6 therefore constitutes a major technical step forward and represents the final developments of the Pre-Deployment Security by Construction Measures (PDSCM) across the three categories of software workloads, thereby fully addressing Task T3.4. In particular, it delivers confidentiality, integrity, and availability mechanisms for containerized workloads, x86 native workloads, and WASM modules.

In the domain of containerized workloads, AI-based secure Kubernetes orchestration and microservice security through dynamic RBAC represent significant advances compared to the state of D3.5. Furthermore, Kubernetes-based fully automatic and zero-touch remote attestation, together with a novel method for inferring CPU contention expressed as the number of (virtual) CPU-bound workers on a CPU, are entirely new subsections with respect to D3.5. Moreover, the runtime integrity of xAPP containers has been significantly enhanced through two new runtime checks introduced since D3.5, namely JSON validation and Falco-based system-call checkers.

Furthermore, the solution of User Association and Resource Allocation for UC4.3 is presented. Mainly, a comprehensive state-of-the-art review has been completed, leading to the clear formulation of the research question and the full development of the corresponding mathematical model and simulation framework. The proposed solution has also been thoroughly evaluated, and the resulting performance outcomes have been obtained and analyzed.

In the domain of x86 native workloads, the initial work presented in D3.5 on integrity and confidentiality has been further advanced through TEEmig, a novel technique for migrating hot processes into a Trusted Execution Environment (TEE). In parallel, a complementary technique based on selective placement of frugal performance-monitoring probes leveraging the LLVM compilation chain has been implemented and validated. Finally, the D-MUTRA blockchain-based

remote attestation framework has been adapted to support the Anomalous Traffic Detection native xAPP workload.

In the domain of WASM modules, D3.5 primarily focused on module integrity. D3.6 extends this work by introducing two novel techniques that provide both confidentiality and availability guarantees.

Finally, this deliverable includes a feasibility study for Use Case 4.6, which investigates the deployment of performance probes along the control flow of a network function to detect DoS attacks. Since the techniques employed fall within the scope of PDSCM, we considered this deliverable to be the appropriate place to present this work. This feasibility study is also a novel contribution compared with D3.5.

2. State of the art analysis

2.1. Introduction

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

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

2.1.1. Definition and usability

2.1.1.1. Definition

As a simple definition, PDSCMs embrace the following criteria:

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

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

Table 1. PDSCM enumeration

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

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

2.1.1.2. Workload migratability

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

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

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

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

2.1.2. Interaction with service-level security solutions

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

2.1.3. Interaction with D2.1 service level security services

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

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

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

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

2.1.3.1. Matching D2.1 priority challenges

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

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

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

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

2.1.4. Interaction with D3.1

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

Table 4. Potential PDSCMs leverage in D3.1 MANO

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

2.1.5. Summary

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

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

2.2. PDSCMs for containers

2.2.1. Introduction

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

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

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

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

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

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

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

2.2.2. Confidential containers frameworks

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

2.2.2.1. Confidential Containers (CoCo)

Security

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

Performance

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

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

Sustainability

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

2.2.2.2. MarbleRun

Security

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

Performance

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

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

Sustainability

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

2.2.2.3. Parma

Security

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

Performance

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

Sustainability

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

2.2.2.4. Trusted Container Extensions (TCX)

Security

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

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

Performance

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

Sustainability

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

2.2.2.5. Conclusions

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

Table 5 Confidential Containers Comparison

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

2.2.3. Pre-deployment Microservice Security by construction

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

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

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

2.2.4. Kubernetes security analysis

2.2.4.1. Main security features

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

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

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

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

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

2.2.4.2. Kubernetes security strengths

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

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

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

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

2.2.4.3. Kubernetes security weaknesses

While Kubernetes offers robust security features, the literature identifies several weaknesses and recurring challenges that can undermine its overall security posture. A key issue arises from insecure or improper configurations, such as weak authentication, poorly defined network policies, and excessive permissions. Sometimes administrators set up Kubernetes with weak or missing security settings. Examples of this include giving users, Pods, or services more permissions than they really need, exposed credentials issues, manifesting by leaving passwords, API keys, or tokens in plain text inside configuration files, and default root access, which means letting containers run as root inside Pods, which makes it easier for attackers to break into the host. These misconfigurations increase the risk of unauthorized access, data breaches, and privilege escalation, leaving clusters open to attack. Closely related is the problem of root access and privilege escalation, where containers often run with elevated privileges by default, enabling malicious actors to break out of their containers and compromise the underlying host system if not properly restricted [18]. The empirical study of Kubernetes manifests [17] finds widespread security misconfigurations, such as missing security context, excessive privileges and hard-coded credentials. These significantly increase the attack surface and confirm that misconfiguration is one of the most critical and common weaknesses.

Another major weakness is the limited strength of namespace isolation. Although namespaces are designed to separate workloads logically, they do not provide strong security isolation. Without additional controls such as strict network segmentation or Pod security restrictions, attackers who gain access to one namespace may be able to move laterally across the cluster. The paper [18] also highlights that improper access control, including weakly configured role-based access control (RBAC), can allow users or services to perform unauthorized actions, undermining both confidentiality and integrity of workloads.

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

2.2.5. O-RAN xAPP security

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

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

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

- O-RAN ALLIANCE TS, “O-RAN Near-RT RIC Architecture [19]
- O-RAN Security Requirements and Controls Specifications [20]
- O-RAN Study on Security for Service Management and Orchestration (SMO) [21]
- O-RAN Study on Security for O-Cloud [22]
- O-RAN Study on Zero Trust Architecture for O-RAN 2.0 [23]
- O-RAN xAPP SDK [24]
- O-RAN Study on Security for Near Real Time RIC and xApps 5.0 [25]
- O-RAN.WG1.TS.OAD-R005-v16.00 – Use Cases and Overall Architecture (2026) [26]
- O-RAN.WG11.TR.CSM-Security-R004-v01.00 – Study on Continuous Security Monitoring (2026) [27]
- O-RAN.WG11.TR.AppLCM-Security-R004-v04.00 – Study on Security for Application Lifecycle Management (2025) [28]
- O-RAN.WG11.TR.CSM-Security-R005-v02.00 – Study on Continuous Security Monitoring (2026) [29]
- O-RAN.WG11.TR.O-CLOUD-Security.0-R005-v09.00 – O-RAN Work Group 11 (Security Work Group) [30]
- O-RAN.WG11.TS.SRCS.0-R005-v14.00 – O-RAN Work Group 11 (Security Work Group) [31]

2.2.6. Evolution of xApp security needs and NATWORK perspective

Variations can be observed across the extensive WG11 publications, particularly regarding remote attestation and runtime integrity verification. These topics are not addressed with the same level of priority or accuracy across documents. Notably, runtime integrity verification is still considered a “nice-to-have” in the mid-term, as it involves unresolved technical challenges. These gaps are addressed in Sections 3.4 (JSON/FALCO runtime checkers), 3.5 (Kubernetes-based attestation), and 4.4 (D-MUTRA for native xApp integrity)

In a general perspective, our analysis of the recently issued document reveals that WG11 is progressively adopting these two concepts, aligning more closely with the ETSI NFV Security Working Group, which is now explicitly referenced. We believe that techniques developed within NATWORK, particularly D-MUTRA—which relaxes operational and technical constraints for both remote attestation and runtime integrity verification—are becoming increasingly relevant.

2.2.6.1. xAPP authentication process

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

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

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

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

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

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

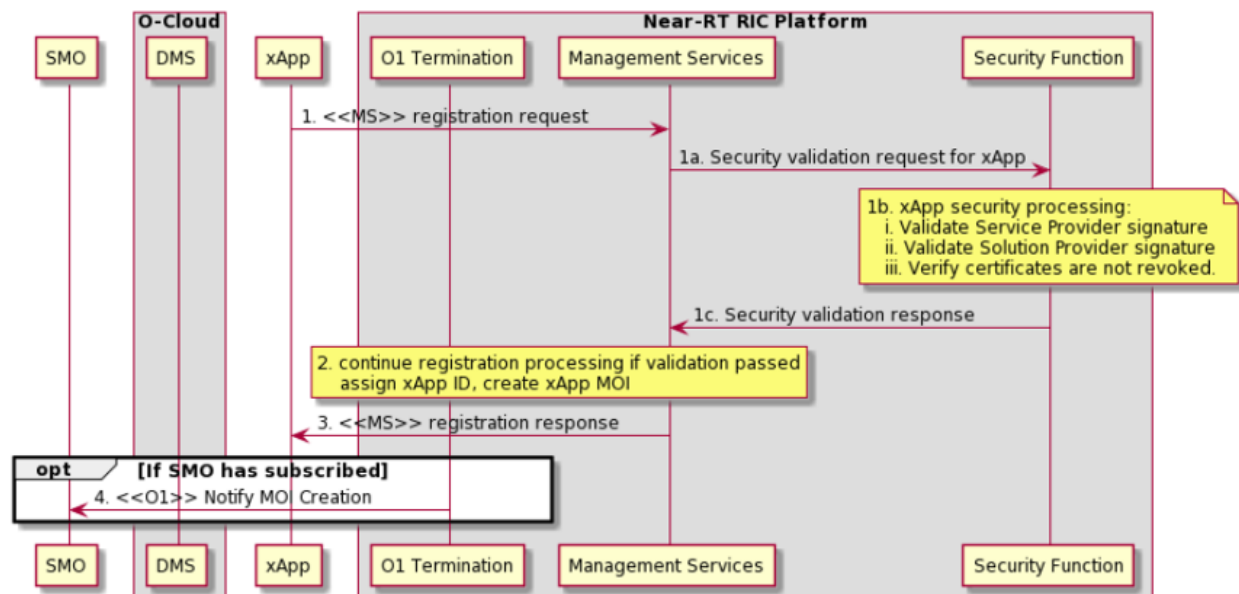


Figure 1. General xAPP authentication in near RT-RIC

xAPP onboarding verification workflow

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

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

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

Takeaways and identified security gaps

- The above-described workflow relates to the authentication process initiated with a registration in a catalogue and the verification of attributes in correspondence with the catalogue-stored artefact during on-boarding.
- The process is multi-stakeholder and complex. The xAPP integrity verification is done by comparing a signed digest stored at the SMO.
- The SMO acts as the security guard, making the peripheral checks before use. Once the xAPP integrity has been checked by the SMO, it is no longer checked thereafter. The SMO is trusted to deliver correct onboarding triggers. A compromise SMO can produce corrupted xAPP onboarding demands luring the near RT-RIC.
- The near RTRIC consumes both and then validates the xAPP MOI, creating a trust anchor issue. A corrupted near RT RIC can vet unauthorized xAPP being trusted.
- The threat model considered by O-RAN excludes a corrupted near RT-RIC and a corrupted SMO.

2.2.6.2. WG11 statement on xAPP remote attestation

In O-RAN Security Requirements and Controls Specifications 11.0 [20], WG11 recommends a conceptually defined remote attestation service (AS) for providing additional benefits besides verifying the O-Cloud platform integrity by Chain of Trust. WG11 stipulates that the remote AS should be extended to include O-RAN Applications integrity as depicted in Figure 2.

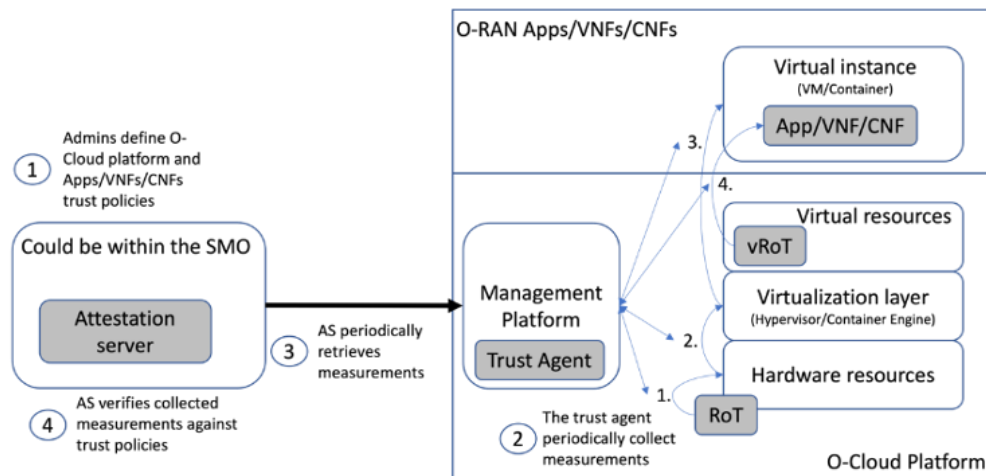


Figure 2. O-RAN theoretical full stack remote attestation framework

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

2.2.6.3. xAPP runtime integrity verification

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

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

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

2.2.6.4. xAPP SBOM management

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

2.2.6.5. Takeaways

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

On xApp authentication

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

On xApp remote attestation

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

On xAPP runtime integrity verification

- WG 11 stresses the requirement for runtime integrity verification, breaking the threat model as employed in authentication (i.e., corrupted SMO, corrupted near RT-RIC)

- WG 11 positions runtime integrity as a research challenge, notably stressing the impact of performance induced by periodic integrity verification. In the NATWORK project, this can be viewed as a security gap to cover.

On xAPP SBOM management

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

2.2.7. Container availability.

2.2.7.1. Domain and Problem Statement

In multi-tenant cloud environments, tenants have limited visibility into the real CPU resources available to their workloads. Traditional Linux metrics and standard monitoring tools either **lack visibility** into external tenant activity or **lack the precision** required to characterize actual CPU contention. The metrics delivered are listed in the Table 6.

Table 6. CPU contention visibility delivered to containers

Command / File	Default Scope	Information
/proc/stat	Node (VM)	Global CPU statistics (user, system, idle, steal, etc.)
/proc/uptime	Node (VM)	Time since boot + global idle time
/proc/loadavg	Node (VM)	Load average (1, 5, 15 min)
/proc/meminfo	Node (VM)	Total, free, buffers, and cache memory
/sys/fs/cgroup/ mpstat	Pod / Container	CPU (usage, throttling), memory, PIDs per cgroup
mpstat	Node (VM)	Aggregated CPU statistics (per CPU, % idle, % iowait, etc.)
top	Pod / Container	List of visible processes
/proc/pressure/ cpu	Node (VM)	CPU Pressure (time spent in CPU contention)
Steal Time (/proc/stat)	Hypervisor / VM	CPU time stolen by other VMs

In addition, is it possible to precisely **quantify the level of CPU contention** and infer **the number of concurrent workloads** competing for shared CPU resources in such environments?

To address this limitation and to attempt to answer this question, this work introduces **xNOPs**, a monitoring approach that estimates CPU contention by analyzing the execution time of a deterministic CPU-bound sidecar workload. Unlike conventional observability mechanisms, **xNOPs** aims to quantify hidden multi-tenant interference by **estimating the equivalent number**

of **concurrent fully utilized workers** (i.e., workloads operating at 100% resource usage) sharing the underlying infrastructure. This abstraction provides a normalized contention metric, where a given value by **xNOPs** may correspond in practice to a larger number of partially utilized real-world workloads.

2.2.7.2. *State of the Art*

The first instinct is often straightforward: we would like to use a tool like **top** and immediately see all running processes and their CPU consumption, but **top** only shows the processes inside the pod by default. Then we might look at other signals such as **/proc/stat** or **/proc/pressure/cpu**, assuming they provide a more global perspective of what the node is doing. Here is an overview of the main Linux commands and metrics in containerized environments and the scope of information they expose from a pod or container. But before diving into their specific limitations, it is important to understand a more fundamental point: all of these metrics operate on a scope that is not fixed. Depending on how the environment is configured, they can shift from a node-wide view to something much narrower, such as a single cgroup, a container namespace, or even being fully blocked through mechanisms like PID and mount namespaces, **cgroup v2 scoping**, **/proc masking**, **seccomp filters**, or **LSM policies** such as **AppArmor** and **SELinux**.

CPU Utilization Rate (%)

CPU utilization rate is one of the most common metrics used in Linux and cloud environments. It indicates how much CPU time is being consumed over a given period, but it provides very limited information about actual CPU contention. For example, what does a CPU utilization of 50% or even **100%** really tell us about contention? In practice, nothing.

A CPU utilization of 100% **may simply mean that the tenant's own workloads are fully** using the available CPU resources without experiencing any interference from neighboring tenants. In this case, the CPUs are busy, but contention can still be close to zero.

Conversely, the exact same 100% utilization may also occur in a heavily contended environment where multiple external workloads continuously compete for the same physical CPUs. The same ambiguity exists with lower utilization values such as 5% or 50%. A workload may appear lightly active while still suffering significant scheduling delays and performance degradation caused by co-located tenants.

Pressure Stall Information (PSI)

Linux Pressure Stall Information (**PSI**) was introduced to provide visibility into resource contention at the scheduler level. For CPU resources, PSI measures the proportion of time during

which tasks are waiting for CPU availability. Compared to CPU utilization, PSI provides a more direct indication of contention. However, it still presents important limitations.

First, PSI values are computed as rolling averages over time windows such as **10 seconds**, **1 minute**, and **5 minutes**. This smoothing effect reduces sensitivity to short and abrupt workload variations, making rapid contention changes harder to detect accurately.

Second, PSI availability is limited. It requires recent Linux kernels (4.20+) and is frequently disabled by default in production or industrial environments. More importantly, PSI quickly saturates under heavy load as shown in Figure 3.

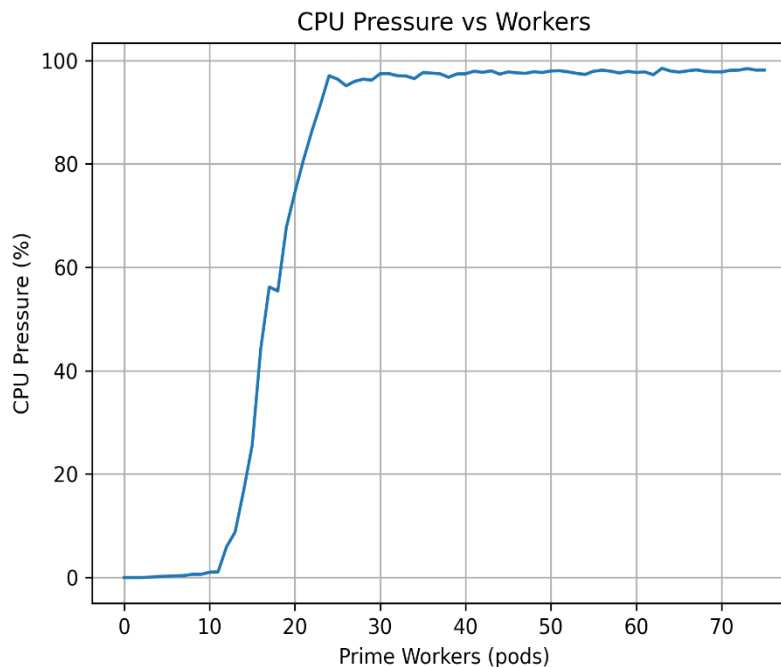


Figure 3. CPU pressure saturation

As the number of concurrent workloads increases, CPU pressure rises progressively until reaching values close to 100%. Beyond this point, the metric no longer differentiates contention levels effectively. For example, the observed CPU pressure may remain almost identical whether 25, 50, or 70 workloads compete on the same node, despite substantial differences in actual contention and application performance degradation.

Steal Time (%)

Steal time indicates the amount of time during which a virtual machine was ready to execute but was not scheduled by the hypervisor. Sounds promising, but it has several interpretation limitations as well.

First, it is an aggregated metric at the VM level (often across all vCPUs), which does not necessarily reflect the actual impact on a specific workload. For example, in the case of a **CPU-bound** workload with limited parallelism, part of the steal time may affect lightly used cores, making the application-level impact lower than what the metric suggests.

Second, this metric is computed over observation windows: it is therefore instantaneous and must be aggregated over time to distinguish structural phenomena from transient variations.

Finally, steal time only measures **inter-VM contention** (time stolen by the hypervisor) and provides no visibility into **intra-VM contention**. In architectures where multiple tenants coexist within the same VM, **this metric becomes unusable**, as it does not capture internal CPU competition effects.

2.2.7.3. *Related work*

Performance variability in multi-tenant cloud systems has been widely studied. In [32], the authors use unsupervised learning over infrastructure-level telemetry such as CPU utilization and latency to detect anomalous behavior consistent with resource contention. While effective for identifying interference, these methods remain correlative and cannot isolate CPU-specific contention or quantify the number of concurrent workloads responsible for degradation.

Similarly, in [33], the authors study interference through controlled VM co-location experiments using host- and hypervisor-level measurements. It shows that shared resources induce non-linear performance degradation but remains offline and descriptive: it does not provide real-time detection, continuous monitoring, or workload-level attribution in operational environments.

2.3. Binary pre-deployment hardening techniques

2.3.1. General

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

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

2.3.2. Confidentiality preservation

The SotA integrates the following PDSCM techniques. In the last three decades, academic surveys [34][35][36][37] have covered the promises, efficiency and performance impact of the different techniques employed in code security.

2.3.2.1. Code encryption

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

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

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

2.3.2.2. Code obfuscation

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

A large variety of code obfuscation techniques have been designed and used over time (e.g., control flow flattening, code virtualization, instruction substitution, opaque predicate/junk code insertion, data structure obfuscation, symbol and string encryption) [38][39][40]. Each of them brings in one specific context (i.e., when applied to one specific executable) a different efficiency and performance impact. The performance impact can be significant (i.e., in the range of 100% for complex code virtualization or control flow flattening), restricting their usage to security-sensitive code and skilled integration teams. Obfuscation requires a specific set up activity on each payload. AI-based tools simplify this workflow [41][42]. AI-based deobfuscation tools

[43][44] de-obfuscate code, recognizing obfuscated code patterns and removing them. Code obfuscation is a direct PDSCM as the code shall be modified through obfuscation patterns.

2.3.2.3. Trusted Execution Environment

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

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

2.3.3. Integrity preservation

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

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

2.3.3.1. Application-level security by PDSCM

The SotA contains the following integrity preservation measures:

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

PDSCM consist in application-level security as stated above.

2.3.4. Availability preservation

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

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

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

2.3.4.2. User-level arbitrary resource allocation

Systems deliver users sufficient administrative rights to adjust the resource allocated to a process (e.g., Linux's *cgroup*, *nice/renice* scheduling priority, *taskset* to bind processes and containers to cores). Hypervisor administrators permit permissioned users to adjust the CPU resource allocation (e.g., allocation of vCPUs assigned to a VM, adjust CPU shares, limits, reservations (e.g., VMware, vSphere, Xen, KVM), and finally by pinning vCPUs to physical CPUs.

2.3.4.3. CPU Performance monitoring by tooling

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

Operating system commands

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

Integrated performance application

A large set of Monitoring applications belongs in the SotA:

- **Nagios [45]**, a widely used infrastructure monitoring (with plugins for CPU usage).
- **Zabbix [46]**, a popular open-source monitoring system with CPU usage metrics.
- **Prometheus + Grafana [47]**, a modern full stack for time-series CPU metrics and dashboards.
- **Datadog [48]**, a cloud-based monitoring with strong CPU profiling and alerts.
- **New Relic [49]**, a SaaS monitoring for applications and infrastructure, including CPU usage.

2.3.4.4. Native code performance (ie, execution speed) instrumentation

To assess the execution speed of a software workload, several instrumentation techniques are employed. These techniques are typically used in the domain of network functions which execution speed must be assessed continuously to infer potential CPU contention. Existing work relevant to the problem can be grouped into five families, summarised in Table 7.

Static compile-time instrumentation (AFL [51], LLVM [52], PGO [53], gcov [54], Sanitizer Coverage [55] achieves basic-block granularity but targets fuzzing, post-mortem profiling or coverage rather than continuous in-process degradation detection.

Dynamic binary rewriting (Pin [56], DynamoRIO [57], Valgrind [58]) is applicable to closed-source binaries but imposes a 10× to 100× runtime cost incompatible with line-rate operation.

Kernel-side and hardware tracing (eBPF/uprobes [59], Intel Processor Trace [60][61], perf_events [62]) leaves the binary untouched but is either too coarse-grained (function entry/exit only) or too data-intensive (gigabytes per minute), and offers no per-basic-block view of application semantics.

Statistical anomaly detection (path-based failure management [63], Pinpoint [64], EWMA/CUSUM control charts [65][66]) contributes the methodology used in the proposed solution but is orthogonal to the question of how the underlying metric stream is to be produced cheaply enough.

Network-probe-specific mechanisms (DPDK statistics [67], OpenTelemetry/Prometheus [68], MMT-Probe’s own routine [69]) report external symptoms or coarse periodic stats and do not identify the internal function in which a slowdown originates.

In [50], a disruptive approach intends to assess the allocated resource level by the payload itself (through payload rewriting) or better through an agent (i.e., sidecar container).

In summary, the state of the art lacks an approach that simultaneously offers (i) basic-block granularity, (ii) compile-time selection of instrumented paths, (iii) bounded production-acceptable overhead, (iv) in-process online detection embedded in the binary, and (v) output through the probe’s existing reporting pipeline.

Table 7. Native code performance monitoring techniques

Family	Representative works	Main limitations
Static compile-time instrumentation	AFL-LLVM, LLVM PGO, gcov, SanitizerCoverage	Designed for fuzzing, post-mortem profiling or coverage; no continuous in-process detection.
Dynamic binary rewriting	Pin, DynamoRIO, Valgrind	10×–100× overhead, incompatible with line-rate probes.
Kernel-side / hardware tracing	eBPF, Intel PT/LBR, perf_events	Coarse granularity or massive data volume; no BB-level view of application logic.
Statistical anomaly detection	Path-based, Pinpoint, EWMA/CUSUM	Methodology only; assumes a low-cost metric stream already exists.
Network-probe-specific	DPDK stats, OpenTelemetry, MMT routine_t	External symptoms or coarse periodic stats; not BB-level, not internal.

2.3.4.5. IPR security

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

Digital Right Management (DRM)

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

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

invariant specific activation token on the permissioned platform. At software start, the machine invariants are collected a second time and reconcile with the activation token through a test, triggering the software start if positive.

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

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

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

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

2.3.4.6. Vulnerability preservation

Vulnerabilities have several origins such as buffer overflow (i.e., exploitation permitted by *Van Neumann* architecture), memory management errors (e.g., use-after-free), Input validation and injection (e.g., SQL injection), race conditions, and higher-level origins (e.g., wrong authentication, misconfiguration, and cryptographic weaknesses).

PDSCM consists of identifying the vulnerability and curating the code (i.e., reprogramming as no curation exists at executable level). Vulnerability detection, however, can be practiced at all steps of the executable file generation (i.e., at programming, on binary executables).

2.4. Web Assembly security

2.4.1. WASM technology history and key design attributes

As stated in D2.1, WASM was defined by Internet browser giants [70], as a substitute to JavaScript highly portable interpreted language. The working group objective was to raise concurrently JavaScript's security, performance and sustainability, in alignment with NATWORK's vision. For that, WASM core asset is its lower-level instruction set, closer to machine atomic operations, faster to interpret and supposedly harder to reverse. Since its standardisation [71] in 2019, WASM has attracted several CPU intensive industries, domains and technologies (e.g., blockchains, FaaS, crypto mining, gaming engines). WASM module interpretation is faster and

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

2.4.2. WASM security analysis

Table 8 provides a rapid view on the key pros and cons of WASM security, deriving from our own study and security surveys [72][73]. We have excluded the generally-cited low-level bytecode instruction set, supposedly making reverse engineering harder, a relative security guarantee (i.e., seasoned reversers are efficient at assembly level, a lower level than WASM instruction set) and AI-enhanced reverse engineering tools will bring instant, near-complete WASM module decompilation in several programming languages outputs [74][75][76].

Table 8. Strengths and weaknesses of WASM security

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

2.4.3. WASM module integrity.

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

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

2.4.3.1. TEE-delivered integrity

By placing the WASM runtime into a TEE, both WASM runtime and module are preserved of confidentiality and integrity attacks. As detailed in [77], using TEE shall be considered and restricted to security-sensitive workloads as it implies higher memory and CPU consumption, leads to novel threat models (e.g., DoS by Raw hammering, evil TCB) and frustrates workload portability (i.e., heterogeneity). The workload-specific performance impact induced by TEE placement also varies with the type of TEE. VM-based optimized TEE (i.e., Intel’s TDX, AMD’s SEV and ARM’s CCA) performance penalty range is generally below 10%. Executable-based TEE (i.e., ARM’s TrustZone, Intel’s SGX) impact is higher and bounded below 30%. In NATWORK, our vision is to consider TEE for these specific security sensitive workloads (e.g., network probe), which deployment can be managed with care and the extra resource consumption measured to be acceptable.

2.4.3.2. Distinction between authentication and remote attestation

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

2.4.3.3. SotA’s Integrity techniques

Table 9 shows the current state of the art related to WASM identity and integrity verification, regrouping the usual techniques of WASM module authentication at onboarding, remote attestation at onboarding, TEE-based remote attestation, and WASM module runtime integrity. Table 9 identifies, for each technique, the verified artefacts attributes, some noticeable operational considerations and the current state of the corresponding SotA.

Table 9. WASM authentication and remote attestation techniques

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

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

2.4.3.4. WASM authentication techniques

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

Browser SubResource authentication

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

Runtime authentication

For WASM runtimes (e.g., WASMTIME, WASMER), programming a DIY (Do It Yourself) protocol leveraging a classical hashing routine is always possible as depicted in Figure 4. Moreover, several tools combine source authentication and payload integrity verification, leveraging a signed hash (i.e., signature) (e.g., Wasm-Sig, WABT, Lucet).

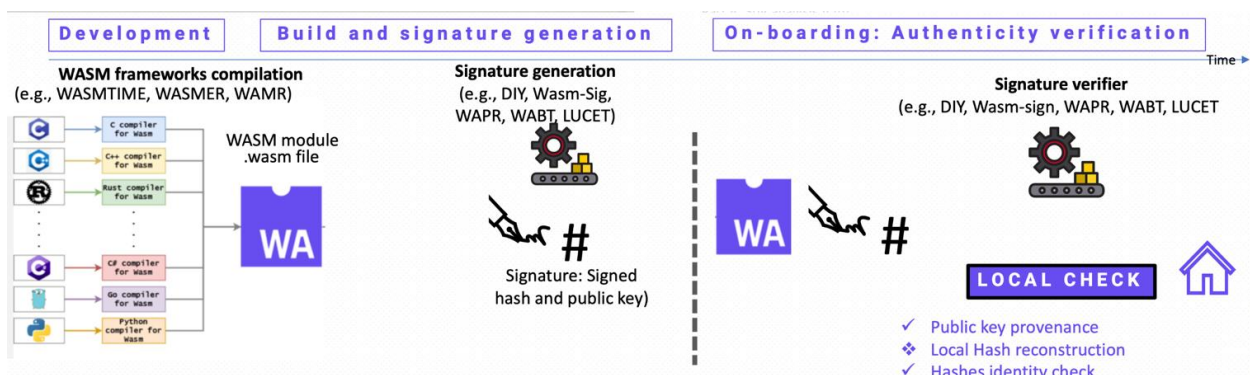


Figure 4. WASM runtime-orchestrated module authentication

2.4.3.5. Remote attestation

To the best of our knowledge, there is no existing remote attestation framework for WASM modules. In networking, WASM modules will be Network Functions and thus ETSI NFV security working group recommendations fully apply [78]. Module authentication suffices when payloads are executed in browsers, while it is not sufficient for networking, where remote attestation is required for service operators.

Theoretical WASM remote attestation implementation

Although we have not found any existing WASM module remote attestation framework, we believe that the first implementations will come soon. A WASM remote attestation framework can be constructed with the support of existing authentication tools, without major technical difficulties. For that, the authentication verifier can be turned into a prover, forwarding signed quotes to a distant verifier, which checks the validity of the prover's public key and the signature by comparison with the same payload's signature stored in its database. This theoretical implementation is illustrated in Figure 5.

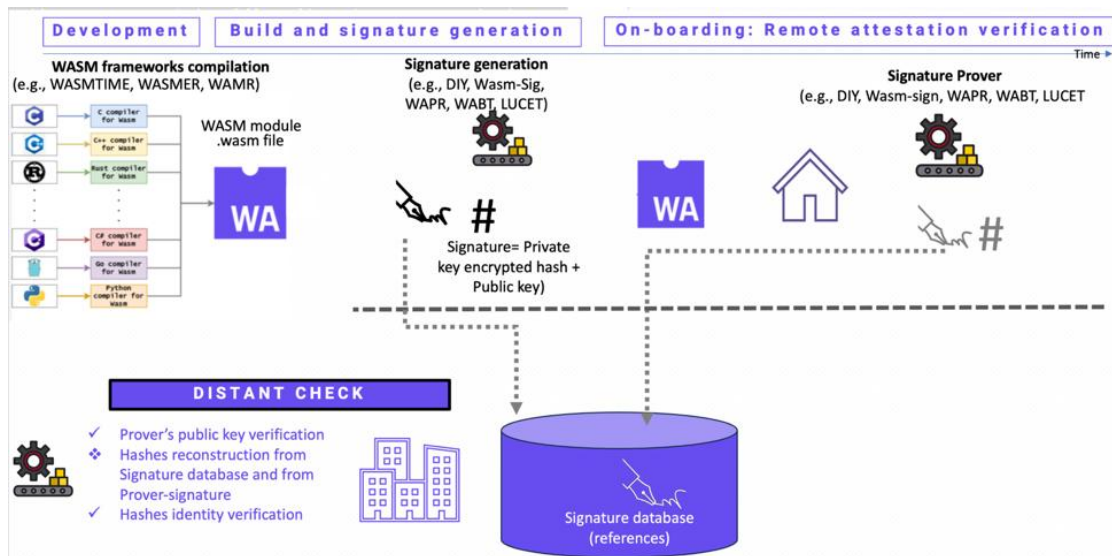


Figure 5. Theoretical WASM remote attestation framework

2.4.4. SotA Takeaways

We have reached the following conclusions:

- WASM authentication is a usual and well-established technique, delivering a security attribute for the workload recipient (i.e., the cloud infra operator) while remote attestation is required for service operators.
- All technical bricks used for authentication can be assembled to construct a remote attestation of the WASM module when they are onboarded.

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

2.4.5. WASM confidentiality preservation

Table 10 summarizes this taxonomy and makes explicit the limitation of each family with respect to the targeted property.

Table 10. WASM confidentiality techniques

Family	Representatives	Limitations	Confidentiality with NETWORK alignment
WASM in TEE	Twine [81](WAMR + Intel SGX); WaTZ (WAMR + ARM TrustZone)	Hardware-bound techniques, restricting deployment with significant memory consumption and CPU cost overhead. The .wasm is assumed to arrive in cleartext within the enclave, or to be protected by TEE-internal primitives — no portable encrypted format	The offered confidentiality does not align with NETWORK security to performance reconciliation principle, due to the limitations stated aside
Hardware-agnostic Wasm computational verification	ZKWASM [82](ZKSNARK over Wasm emulator)	Targets correctness-of-execution proofs, not confidentiality of the distributed module. However, the transformation into arithmetized circuits engender a first layer of obfuscation (ie, possible removal of inputs,	No, these techniques induce very high performance penalty and complex module rewriting without delivering plain confidentiality.

Family	Representatives	Limitations	Confidentiality with NATWORK alignment
		intermediate values, execution traces) which can be enhanced employing additional techniques (e.g., private circuits, encrypting proving systems).	
Hardware-agnostic verified sandbox	WaVe [83](i.e., 45 WASI hostcalls verified with Prusti)	Memory isolation vs the OS, sandbox correctness, with no encryption primitive	No confidentiality is delivered to the WASM module
Homomorphic transformation	None	Homomorphic transformation for WASM are impractical due to the huge CPU costs and difficulty to lower WASM dynamic/runtime model.	Not applicable in practice
WASM obfuscation	[85] WASMixer [86]	Performance overhead varies according to the desired obfuscation level (ie, from 10 to 50%). More importantly, the abstraction level of WASM makes it easy to reverse, being obfuscated or not.	No alignment
Confidential Containers	Kata+SEV-SNP/TDX, CoCo	Runtime confidentiality but hardware-bound, container-level granularity, no native Wasm support	No
Standard Wasm runtimes	Wasmtime, WasmEdge, WAMR	.cwasm precompilation for performance; no official at-rest encryption primitive	Partial (as native code is less lisible)

The review confirms the absence, in the retained literature, of NATWORK-aligned techniques delivering WASM confidentiality, leading us to consider a workflow-fluid simple encryption technique, offering full confidentiality against static analysis and conferring a low overhead, occurring only once prior execution (i.e., by decryption).

2.4.6. WASM availability preservation

Recent research on WebAssembly runtimes mainly focuses on global performance evaluation, runtime optimization, and observability mechanisms. A general overview of these topics is provided by [87].

Several studies analyze the performance characteristics of Wasmtime in cloud and microservice environments, especially regarding cold-start latency and tail-latency trade-offs in production systems [88]. Other works investigate performance debugging and anomaly detection inside WebAssembly runtimes using mutation-based inference techniques [89], or propose flexible non-intrusive dynamic instrumentation methods for WebAssembly [90].

Observability approaches based on eBPF have also been explored for WebAssembly and WASI environments, notably through dedicated performance analysis frameworks [91].

However, existing approaches only partially address the problem considered in this work. Most solutions focus either on offline benchmarking, global runtime measurements, or dynamic instrumentation techniques. They generally do not provide lightweight and fine-grained monitoring of an already compiled runtime binary, nor continuous supervision of long-running or non-terminating executions.

3. NATWORK's PDSCMs on containerized payloads

3.1. General

In this section, we cover our work to secure containerized workloads, complementary to what has been described in D3.5 and which covers many directions including applicated use cases (i.e., containerized xApp) and more generic methods (i.e., hardened K8s orchestration, extended CPU contention visibility in containerized environment, fluidified and globalized D-MUTRA security enforcement leveraging K8s helm charts). Notably, we detail here the work on microservice security, also progressed since D3.5.

We then describe the evolution of our work directed to containerized xApp security. Three directions have been taken:

- xApp two-layer runtime monitoring tooling combining:
 - Layer 1: JSON Runtime Checker: Lightweight Kubernetes API runtime verification through JSON analysis.
 - Layer 2: FALCO-based Runtime Behavioural Monitor: Real-time syscall-level behavioural monitoring through FALCO.
- xApp integrity losses during onboarding and runtime, leveraging K8s helm charts for automatic remote attestation and runtime integrity verification, through declarative security rules for a globalized management of xApps integrity enforcement before onboarding a near RT-RIC.

Finally, we describe a novel method to confer higher visibility on CPU contention status for containerized deployment. This technique usability goes beyond the project as it can be used by anyone deploying a container in a managed and shared environment at the first place.

3.2. Hardened K8s orchestration, through helm charts sanity scans

3.2.1. General

Kubernetes provides several native mechanisms for security, including **Network Policies**, **Role-Based Access Control (RBAC)**, and **Pod Security Admission**. However, studies have shown that security incidents frequently arise not from the absence of these features but from their misconfiguration or misuse. Common problems include exposed credentials, overly broad RBAC assignments, containers running with elevated privileges, and network policies that are either missing or too permissive. These weaknesses are particularly dangerous in distributed environments such as edge clusters and multi-tenant deployments, where the attack surface is naturally wider. Alongside the identified weaknesses presented in D3.5, NATWORK addresses

these challenges by emphasizing **pre-deployment hardening**, ensuring that workloads are validated and secured before they reach the production environment.

To achieve this, we propose the integration of CERTH's AI-based Intrusion Detection System (AI-IDS) into the Kubernetes pre-deployment pipeline. The AI-IDS would act as a policy gatekeeper within the CI/CD process and Admission Controllers, performing in-depth analysis of deployment manifests and configuration files. For example, in Figure 6, **Pod manifests and Helm charts** can be scanned to detect dangerous practices such as privilege escalation (e.g., containers defined with `runAsUser: 0`, running as root), missing resource limits (Pods deployed without `resources.limits`, able to consume unlimited CPU/memory), or the exposure of sensitive credentials (passwords hardcoded in environment variables instead of referencing Kubernetes Secrets).

```

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

```

Figure 6. Pod manifest with root privileges and no resource limits

Another critical domain of analysis involves **Network Policies and RBAC rules**. Here, insecure configurations often allow unrestricted traffic or excessive permissions. For instance, in Figure 7. NetworkPolicy allowing unrestricted ingress traffic a Network Policy with `ingress: { from: [] }` effectively permits all traffic, enabling lateral movement between Pods. Similarly, RBAC bindings that assign `cluster-admin` rights to a microservice service account provide unnecessary and dangerous access to the entire cluster. Even a misconfigured role granting write access to namespaces intended to be read-only can compromise cluster integrity.

```

1  apiVersion: networking.k8s.io/v1
2  kind: NetworkPolicy
3  metadata:
4    name: allow-all-ingress
5  spec:
6    podSelector: {}
7    ingress:
8      - from: [] # <= Allows all sources to connect -> lateral movement risk
9

```

Figure 7. NetworkPolicy allowing unrestricted ingress traffic

```

1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: ClusterRoleBinding
3  metadata:
4    name: insecure-binding
5  subjects:
6    - kind: ServiceAccount
7      name: microservice-sa
8      namespace: default
9  roleRef:
10   kind: ClusterRole
11   name: cluster-admin # <= Grants full cluster-admin rights to a single service
12   apiGroup: rbac.authorization.k8s.io
13

```

Figure 8. RBAC binding granting cluster-admin to a service account

Finally, as shown in Figure 8, **external interfaces and IP bindings** must be reviewed to prevent unintentional exposure of services to the internet. Examples as in Figure 9 include Services of type LoadBalancer created without source IP restrictions, exposing internal APIs publicly, Pods configured with `hostNetwork: true` that bypass the cluster network and bind directly to the host, or workloads mapping host ports such as 22 (SSH) onto every node, unintentionally opening attack vectors at the infrastructure level.

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: public-service
5  spec:
6    type: LoadBalancer # <= Exposes service to the internet by default
7    ports:
8      - port: 8080
9        targetPort: 80
10   selector:
11     app: internal-api # <= An internal API unintentionally made public
12

```

Figure 9. Service of type LoadBalancer exposing an internal API

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

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

The proposed solution integrates **CERTH's AI-IDS** into the Kubernetes pre-deployment pipeline as a policy gatekeeper. By analysing manifests, Helm charts, RBAC rules, and Network Policies

before workloads are deployed, it detects misconfigurations such as privilege escalation, exposed secrets, and overly permissive access. This **shift-left security** approach blocks insecure workloads early, provides immediate feedback to developers, and establishes a hardened baseline for deployment. As a result, NATWORK strengthens Kubernetes security against misconfiguration-driven risks, reducing the likelihood of privilege escalation, data leakage, and lateral movement attacks, while enhancing resilience across cloud-native and edge/O-RAN environments.

3.2.2. Progress beyond the state of the art

The developed Kubernetes pre-deployment security solution advances beyond the state of the art by combining findings from Kubernetes misconfiguration research, Infrastructure-as-Code security analysis, and AI-based intrusion detection into a NATWORK-specific pre-deployment validation workflow. The objective is not to replace existing Kubernetes security mechanisms, but to provide an integrated security gatekeeper that analyses deployment artefacts before they reach cloud-native, edge, and O-RAN production environments.

Several scientific works have shown that Kubernetes security risks frequently originate from configuration weaknesses. Rahman et al. [110] conducted an empirical study of security misconfigurations in open-source Kubernetes manifests, highlighting that Kubernetes configuration files often contain weaknesses that can affect the security posture of deployed applications. Their work demonstrates the importance of systematically analysing Kubernetes manifests, since security risks may be introduced before workloads are even deployed.

The proposed work is also related to the broader field of Infrastructure-as-Code security analysis. Saavedra and Ferreira introduced GLITCH [111], a technology-agnostic framework for detecting security smells in Infrastructure-as-Code scripts. Their work shows the value of transforming configuration artefacts into analysable representations and applying automated security-smell detection. Later work on GLITCH extended this approach to polyglot code smell detection across multiple Infrastructure-as-Code technologies, demonstrating that common security and design smells can be detected across different configuration languages and deployment technologies. This direction is relevant because Kubernetes manifests can be considered part of the Infrastructure-as-Code ecosystem, and the NATWORK prototype follows a similar principle: deployment artefacts are parsed, normalised, and analysed before they are applied to the target environment.

Compared with these research directions, the progress achieved in NATWORK is the development of an integrated pre-deployment security gatekeeper that combines manifest-level analysis, rule-based security validation, AI-IDS positioning, and developer-oriented feedback. The solution analyses multiple Kubernetes artefact types, including Pods, Deployments, DaemonSets, StatefulSets, Jobs, CronJobs, Services, Roles, RoleBindings, ClusterRoles, ClusterRoleBindings,

and NetworkPolicies. This cross-resource approach allows the prototype to detect risks across workload privileges, resource governance, service exposure, access control, and network communication.

The developed solution goes beyond a narrow single-resource scanner by providing a common evaluation workflow for different Kubernetes artefact categories. For example, it can detect when a workload runs as root, when a container lacks resource limits, when sensitive credentials are hardcoded in environment variables, when a NetworkPolicy allows unrestricted ingress, when a service account is bound to cluster-admin, or when a Service of type LoadBalancer is exposed without source IP restrictions. In this way, the prototype connects several misconfiguration classes that are often treated separately in the literature and in operational tools.

The developed solution goes beyond a narrow single-resource scanner by providing a common evaluation workflow for different Kubernetes artefact categories. For example, it can detect when a workload runs as root, when a container lacks resource limits, when sensitive credentials are hardcoded in environment variables, when a NetworkPolicy allows unrestricted ingress, when a service account is bound to cluster-admin, or when a Service of type LoadBalancer is exposed without source IP restrictions. In this way, the prototype connects several misconfiguration classes that are often treated separately in the literature and in operational tools.

3.2.3. Progress towards D3.5

In D3.5, the Kubernetes pre-deployment security solution was presented mainly at concept and design level. The work defined the role of CERTH's AI-based Intrusion Detection System (AI-IDS) as a policy gatekeeper within NATWORK's DevSecOps workflow and described how Kubernetes deployment artefacts could be analysed before reaching the production cluster. The previous version also identified the main categories of Kubernetes misconfiguration to be addressed, including containers running as root, missing resource limits, hardcoded credentials, overly permissive RBAC rules, unrestricted Network Policies, public service exposure, and host-level networking.

Since the previous deliverable, this concept has been implemented as a functional prototype and evaluated through a controlled final evaluation dataset. The main progress consists of three concrete steps. First, a modular Kubernetes manifest scanner was developed to parse YAML deployment artefacts and extract security-relevant configuration fields from Pods, Deployments, DaemonSets, StatefulSets, Jobs, CronJobs, Services, Roles, RoleBindings, ClusterRoles, ClusterRoleBindings, and NetworkPolicies. Second, a set of detection rules was implemented to identify the targeted misconfiguration classes before deployment. Third, a final evaluation

workflow was created to execute the scanner against a representative set of secure and insecure manifests and to produce structured findings, manifest-level decisions, and evaluation metrics.

The implemented prototype produces a structured security assessment for each detected issue. Each finding includes the affected manifest, Kubernetes resource type, resource name, severity level, rule identifier, description of the detected misconfiguration, and recommended remediation action. This makes the solution suitable not only for blocking insecure deployments but also for providing actionable feedback to developers and operators during the CI/CD or admission control process.

The final evaluation was performed using a controlled scenario-based dataset consisting of 50 Kubernetes manifests. The dataset was intentionally expanded compared to the initial prototype validation to provide broader coverage of realistic Kubernetes deployment scenarios. It included 20 secure manifests that were expected to be allowed and 30 insecure manifests that were expected to be blocked. The secure manifests covered properly configured Deployments, Pods, DaemonSets, Services, NetworkPolicies, and RBAC resources. The insecure manifests covered root execution, missing CPU and memory limits, hardcoded credentials, unrestricted ingress traffic, excessive RBAC privileges, wildcard permissions, public LoadBalancer exposure, host networking, host port mappings, privileged containers, privilege escalation, and combined multi-risk configurations.

The dataset also included limitation-oriented cases to evaluate the practical boundaries of the prototype. Particularly, two secure infrastructure-oriented manifests were expected to be allowed but were blocked by the scanner because they used host-level or privileged access. These cases correspond to realistic operational components, such as node-level monitoring or CNI-like agents, which may legitimately require elevated permissions. In addition, three insecure manifests were expected to be blocked but were not detected by the current rule set. These false-negative cases involved sensitive values referenced through ConfigMaps and mutable image tags such as `latest`, which are not yet covered by the implemented checks.

The final evaluation was performed at manifest level. A manifest was considered blocked when the scanner produced one or more findings and allowed when no findings were produced. Out of the 30 insecure manifests, the prototype correctly blocked 27. Out of the 20 secure manifests, the prototype correctly allowed 18. The evaluation therefore produced 27 true positives, 18 true negatives, 2 false positives, and 3 false negatives. The final evaluation results are summarised in Table 11.

Table 11. Final evaluation dataset and manifest-level decisions

Category	Number of manifests
Secure manifests expected to be allowed	20

Category	Number of manifests
Insecure manifests expected to be blocked	30
Total manifests	50
True positives	27
True negatives	18
False positives	2
False negatives	3

Based on these results, the prototype achieved an accuracy of 90.00%, precision of 93.10%, recall of 90.00%, and an F1-score of 91.53%. The metrics are presented in Table 12.

Table 12: Final detection metrics

Metric	Result
Accuracy	90.00%
Precision	93.10%
Recall	90.00%
F1-score	91.53%

The false-positive cases demonstrate that strict pre-deployment security enforcement may incorrectly block legitimate infrastructure workloads when no contextual allow-listing mechanism is available. This is particularly relevant for monitoring agents, CNI components, or other node-level services that may require host networking, host ports, or elevated privileges. Therefore, future versions of the prototype should support trusted workload profiles, namespace-based exceptions, or approval workflows for infrastructure components that require elevated permissions.

The false-negative cases highlight areas where the current detection logic should be extended. Notably, future versions should include checks for sensitive values referenced through ConfigMaps, mutable image tags such as 'latest', missing image digests, unsigned images, insecure registries, and additional supply-chain security indicators. These extensions would reduce the probability of insecure manifests being incorrectly allowed.

Overall, the final evaluation results demonstrate clear progress since the previous deliverable. The work has moved from the design of a Kubernetes pre-deployment hardening concept to an implemented and evaluated prototype. The results confirm that the prototype can detect the majority of the targeted Kubernetes misconfiguration categories before deployment, while the observed false positives and false negatives provide useful guidance for the next development cycle.

3.3. PDSCMs on microservices

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

The project has successfully adopted and implemented Submariner to establish secure inter-cluster communication tunnels, providing encrypted connectivity between clusters. These tunnels provide confidentiality and authentication for cross-cluster service traffic and are integrated into the orchestration workflows. This step guarantees that communication links between clusters are operational and secure at the service level. The use of IPsec has already been validated in initial testbed experiments, enabling secure multi-cluster orchestration.

The automatic (re)configuration of RBAC rules, informed by vulnerability assessment tools, is planned for a subsequent phase of the project to further harden the security baseline. While RBAC rules can already be defined declaratively, NATWORK is exploring automated (re)configuration based on vulnerability assessment results. Lightweight tools such as *Kubesec* can provide recommendations on patching runtimes and tightening access rights, which can then be translated into RBAC policies. This would allow DevSecOps pipelines to dynamically adapt permissions before deployment, aligning the access model with both security requirements and runtime risk assessment.

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

The work on the RBAC policy manager has been mostly completed after D3.5, which included only the initial design and architecture of the policy manager. Work after D3.5 involved, finalising the design and development of the policy manager, along with initial evaluation in lab-based

environment. The policy manager complements the secure-by-design orchestrator to translated security objectives of orchestration into governance policy and persistent RBAC permissions that aligns coherently with the security intents of a 6G slice and the edge-cloud cluster(s) that will deploy segments of it. Together with sFORK for orchestration and SCOUT for cyber hygiene observability, NATWORK achieves wholistic secure-by-design orchestration of 6G slices in edge-cloud continuums.

3.3.1. Beyond the state of the art

NATWORK has developed novel RBAC policy manager, able to audit and translate Rego-based high-level policies into RBAC-structured configurable versions, compatible with ODRL (W3C standard). This enables the translation of complex policies in a set of configurable instructions, enforceable over various cloud custom resources. It further bridges the gap between admission control and API-based authorisation over resources and enables auditing a Rego policy for compliance with RBAC before auto (re)configuring respective resources. The policy manager is a cloud-native software, operates over a combination of Kubernetes or Kubernetes-compatible platforms such as MINIO (data management platform) and Gatekeeper, the cloud-native policy-oriented webhook [92]. It receives a Rego policy from an end-user (via Gatekeeper) and translates into ODRL to identify users, groups, actions and permissions. This is then verified against RBAC structure. If compliance check is successful, a set of RBAC configurations is autogenerated by the manager, which then communicates with Kubernetes to enforce them over the respective Custom Resource Definitions (CRDs). Otherwise, if compliance check fails, a message is sent to the user informing them of the failure and it's the cause.

3.4.O-RAN xAPP security

3.4.1. IS-RD Liquid xAPP threat model

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

We applied the STRIDE methodology [80] with its threat modeling categories – Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service (DoS), and Elevation of Privilege – to identify potential threats to the IS-Wireless xApp in four contexts:

- **Kubernetes Cloud** – The cloud-native infrastructure that orchestrates and secures containers running Liquid RAN, RIC, and xApps.
- **near-RT RIC Platform** – The control framework that hosts xApps and manages near real-time optimization of RAN functions via standardized O-RAN interfaces.
- **3rd party xApp** – Another vendor’s modular microservice deployed on the RIC that consumes RAN data and issues control actions for tasks like traffic steering or slice assurance.
- **Liquid RAN Components** – The disaggregated RAN building blocks (O-DU, O-CU, O-RU) that deliver radio access services and interact with the RIC through E2 and O1 interfaces.

Each STRIDE category is discussed with examples in these contexts in the following Table 13.

Table 13. Liquid near-RT RIC xApp STRIDE analysis

STRIDE Category	Kubernetes Cloud	near-RT RIC Platform (framework & services)	xApp (3rd-party plugin)	Liquid’s RAN Components (O-DU/O-CU and O-RU)
Spoofing	- Rogue pod impersonating a service due to lack of mTLS, allowing attacker to masquerade as xApp or API server. - Compromised credentials used to create malicious pods, appearing as legitimate components.	- Malicious xApp or process impersonates RIC internals via unsecured APIs, injecting commands as if from a trusted module. - Lack of mutual auth allows a fake RIC component to register as part of RIC and intercept traffic.	- Rogue xApp presents stolen or forged credentials during onboarding to pose as a trusted vendor’s xApp (if onboarding process is weak). - One xApp pretends to be another via inter-xApp API if mutual auth isn’t enforced.	- Fake base station (rogue O-DU) tries to connect to RIC’s E2 interface, impersonating a legitimate RAN node to inject false data. - Spoofed gNodeB ID in E2 messages if authentication is missing, misleading the RIC about the sender.
Tampering	- Supply chain attacks inserting malicious code	- Manipulation of RIC message bus traffic (if not	- Malicious xApp tampering with control messages	- Injection of false configuration via O1 or E2:

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

STRIDE Category	Kubernetes Cloud	near-RT RIC Platform (framework & services)	xApp (3rd-party plugin)	Liquid's RAN Components (O-DU/O-CU and O-RU)
	- Sniffing intra-cluster traffic if no encryption, revealing RAN telemetry or credentials.	- performance or user metrics. - An unauthorized user in RIC could query data meant for SMO or operators (like network topology info).	- leaking sensitive info externally (e.g., phone locations). - Supply-chain compromised xApp quietly sends confidential RAN data to attacker's server.	- O1 interface data (config files, performance reports) intercepted by an attacker, leaking network configuration details.
Denial of Service	- Attackers spam the Kubernetes API to overwhelm the control plane (schedule countless pods) causing management outage. - A noisy neighbor container exhausts node CPU/memory, starving RIC components (if no limits).	- Crash of a core RIC service (E2 terminator, routing manager) triggered by malformed xApp message, halting near-RT control. - Multiple xApps issuing heavy compute tasks (like complex AI inferences) freeze the RIC's real-time processing.	- An xApp intentionally consumes excessive RIC resources (e.g., subscribes to every possible event at high frequency) to overwhelm the RIC or E2 node (preventing other xApps from timely processing). - Failure to handle backpressure: a slow or hung xApp causes queue buildup, blocking other xApps' messages (indirect DoS).	- Flooding the RIC with excessive measurement reports or fault indications (a hacked O-DU could spam E2 messages) to overwhelm RIC processing capacity. - Desynchronizing RAN: e.g., a timing sync attack making cells go out of sync, effectively a DoS on radio service.
Elevation of Privilege	- Compromised container escapes to host (if running privileged),	- A bug in RIC (or RMR library) allows code execution, letting	- A compromised xApp exploits an API flaw to elevate its role	- If an O-DU is compromised, it could potentially issue privileged

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

3.4.2. Runtime verification

As discussed in Section 2, O-RAN WG11 increasingly recognizes the necessity of continuous runtime monitoring for xApps deployed in Near-RT RIC environments. However, current WG11 specifications mainly focus on secure onboarding, authentication, SBOM management, and pre-deployment verification, while practical runtime monitoring mechanisms remain insufficiently defined. In this deliverable we address this gap through the integration of Kubernetes runtime security monitoring mechanisms specifically adapted to IS-Wireless LiquidRAN™ environments. The proposed solution introduces a two-layer runtime monitoring architecture combining:

1. **Layer 1: JSON Runtime Checker:** Lightweight Kubernetes API runtime verification through JSON analysis
2. **Layer 2: FALCO-based Runtime Behavioural Monitor:** Real-time syscall-level behavioural monitoring through FALCO

The solution directly addresses several threats identified in the STRIDE analysis presented in Table 13 notably:

- Rogue pod execution (Spoofing),
- Compromised or Excessive RBAC privileges (Elevation of Privilege),

- Privileged container execution (Elevation of Privilege),
- Resource exhaustion attacks (Denial of Service),
- Lateral movement between Pods (Elevation of Privilege),
- Container escape attempts (Elevation of Privilege),
- Runtime tampering after onboarding (Tampering).

Figure 10 illustrates the proposed Kubernetes runtime-security monitoring architecture integrated into the IS-Wireless RAN™ near-RT RIC deployment. The architecture combines Kubernetes API-based runtime verification with syscall-level behavioural monitoring using FALCO to detect runtime anomalies, privilege escalation, and suspicious xApp behaviour. Runtime alerts are exported toward NETWORK security services and future O-RAN Continuous Security Monitoring frameworks.

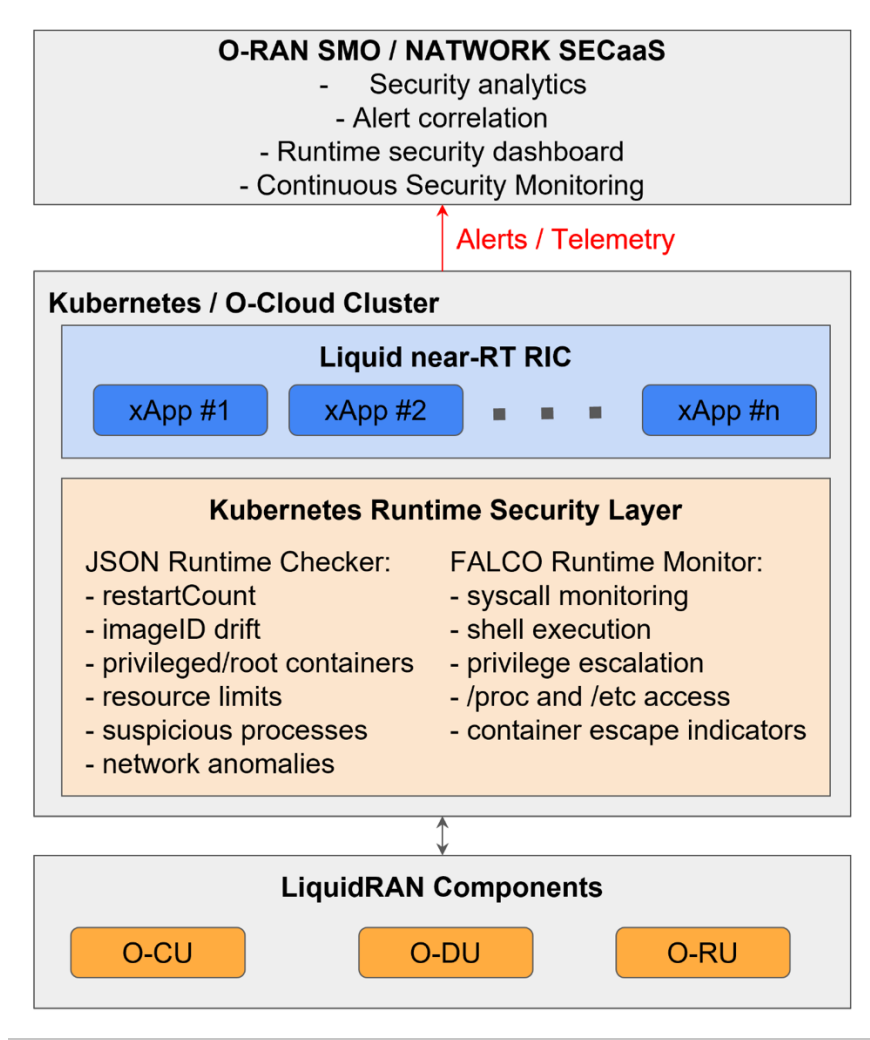


Figure 10. LiquidRAN Kubernetes Runtime Monitoring Architecture

3.4.3. JSON Runtime Checker

The first layer of the proposed runtime-security architecture consists of a lightweight Kubernetes runtime verification mechanism based on periodic JSON analysis of Kubernetes API snapshots. The mechanism cyclically retrieves runtime information from the Kubernetes control plane using commands such as `kubectl get pods -o json` and locally analyses pod-related metadata including pod status, securityContext configuration, containerStatuses, resource definitions, and container image identifiers. The objective of this layer is to detect early indicators of runtime compromise, configuration drift, privilege escalation, or abnormal workload behaviour affecting Kubernetes-hosted xApps and LiquidRAN™ services. The checker verifies several security conditions, including abnormal pod restart activity (`restartCount > 0`), runtime image drift through imageID changes, containers executed as root (`runAsUser == 0`), privileged container execution (`privileged == true`), missing CPU or memory resource limits, suspicious process execution such as `bash`, `sh`, `curl`, or `wget`, and abnormal or unauthorized network connections. This lightweight monitoring layer provides a low-complexity and low-overhead mechanism for continuously validating the runtime state of xApps after onboarding, complementing existing WG11 onboarding authentication and configuration verification mechanisms. Because it operates directly through the Kubernetes API without requiring kernel instrumentation or additional privileged components, the JSON checker can be easily integrated into existing O-RAN cloud-native deployments while providing valuable visibility into runtime anomalies and Kubernetes misconfigurations.

3.4.4. FALCO-based Runtime Behavioural Monitor

The second layer of the proposed runtime-security architecture introduces real-time behavioural monitoring through FALCO [109], a cloud-native runtime security engine operating at Linux kernel level. Unlike the first layer, which periodically analyses Kubernetes API snapshots, FALCO continuously observes syscalls generated by running containers and Kubernetes workloads, providing real-time visibility into runtime activity and low-level system behaviour. The monitoring workflow follows the sequence: Kubernetes Node → Linux Kernel → FALCO → Alert Generation. FALCO is deployed as a privileged Kubernetes DaemonSet, ensuring that one monitoring instance executes on each cluster node while maintaining visibility over all hosted xApp containers and LiquidRAN™ services. The engine analyses kernel events and compares them against predefined security rules to detect suspicious or unauthorized behaviour. The implemented rules focus on the detection of interactive shell execution (`bash`, `sh`), suspicious process spawning, unauthorized access to sensitive filesystem locations such as `/proc` and `/etc`, abnormal outbound network communications, privilege escalation attempts, and indicators of container escape or host compromise. This behavioural monitoring layer is particularly relevant for O-RAN xApp environments

because xApps are dynamically onboarded third-party applications whose behaviour may evolve after deployment, even when onboarding authentication and integrity checks were initially successful. Consequently, FALCO complements onboarding security mechanisms by providing continuous runtime supervision capable of detecting attacks, or malicious activity occurring during operation. Alerts generated by FALCO can be exported through logs, webhooks, or message queues such as Kafka, RabbitMQ, or NATS, enabling integration with NATWORK security services, AI-based IDS components, SIEM platforms, or future O-RAN WG11 Continuous Security Monitoring frameworks. By combining syscall-level behavioural analysis with Kubernetes-native deployment compatibility, this second monitoring layer significantly strengthens the runtime security posture of LiquidRAN™ cloud-native infrastructures while remaining compatible with existing O-RAN and Kubernetes operational models.

3.4.5. Beyond the state of the art

3.4.5.1. JSON checker

The proposed Kubernetes JSON runtime checker goes beyond the current state of the art by introducing a lightweight and continuously deployable runtime-security mechanism specifically adapted to cloud-native O-RAN xApp environments. Existing O-RAN WG11 security specifications and studies mainly focus on secure onboarding, authentication, SBOM verification, remote attestation, and static configuration analysis prior to deployment. While recent WG11 Continuous Security Monitoring documents acknowledge the necessity of detecting compromised xApps during operation, they do not define concrete Kubernetes-native runtime telemetry models or operational runtime verification mechanisms. The proposed Layer 1 mechanism fills this gap by extending runtime supervision directly into the Kubernetes orchestration layer through continuous API-state verification.

Unlike traditional Kubernetes security approaches that primarily operate at deployment time through admission controllers, policy engines, or static manifest analysis, the proposed mechanism continuously evaluates the live runtime state of deployed xApps after onboarding and execution. This enables the detection of runtime drift conditions that remain invisible to pre-deployment controls, including imageID changes after deployment, abnormal restart behaviour, privilege escalation occurring during execution, runtime resource abuse, and suspicious process activation inside containers. The solution therefore bridges the current operational gap between onboarding integrity verification and full runtime integrity verification.

The proposed mechanism also introduces a particularly lightweight runtime-security approach compared to existing kernel-level or cryptographic runtime integrity solutions. Because the checker operates directly through the Kubernetes API server without requiring kernel instrumentation, hypervisor modification, Trusted Execution Environments, or continuous cryptographic measurements, it can be deployed broadly across Kubernetes-hosted O-RAN

infrastructures with minimal operational complexity and negligible performance overhead. This property is especially relevant in telecom edge environments and Near-RT RIC infrastructures where latency, scalability, and resource consumption remain critical operational constraints.

Another novel aspect is the explicit integration of Kubernetes runtime-security indicators into O-RAN xApp security monitoring. Current WG11 specifications discuss compromised xApp detection only at high conceptual level, without defining practical runtime indicators. The proposed checker formalizes a concrete telemetry model based on Kubernetes-native indicators such as:

- pod instability (restartCount)
- runtime image drift (imageID)
- excessive privileges (runAsUser == 0, privileged == true)
- missing resource isolation
- suspicious runtime tools (bash, curl, wget)
- abnormal communication patterns

This establishes one of the first practical Kubernetes-native runtime monitoring frameworks directly aligned with O-RAN WG11 Continuous Security Monitoring objectives.

Finally, the proposal aligns with NATWORK's reconciliation principle between security, performance, and sustainability. Instead of continuously enforcing costly cryptographic integrity verification or hardware-assisted attestation over all workloads, Layer 1 provides a low-cost runtime anomaly detection mechanism capable of operating continuously across large Kubernetes clusters. More advanced mechanisms such as runtime attestation or cryptographic integrity verification can then be selectively activated only when Layer 1 identifies suspicious behaviour or elevated risk conditions. This adaptive and layered approach significantly reduces operational costs while maintaining continuous runtime visibility over O-RAN xApp infrastructures.

3.4.5.2. *FALCO-based Behavioural Monitor*

The proposed FALCO-based runtime behavioural monitoring layer extends beyond the current state of the art by operationalizing continuous syscall-level monitoring specifically for Kubernetes-hosted O-RAN xApp environments. Existing O-RAN WG11 specifications increasingly acknowledge the importance of runtime xApp supervision, compromised xApp detection, and continuous security monitoring. However, current specifications remain mostly focused on onboarding authentication, SBOM verification, policy enforcement, and static application validation. Even the latest WG11 studies discussing runtime integrity monitoring do not define concrete runtime behavioural detection mechanisms, telemetry models, or operational architectures suitable for cloud-native O-RAN infrastructures. The proposed Layer 2 directly

addresses this gap through real-time kernel-level behavioural analysis integrated into the Kubernetes execution environment.

Unlike traditional Kubernetes security approaches that mainly rely on configuration validation, admission control, or static compliance verification, the proposed mechanism continuously observes the actual runtime behaviour of xApp containers after deployment. By monitoring Linux syscalls directly at kernel level, the solution can detect malicious or abnormal runtime activity that remains invisible to Kubernetes API-based monitoring alone. This includes shell execution inside containers, suspicious process spawning, access to sensitive filesystem locations such as /proc and /etc, privilege escalation attempts, abnormal outbound network activity, and indicators of container escape or host compromise.

The proposed architecture also advances beyond existing telecom-oriented runtime security approaches by introducing practical runtime behavioural supervision compatible with O-RAN cloud-native infrastructures. Current O-RAN WG11 specifications mention periodic or on-demand runtime integrity verification but generally assume cryptographic attestation or static verification workflows. In contrast, the proposed FALCO-based layer introduces a behavioural security paradigm where runtime attacks are detected through observation of operational behaviour rather than only through static measurements or cryptographic signatures. This significantly improves visibility into post-onboarding compromise scenarios, insider threats, and zero-day runtime attacks affecting xApps or Kubernetes-hosted RAN services.

Another important contribution beyond the state of the art is the integration of syscall-level runtime monitoring directly into Kubernetes-native O-RAN deployments while preserving cloud-native scalability and operational compatibility. FALCO is deployed as a Kubernetes DaemonSet with one monitoring instance per node, enabling scalable runtime supervision across distributed O-RAN clusters without requiring modifications to xApp source code, Kubernetes internals, or Near-RT RIC application logic. This property is particularly valuable for multi-vendor O-RAN ecosystems where third-party xApps may not expose internal security instrumentation interfaces.

The proposed layer also contributes beyond existing runtime security approaches by integrating behavioural monitoring into a broader NATWORK adaptive security framework. Runtime alerts generated by FALCO can trigger:

- AI-based anomaly analysis
- dynamic risk assessment
- adaptive orchestration responses
- selective activation of stronger security mechanisms
- future runtime attestation workflows

Consequently, Layer 2 is not treated as an isolated monitoring component but as an operational runtime telemetry source feeding NATWORK’s adaptive and sustainability-aware cybersecurity architecture.

Finally, the proposed approach aligns with NATWORK’s reconciliation principle between security, performance, and sustainability. Full continuous cryptographic runtime integrity verification or hardware-assisted attestation may impose non-negligible computational or operational costs when applied systematically to all workloads. In contrast, FALCO-based behavioural monitoring provides continuous runtime visibility with significantly lower complexity while enabling selective escalation toward stronger verification mechanisms only when suspicious runtime activity is detected. This layered and adaptive runtime-security model therefore enables large-scale deployment across distributed O-RAN infrastructures while maintaining operational scalability, low overhead, and continuous runtime supervision.

3.5. Kubernetes-powered automatized remote attestation

3.5.1. Domain and problem statement

This work relates to the simplification and globalization of the workflow for preparing containerized executables to be remotely attested and continuously verified by D-MUTRA, as discussed in D3.5. The approach described below applies to all types of workloads, including, among others, xApps running on near-RT RICs.

In the previous NATWORK stage, attestation had already evolved beyond the isolated binary model and into the container ecosystem. D-MUTRA’s SECaaS tool was initialized from a container image and a `docker-compose.yml` file. It was then able to retrieve the main executable, compute its reference measurement, and augment the compose file with an attestation sidecar. This model works well on a single host, in a lab or demonstration environment, because everything is described in a single file and launched with a single `docker compose up` command.

However, as soon as deployment moves beyond a demo machine, this model quickly reaches its limits. `docker-compose.yml` does not support workload scheduling across multiple nodes, does not manage application auto-restart at the cluster level, does not provide a unified identity and access control model between services, and offers no standardized lifecycle management for sidecars. The attestation sidecar must therefore be described manually: container image, shared volumes, environment variables, and dependencies all need to be explicitly defined. As the stack grows and needs to be either replicated across multiple sites or in for deployment by an operator, each environment increasingly becomes a particular case requiring dedicated maintenance.

The core issue is therefore not the correctness of the Docker-based model itself, but rather its inability to scale efficiently. As long as the application-to-sidecar relationship remains a one-to-one mapping on a single host, the approach remains manageable. However, once deployments extend to edge networks, multi-tenant or multi-cluster environments, large-scale orchestration, or frequent version rotations, Docker Compose becomes too limited: it lacks native concepts such as clusters, namespaces, roles, and a consistent sidecar injection policy.

3.5.2. Leveraging Kubernetes for xApp security enforcement

With Kubernetes, the deployment unit is no longer a compose file but a set of declarative objects: Deployment, Service, ConfigMap, Secret, Namespace, and possibly CustomResource. With Helm, these objects are bundled into a parameterizable, versioned, redeployable chart. Attestation follows the same pattern: instead of editing each compose file, you act at the level of the application chart and the cluster.

The attestation sidecar becomes a Kubernetes container injected into the application pod. It shares the PID, the filesystem or a volume with the target application, exposes its own endpoints (status, metrics), and is managed like any other container: probes, automatic restart, observability, RBAC. The notion of an attestation binary glued to the application disappears completely. On one side you have the application; on the other a standardised sidecar; and between them, a declarative injection mechanism.

That injection can be done in several ways. The first is to add the sidecar manually to the application chart, the direct equivalent of the enriched compose. The second is to automatically rewrite a vanilla chart to produce an attested chart, which is exactly what the attestify-chart.sh pipeline already does. The third, cleaner in the long run, is a declarative overlay approach through a CustomResource or a webhook which states the need for attestation for an application so that the wrapper takes care of adding the sidecar, the volumes, the permissions and the reference measurements in the right namespace. This is the direction explored by the CRD overlay of the Kubernetes wrapper.

The direct benefit is that you no longer instrument a stack by hand: you declare an intent, and the cluster applies it. The sidecar is versioned, observable, redeployable. The reference measurement is carried as a Helm value or a CustomResource field, not as a local file. The stack becomes repeatable across multiple sites, multiple clusters, and multiple operators. The chart is used for all deployments which feature the same security requirements. By templating Kubernetes resources (e.g., Pods, Deployments, RBAC policies, NetworkPolicies, security contexts, and attestation sidecars), Helm enables platform-wide, reproducible, and scalable security enforcement independently of the individual applications being deployed. This perfectly fits with D-MUTRA activation on containerized xApps, where once single helm chart can be

defined and shared to enforce remote attestation or integrity verification during their execution for all xApps onboarded a defined RIC.

Figure 11 compares and contrasts the three modes for D-MUTRA enablement. The actual progress made in the project actually relates to the evolution from a containerized workload (i.e., in the center) to the leverage of K8s automation (i.e., in the right-hand side). From a single `docker-compose.yml` running application and sidecar on one host, we have migrated to a vanilla Helm chart plus a CustomResource and admission webhook that drive a Kubernetes cluster where each pod is uniformly injected with its attestation sidecar. We had categorized four situations/entry cases and the associated merits of Kubernetes in Table 14.

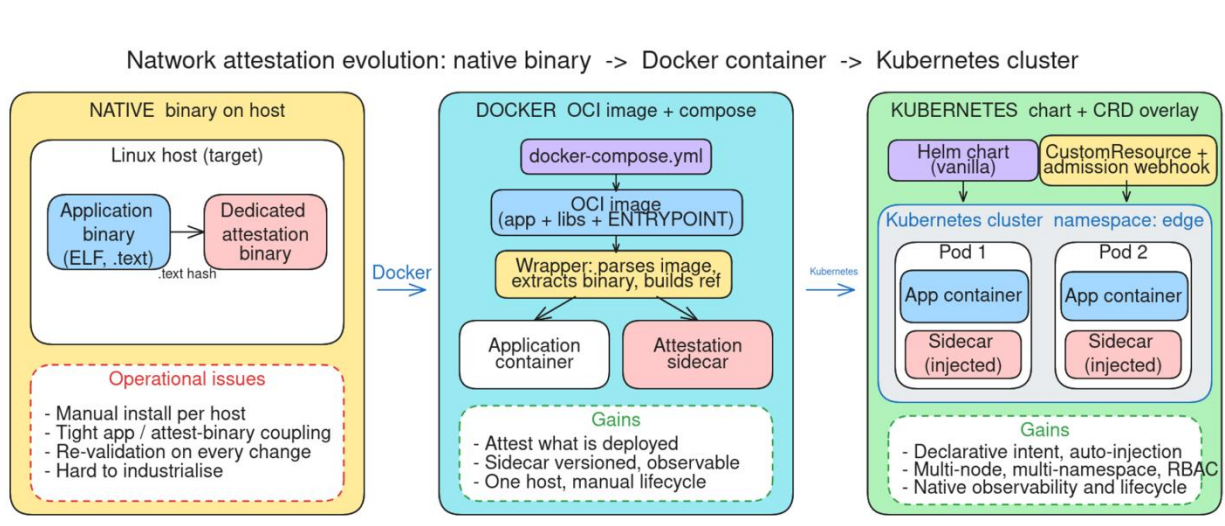


Figure 11. Three modes of workload preparation for D-MUTRA security from native to K8s orchestration

Table 14. Illustration of Kubernetes leverage merits

Situation/case	Available input	Expected workflow
1	Helm chart provided + known application image	The wrapper applies an overlay or patches the chart to add the sidecar, mounts the shared volume and injects the variables (DLT_*, references). The deployment yields an attested chart ready for helm install.
2	Helm chart provided + unknown or inherited image	The wrapper resolves the effective image from the chart (image.repository, image.tag, inherited values), retrieves the main binary via the same logic as the Docker mode, then enriches the chart.
3	No chart, just a Deployment or a legacy docker-compose.yml	The wrapper resolves the effective image from the chart (image.repository, image.tag, inherited values), retrieves the main binary via the same logic as the Docker mode, then enriches the chart.
4	Application already deployed on the cluster	The wrapper converts or wraps the manifest in a minimal Kubernetes overlay (namespace, deployment, service), injects the sidecar, and produces an attested manifest applicable as-is.

The guiding idea is the same as in the binary-to-container transition: we no longer instrument the artefact downstream, we declare the intent upstream and let the system apply the transformation.

3.5.3. D-MUTRA modified workflow

The measurement strategy stays consistent with the Docker mode: dynamic on the main executable (reading the runtime .text of the application process through the namespace shared between containers of the pod), and static on the loaded libraries (file hash inside the application container filesystem, accessed via a shared volume or ephemeralContainers). Kubernetes adds two capabilities: batch attestation across a whole namespace, and exposure of the result to the rest of the stack via Service or Secret.

On the integration side, three evolutions are to be planned. First, the generalisation of the wrapper's CRD overlay to discover the target image, extract the binary and build the reference without modifying the application chart. Second, alignment with the secaas-wrapper chain which already carries reference measurements (--seed-runtime-ref-measures): on the Kubernetes side, those measurements can be injected as a ConfigMap or as a CustomResource field, instead of a

one-shot CLI argument. Third, observability: each sidecar exposes its standard endpoints and publishes its attestations to the DLT via the `DLT_RPC_URL` and `DLT_CONTRACT_ADDRESS` variables carried by the cluster, which makes the mechanism uniform between edge and core.

In short, moving from Docker to Kubernetes does not change the goal of remote attestation; it changes the level at which it is expressed. Before, you enriched a compose file. Now, you express a measurement intent at the cluster level, and a declarative wrapper automatically turns existing charts or deployments into attested charts. It is this shift from file-based certification to cluster-based certification that makes this approach truly scalable for a NATWORK project deployed across multiple operators.

3.5.4. Progress from D 3.5

The progress relates to the workflow simplification and globalization as detailed above, enabling D-MUTRA to be automatically leveraged to secure all xApps onboarding a near real time RIC.

3.6. Work on container availability

3.6.1. General

In contrast with the state of the art, our solution operates directly from a non-privileged pod, without requiring host or node-level visibility assumed by prior work. It uses a deterministic CPU-bound sidecar based on NOP execution to isolate CPU scheduling effects from memory and I/O interference. Beyond prior approaches, xNOPs estimates the number of concurrent external workloads competing for CPU resources, enabling quantitative attribution of neighbor contention through a lightweight, deployable probe suitable for continuous production monitoring.

3.6.2. Restricted container visibility by the deployment architecture

As shown Figure 12, there are two common cloud deployment architectures for containerized workloads, which differ primarily in how compute resources are isolated and scheduled. These architectural choices have a direct impact on the observability and reliability of CPU-related performance metrics.

The first architecture runs containers directly on the host OS using Linux namespaces and cgroups. In this model, Linux metrics and tools such as `top`, `load average`, `CPU usage rates`, or `PSI` could provide a more faithful view of local CPU activity. However, as we already saw this visibility is often restricted to the pod/container scope, making global contention difficult to observe from a tenant perspective.

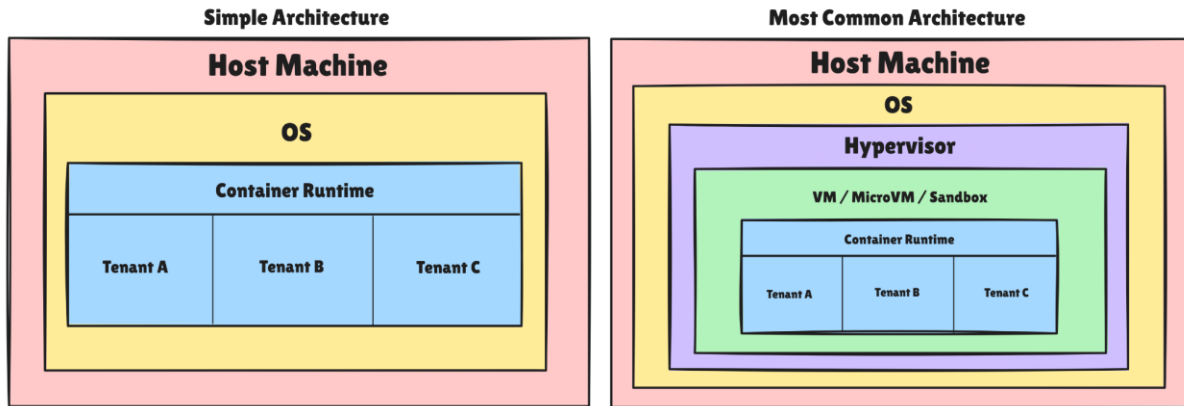


Figure 12 . Two common container deployment architectures

The second architecture, commonly used in public cloud environments, runs containers inside virtual machines managed by a hypervisor. In this layered environment, standard Linux metrics can become incomplete or misleading because part of the scheduling and contention occurs outside the guest VM, at the hypervisor or co-tenant level. Our approach mainly focuses on the first architecture, while noting that the proposed approach can also be extended to virtualized multi-tenant environments.

3.6.3. Solution: xNOPs CPU-bound sidecar.

3.6.3.1. Concept

The proposed solution relies on a lightweight sidecar called xNOPs, periodically executing a deterministic CPU-bound binary composed primarily of NOP instructions. Since the program always executes the exact same instruction sequence without external I/O or significant cache dependencies, its execution time remains directly correlated to the effective CPU capacity available at runtime.

The sidecar is deployed alongside the target application so that both share the same scheduling conditions: Linux CFS arbitration, cgroup hierarchy, Kubernetes quotas and limits, and underlying virtualization constraints. As a result, xNOPs acts as a passive probe of the CPU performance effectively perceived by the application rather than simply measuring exposed CPU utilization metrics.

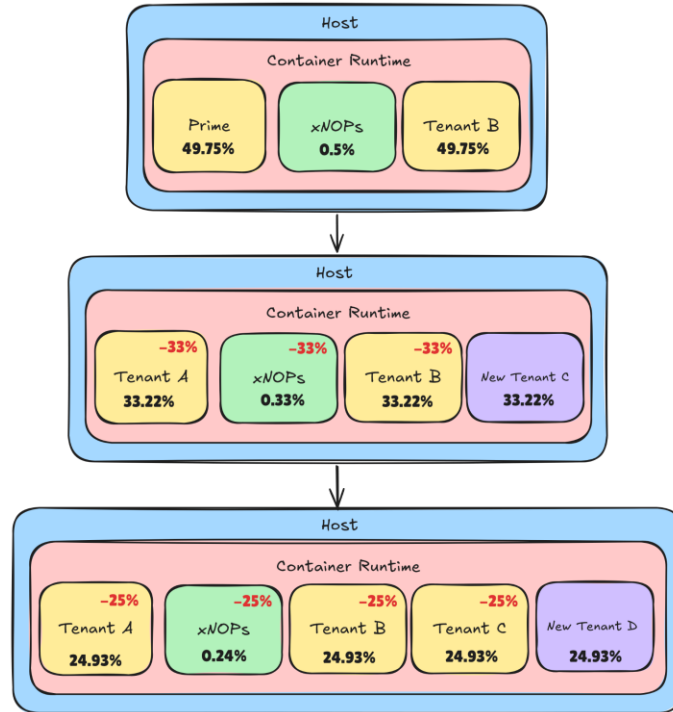


Figure 13. Illustration of performance evolution during CPU contention

Figure 13 illustrates the principle. Initially, the host CPU capacity is shared between two tenants and xNOPs, each receiving roughly half of the available resources.

When a new tenant appears, CPU time is redistributed among more workloads, reducing the share available to every actor, including xNOPs. This reduction increases the execution time of the payload. As additional tenants are introduced, the available CPU fraction continues to decrease, producing progressively longer execution times. By comparing these measurements to an initial calibration state, the system can infer hidden contention and estimate the evolution of the co-resident workload.

Initially, the host CPU capacity is shared between two tenants and xNOPs, each receiving roughly half of the available resources. When a new tenant appears, CPU time is redistributed among more workloads, reducing the share available to every actor, including xNOPs. This reduction increases the execution time of the deterministic payload. As additional tenants are introduced, the available CPU fraction continues to decrease, producing progressively longer execution times. By comparing these measurements to an initial calibration state, the system can infer hidden contention and estimate the evolution of the co-resident workload pressure.

Unlike traditional monitoring approaches, the method observes the final performance resulting from all scheduling and isolation layers combined, including the Linux scheduler, cgroups, Kubernetes resource policies, and the cloud hypervisor.

3.6.3.2. Implementation

The xNOPs sidecar periodically executes a fixed CPU-intensive loop and measures its execution duration with high-resolution timers. A reference execution time t_0 is first established during a calibration phase under nominal conditions. Subsequent measurements t_1 are then compared against this baseline to estimate the loss of effective CPU capacity caused by additional contention.

The estimation model is based on the following relation:

$$n = \frac{shares_c}{1024} \left(\frac{T_1}{T_0} - 1 \right)$$

Where:

- n : number of workers added compared to the calibration state
- $shares_c$: sum of CPU shares of the parent cgroup containing xNOPs during calibration
- T_1 : execution time of xNOPs for a measurement after calibration
- T_0 : execution time of xNOPs during calibration

Definition of shares: CPU shares assigned to a process; these shares determine the proportion of CPU time it receives relative to other processes when there is contention for CPU resources. The model assumes that the execution time of the deterministic workload varies proportionally to the CPU capacity effectively available to the sidecar.

The ratio $\frac{T_1}{T_0}$ therefore reflects the degradation induced by contention, while the CPU shares term normalizes the estimation relative to the scheduling configuration of the environment. This lightweight implementation requires no privileged access, hypervisor visibility, or infrastructure-side instrumentation, making it compatible with standard Kubernetes and multi-tenant cloud deployments. However, this formula requires knowing the sum of CPU shares at calibration time.

We can estimate it using a simple trick based on a differential approach as illustrated in Figure 14.

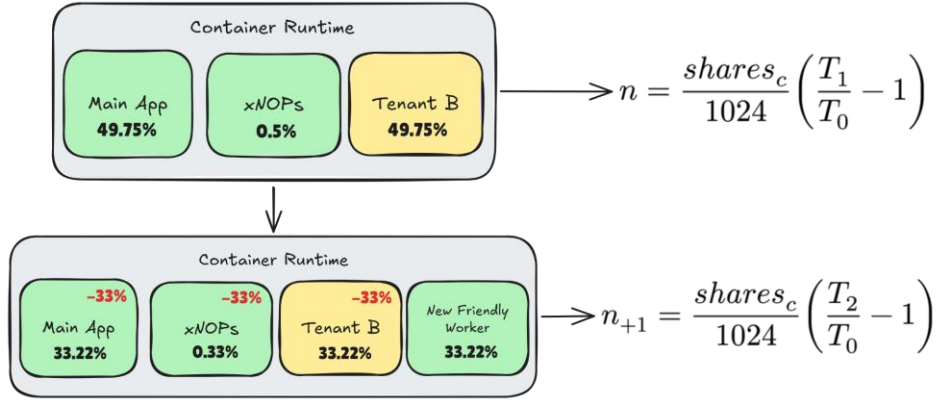


Figure 14. Illustration of calculating total CPU shares in an unknown environment

A first execution of xNOPs is performed with n active workers, producing an execution time **T1**. One additional worker is then injected into the same workload pool, and a second execution is performed under $n+1$ concurrent workers, yielding **T2**.

$$shares_c = 1024 \cdot \frac{(n_{+1} - n)T_0}{T_2 - T_1}$$

Comparing **T1** and **T2** allows us to isolate the impact of a single worker, which can then be used to infer the effective total number of workers in the system.

This formulation is a first-order approximation valid only in a single-core contention context. In multi-core environments, where workers may be distributed across multiple logical cores with effects such as hyperthreading, CPU affinity, and more complex scheduling behavior, this model becomes insufficient. To more accurately estimate capacity under these conditions, we use the following more advanced formula:

$$n = \left(C \times \frac{T_1}{T_0 \times P_{smt}} \times \frac{shares_x}{shares_w} \right) - \frac{shares_c}{shares_w}$$

Where:

- $shares_x$: number of CPU shares assigned to **xNOPs**
- C : number of logical CPUs on the machine
- P_{smt} : hyperthreading efficiency coefficient

This formulation extends the original model to a multi-core environment by introducing the number of available CPUs (C) and a hyperthreading efficiency factor (P_{smt}) to reflect the reduced performance gain of logical cores compared to physical ones. It accounts for Linux CPU scheduling

behavior through CPU share weighting, enabling execution time variations to be translated into an equivalent concurrent load while respecting resource allocation differences between workloads.

3.6.4. Results

Figure 15 shows an estimate of the number of concurrent *prime_infinite* pods derived from xNOPs execution time using the previously introduced analytical model. The evaluation is conducted on a single-node K3s cluster in a multi-core environment, with 12 logical cores (6 physical cores) and Simultaneous Multithreading (SMT) enabled. No CPU requests or limits are defined for the workloads. This method relies on a single reference measurement ($T_0 = 7.422112$ s) obtained during calibration.

The blue points represent observed data, while the red curve shows the theoretical prediction. Their strong agreement (MAPE = 1.9% corresponding to approximately 98% prediction accuracy) demonstrates that the analytical model accurately estimates concurrent load, with performance comparable to the full regression-based approach.

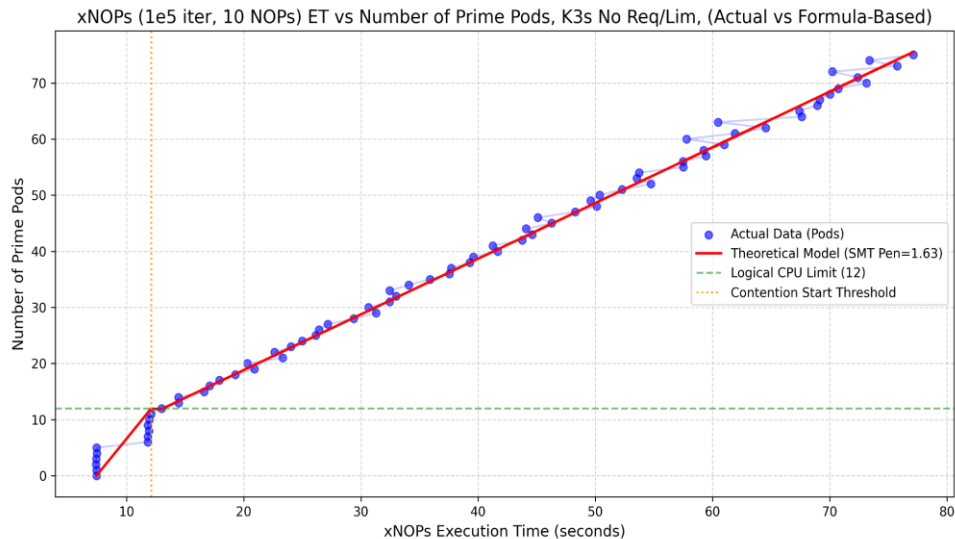


Figure 15. Comparison of xNOPs execution time against the formula-based model

The graph also reveals two plateau regions: the first during physical core saturation ($\approx 1-6$ workers), and the second when SMT/Hyper-Threading logical cores are utilized ($\approx 7-12$ workers). Beyond these thresholds, contention becomes dominant and execution time increases linearly again. The model successfully captures these transitions, showing that a lightweight calibration based on a single T_0 measurement can accurately reconstruct node load from one tenant-side pod.

3.6.5. Penalty/Measurement overhead

The degradation induced by **xNOPs** is depending primarily on two configurable parameters: the number of NOP instructions executed (i.e., loop iterations and per-loop NOP count), and the CPU shares allocated to the xNOPs process within the cgroup scheduling hierarchy. This experiment below evaluates the overhead introduced by xNOPs when co-located with a prime number calculator (10^5 prime numbers) in a Docker environment pinned to the same CPU core, with default CPU shares for both (1024).

Table 15. Penalty induced by xNOPs on Prime (10^5)

NOPs executed	Penalty to Prime Calculator	Spearman coef.	Spearman p-value	RMSE
1 000	≈ 0	≈ 0	5.3e - 1	0.458
10 000	0.03 %	0.018	9.5e - 1	0.427
100 000	0.03 %	0.986	4.1e - 9	0.060
1 000 000	0.57 %	0.981	8.4e - 9	0.068

Table 15 shows a clear threshold effect in statistical validity and measurement stability. At 1,000 and 10,000 NOPs, xNOPs remains in a noise-dominated regime, with no meaningful correlation to the reference workload and high dispersion, indicating that the signal is not yet reliable despite negligible overhead ($\sim 0.03\%$). At 100,000 NOPs, the system transitions to a stable operating regime, with strong statistical agreement (Spearman ≈ 0.986 , $p \approx 4.1e-9$) and low error (RMSE ≈ 0.060), while maintaining a negligible performance penalty ($\sim 0.03\%$). At 1,000,000 NOPs, correlation remains strong, but RMSE increases slightly and overhead rises to $\sim 0.57\%$ without improving measurement quality. Overall, all configurations remain well within KPI NATWORK 1.3.3 constraints ($\leq 10\%$ combined penalty budget), with xNOPs monitoring overhead staying below 1% in all tested regimes. This overhead can be further reduced by lowering xNOPs CPU shares, providing an additional tuning lever to minimize performance impact.

3.6.6. Beyond the state of the art

xNOPs addresses the limitations of existing approaches to CPU observability by moving beyond both traditional system metrics and infrastructure-dependent monitoring frameworks toward a direct estimation of effective CPU contention. Unlike prior work that relies on host-, VM-, or hypervisor-level telemetry, it operates entirely from a non-privileged pod and does not require node-level visibility.

In contrast to conventional signals such as `top`, `/proc/stat`, or pressure metrics, which are partial, scope-limited, and affected by namespace isolation, cgroup boundaries, or security constraints,

xNOPs infers CPU competition through a controlled deterministic NOP instruction-based probe that isolates CPU scheduling effects from memory and I/O interference at the execution level.

In addition, it provides information that is not available in prior detection- or benchmarking-based approaches: an explicit estimate of the number of concurrent external workloads effectively competing for CPU resources. This goes beyond simple detection by introducing a quantitative attribution of co-tenancy pressure. As a result, it opens the door to practical downstream uses such as cluster qualification, autoscaling decisions, and CPU sizing, while operating under the same unprivileged, pod-level constraints as production workloads.

Future works considered after NATWORK will focus on extending the current approach toward multi-VM and distributed cloud architectures. The present study was limited to mono-node Docker and K3s deployments in bare-metal environments, where contention remains local to a single host. Adapting the methodology to virtualized and multi-node infrastructures will likely require revisiting the current analytical formulations to account for hypervisor scheduling, distributed resource contention, and cloud-specific variability. This extension is essential to better match real-world cloud-native deployments.

Another important direction concerns workload diversification. The current xNOPs implementation is CPU-bound, but future variants could target other shared resources such as memory, storage I/O, cache, or networking. Such specialized xNOPs workloads would enable the framework to characterize a broader range of contention scenarios commonly encountered in multi-tenant cloud environments.

3.6.7. Progress versus D3.5

This development was not initiated and discussed in D3.5. It is a novel promising technique devised and developed since then.

4. NATWORK's PDSCMs on native workloads

The work initiated in D3.5 has been continued with the following evolutions, dealing with two native security workloads delivered by NATWORK partners (i.e., MMT probe and Anomalous Traffic Detector xApp), both able to detect anomalous traffic:

1. MMT placement in SEV-SNP trusted execution environment, offering both integrity and confidentiality to MMT traffic security probe.
2. MMT availability and performance monitoring, leveraging control flow probes leveraging LLVM compilation.
3. Native xApp remote attestation and runtime integrity verification by D-MUTRA, offering continuous integrity for the Anomalous Traffic Detection xApp.

4.1. TEE-based confidentiality and integrity for MMT probe

4.1.1. MMT threat model

Even though this threat model has not changed since the edition of D3.5, it is recalled for simplicity here.

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

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

The MMT framework can be deployed in two main ways depending on the needs of the environment. One option is native installation, where users compile the binaries such as *MMT-probe*, *MMT-security*, and the associated .so modules, and then run them directly on Linux systems. This approach provides maximum flexibility for integration with custom setups and

allows fine-grained control over configuration and optimization. Alternatively, a more modern and reproducible method is containerized deployment using pre-built Docker images. With this approach, all dependencies are packaged together, making it easier to install, test, and update the framework while ensuring consistency across environments. Containerized deployment also simplifies orchestration with tools such as Kubernetes or Docker Compose, which is particularly valuable in infrastructures that need to scale dynamically or enforce standardized deployment practices.

When deploying the MMT framework, whether through native binaries or containerized environments, the security of the software artefacts and associated rulesets is a critical concern. A binary or shared library that has been tampered with, replaced, or misconfigured can undermine the entire monitoring and enforcement pipeline. To evaluate these risks systematically, we apply the STRIDE methodology [80], which helps us reason about potential threats across six key categories. We discuss in detail the threat model of deploying MMT that highlights multiple pre-deployment security considerations.

Spoofing is a significant concern during the distribution and deployment of MMT. Without proper controls, an attacker could impersonate a legitimate developer or repository and trick operators into installing a maliciously crafted version of *MMT-probe*, *MMT-security*, or one of the .so detection modules. In practice, this can occur if binaries are downloaded from unofficial mirrors or if Docker images are pulled from unverified registries. Preventing spoofing requires strong authentication of both the source repository and the individuals who build and release MMT.

Tampering focuses on the risk of modification to binaries or rulesets before they are deployed. Because MMT-Security relies on compiled .so rules to enforce detection logic, even a subtle modification in a library could result in rules being disabled, altered, or replaced with logic that intentionally bypasses threats. For example, a tampered library could silently allow specific malicious traffic through, creating a blind spot in monitoring. To counter this risk, deployments must incorporate artifact integrity validation, such as checksum verification, digital signatures, or trusted build pipelines.

Repudiation arises when there is no clear accountability for changes made to binaries or configurations. In environments without proper logging and version control, it may be impossible to prove whether a binary was modified by a malicious actor or simply updated by a developer. This lack of traceability hinders incident response and weakens confidence in the security posture. Implementing auditable pipelines, logging all artifact changes, and enforcing commit signing are crucial to address repudiation threats.

Information disclosure represents another serious category of risk. The detection rulesets themselves may encode proprietary intellectual property or sensitive patterns used for anomaly detection. If these .so libraries or associated configuration files are leaked, an adversary could gain insights into the organization’s detection strategy, allowing them to craft evasive attacks. Moreover, improper containerization or configuration could inadvertently expose sensitive credentials used by MMT to external parties. Ensuring proper access control, encrypting secrets, and limiting container privileges are necessary steps to reduce exposure.

Denial of Service (DoS) can result from the deployment of corrupted or malicious artifacts that cause instability in the monitoring stack. For example, a malformed ruleset could trigger a crash loop in *MMT-security*, rendering the detection system unavailable. Similarly, a binary modified to consume excessive resources could degrade the overall monitoring environment. DoS threats highlight the importance of testing rulesets in staging environments before production rollout, as well as implementing resource isolation through container orchestration platforms like Kubernetes.

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

In the perspective of Use Case 4.5 dedicated to optimized and explainable MTD, ZHAW and Montimage are closely working on the dynamic placement of MMT in and out of a TEE enclave with AMD’s SEV-SNP (Secure Encrypted Virtualization with Secure Nested Paging [79]).

The proposed system, TEEMig, combines container migration with TEE execution using SEV-SNP. TEEMig is designed as a multi-cluster architecture controlled by a central coordination component. The clusters can have different heterogeneous hardware, specifically, a TEE-based cluster running inside an AMD SEV-SNP protected environment serves as the confidential execution target. This separation allows a clear distinction between regular execution and confidential computing execution while keeping the overall workflow consistent across clusters.

Since D3.5, the TEEMig architecture has been fully specified and its multi-cluster control plane designed. It is represented in Figure 16 the key architectural elements are as follows. A central coordination component monitors the security state of the host environment and receives threat signals from the MTD decision logic. Based on these signals, it orchestrates the live migration of

the MMT workload between the standard execution cluster and the SEV-SNP-protected cluster. The migration mechanism leverages CRIU (Checkpoint/Restore In Userspace) to capture the full process state of MMT-probe, transfer it across the cluster boundary, and restore it inside the confidential VM, thereby avoiding a cold restart and preserving ongoing monitoring continuity.

A critical property of TEEMig is that no modification to the MMT binary is required to enable TEE sheltering. The MMT-probe executable is placed inside the SEV-SNP-protected VM without recompilation or any code-level adaptation. This is a direct consequence of using a VM-type TEE rather than a process-level TEE such as Intel SGX, which would require code restructuring to fit within enclave boundaries. The entire MMT process, including its packet capture engine and protocol dissection modules, is transparently sheltered by the hardware memory encryption provided by SEV-SNP.

The experiments are conducted on AMD EPYC v4 processors, which provide native hardware support for SEV-SNP. A testbed has been set up instantiating the two-cluster topology: one standard Kubernetes cluster with Linux host nodes serving as the baseline execution environment, and one microK8s SEV-SNP-enabled cluster acting as the confidential execution target.

The continuity of the MMT probe service during the migration is ensured by the implemented Istio-based traffic redirection mechanism. Using the *primary-remote* deployment model, in which one cluster hosts the main control plane while one or more remote clusters connect to and use that central control plane, TEEMig coordinates the data plane traffic orchestration with the live migration of the MMT probe container in Kubernetes. This enables a continuity of both the analysis of the mirrored traffic as well as the data reporting to the MMT Operator, the central GUI providing network administrators with real-time to MMT insights of traffic and eventual anomalies and attacks.

Initial integration tests have confirmed the feasibility of the migration workflow and the correct initialisation of MMT-probe inside the confidential VM. Performance measurements quantifying the overhead introduced by SEV-SNP memory encryption on MMT's packet capture throughput have been conducted and are detailed in Section 4.1.3 below.

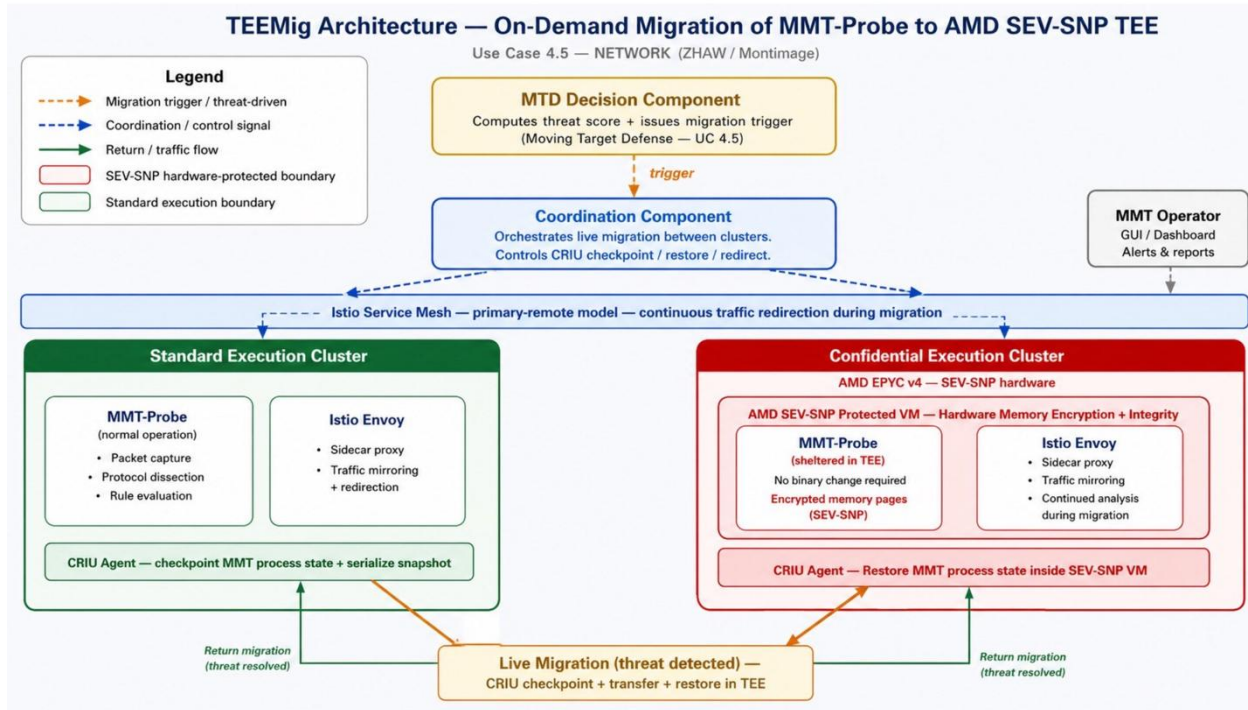


Figure 16. TEEMig multi-cluster architecture

4.1.2. Beyond the state of the art

The TEEMig approach introduces several properties that, taken together, constitute a meaningful advance beyond the current state of the art in TEE-based workload protection for network security functions.

On-demand, threat-triggered TEE sheltering. Existing deployments of TEE-based workload protection apply the TEE systematically and permanently, i.e., the workload always runs inside the confidential environment by default. TEEMig departs from this paradigm by treating TEE sheltering as a dynamic, on-demand response to a detected or presumed security threat. Under normal operating conditions, MMT-probe runs in a standard execution environment with no TEE overhead. Upon detection of a threat — or upon a probabilistic presumption of one, as defined by the MTD decision logic — the coordination component triggers a live migration of MMT into the SEV-SNP-protected VM. Once the threat condition subsides, the workload can be migrated back to the standard environment. This approach directly implements NATWORK’s concept of adaptive security: avoiding overprotection and its associated resource costs, while ensuring that confidentiality and integrity guarantees are available precisely when they are needed.

No modification to the protected binary. A significant limitation of process-level TEEs such as Intel SGX is that they require the application to be restructured so that security-sensitive code and data are explicitly partitioned into enclave regions. This imposes a non-trivial development

effort and restricts the applicability of TEE protection to workloads for which source code access and refactoring are feasible. TEEMig avoids this entirely. Because AMD SEV-SNP operates at the VM level, encrypting and integrity-protecting the entire memory space of the guest VM, MMT-probe can be sheltered without any change to its source code or binary. The PDSCM is applied at the deployment level — by placing the binary inside a pre-configured SEV-SNP VM — rather than at the code level. This property is particularly valuable for commercial or proprietary components, such as the pre-compiled libmmt_core SDK distributed by Montimage, where source-level modification is not an option.

Heterogeneous multi-cluster operation. TEEMig operates across clusters with heterogeneous hardware capabilities: a standard compute cluster and a TEE-equipped cluster. This design reflects the realistic infrastructure of telecom operators and cloud providers, where not all nodes are equipped with confidential computing hardware. Rather than requiring a homogeneous TEE deployment, TEEMig enables operators to designate a dedicated pool of SEV-SNP-capable nodes as the confidential execution target, while the bulk of the workload runs on standard hardware. The migration mechanism bridges these two environments transparently, without requiring changes to the network monitoring pipeline upstream or downstream of MMT-probe.

Integration with MTD decision logic. TEEMig is designed to be driven by the Moving Target Defense (MTD) decision component developed within NATWORK (Use Case 4.5). The MTD component continuously evaluates the threat landscape and computes a security posture score. When this score crosses a configurable threshold, it issues a migration trigger to the TEEMig coordination component. This closed-loop integration between threat detection and confidential execution is a key architectural novelty: it transforms TEE sheltering from a static pre-deployment configuration into a dynamic, runtime-responsive security mechanism. The performance and sustainability cost of TEE protection is thereby incurred only when justified by the threat assessment, in direct alignment with NATWORK’s principle of reconciling security, performance, and energy efficiency.

4.1.3. Progress from D3.5

The progress since D3.5 consists in the full implementation of the TEEMig architecture described above and in the performance characterisation of the SEV-SNP overhead on MMT-probe, reported in the following subsections.

4.1.3.1. Performance evaluation — testbed and methodology

The benchmark topology comprises a dedicated traffic source host running tcpreplay to inject PCAP traffic at controlled rates, a standard Linux VM serving as the performance baseline, and an AMD SEV-SNP-protected confidential VM with full hardware memory encryption active. MMT-

probe operates in libpcap capture mode on a dedicated vNIC (eth0) in both environments. Statistics are collected from MMT-probe's internal one-second period CSV output, supplemented by /proc/<pid>/stat sampling for CPU accounting and VmRSS sampling for memory.

Traffic was replayed from a representative PCAP trace at seven discrete rate steps: 50k, 100k, 200k, 300k, 400k, 500k, and 600k packets per second (pps). Each step was sustained for 50 seconds. The first 5 seconds of each step were discarded as ramp-up transients before computing statistics, yielding 50 steady-state samples per step per environment (350 samples total per environment). The primary metric is MMT-probe's own packets-per-second captured counter. The overhead formula is: $(\text{std_mean_pps} - \text{sevsnp_mean_pps}) / \text{std_mean_pps} \times 100\%$.

4.1.3.2. Overall throughput results

Across all 350 measurement samples per environment, MMT-probe achieved a mean throughput of 293,169 pps in the standard VM and 270,710 pps in the SEV-SNP VM. The overall mean SEV-SNP overhead is 7.66%. The standard deviations are proportionally similar between environments (189,141 pps standard vs. 172,360 pps SEV-SNP), confirming that SEV-SNP introduces a near-constant multiplicative penalty rather than increased measurement variance.

Table 16 presents the per-rate-step throughput comparison. All values are in packets per second. The P5–P95 range reflects the steady-state spread after ramp-up trimming.

Table 16. Per-rate-step throughput comparison: Standard VM vs AMD SEV-SNP VM. All values in packets per second. P5–P95 range after ramp-up trimming (first 5 s per step discarded). n = 50 steady-state samples per step per environment.

Rate step	Std mean (pps)	Std P5–P95 (pps)	SEV mean (pps)	SEV P5–P95 (pps)	Overhead	n (samples)
~50k pps	47,947	30,332 – 51,724	45,253	27,545 – 48,387	5.62%	50
~100k pps	95,716	58,599 – 102,698	90,128	55,615 – 96,221	5.84%	50
~200k pps	190,601	116,934 – 206,454	180,451	112,439 – 193,899	5.33%	50
~300k pps	286,998	180,722 – 308,279	266,920	176,047 – 286,614	7.00%	50
~400k pps	381,442	225,826 – 408,753	352,014	220,466 – 382,792	7.72%	50

Rate step	Std mean (pps)	Std P5–P95 (pps)	SEV mean (pps)	SEV P5–P95 (pps)	Overhead	n (samples)
~500k pps	477,143	290,041 – 517,427	439,212	274,621 – 469,334	7.95%	50
~600k pps	572,338	352,216 – 623,276	520,992	333,122 – 570,044	8.97%	50

4.1.3.3. *Overhead trend and interpretation*

A clear monotonic increase in overhead with traffic rate is observable across the seven steps, from 5.33% at 200k pps to 8.97% at 600k pps. This scaling behaviour is consistent with a memory-bandwidth bottleneck effect: AMD SEV-SNP encrypts all VM memory in-flight, and as packet buffer volume grows with traffic rate, the fraction of memory-bus bandwidth consumed by the encryption pipeline increases proportionally. This effect is particularly relevant for libpcap-based tools, which rely on kernel-to-userspace memory copies for every captured packet. The near-constant multiplicative nature of the overhead — confirmed by the proportional standard deviations between environments — indicates that SEV-SNP does not introduce additional scheduling jitter or non-determinism beyond the baseline encryption cost.

The measured 7.66% mean overhead falls within the 5–15% range reported in the literature for AMD SEV-SNP VM-level TEEs under network-intensive workloads and is significantly lower than process-level TEE solutions such as Intel SGX (15–30% overhead due to enclave context-switching costs). The TEEMig design choice of a VM-level confidential computing approach is therefore justified from a performance standpoint. In the TEEMig on-demand model, this overhead is incurred only when the MTD decision component triggers migration into the confidential environment and is bounded to the duration of the active threat window. For continuous baseline operation — MMT-probe running outside the TEE — NATWORK KPI 1.3.3 ($\leq 1\%$ monitoring overhead) is fully satisfied.

For deployments where sustained high-rate capture above 500k pps is expected during threat windows, the adoption of SR-IOV or DPDK capture modes — which reduce per-packet memory copies — is expected to lower the SEV-SNP overhead by 2–4 percentage points, bringing it below 7% even at peak load. This constitutes a future optimisation direction within the TEEMig work.

4.2. Flexible and cost-effective native workload performance monitoring

4.2.1. Domain

To assess software performance, a common method measures execution time by placing timestamps at the entry and exit of finite processes. In telecom systems, however, most processes are continuous and never terminate, making this approach unsuitable. Instead, probes must be inserted into selected finite code blocks within the workflow. This introduces two major challenges: minimizing the overhead induced by monitoring and identifying code blocks that accurately represent overall performance. Effective probe placement therefore requires restraint and a strong understanding of the software's internal logic.

For example, the performance of a traffic parsing and filtering function can be evaluated by monitoring the loop that parses packets accumulated in the input buffer. Following the NATWORK reconciliation principle, we propose a method for defining efficient software speed indicators while keeping monitoring overhead negligible. The approach is illustrated using the native workload of MMT-Probe.

This work lies at the intersection of three domains: high-performance network probe monitoring (packet capture, DPI, filtering, and export through Kafka or MQTT), runtime observability of x86_64 binaries, and LLVM-based compile-time instrumentation for inserting measurement probes independently of the execution environment. The target application, MMT-Probe, is an open-source traffic analyzer developed by MONTIMAGE, featuring multi-process and multi-threaded operation, DPI support through libmmt_core, and configurable reporting pipelines.

4.2.2. Problem statement

In this section we are addressing the following research question: how can a network probe running on x86_64 architectures guarantee its functional availability — namely, sustaining the expected throughput — without relying on external monitoring infrastructure, and detect degradations early enough to prevent data loss?

This raises several research sub-questions: what observation granularity (packet, function, or basic block) best characterizes degradation; what monitoring overhead remains acceptable at rates of 10^6 – 10^9 packets/s; whether instrumentation can be limited to critical execution paths; which statistical model can separate genuine degradation from normal traffic fluctuations; and how the resulting availability signal can be exported without an external observability stack. The novelty lies in making the binary itself produce its own availability indicators, instead of depending on system-level or service-level monitoring.

4.2.3. Solution

This section presents a self-monitoring approach for network probes on x86_64 architectures, focusing on MMT-Probe. The objective is to allow a production binary to autonomously detect throughput degradation, with fine spatial resolution (basic-block level) and sub-second temporal resolution, without external observability infrastructure. Existing approaches in software instrumentation, binary rewriting, kernel tracing, and anomaly detection are reviewed and shown not to satisfy these constraints jointly.

The proposed solution relies on an out-of-tree LLVM pass inserting inline counters into selected basic blocks. Instrumentation is restricted to hot paths through a two-level selection mechanism combining allowlists and source-code annotations. The counters are processed in-process by an EWMA detector integrated into the probe's periodic statistics thread, while alerts are raised using MMT-Probe's native reporting pipeline. A locally validated prototype demonstrates the feasibility of the approach, and several extensions are identified, including per-worker counters, embedded ELF metadata, and profile-guided instrumentation.

Overall, the method enables a network probe to monitor itself internally with minimal overhead. Measurement points are inserted at compile time into selected hot paths, execution frequencies are tracked at runtime, and anomalies are propagated through the probe's existing output channels such as files or message buses.

4.2.4. Detection model

For each instrumented basic block i , at every period T of the `routine_t` thread, the detector computes:

- $\text{delta}_i(t) = \text{counters}[i](t) - \text{counters}[i](t - T)$
- $\text{ewma}_i(t) = (1 - \alpha) \cdot \text{ewma}_i(t - T) + \alpha \cdot \text{delta}_i(t)$ (typically $\alpha = 0.1$)
- An alert is raised when $\text{delta}_i(t) < \beta \cdot \text{ewma}_i(t)$ (β typically 0.5) and $\text{ewma}_i(t) > \theta$ (where θ is a floor threshold filtering out rarely-executed basic blocks).

This criterion detects a sudden drop relative to the recent trend; it requires neither a seasonal model nor a learning phase. The sensitivity of the detector is parameterised through three independent constants. The smoothing factor α controls how quickly the EWMA tracks changes: a small α gives a stable but slow-reacting estimate, a larger α reacts faster at the cost of more noise. The relative threshold β sets how large a drop must be to trigger an alert: it expresses the fraction of the recent trend below which the most recent observation is considered abnormal. The floor threshold θ suppresses alerts on basic blocks whose smoothed activity is too low to be

statistically meaningful, preventing noisy alarms on cold paths. The values of (α, β, θ) are tuned per deployment from a baseline calibration run.

4.2.5. Implementation

4.2.5.1. Compilation pipeline

Figure 17 depicts the integration of the BBProbeV2 pass into the standard Clang/LLVM toolchain. The main flow (left column) follows the conventional path from C source files through the Clang frontend, the LLVM optimisation pipeline, the backend, and the ELF binary. The proposed pass is hooked into the optimisation pipeline at the `OptimizerLastEPCallback` extension point (orange dashed arrow on the figure): it receives the LLVM module after standard optimisations (mem2reg, instcombine, simplifcfg, gvn, and others) have been applied. The configurable inputs of the pass — the `-fpass-plugin=` flag and the environment variables `BBPROBE_ALLOWLIST` and `BBPROBE_ANNOTATION` — are passed to it when Clang loads the plugin.

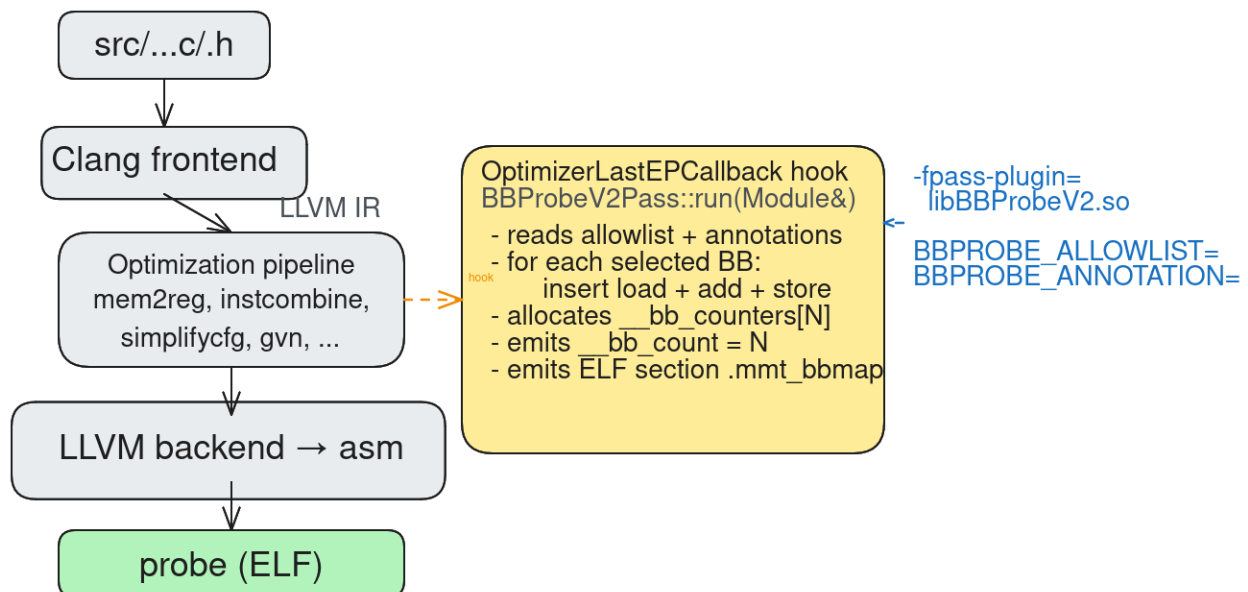


Figure 17. Detailed compilation pipeline showing the placement of the BBProbeV2 pass.

The decision to insert the instrumentation post-optimisation rather than pre-optimisation is deliberate. The pass does not instrument code that would have been eliminated by standard optimisation passes (dead basic blocks, inline functions, unrolled loops); the instrumentation cost therefore matches the cost actually paid at runtime.

4.2.5.2. Runtime view

Figure 18 depicts the data flow inside the running probe process. The three hot paths — capture loop, DPI dispatch, output writing — emit a tick whenever execution enters an instrumented

basic block; the tick corresponds to the inline `__bb_counters[i]++` increment. The cumulative values accumulate in the global `__bb_counters[N]` array. Periodically, every stats-period second (an existing parameter of MMT-Probe), the routine_t thread reads the counter array, updates the EWMA per counter, and raises an alert if the latest delta falls below a fraction of the EWMA while the EWMA stays above the floor threshold. The alert is forwarded to `output_write_report()` and from there to the configured output channels (file, socket, Kafka, MQTT), thus reusing the existing reporting pipeline without introducing any specific infrastructure.

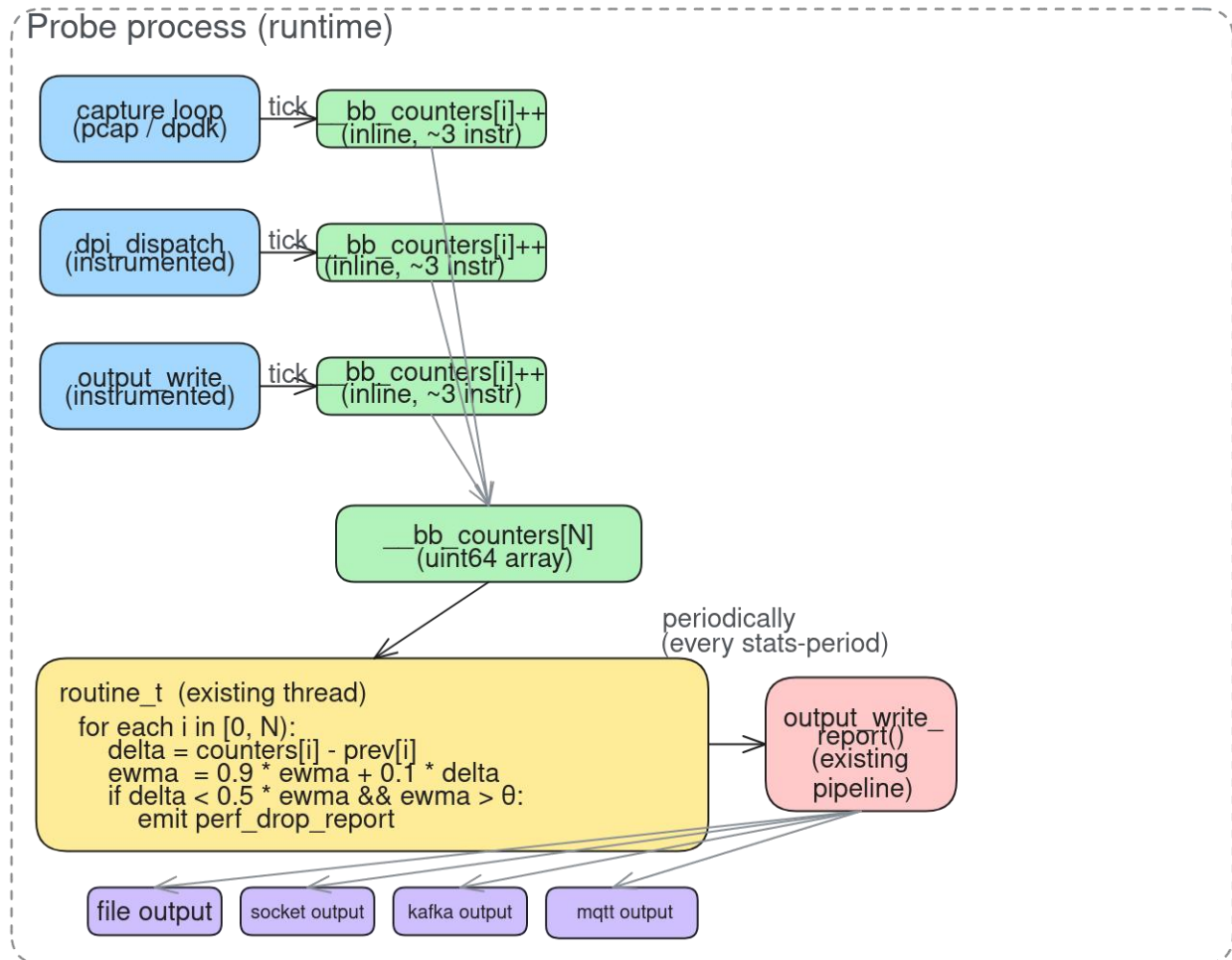


Figure 18. Detailed runtime pipeline including EWMA computation and reporting output channels

Figure 19 shows the content added by the pass in three ELF sections of the produced binary. The `.text` section contains the executable code, now including the inline counter increment in each selected basic block (approximately nine additional bytes per instrumented basic block). The `.bss` section allocates the `__bb_counters[N]` array ($N \times \text{uint64_t}$); placement in `.bss` is appropriate since the array is zero-initialised, incurring no disk cost and only a memory mapping operation at process startup. The `.rodata` section contains the metadata: the constant `__bb_count = N` and

the `__bb_map[id → (function, file:line)]` table. This table is emitted in a custom ELF section, `.mmt_bbmap`, which is read at process startup through `dl_iterate_phdr`, making the binary self-contained and not requiring any companion file to interpret the counters.

ELF segments (after BBProbeV2 pass)

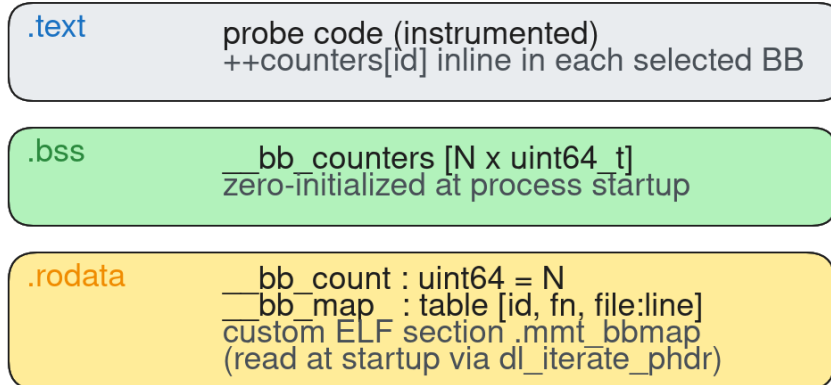


Figure 19. ELF memory layout produced by the BBProbeV2 pass

4.2.6. Per-basic-block elementary cost

On x86_64, the counter increment compiles to three instructions: a load of the current counter value into a register, an addition by one, and a store back to memory. On a modern out-of-order core, the three instructions are issued in parallel with the surrounding basic-block code, yielding an effective cost of approximately one cycle and an instruction footprint of approximately nine bytes per instrumented basic block. For a basic block executed at very high frequency, the unamortised marginal overhead would be significant; in practice, the load and store operations overlap with the basic block's own code, and the measured overhead remains low when the allowlist is focused on the hot paths. Precise overhead figures will be reported against the reference testbed as part of the validation phase described in 0.

4.2.7. Validation methodology

The validation strategy is organised into three phases aligned with the implementation maturity.

The **first phase**, already completed, validates the instrumentation mechanism on the open-source components of MMT-Probe independent of the proprietary `libmmt_core` library. The BBProbeV2 pass was successfully tested on `src/lib/hash.c`, selecting two of nine functions

from the allowlist and inserting forty-nine probes across their basic blocks. End-to-end execution confirmed correct runtime behaviour.

The **second phase**, pending access to a compatible libmmt_core build, will validate compatibility between the Clang-compiled probe and the proprietary library, using representative traffic traces. Functional correctness will be verified by comparing instrumented and non-instrumented binaries on identical traces, while overhead will be evaluated t

hrough wall-clock and CPU-time measurements under multiple instrumentation configurations. The **third phase** will validate the in-process detector by introducing synthetic degradations through artificial delays in selected basic blocks. The EWMA detector will then be evaluated in terms of detection latency, false-positive rate, and parameter sensitivity under realistic traffic variations.

4.2.8. Test infrastructure

Local validation is performed on a Linux Arch host using Clang 21.1.8, while production validation requires a higher-performance environment provided by MONTIMAGE, including a reference deployment of MMT-Probe, representative traffic captures, and potentially DPDK-enabled NIC hardware for line-rate replay.

Compatibility between Clang/LLVM and MMT-Probe has been validated on the open-source codebase. The build system supports Clang transparently, GCC-specific extensions remain compatible, and only a minor linker-flag adjustment is required. The proprietary libmmt_core SDK also links correctly thanks to GCC/Clang C ABI compatibility.

Two LLVM 21 out-of-tree passes have been developed. The first, BBCount, performs static analysis by reporting the number of basic blocks and instructions per function, providing a structural map for instrumentation planning. The second, BBProbeV2, injects probes at basic-block entry points and supports selective instrumentation through allowlists or source-level annotations. Both mechanisms were validated end-to-end and confirmed to instrument only the intended functions.

Based on guidance by MONTIMAGE, the main instrumentation targets are the packet capture loop, protocol dissection engine, rule evaluation engine, and reporting pipeline, which together form the critical execution path of the probe. Selective instrumentation limits overhead while preserving representative measurements on latency-sensitive code paths.

The next stage consists in deploying the instrumented binary on a realistic testbed to collect execution-time and call-frequency profiles. These measurements will identify hot and cold paths, quantify LLVM instrumentation overhead, and establish a baseline for evaluating TEE placement

and D-MUTRA integrity verification, while ensuring runtime monitoring overhead remains below 1% in line with NATWORK KPIs.

Reference testbed provisioned on Hetzner Cloud is detailed in Table 17.

Table 17. MMT probe self-monitoring used testbed specifications

Component	Value
Host class	Hetzner Cloud cx33 (4 vCPU shared, 8GiB RAM)
Region	nbg1 (Nuremberg)
Operating system	Ubuntu 24.04 LTS
Compilers	GCC 13.2.0, Clang 18.1.3 (LLVM 18)
Optimisation level	-O3 (Makefile default)
MMT-DPI	V1.7.10 (Debian package)
MMT-Probe modules linked	KAFKA, REDIS, MQTT
Threading mode	Nb-thread = 0 (single-threaded data plane)
Input	Test/UA-Exp01.pcap (953 packets, $\approx$ 250ms wall-clock)
Runs per condition	10
Execution wrapper	Chrt -f 50 taskset -c 1 (CPU 1 pin + SCHED_FIFO)
Resource sampling	/proc/<pid>/stat every 200ms

Notes on the execution wrapper:

`taskset -c 1`: pins the probe to a single logical CPU so the per-core Time-Stamp Counter stays synchronised between entry and exit `rdtsc` readings.

`chrt -f 50`: raises the probe to SCHED_FIFO priority 50 to limit preemption by other intra-VM processes.

Applied uniformly to every condition. Any residual jitter contributes identically to the control and to the variant under test, and cancels out in the relative overhead computation.

4.2.9. Measures

Five binaries of `./probe` are compiled from the same source tree and compared in Table 18.

Table 18. Five different compiled MMT probes for testing

#	Label	Compiler	LLVM pass	Backend	Allowlist
1	baseline	gcc -O3	—	—	—
2	clang-no-pass	clang -O3	—	—	—
3	counter-hotpaths	clang -O3	active	counter	hot-paths
4	T	clang -O3	active	timer (rdtsc only)	hot-paths
5	TC	clang -O3	active	timer-counter	hot-paths

4.2.9.1. Test scenario

Hot-paths allowlist (per MONT guidance — capture, dissection, worker orchestration):

- src:src/modules/packet_capture/*
- src:src/modules/dpi/dpi.c
- src:src/modules/dpi/dpi.h
- src:src/worker.c

→ 27 functions instrumented.

Per-function injection (applied identically to T and TC):

At function entry: %t = llvm.readcyclecounter(); fence seq_cst.

Before every ret: fence seq_cst; %t' = readcyclecounter(), then bucket = 63 - ctz(%t' - %t), histogram[bucket]++, calls++.

TC additionally inserts ++counters[bb_id] at the head of each instrumented basic block; T omits this step. Histograms are emitted into the ELF section __bb_probe_funcs, whose bounds are exposed by the linker as __start__bb_probe_funcs / __stop__bb_probe_funcs.

A constructor-registered atexit handler in src/lib/bb_probe_rt.c walks the section at shutdown and writes one CSV per run.

Choice of T as control for the latency measurement (not clang-no-pass):

Both T and TC contain the same two rdtsc calls and the same two fence seq_cst barriers in the same positions.

Their contribution cancels exactly in (median_TC – median_T) / median_T.

The remaining delta is attributable to the basic-block counters alone, which is what KPI 1.3.3 actually measures.

4.2.9.2. Global resource measurements

Table 19 shows the CPU expenses over the different CPU extracted elements and MMT measured throughput. Median over ten runs per condition. Wall-clock measured by the bench harness; cpu_mean and rss_peak aggregated from periodic /proc/<pid>/stat samples.

Table 19. CPU expense of MMT probe self monitoring

Metric	unit	baseline	clang-no-pass	T	TC	counter-hotpaths
wall_clock	ms	246.0	255.0	276.5	254.0	248.5
cpu_mean	%	97.15	96.77	95.69	98.59	94.79
cpu_peak	%	97.15	96.77	95.69	98.59	94.79
rss_mean	MiB	17.59	17.06	14.31	17.50	17.00
rss_peak	MiB	26.25	26.38	26.38	26.50	26.38
throughput	pkt/s	3 874	3 738	3 447	3 752	3 835

KPI 1.3.4 (CPU / RAM overhead < 10 %)

- Control = clang-no-pass, isolating pass cost from GCC↔Clang compiler differences.
- Worst overhead observed across the three instrumented variants: 1.88 % (TC vs clang-no-pass).
- Verdict: PASS with a factor-of-5 margin to the bound.

4.2.9.3. Per-function latency measurements

Wall-clock alone cannot isolate the basic-block-level overhead targeted by KPI 1.3.3:

A 10 % degradation on a single hot function is averaged out at the global scale by surrounding traffic.

The testbed therefore reports per-function median execution time in CPU cycles, derived from the 64-bucket logarithmic histograms emitted at runtime by the pass.

Median interpolated linearly across the histogram CDF. Bucket b covers cycle deltas in $[2^b, 2^{b+1})$ → per-bucket resolution ± 50 %, sufficient to resolve overheads on the order of a few percent.

Filtering rule:

Functions called fewer than $n_{\min} = 100$ times during the run are excluded from the verdict (BBPROBE_MIN_CALLS env override).

At $n_{\min} = 100$ the median estimator is stable to within one bucket width; below this, figures stay in the CSV for transparency but do not set the verdict.

9 of the 27 instrumented functions satisfy $n_{\text{calls}} \geq 100$ on the current trace.

Median cycle counts and relative overhead, ordered by descending overhead as defined in Table 20.

Table 20. Penalty induced by the monitoring on MMT functions

Function	n_{calls}	median_T (cyc)	median_TC (cyc)	$\Delta \%$
worker.c:worker_process_a_packet	953	3 910	4 065	+3.97 %
pcap_capture.c:_got_a_packet	953	27 308	27 502	+0.71 %
dpi.c:_period_session_report	94 391	193	193	-0.03 %
worker.c:worker_update_timer	953	22 751	22 683	-0.30 %
dpi.c:_do_stat_reports	952	22 667	22 580	-0.38 %
dpi.c:dpi_update_timer	953	22 715	22 622	-0.41 %
dpi.c:_packet_handler	953	200	198	-0.95 %
dpi.c:_starting_session_handler	171	896	878	-1.96 %
dpi.c:_ending_session_handler	171	285	239	-16.15 % *

* Negative figures = run-to-run jitter on a shared-vCPU host (residual variance floor at $n = 10$ runs is $\approx \pm 3 \%$), not a speed-up. At $n = 171$ the median estimator is at the lower edge of the statistical stability range admitted by $n_{\min}$. The high- n rows at the top (worker_process_a_packet at $n = 953$ and _period_session_report at $n = 94\,391$) carry essentially all of the verdict's information.

4.2.9.4. Verdict against NETWORK KPIs

- KPI 1.3.3 (runtime performance degradation $< 10 \%$, control = T)
TC: worst-function overhead = +3.97 % >> PASS
function = worker.c:worker_process_a_packet
- KPI 1.3.4 (CPU / RAM overhead $< 10 \%$, control = clang-no-pass)
T: worst overhead = +0.00 % >> PASS
TC: worst overhead = +1.88 % >> PASS
counter-hotpaths: worst overhead = 0.00 % >> PASS

Both KPIs pass with margin.

Worst-function instrumentation overhead on the data plane = **+3.97 %**, below half of the 10 % NETWORK bound, factor-of-2.5 margin for future deployment variability.

4.2.10. Interpretation

We provide an interpretation of the above attained results.

Top row — **worker.c:worker_process_a_packet**:

- Per-packet dispatcher: 1 call per input packet (953 on this trace).
- Body executes in $\approx 4\,000$ cycles ($\approx 2\,\mu\text{s}$ at 2 GHz).
- The basic-block counters distinguishing TC from T add a fixed load–add–store per entered block.
- Overhead remains under 4 %

Near-zero row — **dpi.c:_period_session_report**:

- 94 391 calls per run, body ≈ 190 cycles.
- Load–add–store issued in parallel with the function’s own work by the out-of-order pipeline.
- Observable cost collapses to within the measurement floor: $\text{median}(\text{TC}) = \text{median}(\text{T})$ to within one bucket.

General pattern across data-plane functions:

- Cluster within a few percent of zero.
- Overhead scales with *number of executed basic blocks*, not with *number of instrumented basic blocks*.
- A counter inserted into a basic block that does not execute costs nothing $\rightarrow$ the hot-paths allowlist is, in this respect, a sound choice.

4.2.11. Conclusion

Phase 2 of the validation strategy is complete. The Clang-compiled probe links correctly against the proprietary libmmt_core, produces bit-identical outputs to the non-instrumented baseline on the reference trace, and exhibits an overhead well below the NATWORK D2.2 bounds.

Per-function instrumentation cost on the data plane stays under 4 % in the worst case, with most hot functions showing no measurable overhead. CPU and RAM stay within 2 %.

Both KPI 1.3.3 and KPI 1.3.4 pass with a comfortable margin, validating the feasibility of an in-binary self-monitoring mechanism for high-throughput network probes on x86_64.

Remaining work for the third phase — synthetic-degradation injection and EWMA detector evaluation — is unblocked and can proceed against the same testbed.

4.2.12. Beyond the state of the art

Beyond the immediate deliverable, several extensions open the door to original research contributions. The first axis of progress concerns scalability and robustness: storing counters per worker rather than in a shared array, so that the probe remains fast on multi-core machines; embedding the symbol mapping directly inside the produced binary, so that no auxiliary file needs to be deployed alongside it; and adding the ability to turn monitoring on or off at runtime, so that an instrumented binary can run in production with zero overhead until monitoring is explicitly enabled. The second axis concerns smarter detection: replacing the simple EWMA with a model aware of seasonal traffic patterns in order to reduce false positives during business cycles and cross-checking the internal signal with external indicators such as NIC drop counts or hardware performance counters in order to confirm that a slowdown is real before alerting. The third axis concerns smarter targeting: using a preliminary profiling run to automatically discover the hot paths and generate the allowlist for the next build, so that only the basic blocks that genuinely matter carry the instrumentation cost. Together, these directions sketch a research roadmap toward a publishable contribution, focusing in particular on the combination of adaptive sampling, automatic generation of authorisation lists and external cross-validation.

4.2.13. Progress towards D3.5

LLVM-based probe insertion for native code performance analysis was not started and discussed in D3.5 and is therefore a novel item and NATWORK project outcome.

4.3. Dynamic User Association and Resource Allocation in Modern Heterogeneous Cellular Networks

4.3.1. Domain

Modern heterogeneous cellular networks (HetNets) must continuously allocate users and manage radio resources while operating under highly dynamic traffic conditions. Since several classes of Base Stations (BSs) with different transmit powers coexist within the same deployment, user association decisions directly influence network throughput, Quality-of-Service (QoS), interference levels, and energy consumption. Consequently, optimizing User Association and

Resource Allocation (UARA) has become one of the fundamental challenges in Beyond 5G (B5G) systems [112].

Conventional association techniques, including maximum received Signal-to-Interference-plus-Noise Ratio (SINR) and Cell Range Expansion (CRE), optimize only a subset of these objectives and therefore cannot adequately respond to changing network conditions [113]. More sophisticated optimization and Reinforcement Learning approaches improve decision quality but often introduce high computational complexity, limited scalability and adaptability, or increased dependence on detailed User Equipment (UE) information.

This solution provides a hybrid resource management architecture that combines distributed Game Theory with centralized Deep Reinforcement Learning (DRL). Instead of directly assigning users to BSs, the centralized controller guides the performance of a distributed non-cooperative game executed by the Ues, allowing the network to adapt its behaviour according to different operational objectives while maintaining low execution complexity.

4.3.2. Problem Statement

This section addresses the following research question: How can a HetNet dynamically optimize UARA while ensuring scalability, low computational complexity, and adaptability to diverse operational objectives under varying traffic conditions?

This objective introduces several key research challenges. Classical mathematical optimization methods typically rely on iterative procedures that become computationally intensive, limiting their feasibility for real-time deployment. In contrast, Reinforcement Learning-based approaches often face challenges related to combinatorial action spaces, user-dependent state representations, and rapidly increasing computational complexity as network size grows [114]. In addition, many existing frameworks depend on detailed individual UE measurements, such as SINR vectors or Channel State Information (CSI), which raises privacy-related concerns. Moreover, Multi-Agent Reinforcement Learning (MARL) methods may require on-device neural network inference at the UE level, which is often impractical for resource-constrained devices [115].

A further limitation of current UARA schemes lies in their reliance on fixed optimization objectives. In real-world deployments, network operators frequently need to adaptively prioritize different goals, such as maximizing throughput under high traffic load, reducing power consumption during low-load conditions, or enhancing coverage in interference-dominated environments [112]. However, most existing approaches lack a flexible mechanism that enables effective transitions between such operational modes without requiring redesign or retraining of the underlying optimization framework.

Therefore, the goal of this solution is the design of a scalable orchestration framework that can dynamically guide the distributed UARA process toward varying macroscopic network objectives, while maintaining low computational overhead, robustness under heterogeneous traffic conditions, and practical deployability in modern HetNet scenarios.

4.3.3. Solution

To address the aforementioned challenges, this work introduces a bilevel orchestration framework for game-theoretic UARA in HetNets. At the execution layer, Ues autonomously select their serving BS and radio channel through a distributed multi-objective non-cooperative game. Each UE evaluates the available BS–channel combinations using a utility function that accounts for achievable data rate, received SINR, and BS power consumption. Strategy updates are carried out asynchronously via Best-Response Dynamics, continuing until the system converges to a stable equilibrium point.

At the orchestration layer, a centralized Deep Q-Network (DQN) controller dynamically tunes the weighting parameters of the distributed utility function. Instead of directly optimizing individual UE associations, the controller regulates the relative emphasis assigned to throughput, link quality, and power cost. This control mechanism allows the network to shift between different performance modes, employing balanced, power-aware, and coverage-enhancing operation, while preserving the decentralized structure of the UARA process.

In contrast to typical centralized Reinforcement Learning approaches, the proposed architecture ensures that both state and action spaces remain relatively independent of the number of active users. The DQN controller operates on BS load information rather than UE-level measurements, thereby limiting exposure of sensitive information and enabling efficient operation across different traffic loads.

4.3.4. Distributed game-theoretic decision model

The UARA process is modelled as a multi-objective non-cooperative game in which each UE selects a single BS–channel pair to maximize a utility function that combines normalized data rate, received SINR, and BS power consumption. The system operates through Asynchronous Best-Response Dynamics, where Ues iteratively update their associations whenever a strictly better utility is found. Each unilateral update immediately affects the corresponding BS load state, which in turn influences subsequent decisions of other users. This distributed interaction continues until no UE can improve its utility through a unilateral change. The process converges to a Pure Strategy Nash Equilibrium (PSNE), where all users reach stable associations. The

decentralized nature of the decision-making ensures scalability through manageable computational complexity that scales linearly with the network size.

4.3.5. Reinforcement-learning orchestration scheme

The HetNet Controller incorporates a DQN agent that dynamically configures the weighting parameters of the SINR and power consumption components in the utility function, while keeping the rate-related weight fixed to ensure consistent throughput-oriented behavior. In this way, the controller steers the distributed UARA process toward different macroscopic network objectives. Specifically, by adjusting these weighting parameters, the network can transition between power-aware, coverage enhancing, and balanced operation, depending on the observed traffic conditions.

The developed DQN operates using aggregated BS traffic information rather than UE-specific measurements, thereby respecting privacy standards. Moreover, this abstraction allows the learned policy to remain effective across different traffic distributions, including scenarios not explicitly seen during training. The algorithmic flow of the developed DQN orchestration scheme is illustrated in Figure 20.

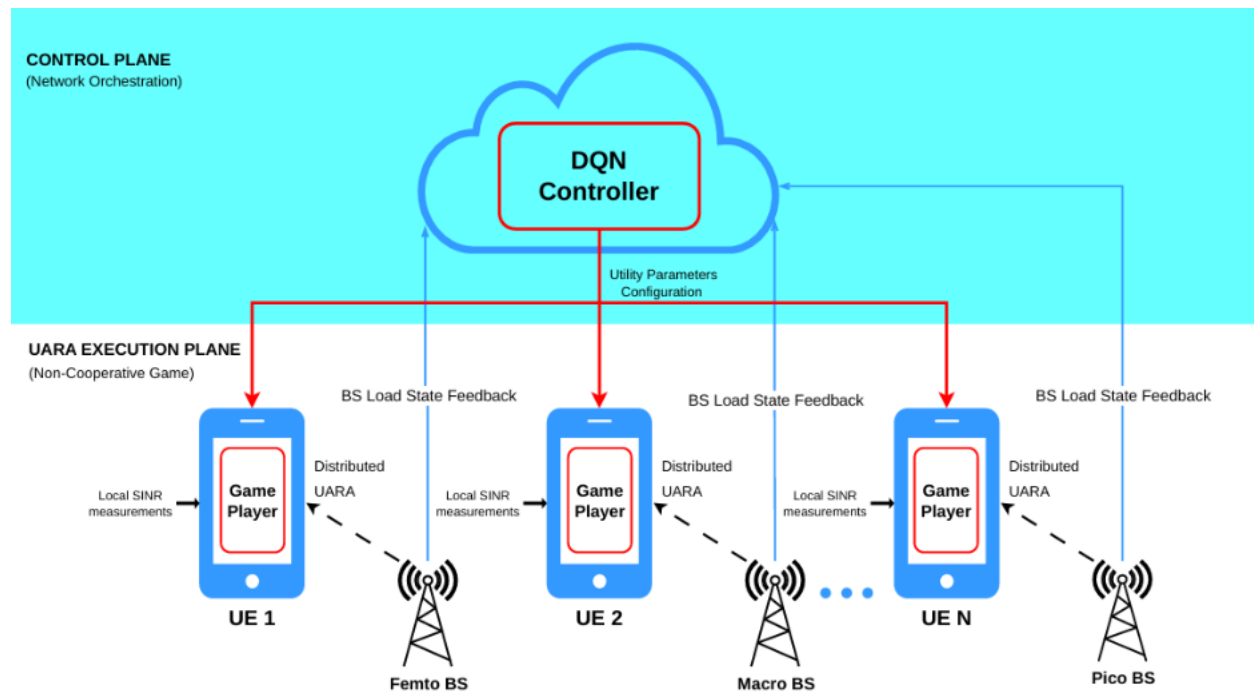


Figure 20 Algorithmic flow of the proposed DQN orchestration scheme

4.3.6. Evaluation Methodology

The framework is evaluated in a realistic three-tier urban HetNet scenario based on 3rd Generation Partnership Project (3GPP) TR 38.901-standardized urban channel models using the Quadriga simulator. The BS layout remains fixed across different evaluated instances and large-scale fading effects remain consistent. Ues are randomly placed inside the network area, with a higher probability inside areas of concentrated traffic, referred to as Hotspots. System performance is assessed in terms of average UE SINR, aggregate network throughput, total network power consumption, and execution time under different traffic densities and network operating objectives. System performance is examined in terms of network throughput, average UE SINR, total power consumption, and response time.

4.3.7. Results

The proposed solution (DQN-GT) is evaluated against four representative UARA approaches, namely Max-SINR association, Cell Range Expansion (CRE), a static game-theoretic formulation, and an Exhaustive Search-orchestrated Game (ES-GT), which provides the optimal utility-parameter configuration for comparison. Performance is assessed in terms of network throughput, average UE SINR, total network power consumption, and response time under different traffic densities and network operating objectives.

Table 21: System Metrics Relative to the Balanced Policy for 20 Ues

Method	SINR (dB)	Power (W)	Throughput (Mbps)	Response time (ms)
Max-SINR	7.35	61.02	6.07	<0.01
CRE	6.97	47.76	6.29	<0.01
ES-GT	6.95	48.40	7.31	49.73
DQN-GT	6.98	50.76	7.26	0.17
Static Game	6.55	44.34	7.46	0.14

The obtained results demonstrate that, under the balanced policy reported in Table 21, Table 22, and Table 23, the controller-orchestrated game-theoretic frameworks consistently achieve higher aggregate throughput than the signal-based association schemes, while maintaining comparable average UE SINR and network power consumption to CRE. This improvement highlights the benefit of the proposed multi-objective utility formulation, which jointly considers throughput, signal quality, and power cost, instead of relying solely on radio signal measurements.

Table 22: System Metrics Relative to the Balanced Policy for 40 Ues

Method	SINR (dB)	Power (W)	Throughput (Mbps)	Response time (ms)
Max-SINR	7.40	117.15	8.98	<0.01
CRE	7.01	88.11	9.09	<0.01
ES-GT	6.89	87.22	10.24	89.79
DQN-GT	6.87	90.42	10.20	0.27
Static Game	5.78	68.66	10.37	0.24

Table 23: System Metrics Relative to the Balanced Policy for 80 Ues

Method	SINR (dB)	Power (W)	Throughput (Mbps)	Response time (ms)
Max-SINR	7.30	239.16	11.04	<0.01
CRE	6.90	178.61	10.85	<0.01
ES-GT	6.74	173.59	11.19	186.05
DQN-GT	6.52	178.14	11.22	0.48
Static Game	4.87	120.94	10.94	0.45

A significant advantage of the proposed bilevel architecture compared to existing mathematical optimization and reinforcement learning methods is its computational efficiency. By restricting the DQN to the orchestration of utility-function parameters while delegating user association to the distributed game, the controller maintains state and action spaces that are independent of the number of active users. As a result, the framework preserves low execution latency even for the traffic load of 80 users, as reported in Table 23 and Table 24, therefore remaining suitable for low-latency operation in dense network deployments.

Table 24: System Metrics of the Coverage-Enhancing Policy for 40 and 80 Ues

Number of users	Method	SINR (dB)	Power (W)	Throughput (Mbps)	Response time (ms)
40	ES-GT	7.29	104.46	10.14	88.90
	DQN-GT	7.19	102.55	10.20	0.30
80	ES-GT	7.22	217.85	11.52	187.92
	DQN-GT	7.14	213.98	11.45	0.49

DQN-GT effectively approximates the optimal policy identified by the exhaustive-search solution while reducing the computational cost by several orders of magnitude. High approximation

accuracy is maintained not only for the traffic density of 40 Ues used during training, but also for previously unseen network loads of 20 and 80 users, demonstrating the generalization capability of the developed controller without requiring retraining. These observations hold for all the evaluated traffic loads regarding the performance policies of balanced, power-aware, and coverage-enhancing operation, with the corresponding results being reported in Table 21, Table 22, Table 24 and Table 25.

Table 25: System Metrics of the Power-Aware Policy for 20 and 40 Ues

Number of users	Method	SINR (dB)	Power (W)	Throughput (Mbps)	Response time (ms)
20	ES-GT	2.10	23.19	7.27	47.64
	DQN-GT	3.27	27.86	7.57	0.18
40	ES-GT	1.32	34.44	10.83	89.41
	DQN-GT	1.51	36.83	10.94	0.30

The evaluated operational policies demonstrate the flexibility of the proposed orchestration framework. The Balanced Policy, presented in Table 22, Table 23 and Table 24, provides an effective trade-off between throughput, coverage, and power consumption under general operating conditions. The Power-Aware Policy, presented in Table 25, reduces network energy consumption while maintaining satisfactory communication performance, making it particularly suitable for low-congestion periods, such as reduced nighttime traffic. Conversely, the Coverage-Enhancing Policy, presented in Table 25, prioritizes radio-link quality and improves network coverage and throughput under highly congested conditions, making it well suited for temporary traffic surges, such as those occurring after large public events.

4.3.8. Conclusion

The developed DQN-orchestrated framework addresses several limitations of existing UARA solutions by combining distributed Game Theory with centralized Deep Reinforcement Learning. In contrast to centralized optimization and conventional Reinforcement Learning approaches, it maintains low computational complexity and scalability while avoiding dependence on detailed UE-specific information and preserving adaptability across different network operating objectives and traffic densities.

4.3.9. Beyond the state of the art

The developed solution advances beyond existing UARA approaches in several dimensions. Compared to signal-based association schemes such as Max-SINR and Cell Range Expansion, that are signal-oriented, the solution jointly optimizes throughput, link quality, and power

consumption, provides more efficient resource management, and supports runtime transitions between operational objectives without requiring redesign or retraining.

Regarding the mathematical optimization approaches, which typically rely on iterative procedures that become computationally prohibitive as network size grows, the solution operates in the sub-millisecond regime even under high congestion, remaining well within real-time operational constraints. As concerns the centralized RL approaches, which typically define action spaces that grow combinatorially with the number of active users, the proposed DQN orchestration mechanism maintains constant state and action space dimensionality regardless of network density, enabling scalability while maintaining stable performance.

Furthermore, unlike existing optimization- and RL-based UARA frameworks that depend on detailed UE-level measurements such as CSI or SINR vectors, the proposed DQN controller operates on aggregated BS-level traffic load information, thereby limiting exposure of sensitive user data while remaining effective across varying traffic conditions.

4.3.10. Progress from D3.5

During the development of Deliverable D3.5, the research question had not yet been formally defined. Since then, a comprehensive state-of-the-art review on UARA in HetNets has been completed, the research question has been clearly formulated, and both the mathematical modelling and simulation framework of the proposed solution have been fully developed. In addition, the evaluation phase has been conducted and the corresponding results have been obtained.

4.4. Native xApp remote attestation and runtime integrity verification

4.4.1. Domain and problem statement

Security Domain. The security domain of this work focuses on runtime trust assurance for native O-RAN xApps deployed in a Near-RT RIC environment. More specifically, the domain includes:

- Near-RT RIC security, because the xApp operates as part of the RAN intelligent control environment.
- Native xApp integrity verification, since the xApp runs directly on a host server rather than inside a Kubernetes-managed container.

- 5G network slice security, because the xApp monitors and classifies slice-level traffic. Host-based remote attestation, where the trustworthiness of the server, operating system, and xApp execution environment must be verified.
- Runtime integrity monitoring, ensuring that the xApp binary, configuration, model files, and critical dependencies remain unchanged during execution.
- Security monitoring trustworthiness, since the classification results produced by the xApp must be reliable before they are used for alerting or mitigation.

Figure 21 shows the position and interactions of the Anomalous Traffic Detection xApp with the near real time RIC.

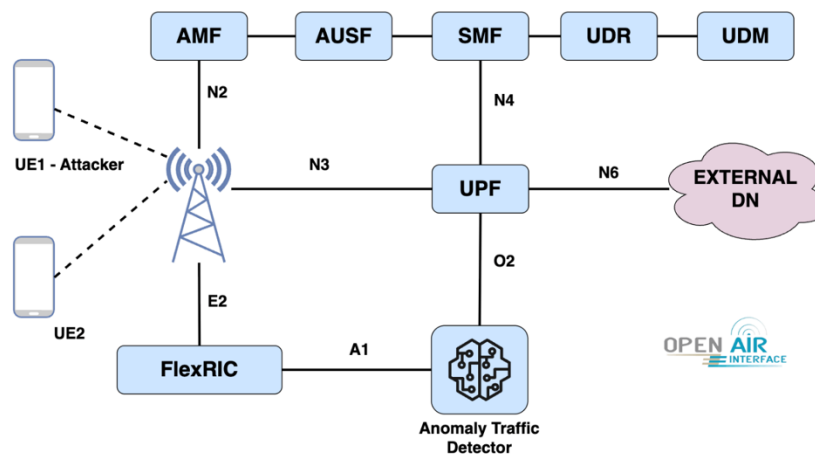


Figure 21. Native xApp for Anomaly Traffic Detection and its interaction with the near RT RIC elements

In this context, D-MUTRA can be employed to provide remote attestation and runtime integrity verification for the native xApp and its hosting environment. Instead of verifying a Kubernetes pod or container image, D-MUTRA would verify the integrity of the executable files, configuration files, model artifacts, and runtime process state associated with the xApp. Below is the description of security threat, domain and problem statement:

Security Threat. The considered security threat concerns the tampering, compromise, or unauthorized modification of a native xApp deployed alongside a Near-RT RIC on a physical server. The xApp aka Anomaly Traffic Detector is responsible for monitoring traffic in a 5G network and classifying slice-level traffic as either benign or abnormal/malicious. Since the xApp contributes to security-relevant decision-making, its integrity is critical for the correctness and reliability of the overall detection workflow. In this deployment model, the xApp does not run as a containerized Kubernetes workload, but as a native software component installed and executed directly on the host server. Therefore, the attack surface includes unauthorized

changes to the xApp binary, execution scripts, configuration files, machine learning model files, threshold parameters, libraries, runtime dependencies, or communication interfaces with the Near-RT RIC. An attacker who gains access to the server or to the xApp deployment path could modify these assets in order to influence the classification outcome. A compromised xApp may misclassify malicious slice traffic as benign, suppress alerts, modify anomaly thresholds, alter the loaded detection model, or manipulate the reporting logic toward the Near-RT RIC. As a result, attacks against the 5G network slice may remain active and undetected. Conversely, the attacker could also force benign traffic to be classified as malicious, causing unnecessary alarms, service disruption, or incorrect mitigation actions. This threat is particularly important because the Near-RT RIC relies on the xApp's output for near-real-time traffic awareness and potentially for triggering security-related responses. Therefore, it is not sufficient to assume that the xApp is trustworthy simply because it has been deployed on the server. Its integrity must be verified before execution and continuously monitored during runtime.

4.4.2. NATWORK solution. D-MUTRA for xApp runtime integrity verification

To address this problem, D-MUTRA is considered as a mechanism for enabling host-level remote attestation and runtime integrity verification of the native xApp. The objective is to verify that the xApp is executed on a trusted server, that its binary and supporting files match the expected trusted measurements, and that no unauthorized modification occurs during runtime. In this way, the system can establish and maintain trust in the xApp before its classification outputs are used for traffic monitoring, anomaly detection, and security policy enforcement. We want to guarantee that an xApp deployed on a Near-RT RIC is the exact version we validated and signed, and that nobody has modified it between the moment we shipped it and the moment its decisions are consumed by the RIC. When the xApp is a traffic classifier that labels 5G slice flows as benign or malicious, its trustworthiness directly conditions slice security — a tampered xApp can silently flip malicious traffic to benign or trigger false alarms. Hence remote attestation: not trust yourself, but prove to an independent third party that you have not been modified.

4.4.3. Implementation and workflow

Figure 22 shows the general layout, including the different components dealing with remote attestation of the xApps. Noticeably, it also shows what is actually verified inside the xApp.

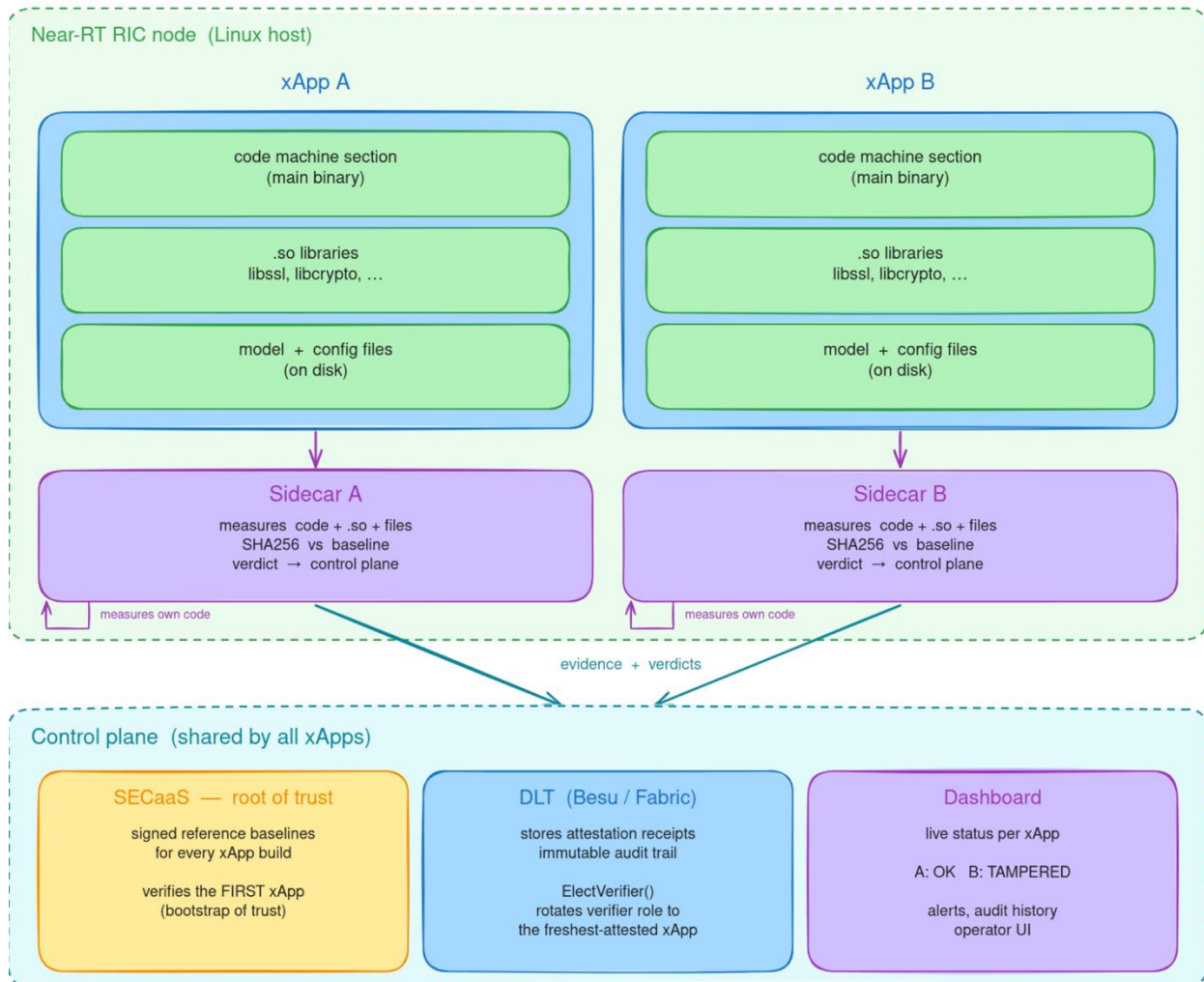


Figure 22. D-MUTRA layout for xApp integrity verification

4.4.3.1. Measures

Four families of assets, represented by the three green sub-blocks inside each xApp (ie, Block 1-3 below), complemented with the sidecar itself (ie, Block 4 below):

Block 1 — code machine section (main binary)

The xApp's executable code, as it is loaded in memory (not the on-disk file). We read the code section of the process through `/proc/<pid>/mem` and compute its SHA256. This is what catches an in-memory hot-patch, a code injection, or a binary substituted at launch.

Block 2 — shared libraries (.so)

The shared libraries loaded by the process (libssl, libcrypto, app-specific .so files). Same mechanic as for the main binary: SHA256 over the executable portion mapped into the process. A library replaced with a backdoored version is detected exactly the same way as a tampered binary.

Block 3 — model and config files (on disk)

The on-disk artefacts the xApp depends on: the trained ML model (weights), configuration files, anomaly thresholds, shell scripts. For these we hash the file directly on the filesystem — not memory — at startup and periodically. This covers the scenario where an attacker swaps the ML model to bias detection without touching the binary itself.

Block 4 — the sidecar itself (self-attestation)

On the same cycle, the sidecar also measures its own code section. Without this step, the trust chain has a missing link: a tampered sidecar could silently approve a tampered xApp. Self-measurement closes the loop — if the sidecar binary is modified, its own verdict raises TAMPERED before it has any chance to lie about the xApp it is supposed to watch. This is the small self-loop arrow drawn on each sidecar in the diagram.

4.4.3.2. *Measuring entity*

A dedicated attestation sidecar runs alongside each xApp, independent of the xApp process. It is a Go binary that:

1. reads the xApp's memory through /proc/<pid>/mem (for code and shared libraries)
2. reads the relevant files on disk (for model and configs)
3. computes a SHA256 on each target
4. compares against the baseline signed at delivery time
5. emits an OK or TAMPERED verdict

On the same cycle, the sidecar produces a combined digest: SHA256(xApp digest || sidecar digest). The xApp and the sidecar are bound together into a single measurement. This combined digest is then compared against a combined reference baseline (xApp + sidecar) signed at delivery time. If either side has been tampered with — the xApp or the sidecar itself — the combined hash diverges from the baseline and the verdict turns to TAMPERED. Neither component can claim integrity on its own. This is the self-attestation pass shown as the small loop arrow in the diagram.

The sidecar does not touch the xApp: no injection, no hooks, no instrumentation. It observes from the outside. That is what makes the solution non-intrusive and compatible with any existing xApp, whether it ships as a container or as a native systemd binary.

4.4.3.3. *Control plane*

A local sidecar saying "my xApp is fine" provides no guarantee on its own: if it is compromised, it will lie. The decision must therefore be externalised to a shared control plane. That is the role of the cyan band at the bottom of the diagram.

There are three components, as described next: SECaaS — root of trust, DLT — Besu QBFT or Fabric, and Dashboard.

4.4.3.4. *SECaaS — root of trust*

Holds the signed reference baselines computed at build time (the expected SHA256 of the code, libraries, model, and configs for every version of every xApp). It is our off-chain root of trust. Its critical role: verify the very first xApp at system startup, when the attestation chain is empty and no trusted peer exists yet. Without SECaaS, there is nothing to bootstrap against.

4.4.3.5. *DLT — Besu QBFT or Fabric*

Stores every attestation receipt on a blockchain. Two contributions:

Immutable trail: we can prove after the fact which xApp was healthy at which point — full audit.

Verifier election: at every new cycle, the smart contract's ElectVerifier() looks at which agent was last successfully attested, and hands the verifier role to that one. The role rotates automatically among healthy xApps, which removes any fixed trusted point from the system.

4.4.4. *Dashboard*

Operator UI. Shows the live OK / TAMPERED status per xApp, history, and alerts. No logic of its own — it consumes the DLT and SECaaS.

4.4.4.1. *The full workflow — rotating verifier*

The flow across these components, adapted to xApps:

- Step 1: xApp A sends Evidence — SECaaS is elected verifier (no past attestation)
- Step 2: SECaaS compares Evidence against the signed baseline
- Step 3: SECaaS closes — lastSuccess[A] = timestamp
- Step 4: xApp B sends Evidence — DLT elects A (freshest attested)
- Step 5: SECaaS publishes RefSignatures
- Step 6: xApp A (elected verifier) listens, compares, closes — lastSuccess[B]
- Step 7: xApp C sends Evidence — DLT elects B ...
- Step 8: and so on, every attested xApp becomes verifier for the next.

In practice, once bootstrap is done by SECaaS, the xApps attest each other — whichever one just proved its integrity validates the next one. If an xApp cheats, its TAMPERED verdict is recorded on the DLT and it will never be elected as verifier again.

4.4.4.2. *Permissions — the orange line on the diagram*

Reading `/proc/<pid>/mem` is a privileged operation under Linux. The sidecar must satisfy one of three conditions:

- run as root (simple but broad)
- carry the `CAP_SYS_PTRACE` capability (recommended: dedicated system user with the granular capability)
- run under the same UID as the xApp (handy in dev, not in production)

This is the isolation knob: we want the prover to have the least privilege possible while still being able to observe its target.

4.4.5. KPI compliance verification

The three UC1.2 KPIs defined in D2.2 — remote attestation cycle latency (1.3.1, < 1 s), runtime performance degradation (1.3.3, < 10 %), and resource / energy footprint (1.3.4, < 10 %) — were measured end-to-end on the OAI FlexRIC native xApp `xapp_rc_slice_ctrl`, attested by the D-MUTRA Go sidecar against a Hetzner-hosted Besu QBFT core. All three pass with substantial margin under the production configuration retained for deployment.

4.4.5.1. *Retained configuration: cache-isolation pinning*

Among the mitigation axes explored, the decisive one is pinning the sidecar's prover loop to a separate L2-cache cluster via the `systemd` directive `AllowedCPUs`. The other axes — CPU quota, interval throttle, `nice` / IO scheduling — are kept as conservative defensive bounds but do not carry the KPI compliance.

The sidecar's per-cycle work is dominated by reading `/proc/<xapp-pid>/mem` (~ 10 MiB) and computing SHA256. When the sidecar and the xApp share the same L1/L2 cache domain, those reads invalidate the xApp's hot lines and inflate the latency of its decision loop. On the lab host (Intel i7-14700, hybrid 8 P-cores + 12 E-cores) the E-cores expose three independent L2 clusters (16-19, 20-23, 24-27). Pinning the sidecar to cluster 24-27 gives it a private L1+L2 domain while the xApp keeps the P-cores. L3 and DRAM remain shared but do not dominate at sub-millisecond resolution — verified experimentally by isolating this axis alone: the worst-case (p99) runtime penalty collapses from +10 % at baseline to +0.05 %.

4.4.5.2. *Headline results*

Table 26. KPI validation for native xAPP remote attestation and runtime integrity verification

KPI	NATWORK target	Measured	Margin	Verdict
1.3.1 in-scope cycle (hash + verify + write)	< 1 s	225 ms	4.4 ×	PASS
1.3.3 performance degradation (p99)	< 10 %	+0.05 %	200 ×	PASS
1.3.4 sidecar CPU footprint (mean)	< 10 %	1.78 %	5.6 ×	PASS

4.4.5.3. *KPI 1.3.1 — Attestation cycle latency*

The cycle latency covers the work performed locally by the sidecar to produce and submit one attestation: hashing the xApp .text and the sidecar's own .text (self-integrity), computing the combined digest, and writing the evidence transaction to the DLT. The on-chain close, oracle reference-digest lookup and verifier election are out of scope, in line with the proposal's framing — the remaining latency depends on the network and DLT backend.

Table 27. KPI 1.3.1 compliance for xApp remote attestation

Phase	Latency
Hash xApp .text (~ 10 MB, SHA256)	~ 20 ms
Hash sidecar self .text (~ 10 MB, SHA256)	~ 20 ms
Combined digest + local compare	< 1 ms
bc.SendEvidence DLT commit (signed tx → Besu RPC)	183 ms ($\sigma = 2$ ms, $n = 8$)
TOTAL in-scope (vs target < 1 s)	~ 225 ms — PASS

4.4.5.4. *KPI 1.3.3 — Performance degradation*

Per-call clock_gettime brackets around parse_and_apply_slicing() in a sentinel-wrapped copy of the xApp source. 180 s × 2 phases (sidecar OFF / sidecar ON), ~ 906 samples per phase, two mock anomaly-detection clients at 5 Hz driving the xApp decision loop. The p99 (worst-case 1 % of calls) is reported as the load-bearing metric since the mean is noise-dominated at sub-millisecond resolution on a non-isolated lab host.

Table 28. KPI 1.3.3 compliance for xApp remote attestation

Statistic	OFF (sidecar stopped)	ON (sidecar active, cpuset 24-27)	Penalty
mean	284.7 μ s	304.3 μ s	+6.9 %
p99	480.7 μ s	481.0 μ s	+0.05 %

At the p99 the difference (+0.3 μ s over a 480 μ s envelope) is well within lab-host noise — the sidecar is, for the xApp's hot path, effectively invisible.

4.4.5.5. KPI 1.3.4 — Sidecar resource / energy footprint

Sampling /proc/<sidecar-pid>/stat at 1 Hz over 300 s. CPU% is computed as $\Delta(\text{utime} + \text{stime ticks}) / \text{TICKS_PER_SEC} / \Delta t \times 100$ (per-core scale); RSS from the resident-set-size field $\times \text{PAGESIZE}$. Energy is reported via CPU% as the standard proxy.

Table 29. KPI 1.3.4 compliance for xApp remote attestation

Metric	mean	max	min
CPU %	1.78 %	5.0 %	0.4 %
RSS (MiB)	76	90	63

The CPUQuota=10 % directive caps the sidecar at 10 % of one core (0.36 % of system) as a hard ceiling, independent of attestation cadence. The observed 1.78 % mean stays comfortably below that bound and well under the 10 % NETWORK target.

4.4.6. Beyond the state of the art

As recalled in the state of the art for xAPP security detailed in section 2.2.5 and in perfect alignment with O-RAN WG 11's recommendations and expectations for future security enforcement on xApp in near RT RIC, D-MUTRA brings two recommended services of remote attestation at onboarding and continuous runtime integrity verification, fixing WG 11's challenge of penalty. The state of the art is overcome by the following features and proprieties of equivalent importance:

- Zero hardware and system dependency enabling secure deployment anywhere
- Zero penalty for runtime continuous verifications (which is meeting WG11 defined challenge)
- Zero touch for signature and key management, automatically handled through the blockchain, totally relaxing DevSecOps

4.4.7. Progress beyond D3.5

The integration work between D-MUTRA and Anomalous Traffic Detection has not been started before and described in D 3.5.

5. NATWORK's PDSCM on WASM workloads

In D3.5, we have described architectural design of a novel WASM integrity verification technique, relying on a novel ad hoc functionality implementation on the WASMTIME open source WASM runtime, forcing the runtime JIT compilation and serialization to reconstruct a native ELF-formatted workload. In the domain of WASM integrity verification, NATWORK's technique sets a new state of the art.

Pursuing the same approach (i.e., enriching the WASM runtime), we have defined two advances on the state of the art looking at WASM confidentiality and WASM availability, described below.

5.1. WASM integrity

5.1.1. General

D3.5 describes a novel technique to confer runtime integrity to WASM module. We are currently working in an integration of it inside a WASM full stack attestation mechanism, employing iMEC WASM runtime integrity verification and our WASM module integrity verification.

5.1.2. Progress versus the state of the art

As described in D 3.5, the method is novel, bringing a step towards the state of the art which does not contain such existing-platform agnostic integrity enforcement solution. The solution implies a change of the WASM runtime, a valid assumption for networking.

5.1.3. Progress versus D3.5

Since the delivery of D3.5, we are working on the integration of our WASM module integrity solution (i.e., based on our modified WASM runtime) on iMEC FLOCKY/PROVE suite for secure orchestration of workload is underway.

5.2. WASM confidentiality

5.2.1. Introduction

WebAssembly (WASM) is increasingly adopted as a portable execution layer in cloud-edge architectures, where compiled .wasm modules circulate as binary artifacts between continuous integration pipelines, image registries, and execution nodes. In current practice, these modules transit in cleartext, exposing proprietary algorithms, any hard-coded secrets, and internal data structures to any actor with read access to the registry, a mirror, or the network transfer. Existing approaches to Wasm security either (i) rely on a hardware trusted execution environment (Intel SGX, ARM TrustZone) to provide runtime confidentiality, restricting deployment to platforms

equipped with such extensions; or (ii) target computational verification rather than artifact confidentiality, as in zero-knowledge Wasm emulators; or (iii) provide provenance signatures without any confidentiality guarantee. This work introduces a hardware-agnostic at-rest confidentiality scheme for portable WebAssembly modules, instantiated as a modification of the Wasmtime runtime. The proposed scheme combines AES-256-GCM authenticated encryption with a self-describing binary envelope (WASMENC1 | version | nonce | ciphertext + tag) and integrates decryption transparently into the runtime's unique loading path, such that the GCM authentication tag is verified prior to compilation and that unmodified .wasm modules continue to load unchanged. The contribution comprises (a) the definition of a portable encrypted format, suitable for storage in any OCI-compatible registry, (b) its reference implementation as a Wasmtime fork together with a CLI subcommand wasmtime encrypt, and (c) a literature analysis showing, to the best of the authors' knowledge, the absence of any equivalent scheme in open-source Wasm runtimes. The approach preserves full backward compatibility with cleartext modules and introduces no hardware dependency, making it deployable across heterogeneous edge fleets.

5.2.2. Problem statement

By its very nature, WebAssembly (Wasm) is per-se a portable execution layer in cloud-edge continuum architectures. Compiled .wasm modules circulate as binary artifacts between the continuous integration pipeline, image registries, and execution nodes. These artifacts are distributed in cleartext, exposing intellectual property (algorithms, internal data structures, hard-coded secrets) to inspection, copying, and adversarial static analysis whenever an unauthorized actor obtains access to the registry, to a mirror, or intercepts the network transfer.

This work addresses at-rest confidentiality of the Wasm module (registry, disk, mirror) and confidentiality in transit (network transfers), while preserving backward compatibility with unencrypted .wasm modules and remaining hardware-agnostic, that is, independent of any hardware trusted execution environment (Intel SGX, Intel TDX, AMD SEV-SNP, ARM TrustZone). This constraint enables deployment on heterogeneous *edge* fleets where such extensions are not universally available.

The targeted property is deliberately scoped to the at-rest confidentiality of a portable binary; it is distinct from the related properties of execution integrity, computational verification, and platform attestation, which fall under separate work. Our work can be analyzed in front of WASM confidentiality measures as detailed in 2.4.5.

5.2.3. Solution

The proposed approach integrates, within the unique loading path of the Wasmtime runtime, a symmetric authenticated encryption primitive. This integration differs from an external pre-processing step in that it guarantees cryptographic verification of module integrity *prior to* any compilation, and preserves compatibility with unencrypted modules without modifying the standard loading path.

Cryptographic choice. The scheme is based on AES-256-GCM, an AEAD (*Authenticated Encryption with Associated Data*) mode standardized by NIST SP 800-38D, providing both confidentiality and integrity of the payload in a single atomic call. The symmetric key, of 32 bytes, is distributed out-of-band between the encrypting party (continuous integration pipeline) and the decrypting party (execution node). Key resolution follows a deterministic hierarchy: CLI argument `--key` (hexadecimal); CLI argument `--key-file` (path); environment variable `WASMTIME_WASM_KEY`; environment variable `WASMTIME_WASM_KEY_FILE`. A random 12-byte nonce, generated per module via the system entropy source (`OsRng`), satisfies the uniqueness requirement of the GCM mode. The 16-byte authentication tag, appended to the ciphertext by the `aes-gcm 0.10.3` implementation, enables detection of any alteration.

Binary format. The encrypted envelope follows a self-describing format carrying a *magic number*, a version byte, and the cryptographic material:

WASMENC1(8) | version=1(1) | nonce(12) | ciphertext + tag(16)

The format is auto-detectable both by file extension (`*.wasm.enc`) and by inspection of the binary preamble, which makes the mechanism robust to mislabeling and compatible with transports without a file name (standard input, streams). Transparent decryption is performed in `RunCommon::load_module` (`common.rs`), without modifying the compilation pipeline or the public API.

User interface. A `wasmtime encrypt <module.wasm>` subcommand is added to the CLI. An end-to-end script `scripts/wasm-encrypt-smoke.sh` exercises the full cycle of key generation, compilation, encryption, and execution.

5.2.4. Implementation

The implementation is described through two complementary figures. Figure 23 presents the operational end-to-end flow, from the source artifact in the continuous integration pipeline to Figure 23 shows the encryption (i.e., at BUILD time) and decryption (i.e., at RUNTIME) flow of a WebAssembly module between the continuous integration pipeline and the execution node. The symmetric key *K*, represented by the central box, is shared out-of-band between the two

contexts. Verification of the GCM tag precedes any compilation, which rejects at runtime entry any corruption or substitution of the artifact.

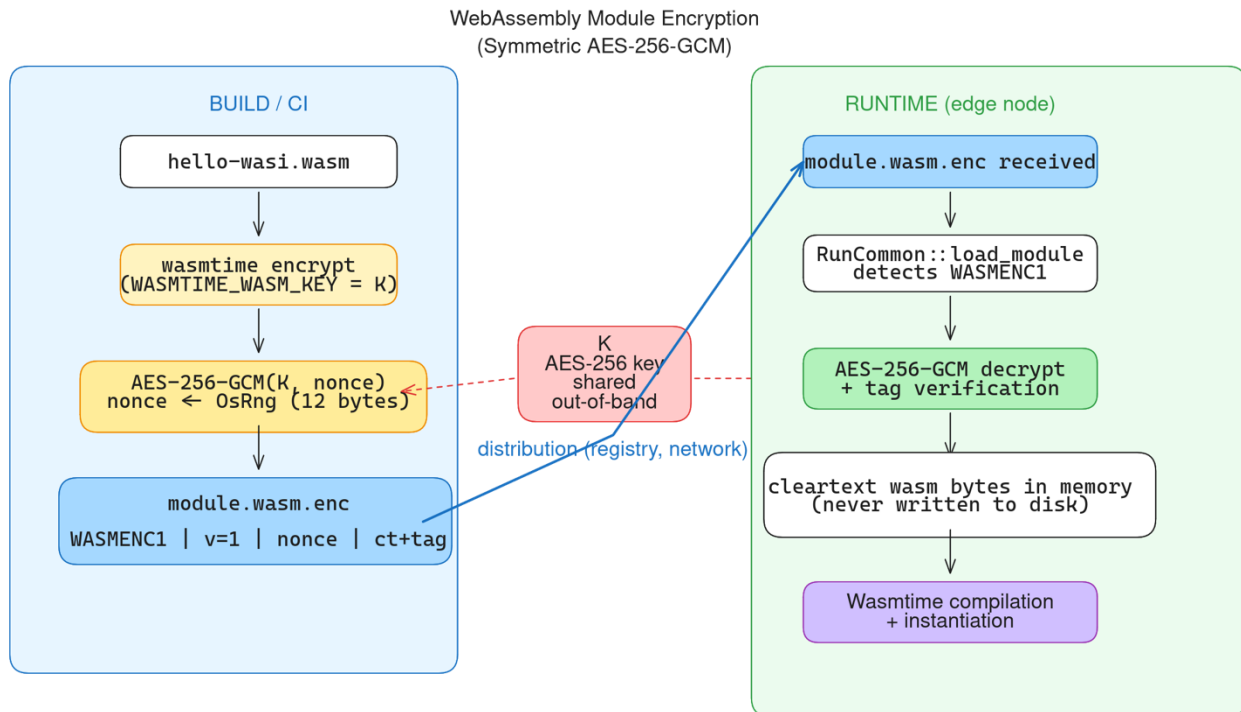


Figure 23. General workflow for module encryption (Build and Runtime)

Figure 24 shows the binary structure of the `.wasm.enc` envelope. The format is minimal so as to maximize portability and auditability: an 8-byte magic number enables detection without a file extension; the version byte allows future format evolution without breaking compatibility; the nonce and tag follow the standard recommendations of the GCM mode.

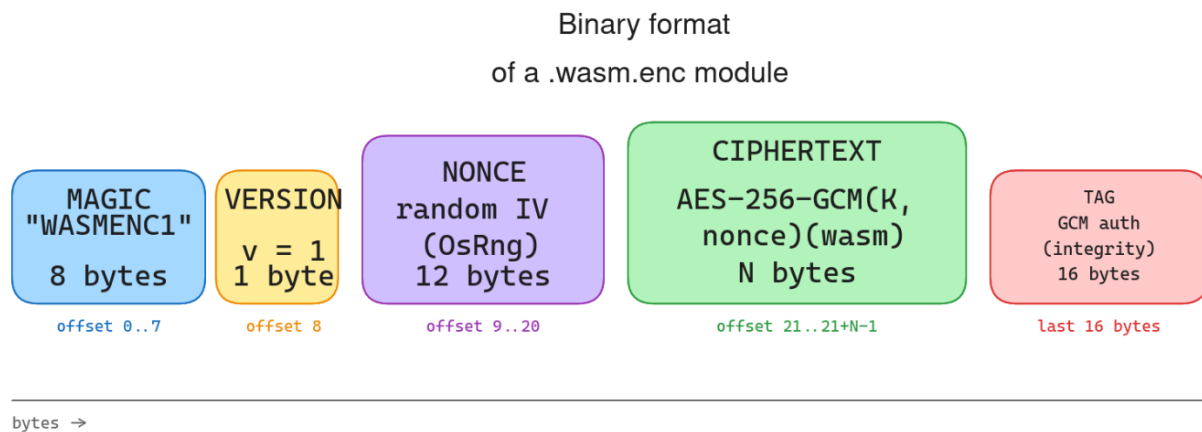


Figure 24. Binary format for an encrypted WASM module

5.2.5. Validation Methodology

5.2.5.1. Approach

Validation is organized along three complementary axes. The first is the measurement of the temporal overhead introduced by the decryption step inserted in the loading path: it quantifies the time elapsed between the reading of the encrypted module from disk and the availability of cleartext bytes to the runtime's compiler, in order to verify that this step remains compatible with the operational constraints of an *edge* node. The second is a functional smoke test that exercises the full cycle — key generation, compilation, encryption, persistence, loading, and execution — on a reference payload, and confirms that the decrypted module produces the same execution trace as the source module. The third consists of negative tests targeting the error paths of the loader (missing key, malformed key, alteration of the ciphertext or of the tag, truncated header, unknown version), which establish that any unauthenticated data is rejected by the runtime before invocation of the compiler.

5.2.5.2. Testbed description

Linux x86_64 platform without active TEE extensions; reference payload `hello-wasi` (Rust → `wasm32-wasip1`, `--release` build); end-to-end smoke test `scripts/wasm-encrypt-smoke.sh`; format inspection script `scripts/wasm-encrypt-check.sh`; negative tests covering missing key, malformed key, and alteration of the ciphertext or of the tag.

5.2.5.3. Measures with respect to KPIs

The results presented in Table 30 evaluate the overhead introduced by the payload decryption and remote attestation mechanisms across a representative set of WebAssembly applications. Measurements were performed on different workload types, ranging from simple programs (*hello-wasm*) to computationally intensive benchmarks (*prime1e6*), as well as a HTTP server application. The objective is to verify compliance with the project performance indicators, namely a payload decryption time below 3 seconds (KPI 1.3.2) and a remote attestation cycle time below 1 second (KPI 1.3.1) for x86 payloads.

The measured payload decryption time ranges from 261 μ s to 530 μ s depending on the benchmark. Even in the worst-case scenario (*http_server*), the observed decryption time remains below 1 ms, which is far beneath the threshold defined by KPI 1.3.2 (3 s). The computational benchmarks exhibit highly consistent decryption times, between 309 μ s and 315 μ s, indicating that the decryption overhead is primarily determined by the binary size and remains negligible compared to the overall application execution time.

Table 30. WASM module decryption time measurements

Benchmark	Decryption Time (us)	Attestation Cycle Time (ms)
hello-wasm	261	159
sha256-bench	381	162
loop_worker	272	160
prime1e3	311	160
prime1e4	310	159
prime1e5	309	159
prime1e6	315	160
http_server	530	161

Regarding remote attestation, all benchmarks exhibit a constant attestation cycle time of around 160 ms. This value corresponds to only 16% of the maximum allowable threshold specified by KPI 1.3.1 (1 s), providing a comfortable margin for deployment scenarios.

Table 31 following results evaluate KPI 1.3.3, which defines the acceptable performance degradation induced by runtime verification and performance monitoring mechanisms for x86 payloads (threshold < 10%). Measurements were performed on the same machine using 100 executions per configuration, and report mean execution times for three setups: original wasmtime, wasmtime with local integrity checks, and wasmtime with both local integrity checks and remote attestation. The reported execution times are obtained from internal measurements within the payload itself.

Table 31. WASM confidentiality performance penalty measurements

Benchmark	Vanilla Mean Time (s)	Integrity Mean Time (s)	Integrity + Attestation Mean Time (s)
prime1e3	0.00027484	0.00027483	0.00027488
prime1e4	0.00855530	0.00855166	0.00852244
prime1e5	0.27694117	0.27692832	0.27671894
prime1e6	9.14378697	9.14059956	9.14680852
prime1e7	304.849242	304.541354	304.839320

Across all *prime* benchmarks, execution times remain highly stable between configurations, with only marginal variations observed. These differences are not systematic and fall within the expected measurement noise and inter-run variability of the execution environment. Deviations remain minimal and no consistent slowdown is observed when enabling integrity verification or attestation.

This observation is further confirmed by the results obtained on the HTTP server workload, executed on native Linux (ELF 64) as well as under Wasmtime in both vanilla and security-enhanced configurations (integrity checking and remote attestation). The benchmark was conducted on the same machine using 30,000 mixed GET and POST requests to the server, ensuring a consistent and comparable execution environment across all setups. Table 31 shows the impact on the confidentiality-protected WASM HTTP server against non protected variants.

Table 32. Confidentiality-protected WASM HTTP server penalty assessment

http_server payload	Mean Response Time (ms)	Throughput (req/s)
Native Linux (ELF 64)	0.810	1229.13
Vanilla Wasmtime	0.811	1228.39
Wasmtime Integrity Check	0.809	1231.44

The measured mean response times and throughput are virtually identical across all configurations. The observed differences are fully contained within the confidence interval ($IC_{90} \pm 13.5$ req/s), indicating that they are not statistically significant.

Taken together, the results from both the prime computational benchmarks and the HTTP server workload consistently demonstrate that the introduced runtime verification mechanisms do not produce measurable performance overhead. In both cases, all observed variations remain within statistical noise and confidence intervals, with no systematic slowdown detected. Consequently, the performance degradation remains well below the 10% threshold defined by KPI 1.3.3, which can therefore be considered fully satisfied.

5.2.6. Progress Beyond the State of the Art

This work delivers a contribution that none of the approaches surveyed in our state-of-the-art analysis provides: a portable, self-describing, at-rest encrypted WebAssembly module format, transparently decrypted in the unique loading path of an open-source runtime, with no hardware dependency.

With respect to TEE-bound runtimes such as TwiNE [84] and WaTZ [85], the proposed scheme achieves at-rest confidentiality on commodity x86_64 hardware, without requiring Intel SGX or ARM TrustZone, and closes the gap upstream of the enclave (registry, network), where TEE primitives do not protect the artifact.

With respect to hardware-agnostic verification approaches such as ZKWASM [82] and WaVe [83], it adds artifact confidentiality at rest — a property that neither zero-knowledge proofs nor sandbox isolation deliver — while remaining orthogonal and combinable with both. With respect to provenance-only signing (Sigstore, Notary v2, in-toto), it adds confidentiality on top of integrity and provenance; a `.wasm.enc` envelope can itself be signed, superseding signing alone for any threat model requiring intellectual property protection.

With respect to Confidential Containers (Kata+SEV-SNP/TDX, CoCo), it operates at module granularity on commodity hardware, and is therefore deployable in contexts where SEV-SNP or TDX are not available. With respect to mainstream Wasm runtimes (Wasmtime, WasmEdge, WAMR), it introduces an official at-rest encryption primitive while preserving full backward compatibility: unmodified `.wasm` modules continue to load, compile, and execute through the standard path.

Finally, with respect to the bare use of AES-256-GCM as a library primitive, the contribution defines a portable envelope format, a deterministic key-resolution hierarchy, a magic-based auto-detection mechanism, and a transparent runtime integration point, turning a code-level building block into a deployment-grade scheme.

The distinctive properties of the contribution are:

- AEAD integration in the loading path. The GCM tag is verified before any compilation; corrupted or substituted modules are rejected at runtime entry.
- Self-describing format. Magic number plus version byte enable native coexistence of cleartext and encrypted modules, with extension-independent detection.
- Independence of published artifacts. A random 12-byte nonce per module eliminates correlation attacks across re-encryptions.
- Backward compatibility. The cleartext `.wasm` loading path is unchanged; migration to the encrypted format is strictly non-disruptive.

5.2.7. Conclusion

This work contributes a portable, self-describing, hardware-agnostic at-rest encrypted WebAssembly module format, whose transparent integration into the loading path of the Wasmtime runtime preserves compatibility with existing cleartext modules. No prior contribution combines these properties on an open-source Wasm format. The design opens several reinforcement directions (e.g., precompiled artifacts, propagation to other binaries) without challenging the underlying cryptographic primitive. The described technique is perfectly

aligned with NATWORK performance to security principle as security is attained at very low performance overhead and meeting the associated KPI 1.3.2.

5.2.8. Evolutions from D3.5

This section covers WASM confidentiality, recently developed and which was not started hence discussed in D3.5 version.

5.3. WASM availability

5.3.1. Domain and problem statement

Ensuring that a program continues to execute correctly and at an acceptable speed is a challenge for long-running or indefinitely executing WebAssembly applications. Instead of relying only on global execution time measurements, this work proposes a monitoring approach for Wasmtime based on static binary rewriting with **e9patch**.

The solution inserts probes at selected locations inside the Wasmtime binary in order to continuously observe runtime behavior, verify in real time that the program is still executing correctly, and detect performance degradation during execution. By instrumenting specific runtime functions, the approach also provides fine-grained and precise execution measurements, which are more suitable for variable-duration or non-terminating workloads than traditional end-to-end timing methods.

5.3.2. Solution

Our solution is novel compared to all previous state of the 2.4.6, notably through the modification of the runtime it involves.

5.3.2.1. Concept

As shown in Figure 25, Wasmtime is operating in two modes: compiled execution (JIT/AOT) and interpreted execution. The monitoring strategy depends strongly on these modes, as they determine where instrumentation can be applied.

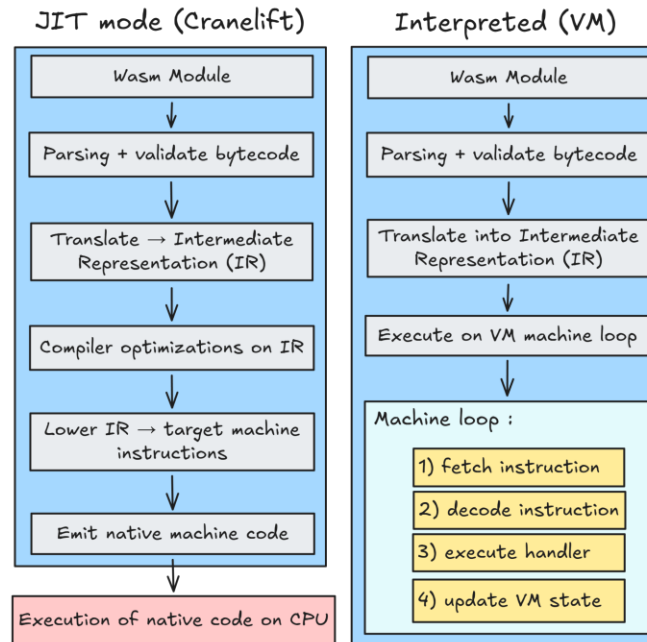


Figure 25. WASMTIME two execution modes

5.3.2.2. Compiled mode (JIT / AOT)

By default, a Wasm module follows a standard compilation chain. Once compiled, execution runs directly on the CPU, with no continuous involvement of Wasmtime except at function boundaries. This significantly limits fine-grained instrumentation: there is no persistent runtime execution loop that can be used for continuous monitoring, making it difficult to observe long-running or infinite loops. Two alternatives exist but are constrained:

Injecting probes at the IR or generated code level, but granularity is workload-dependent and requires manual tuning, reducing generality.

Using internal mechanisms such as ***wasmtime_internal_unwinder***, which provide stack visibility but introduce high overhead and additional complexity.

5.3.2.3. Interpreted mode (Pulley VM)

The interpreted mode (forced via **-target pulley32/64**) executes the module inside a VM loop where each instruction is fetched, decoded, and executed within the runtime. Unlike compiled mode, every execution step remains observable, enabling continuous instrumentation at the instruction dispatch or handler level. Although this mode is slower than compiled execution (JIT/AOT), it ensures universal compatibility for Wasmtime across all platforms and allows for the instrumentation of static processing functions within the VM via static binary rewriting probes. Development was therefore continued using this execution model.

5.3.3. First implementation and results

To achieve the objective, a patched version of Wasmtime was instrumented using e9patch. The instrumentation added probes to internal functions in order to monitor and count their invocations at runtime. The runtime was executed exclusively in interpreted mode.

A WebAssembly module was generated from a C program implementing a prime number computation algorithm, and several executions were performed with increasing workloads corresponding to different values of x , ranging from 10 to 10 0000. This experiment focused on the function `pulley_interpreter::decode::decode_one_extended`, which is part of Wasmtime's interpreter execution path. To monitor its activity, we patched the entry point of the function by inserting a trampoline, as illustrated in the Figure 26

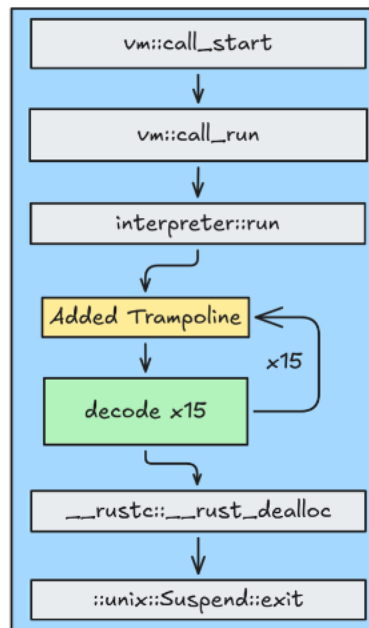


Figure 26. Probe insertion into WASMTIME runtime decoding loop

The trampoline itself performs only a minimal operation: printing a message to the console to trace execution events during interpretation. Since the trampoline is executed on every entry into the decode routine, its invocation count (15 times in this example) directly reflects the number of interpreted instructions processed by Wasmtime.

The collected results given in Table 33 show a consistent and stable growth pattern. The number of interpreter decode calls increases steadily as the workload grows, while remaining relatively well-behaved across multiple orders of magnitude. In particular, the Calls / x ratio grows slowly, from 1.5 for $x=10$ to approximately 6.5 for $x=100000$. This indicates that interpreter overhead

scales progressively as shown in Figure 27. Number of calls by Number of delivered Primes by Prime (x)ressively rather than explosively.

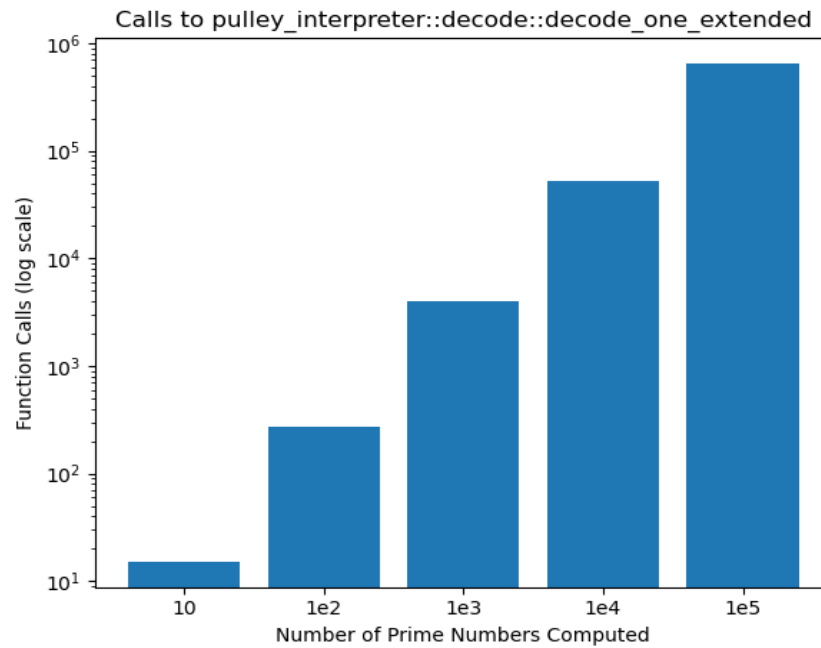


Figure 27. Number of function calls versus Number of primes in Prime(x)

Table 33. Function calls versus x (for PRIME(x) wasm module

x	Function calls (decode function)	Calls / x	Multiplier (x+1/x)
10	15	1.5	N/A
100	271	2.71	18.1
1000	3960	3.96	14.6
10000	52365	5.24	13.2
100000	649855	6.5	12.4

Another notable observation is that the growth multiplier decreases as the workload increases. While the workload itself is multiplied by 10 between each measurement point, the number of decode calls grows by factors gradually decreasing from 18.1 to 12.4. This suggests that the interpreter behavior remains stable and does not exhibit pathological scaling characteristics.

Overall, the measurements indicate that the instrumentation process behaves correctly and that the patched Wasmtime interpreter can successfully expose meaningful execution metrics through function-level probing. The observed scaling behavior is coherent with the structure of

the tested algorithm and confirms that monitoring this function provides a relevant indicator of runtime activity during WebAssembly execution.

5.3.4. Consistency validation across payloads

In a previous study, instrumentation was applied at the level of a decode function, which initially provided a useful proxy for execution intensity, particularly for a specific benchmark (the prime calculator). In that setting, a near-proportional relationship was observed between the number of function invocations and the number of executed instructions. However, further analysis revealed that this correlation does not generalize across different WebAssembly binaries and is in fact largely specific to that particular workload.

To address this limitation, we increased the granularity of observation by moving from function-level instrumentation to basic block-level instrumentation. A basic block is defined as a straight-line sequence of instructions with a single entry point and a single exit point, with no internal branches. This finer-grained view allows for a more consistent approximation of runtime activity across diverse binaries. In this context, we identified a more appropriate instrumentation point inside the Wasmtime execution loop, specifically within `pulley_interpreter::match_loop::run`. This location corresponds to a frequently executed interpreter dispatch path that is common across all WebAssembly workloads, making it a more generic candidate for runtime monitoring.

The objective of the following experiment is therefore to evaluate whether the execution count of this patched basic block can serve as a reliable and workload-agnostic proxy for the amount of work performed by a WebAssembly program.

The evaluated workload consists of a tight loop executing a linear congruential computation for a configurable number of iterations (x). This workload was chosen because its execution cost scales linearly with the iteration count, providing a straightforward reference for proportionality analysis.

Table 34. WASM monitoring validation runs

Iteration s	Block 0x250d631	Block 0x250b63f	Ratio (block 2 / block 1)
100	100	1182	11,82
1000	1000	11082	11,082
10000	10000	110082	11,0082
100000	100000	1100081	11,00081

Table 34 above reports the number of executions observed for two instrumented blocks located inside `pulley_interpreter::match_loop::run`, the main interpreter loop responsible for executing

WebAssembly instructions. Block 0x250b63f corresponds to the most frequently executed block within the interpreter, while block 0x250d631 is executed less often. For a given binary and input, the execution counts of both blocks remain constant across repeated runs, providing a stable basis for proportionality analysis.

The results demonstrate a clear linear relationship between the workload iteration count and the execution counts of both instrumented blocks. Block 0x250d631 is executed exactly once per workload iteration across all tested configurations, making it a highly representative indicator of application activity. Although block 0x250b63f is the most frequently executed block in the interpreter, its execution count remains consistently proportional to that of 0x250d631, with the ratio converging toward a stable value of approximately 11 as the workload size increases. This stable proportionality confirms that block 0x250d631 accurately reflects the behavior of the hottest execution path while requiring fewer probe activations, making it an effective and low-overhead instrumentation target.

5.3.5. Challenging our proof of concept. Tests.

This section aims to validate that the patched basic block at address 0x250d631 inside the Wasmtime run function is effectively and consistently executed as part of the actual execution flow of compiled WebAssembly binaries. In other words, the objective is to demonstrate that this instrumentation point is causally linked to payload execution rather than being an isolated or spurious monitoring artifact.

As a first validation step, we verified that the patched block is executed regardless of the specific binary being run inside Wasmtime. This behavior is expected, as execution occurs within the main run loop of the Wasmtime runtime, ensuring that any dispatched WebAssembly payload necessarily triggers the instrumented path. This confirms that the monitoring hook is correctly placed within the critical execution path of the runtime. To further characterize this relationship, we evaluated the sensitivity of the monitoring signal with respect to execution performance variations. Using the prime1e3 payload as a baseline workload, we introduced controlled CPU contention through stress-ng workers in order to progressively degrade execution performance. The goal was to observe whether variations in the execution time of the payload would be reflected in the frequency of execution of the patched block.

The resulting measurements, illustrated in Figure 28, compare the normalized execution time of prime1e3 (measured internally within the payload) against the normalized frequency of execution of the patched block. The two curves exhibit a strong inverse relationship, indicating that as execution time increases under CPU stress, the block execution frequency decreases accordingly due to reduced instruction throughput.

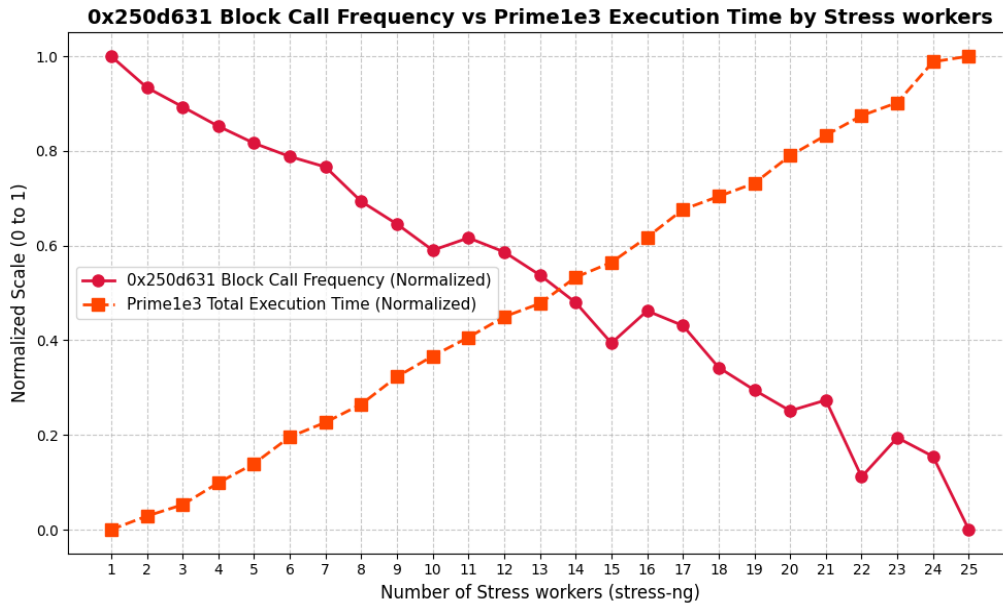


Figure 28. Patched block frequency vs. Prime1e3 execution time

This relationship is further confirmed by statistical analysis, which shows a very strong negative correlation between the two signals: Pearson Correlation Coefficient = -0.9894 with a p-value of 8.433713e-21. This provides strong evidence that the monitoring signal is tightly coupled with actual runtime behavior and accurately reflects execution dynamics of the underlying binary.

Figure 29 improves the previous analysis by converting the block execution frequency into the mean time between consecutive executions of the patched block.

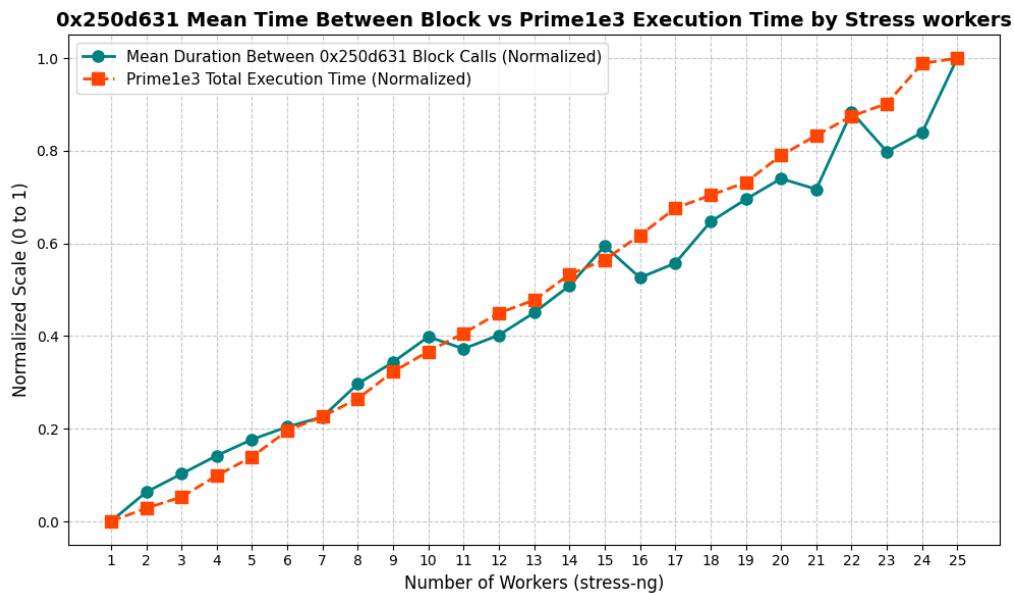


Figure 29. Mean inter-block duration vs. Prime1e3 execution time.

As CPU contention increases, the mean time between block executions closely follows the increase in payload execution time. The two curves exhibit a very similar evolution, showing that the instrumented block reflects the execution rate of the Wasmtime runtime's run function and, as a consequence, provides an indirect but reliable proxy for the execution speed of the underlying WebAssembly payload.

A larger delay between block executions indicates a slowdown of the workload, while a delay tending toward infinity (i.e., a block frequency approaching zero) would indicate that the payload is no longer making progress. Consequently, the monitoring mechanism can be used not only to estimate performance degradation but also to infer the availability and liveness of the running application.

5.3.6. Measured Penalty

To evaluate the runtime overhead introduced by the static patch-based instrumentation, we measured execution penalties across multiple benchmarks, including the same prime calculator, the linear workload described earlier and a sorting workload (*sort*) that applies many sorting algorithms to arrays of size x .

Two instrumentation configurations were evaluated:

- (1) a patched version of block 0x250d631 where the inserted probe redirects execution through a trampoline to extract a timestamp via shared memory
- (2) a reduced-frequency of the same instrumentation using modulo-8 sampling (one probe activation every 8 executions).

For each benchmark, the execution time of both instrumented configurations was compared against a reference execution performed with the original, unmodified Wasmtime runtime. All values reported in the table below correspond to averages computed over 100 independent executions.

Table 35. Measured penalty induced by WASM monitoring

Benchmark	Patch (1) penalty	Patch (2) penalty
prime1e3	+4.19%	+2.01%
prime1e4	+0.81%	+0.21%
prime1e5	+5.78%	+1.15%
prime1e6	+5.74%	+0.30%
simple_workload_10	+0.10%	—
simple_workload_1e2	+1.18%	—
simple_workload_1e3	+0.33%	—
simple_workload_1e4	+0.72%	—

Benchmark	Patch (1) penalty	Patch (2) penalty
simple_workload_1e5	+0.39%	–
simple_workload_1e6	+0.41%	–
sort_1e2	+0.74%	+0.43%
sort_1e3	+3.28%	+0.71%

Overall, the results indicated in Table 35 show that the instrumentation overhead remains low and generally stable across workloads, all measured configurations remain well within the Network KPI 1.3.3 constraint, which specifies a combined penalty budget of no more than 10%. However, significant overhead is observed in compute-intensive workloads such as the prime calculator, where the fully instrumented configuration introduces up to +5.78% overhead. To tackle this, a mod-8 sampling strategy can be applied, reducing probe activations to one out of every eight executions. This approach significantly lowers the overhead, for example from +5.74% to +0.30% for *prime1e6*.

5.3.7. Beyond the state of the art

This approach extends existing WebAssembly monitoring techniques by using static binary rewriting of Wasmtime via e9patch, enabling lightweight and deterministic instrumentation directly inside the runtime. Compared to prior work based on dynamic tracing or coarse benchmarking, it provides more stable and fine-grained observation without modifying code the WebAssembly module or relying on heavy runtime introspection.

Initial results confirm that the instrumentation produces meaningful execution signals correlated with runtime activity. However, several limitations remain open: potential noise introduced by variability at the IR or instruction-dispatch level, and the difficulty of extending the approach to fully support compiled execution modes, which remains a key direction for future work.

5.3.8. Progress from D3.5

This work was not inserted in D3.5 and has been developed and continued since then.

5.3.9. Conclusion

This work extends existing WebAssembly monitoring techniques by using static binary rewriting of Wasmtime via e9patch, enabling lightweight and deterministic instrumentation directly inside the runtime. Compared to prior approaches based on dynamic tracing or coarse-grained benchmarking, it provides more stable and fine-grained observation without modifying the WebAssembly module or relying on heavy runtime introspection.

Beyond observability, the approach targets long-running or effectively infinite-lifetime workloads where traditional execution time metrics are insufficient. It enables continuous assessment of payload activity and performance through lightweight runtime-derived signals.

Results confirm that the instrumentation produces signals strongly correlated with runtime activity and workload intensity, enabling both execution monitoring and performance tracking. These results remain well within the penalty bounds defined by the NATWORK KPI 1.3.3.

A remaining limitation is the need to modify the Wasmtime runtime itself. However, this does not constitute a significant constraint in the considered context, as the runtime is already instrumented for integrity checking, and such modifications are typically acceptable in controlled deployment environments such as telecommunications infrastructures.

6. Conclusion

D3.6 “Pre-Deployment Security per Construction Measures.r2” completes the work conducted on PDSCM initiated in D3.5 “Pre-Deployment Security per Construction Measures.r1”, either by developing further techniques or alternatively by bringing novel developments as shown in Table 36.

Table 36. Summary of D3.6 outcomes versus D3.5

Payload type	PDSCM	D3.5	D3.6
Containers	Hardened orchestration by K8s helm chart anomaly detection	Present	Improved
Containerized microservices	RBAC prone microservice orchestration	Present	Improved
Containerized xApp	Remote attestation and runtime integrity	Present	Novel globalized K8s helm chart-based orchestration
Containerized xApp	JSON checker and FALCO-enabled syscalls checker	Not present	Novel onboarding and runtime security
Container	Sidecar for CPU bound contention probe	Not present	Novel availability improvement offering novel CPU contention visibility
x86 native	D-MUTRA based remote attestation and runtime integrity verification	Present	Present with Anomalous Traffic Detection xApp
x86 native	LLVM enabled method for easier monitoring probe insertion	Not present	Novel technique applied on MMT probe
X86 native	Integrity and confidentiality by TEE (ie, AMD’s SEV SNP)	Present	Significantly extended with TEEMig hot migration architecture
WASM	Remote attestation and runtime integrity	Present	Improved by integration for full stack remote attestation
WASM	Confidentiality	Not present	Novel technique preventing static analysis through encryption
WASM	Availability check by performance monitoring	Not present	Novel technique instrumenting WASM runtime

As shown in the table, a very significant amount of research has been carried out during the nine months since the release of D3.5, resulting in promising techniques that frequently advance beyond the state of the art (e.g., enforcement mechanisms against WASM CIA attacks, enhanced visibility into CPU contention in containerized environments). These developments collectively address the three targeted workload categories. The contributions presented in D3.6 were either initiated or completed during the nine months preceding the delivery of D3.6 (M30).

Combined with the outcomes already delivered at M24 and reported in D3.5, these results fulfil the objectives of NATWORK Task 3.4. The three core security properties—Confidentiality, Integrity, and Availability—have been significantly strengthened across containerized, native x86, and WASM workloads. The proposed techniques have also been demonstrated using partners' representative workloads, including the MMT probe and the xAPP anomalous traffic detection application.

For WASM workload hardening, all proposed techniques rely on modifications to the WASM runtime, a practical approach for networking environments. When integrated into the IMEC secure orchestration framework, the modified runtime itself can be remotely attested before deployment, enabling end-to-end security for the complete WASM software stack, including both the runtime and the deployed modules. The work also significantly advances the security of O-RAN deployments, for both containerized and native xAPP workloads. Several complementary protection mechanisms have been developed, including JSON validation, Falco-based system call monitoring, remote attestation, and runtime integrity verification.

Most of the proposed techniques have already been thoroughly validated against NATWORK's KPIs. A limited number of contributions still require additional experimental measurements, which are planned to complete their evaluation.

All developed techniques are consistent with NATWORK's reconciliation principles, preserving workload portability while minimizing performance overhead. For example, TEEMig enables the hot migration of workloads between Trusted Execution Environments (TEEs) without introducing significant execution penalties. The proposed PDSCM mechanisms also introduce new capabilities for the automated deployment of security services by leveraging Kubernetes orchestration. In particular, a Kubernetes-based approach enables the fully automated provisioning of remote attestation for containerized workloads, allowing D-MUTRA to achieve a highly scalable architecture in which workload integrity is automatically protected throughout both onboarding and runtime execution.

Finally, a novel technique is introduced to improve observability in containerized environments by providing unprecedented visibility into CPU contention. By estimating the number of

competing CPU-bound co-located workloads, this approach offers valuable operational insights for stakeholders deploying applications in managed container platforms.

References

- [1] Intel. (n.d.). Confidential containers made easy. Retrieved September 21, 2025, from <https://www.intel.com/content/www/us/en/developer/articles/technical/confidential-containers-made-easy.html>
- [2] RedHat. (n.d.). Zero trust starts here: Validated patterns for confidential container deployment. Retrieved September 21, 2025, from <https://www.redhat.com/en/blog/validated-patterns-confidential-container-deployment>
- [3] Falcao, E., Silva, F., Pamplona, C., Melo, A., Asadujjaman, A. S. M., & Brito, A. (2025). Confidential Kubernetes deployment models: Architecture, security, and performance trade-offs. Applied Sciences, 15(18), 10160. <https://doi.org/10.3390/app151810160>
- [4] Intel. (n.d.). TDX documentation. Retrieved September 21, 2025, from <https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html>
- [5] AMD. (n.d.). SEV-SNP documentation. Retrieved July 6, 2026, from <https://www.amd.com/fr/developer/sev.html>
- [6] Ye, M. (2024). Enabling Performant and Secure EDA as a Service in Public Clouds Using Confidential Containers. Retrieved September 21, 2025, from <https://arxiv.org/pdf/2407.06040v1>
- [7] Confidential Containers. (2024). Introduction to Confidential Containers (CoCo). Retrieved September 21, 2025, from <https://confidentialcontainers.org/blog/2024/02/16/introduction-to-confidential-containers-coco/>
- [8] Pronteff. (n.d.). Openshift confidential containers now on Microsoft Azure. Retrieved September 21, 2025, from <https://pronteff.com/openshift-confidential-containers-now-on-microsoft-azure/>
- [9] Edgeless Systems. (n.d.). Marblerun. Retrieved July 6, 2026, from <https://www.edgeless.systems/products/marblerun>
- [10] Johnson, M. A., et al. (2024). Confidential Container Groups: Implementing confidential computing on Azure container instances. ACM Queue 22(2). <https://spawn-queue.acm.org/doi/pdf/10.1145/3664293>
- [11] Sanctuary. (n.d.). Trusted container extensions for container-based confidential computing. Retrieved September 21, 2025, from <https://sanctuary.dev/en/blog/container-based-confidential-computing/>
- [12] Brasser, F. (2022). Trusted Container Extensions for Container-based Confidential Computing. <https://arxiv.org/pdf/2205.05747>

- [13] Budigiri, G., Baumann, C., Mühlberg, J. T., Truyen, E., & Joosen, W. (2021). Network policies in Kubernetes: Performance evaluation and security analysis. In 2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit).
- [14] Koukis, G., Skaperas, S., Kapetanidou, I. A., Mamatas, L., & Tsaoussidis, V. (2024). Performance evaluation of Kubernetes networking approaches across constrained edge environments. In 2024 IEEE Symposium on Computers and Communications (ISCC).
- [15] Shamim, M. S. I., Bhuiyan, F. A., & Rahman, A. (2020). Xi commandments of Kubernetes security: A systematization of knowledge related to Kubernetes security practices. In 2020 IEEE Secure Development (SecDev) (pp. 58–64).
- [16] Carrión, C. (2022). Kubernetes scheduling: Taxonomy, ongoing issues and challenges. ACM Computing Surveys, 55, 1–37.
- [17] Rahman, A., Shamim, S. I., Bose, D. B., & Pandita, R. (2023). Security misconfigurations in open source Kubernetes manifests: An empirical study. ACM Transactions on Software Engineering and Methodology, 32, 1–36.
- [18] Kampa, S. (2024). Navigating the landscape of Kubernetes security threats and challenges. Journal of Knowledge Learning and Science Technology, 3, 274–281.
- [19] O-RAN Alliance. (n.d.). Near-RT RIC architecture 7.0. Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [20] O-RAN Alliance. (n.d.). Security requirements and controls specifications 12.0. Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [21] O-RAN Alliance. (n.d.). Study on security for Service Management and Orchestration (SMO) 6.0. Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [22] O-RAN Alliance. (n.d.). Study on security for O-Cloud 7.0. Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [23] O-RAN Alliance. (n.d.). Study on Zero Trust Architecture for Secure O-RAN. Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [24] O-RAN Software Community. (n.d.). O-RAN xAPP SDK. Retrieved July 6, 2026, from <https://lf-o-ran-sc.atlassian.net/wiki/spaces/ORANSdk/pages/14516830/xAppFramework>
- [25] O-RAN Alliance. (n.d.). Study on security for Near Real Time RIC and xApps 5.0. Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [26] O-RAN.WG1.TS.OAD-R005-v16.00 – Use Cases and Overall Architecture (2026). Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [27] O-RAN.WG11.TR.CSM-Security-R004-v01.00 – Study on Continuous Security Monitoring (2026). Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [28] O-RAN.WG11.TR.AppLCM-Security-R004-v04.00 – Study on Security for Application Lifecycle Management (2025). Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>

- [29] O-RAN.WG11.TR.CSM-Security-R005-v02.00 – Study on Continuous Security Monitoring (2026). Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [30] O-RAN.WG11.TR.O-CLOUD-Security.0-R005-v09.00 – O-RAN Work Group 11 (Security Work Group). Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [31] O-RAN.WG11.TS.SRCS.0-R005-v14.00 – O-RAN Work Group 11 (Security Work Group). Retrieved July 6, 2026, from <https://specifications.o-ran.org/specifications>
- [32] Llorido-Botran, T., Huerta, S., Tomás, L., Tordsson, J., & Sanz, B. (2017). An unsupervised approach to online noisy-neighbor detection in cloud data centers. *Expert Systems with Applications*, 89, 188–204. <https://doi.org/10.1016/j.eswa.2017.07.038>
- [33] Koh, Y., Knauerhase, R., Brett, P., Bowman, M., Wen, Z., & Pu, C. (2007). An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software* (pp. 200–209). IEEE. <https://doi.org/10.1109/ISPASS.2007.363750>
- [34] Collberg, C., Thomborson, C., & Low, D. (1997). A taxonomy of obfuscating transformations (Technical Report No. 148). Department of Computer Science, University of Auckland.
- [35] Barak, B. (2016). Hopes, fears, and software obfuscation. *Communications of the ACM*, 59 (3), 88–96. <https://doi.org/10.1145/2757276>.
- [36] Xu, H., Zhou, Y., Kang, Y., & Lyu, M. R. (2017). On Secure and Usable Program Obfuscation: A Survey. *arXiv:1710.01139*.
- [37] De Sutter, B., Schrittwieser, S., Coppens, B., & Kochberger, P. (2024). Evaluation methodologies in software protection research. *ACM Computing Surveys*, 57 (4), Article 86. <https://doi.org/10.1145/3702314>.
- [38] ARXAN Technologies. (n.d.). Application security. Retrieved July 6, 2026, from <https://digital.ai/products/application-security/>
- [39] VMProtect. (n.d.). VMProtect. Retrieved July 6, 2026, from <https://vmpsoft.com/>
- [40] SOLIDSHIELD. (n.d.). SOLIDSHIELD. Retrieved July 6, 2026, from <https://www.solidshield.com/>
- [41] AppDome. (n.d.). AppDome. Retrieved July 6, 2026, from <https://www.appdome.com>
- [42] Obfuscator-ai. (n.d.). Obfuscator-ai. Retrieved July 6, 2026, from <https://pypi.org/project/obfuscator-ai>
- [43] Li, R., Zhang, C., Chai, H., Ying, L., Duan, H., & Tao, J. (2024). PowerPeeler: A precise and general dynamic deobfuscation method for PowerShell scripts. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security* (pp. 4539–4553). Association for Computing Machinery. <https://doi.org/10.1145/3658644.3670310>.
- [44] Garba, P., & Favaro, M. (2019). SATURN: Software deobfuscation framework based on LLVM. In *Proceedings of the 3rd ACM Workshop on Software Protection* (pp. 27–38). Association for Computing Machinery. <https://doi.org/10.1145/3338503.3357721>

- [45] Nagios. (n.d.). Nagios monitoring tool. Retrieved July 6, 2026, from <https://www.nagios.org/>
- [46] Zabbix. (n.d.). Zabbix monitoring tool. Retrieved July 6, 2026, from <https://www.zabbix.com/documentation/current/en/>
- [47] Prometheus. (n.d.). Prometheus and Grafana. Retrieved July 6, 2026, from <https://prometheus.io/>
- [48] Datadog. (n.d.). Datadog monitoring tool. Retrieved July 6, 2026, from <https://www.datadoghq.com/>
- [49] New Relic. (n.d.). New Relic monitoring tool. Retrieved July 6, 2026, from <https://newrelic.com/>
- [50] DESIRE-6G Project. (n.d.). Deliverable D3.3. Retrieved July 6, 2026, from <https://zenodo.org/records/17077365>
- [51] Fioraldi, A., Maier, D., Eißfeldt, H., & Heuse, M. (2020, August). AFL++: Combining incremental steps of fuzzing research. In 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association.
- [52] Lattner, C., & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, USA, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.
- [53] Chen, D., Li, D. X., & Moseley, T. (2016). AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (pp. 12–23). Association for Computing Machinery. <https://doi.org/10.1145/2854038.2854044>.
- [54] Free Software Foundation. (n.d.). gcov: A test coverage program*. In **Using the GNU Compiler Collection (GCC)*. Retrieved July 6, 2026, from <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>.
- [55] Serebryany, K. (2016). Continuous fuzzing with lib Fuzzer and Address Sanitizer. In 2016 IEEE Cybersecurity Development (SecDev) (p. 157). IEEE. <https://doi.org/10.1109/SecDev.2016.043>.
- [56] Luk, C.-K., Cohn, R. S., Muth, R., Patil, H., Klauser, A., Lowney, P. G., Wallace, S., Reddi, V. J., & Hazelwood, K. M. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (pp. 190–200). Association for Computing Machinery. <https://doi.org/10.1145/1065010.1065034>.
- [57] Bruening, D. L. (2004). Efficient, transparent, and comprehensive runtime code manipulation [Doctoral dissertation, Massachusetts Institute of Technology]. MIT DSpace. <https://dspace.mit.edu/entities/publication/b63c9921-1f04-4cd2-bb8b-75c5d2f6df3b>.

- [58] Nethercote, N., & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (pp. 89–100). Association for Computing Machinery. <https://doi.org/10.1145/1250734.1250746>.
- [59] Gregg, B. (2019). *BPF performance tools: Linux system and application observability*. Addison-Wesley.
- [60] Intel Corporation. (2026). Intel® 64 and IA-32 architectures software developer’s manual Retrieved July 6, 2026, from <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [61] Kleen, A. (2015, July 1). Adding Processor Trace support to Linux. LWN.net. <https://lwn.net/Articles/648154/>.
- [62] de Melo, A. C. (2010). The new Linux “perf” tools. Linux Kongress. <https://oldvger.kernel.org/~acme/perf/lk2010-perf-paper.pdf>.
- [63] Chen, M. Y., Accardi, A., Kiciman, E., Patterson, D. A., Fox, A., & Brewer, E. (2004). Path-based failure and evolution management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)* (pp. 309–322). USENIX Association.
- [64] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., & Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings International Conference on Dependable Systems and Networks* (pp. 595–604). IEEE. <https://doi.org/10.1109/DSN.2002.1029005>.
- [65] Page, E. S. (1954). Continuous inspection schemes. *Biometrika*, 41(1–2), 100–115. <https://doi.org/10.1093/biomet/41.1-2.100>.
- [66] Roberts, S. W. (1959). Control chart tests based on geometric moving averages. *Technometrics*, 1(3), 239–250. <https://doi.org/10.1080/00401706.1959.10489860>.
- [67] DPDK Project. *DPDK Programmer’s Guide*, ch. on Statistics. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/
- [68] OpenTelemetry Authors. (n.d.). OpenTelemetry specification. Retrieved July 6, 2026, from <https://opentelemetry.io/docs/specs/otel/>
- [69] Montimage. (n.d.). MMT toolchain documentation and publications. Retrieved July 6, 2026, from <https://www.montimage.com/>.
- [70] World Wide Web Consortium. (n.d.). WebAssembly Working Group. Retrieved July 6, 2026, from <https://www.w3.org/groups/wg/wasm>.
- [71] W3C. (n.d.). WASM specifications. Retrieved July 6, 2026, from <https://www.w3.org/TR/wasm-core-2/WASM>

- [72] Perrone, G., & Romano, S. P. (2025). WebAssembly and security: A review. *Computer Science Review*, 56, Article 100728. <https://doi.org/10.1016/j.cosrev.2025.100728>.
- [73] Michaud, Q., Pipereau, Y., Levillain, O., & Ayed, D. (2024). Securing stack smashing protection in WebAssembly applications. *arXiv*. <https://doi.org/10.48550/arXiv.2410.17925>.
- [74] She, X., Zhao, Y., & Wang, H. (2024). WaDec: Decompiling WebAssembly using large language model. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (pp. 481–492). Association for Computing Machinery. <https://doi.org/10.1145/3691620.3695020>.
- [75] Werner, B. (n.d.). WasmRev. Retrieved July 6, 2026, from <https://github.com/benediktwner/reasm>.
- [76] Fang, W., Zhou, Z., He, J., & Wang, W. (2024). StackSight: Unveiling WebAssembly through large language models and neurosymbolic chain-of-thought decompilation. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235, pp. 13010–13028)*. PMLR. <https://proceedings.mlr.press/v235/fang24e.html>.
- [77] Lacoste, M., & Lefebvre, V. (2023). Trusted execution environments for telecoms: Strengths, weaknesses, opportunities, and threats. *IEEE Security & Privacy*, 21(3), 37–46. <https://doi.org/10.1109/MSEC.2023.3259801>.
- [78] ETSI. (2019, November). Network functions virtualisation (NFV); Security; Report on NFV remote attestation architecture (ETSI GR NFV-SEC 018 V1.1.1). https://www.etsi.org/deliver/etsi_gr/NFV-SEC/001_099/018/01.01.01_60/gr_NFV-SEC018v010101p.pdf.
- [79] AMD. (n.d.). SEV-NP TEE user documentation. Retrieved July 6, 2026, from <https://www.amd.com/fr/developer/sev.html>
- [80] Conklin, L. (2026). *Threat Modeling Process (Historical)*. OWASP Foundation. https://owasp.org/www-community/Threat_Modeling_Process.
- [81] Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V., Mazzeo, G., Hollum, A., & Vaydia, D. (2023). A comprehensive trusted runtime for webassembly with intel sgx. *IEEE Transactions on Dependable and Secure Computing*, 21(4), 3562-3579.
- [82] Gao, S., Li, G., & Fu, H. (2024). Zkwasm: A zksnark wasm emulator. *IEEE Transactions on Services Computing*, 17(6), 4508-4521.
- [83] Johnson, E., Laufer, E., Zhao, Z., Gohman, D., Narayan, S., Savage, S., ... & Brown, F. (2023, May). WaVe: a verifiably secure WebAssembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)* (pp. 2940-2955). IEEE.
- [84] Ménétrey, J. (2024). Webassembly as an abstraction for secure and efficient computing in trusted execution environments.

- [85] Ménétrey, J., Pasin, M., Felber, P., & Schiavoni, V. (2022, July). Webassembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge* (pp. 3-8).
- [86] Cao, S., He, N., Guo, Y., & Wang, H. (2024, September). WASMixer: Binary Obfuscation for WebAssembly. In *European Symposium on Research in Computer Security* (pp. 88-109). Cham: Springer Nature Switzerland.
- [87] Zhang, Y., Liu, M., Wang, H., Ma, Y., Huang, G., & Liu, X. (2025). Research on webassembly runtimes: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(8), 1-47.
- [88] Azmi, S. K. (2024). Wasmtime at Scale: Isolation, Cold-Start and Tail-Latency Trade-offs of WebAssembly Microservices in Production.
- [89] Zeng, R., Jiang, S., Zhao, W., & Zhou, Y. (2026). Debugging Performance Issues in WebAssembly Runtimes via Mutation-based Inference. *arXiv preprint arXiv:2604.13693*.
- [90] Titzer, B. L., Gilbert, E., Teo, B. W. J., Anand, Y., Takayama, K., & Miller, H. (2024, April). Flexible non-intrusive dynamic instrumentation for webassembly. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (pp. 398-415).
- [91] Mao, C., Su, Y., Shan, S., & Li, D. (2024, July). eWAPA: an eBPF-based WASI performance analysis framework for web assembly runtimes. In *2024 IEEE International Conference on Software Services Engineering (SSE)* (pp. 323-333). IEEE.
- [92] The Gatekeeper Authors. (2026). *Gatekeeper: A customizable cloud native policy controller*. Cloud Native Computing Foundation (CNCF). <https://open-policy-agent.github.io/gatekeeper/website/>
- [93] Nanos, A., & Papagianni, C. (2025). D3.3: Final report on the DESIRE6G intelligent and secure management, orchestration, and control. Zenodo. doi.org
- [94] Das, S., Chen, B., Chandramohan, M., Liu, Y., & Zhang, W. (2018). ROPSentry: Runtime defense against ROP attacks using hardware performance counters. *Computers & Security*, 73, 374-388.
- [95] Elsabagh, M., Barbara, D., Fleck, D., & Stavrou, A. (2017, March). Detecting ROP with statistical learning of program characteristics. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (pp. 219-226).
- [96] Davi, L., Sadeghi, A. R., & Winandy, M. (2011, March). ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (pp. 40-51).
- [97] Zhang, J., Chen, W., & Niu, Y. (2019). DeepCheck: A non-intrusive control-flow integrity checking based on deep learning. *arXiv preprint arXiv:1905.01858*.

- [98] Roy, K. C., & Chen, Q. (2024, August). LogSHIELD: A graph-based real-time anomaly detection framework using frequency analysis. In *International Conference on Science of Cyber Security* (pp. 56-75). Singapore: Springer Nature Singapore.
- [99] Ge, X., Cui, W., & Jaeger, T. (2017). Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices*, 52(4), 585-598.
- [100] Chen, L., Sultana, S., & Sahita, R. (2018). HeNet: A Deep Learning Approach on Intel $\text{\$}^{\circledast}$ Processor Trace for Effective Exploit Detection. *arXiv preprint arXiv:1801.02318*.
- [101] Wang, X., Huang, F., & Chen, H. (2019). DTrace: Fine-grained and efficient data integrity checking with hardware instruction tracing. *Cybersecurity*, 2(1), 1.
- [102] Cheng, L., Liljestr nd, H., Ahmed, M. S., Nyman, T., Jaeger, T., Asokan, N., & Yao, D. (2019, September). Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Cybersecurity Development (SecDev)* (pp. 114-128). IEEE.
- [103] Gu, Y., Zhao, Q., Zhang, Y., & Lin, Z. (2017, March). PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (pp. 173-184).
- [104] Cisco Systems. (2026). *Snort: Open Source Intrusion Prevention System*. <https://www.snort.org>
- [105] Open Information Security Foundation (OISF). (2026). *Suricata: Open Source Network Analysis and Threat Detection Software*. <https://suricata.io>
- [106] Fu, C., Li, Q., Shen, M., & Xu, K. (2021, November). Realtime robust malicious traffic detection via frequency domain analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (pp. 3431-3446).
- [107] Contributeurs de Wikip dia. (2026). *Cloudflare*. Wikip dia. <https://fr.wikipedia.org/wiki/Cloudflare>
- [108] Contributeurs de Wikip dia. (2026). *Akamai Technologies*. Wikip dia. [https://fr.wikipedia.org/wiki/Akamai Technologies](https://fr.wikipedia.org/wiki/Akamai_Technologies)
- [109] The Falco Authors. (2026). *Falco: Cloud Native Runtime Security*. The Falco Project. <https://falco.org/>
- [110] Rahman, A., Shamim, S. I., Bose, D. B., & Pandita, R. (2023). Security misconfigurations in open source kubernetes manifests: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 32(4), 1-36.
- [111] Saavedra, N., & Ferreira, J. F. (2022, October). Glitch: Automated polyglot security smell detection in infrastructure as code. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering* (pp. 1-12).

- [112] Liu, D., Wang, L., Chen, Y., Elkashlan, M., Wong, K. K., Schober, R., & Hanzo, L. (2016). User association in 5G networks: A survey and an outlook. *IEEE Communications Surveys & Tutorials*, 18(2), 1018-1044.
- [113] Xu, Y., Gui, G., Gacanin, H., & Adachi, F. (2021). A survey on resource allocation for 5G heterogeneous networks: Current research, future trends, and challenges. *IEEE Communications Surveys & Tutorials*, 23(2), 668-695.
- [114] Kim, T., & Chang, J. M. (2017). QoS-aware energy-efficient association and resource scheduling for HetNets. *IEEE Transactions on Vehicular Technology*, 67(1), 650-664.
- [115] Zhao, N., Liang, Y. C., Niyato, D., Pei, Y., Wu, M., & Jiang, Y. (2019). Deep reinforcement learning for user association and resource allocation in heterogeneous cellular networks. *IEEE Transactions on Wireless Communications*, 18(11), 5141-5152.

Annexes

Annex A List of DoS Attacks (non-exhaustive)

Attack Type	Category	Target	Basic Mechanism	Impact on Network Functions
UDP Flood	Volume	Bandwidth, routers	Sends massive UDP packets to random ports	Link saturation, router overload
ICMP Flood	Volume	Bandwidth, network stack	Floods ICMP echo requests	Network congestion, CPU exhaustion
Amplification (DNS/NTP/etc.)	Volume	Bandwidth, edge devices	Spoofs victim IP and triggers long replies	Massive bandwidth exhaustion
SYN Flood	Protocol	TCP stack, firewalls	Exploits half-open handshake	State table exhaustion
ACK/RST Flood	Protocol	Firewalls, IDS	High-rate ACK/RST packets	Makes stateful devices track useless flows
TCP Connection Exhaustion	Protocol	Servers, load balancers	Many connections opened but never closed	Connection table exhaustion
Fragmentation (Teardrop, etc.)	Protocol	IP stack	Sends malformed/overlapping fragments	Crashes or loops IP reassembly
Ping of Death	Protocol	Network stack	Oversized ICMP packets	Buffer overflow in legacy stacks
Smurf Attack	Protocol	Networks with broadcast	ICMP to broadcast → replies to victim	Amplified congestion on LAN
BGP Session Flood	Routing	BGP routers	Floods BGP TCP sessions	Routing instability, reconvergence
OSPF LSA Flood	Routing	Internal routers	Floods fake LSAs	Routing table overload
HTTP Flood	Application	Web servers, proxies	Many valid-looking HTTP requests	CPU and thread exhaustion
Slowloris	Application	Web servers	Holds HTTP headers open slowly	Exhausts connection slots
Slow POST/RUDY	Application	Web servers	Very slow body uploads	Thread exhaustion in input handlers
DNS Query Flood	Application	DNS resolvers	Rapid queries (often random domains)	Resolver overload, cache busting
SIP/VoIP INVITE Flood	Application	VoIP servers	Mass INVITE requests	SIP state overload, call denial
Algorithmic complexity targeted attack	Application	Network functions	Drive the application into corner cases, fostering for algorithmic complexity	Congest network functions

Annex B Feasibility study. Using control flow probes to detect DoS attacks

UC4.6 describes a considered method to detect DoS attacks looking how the traversed network functions are affected. As this use case realization is conditioned by an initial feasibility study, this section constitutes the feasibility study. This **feasibility study** was motivated by the identified technical risks in the proposal and which were confirmed by this study.

B.1 Introduction. Use case 4.6 description

B.1.1 General

The use case has been detailed in D2.2. The following provides the essential elements for its fast and easy grasp.

B.1.2 Principle and core ideas

- Probing the control flow graph for performance assessment or specific attack characterization-classification (eg, Return Oriented Programming)
- Assessed impact of DoS attack, dropping network function performance
- Disambiguation between DoS attack and platform resource attrition, by expanding the Definition and scope of collected-analyzed metrics
- Extracting time series on Time to Execute, Code block call frequency, sequence of traversed code blocks.
- Bufferization before post processing
- Post-processing leveraging AI to classify a DoS attack

B.1.3 KPIs

- KPI 4.3 Delivery of specification PoC related to Software Control Flow Monitoring. An initial feasibility study will be produced, notably in consideration of the identified technical risks.
- KPI 4.4 Probability of detection of DoS attack inference:>80%. This KPI will be conditioned by the outcome of KPI 4.3 (i.e., initial feasibility study)
- KPI 4.5 Probability of false detection <5%. This KPI will be conditioned by the outcome of KPI 4.3 (i.e., initial feasibility study)
- Additional KPIs must be added as: A-KPI 18. False alarm rate for DoS attack detection

B.1.4 Identified risks

- Performance degradation
- Low probability of detection
- High false alarms, reflecting the inability to disambiguate causes such as legit platform resource attrition
- Curated and relevant DoS attacks used for AI classification training

B.1.5 Control flow timestamping

B.1.5.1 General

This section details the motivations of probing the control flow, with probes set up inside code blocks, making it possible to assess the speed of execution of a software and determine if the software instructions are used as they should.

B.1.5.2 Principle of performance monitoring by control flow probing

As shown below, code blocks 1 to 4 are executed in a sequence. The aggregation of all possible execution sequences traversed during the execution of a program consists in its control flow graph. Code blocks are supposedly elementary and minimal code sequence traversed from their entry point to their exit.

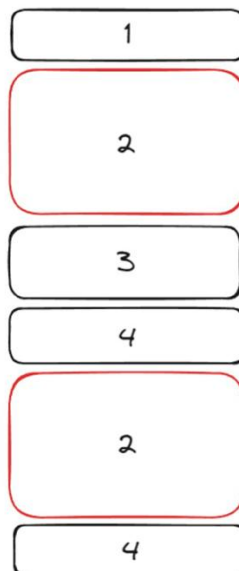


Figure 30. Control flow probe insertion

- A specific code block (ie, here code block #2) Execution Time (ET) measured with two timestamps located at the entry and exit points.
- The Call Frequency (CF) of a code block.

Both metrics characterize different the speed of execution of a software by the platform. Noticeably a software may have a continuous execution and never stops, disallowing a measure on time basis (ie, execution time). In this case, operating at code block level enables to handle a finite time element on which the ET measure is produced. CF on the other hand, when timestamps are placed on speed-marking code blocks, bring an idea of the throughput of the network function. Typically, if the main packet processing sequence is probed, collecting its call frequency directly reflects the speed of processing of incoming packets, which can be influenced by several factors such as the incoming traffic flow rate and the allocated resource delivered to the process by the processor.

Moreover, collecting the sequence of traversed code blocks is used to infer anomalous execution behaviour. Typically return oriented programming attacks takes control of the control flow, placing gadgets on the stack. Code blocks will be called in anomalous sequences.

B.1.5.3 Need for selective placement for sustainability

As demonstrated by [93], inserting timestamps on each code block may totally break the performance of the supporting software (i.e., to measure its performance!). Probing blindly each different code blocks derive in performance penalty from two digits of percents (%) to three digits of percents (%), according to the monitored code structure. As a consequence, a selective placement of these probes must be done, through a hybrid (i.e., static and dynamic) analysis of the code to discern performance-marking code blocks (e.g., a central packet parsing or processing loop). The placement analysis is not straightforward and sometimes not practicable in standard deployment conditions.

B.1.5.4 Control flow graph undecidability, breaking accuracy

However, the control flow is not a static structure and is prone to be dynamically modified. Code blocks may broken down during execution as the control flow may get into these code blocks after their entry and leave them before the exit. Hence, selected code blocks for probing may be broken, delivering erroneous measures. This risk can be reduced significantly through the knowledge of the semantic of a code structure (e.g., parsing the packet content, processing the routing). Semantic analysis derives from the same hybrid code analysis as stated above, with the same operational constraints.

B.1.5.5 Principle of code block tracing for anomalous behavior detection

ROP (or JOP) attacks disrupt the control flow to execute small code sequences (i.e., in fact one single instruction) located just before the code block exit (i.e., Return) and its return to stack pointer for the next jump of that sort. These high frequency jumps inside code blocks can be traced and reveal these forms of attack. Typically, ROP attacks reveal and exploit the control graph undecidability as stated above, as code blocks can indeed be shortened and exited before they are completed.

For such control flow graph anomaly detection, several types of techniques are existing in the state of the art, employing different monitoring metrics produced by the processor, the execution trace and the operating system:

Post execution analysis with extraction of vectors on the execution trace and AI inference for detection. Because it does not occur at runtime, no performance penalty is produced. No modification of the software is required for this post execution analysis.

During execution with time series on Hardware Performance Counter (HPC) (e.g., branch misprediction, return misprediction and instruction/branch counts [94][95]). An alternative derives from the domain of Control Flow Integrity with metrics collected from Intel Processor Trace (IPT) with ML detection of an anomalous trace [97]. These techniques do not impose changes on the code as they all collect and use processor-delivered metrics. An alternative is to set up probes on return instruction [96] on the binary code.

B.2 Scoping phase

B.2.1 General

This section details the collaborative work produced by MONT and TSS, as part of a scoping phase which took place at MONT's offices in the 24th of February 2025. This kick-off work session aimed at defining the collaborative work, reasserting the considered techniques, pointing out already discovered difficulties. These elements derived from an initial conceptualization phase by TSS, presented and challenged by MONT.

B.2.2 Scoping the objective

A recall of the main objective of the use case U.C 4.6 was made. In alignment with NATWORK concept and its reconciliation of security and performance, the objective is to define how network function control metadata can be used to produce a novel method for DoS detection. Graph based detection is explored [98], where graph refers to the traffic patterns but without

entering into a network function and its internal execution flow graph. Leveraging internal control flow graph is today unexplored. The motivation is to assess if and how tracing the code execution can bring any sign that a DoS attack is under way, with limited false positives and false negatives.

B.2.3 Discussed technical statements

The meeting has enabled to issue the following statements:

There is no one DoS attack type but several and in fact many. The different DoS attacks impact the network function performance differently. Under DoS attacks, the performance ratio of a targeted network function can either increase or decrease according to the DoS attack type. As a consequence, our mis-conceived idea that DoS attacks forcibly degrade the throughput of a network function was uncovered. All DoS attacks using repeated packets lead to a higher cache hit rate, augmenting the network function performance. Sophisticate DoS attacks that spawn randomized packets reversely drop the cache hit rate, hence degrade the network function performance ratio. Detection of cache hit rate variation (i.e., positively or negatively) may be used as a DoS attack detection metrics, with the assumption that a normal traffic data distribution can be distinguished from a highly varying (i.e., randomized packet attack) or extremely narrowed distribution (i.e., repeated packet attack).

A list of possible collected metrics was enumerated during the meeting.

- NF performance, via a *reference* code block injection over the CFG at performance-prone section of the code (e.g., main packet processing loop)
- Hardware Performance Counters (HPC): (e.g., cache misses, indirect branches, retired instructions, number of traversed branches), available on x86 processors (i.e., Intel and AMD)
- Instruction Per Cycle via PMCs (i.e., in relation to cache misses, branch predictions hit rate)
- Process-related CPU use rate (i.e. time allotted by the processor on one process in percent): Via O.S and perf
- Intel Processor Trace metrics (e.g., branches traversed, interruptions) available on Intel processors and requiring a post processing of the extracted metadata by tools (e.g., libipt, perf)

B.2.4 Conclusions

While the UC 4.6 main challenge was to first identified as the required disambiguation between a DoS attack and a normal resource attrition, our scoping phase has revealed another challenge.

As a matter of fact, we cannot bet on the fact that a DoS attack degrades the performance of the targeted network function as it may also augment it. Consuming the listed extra metrics must be further explored, studied and progressed, as means reduce false positives and negatives. Last, defining and leveraging a cache-agnostic process is promising as it reveals unambiguously a platform resource attrition.

B.3 Time-domain analysis

Time domain analysis refers to our study of control flow-based time series candidates. This analysis starts with a reminder of a succinct UC 4.6 objective for clarity, the enumeration of candidate metrics, the relevance of the metrics for DoS detection, and finally their usability with respect to the use case defined KPIs.

B.3.1 Succinct reminder of the Use Case 4.6 objective

Control flow metrics, extracted on a traversed network function are collected to provide an early detection of DoS attacks with sufficient accuracy. The objective of the detection KPIs are given as:

- KPI 4.4 Probability of detection of DoS attack inference:>80%.
- KPI 4.5 Probability of false detection <5%.
- These two KPIs are conditioned by the outcome of a feasibility study

B.3.2 Enumeration of time domain metrics

In the time-domain, we dissociate the metrics related to the control graph from the performance related metrics.

Control-flow based metrics, possibly collected through control graph probes or through processor registers are listed below:

- Basic block timestamps as shown in Figure 30, generates two metrics of Execution Time (ET) and Call Frequency (CF)
- Processor based Hardware Performance Counters, delivers branch counts and processed instructions (aka retired instructions)
- Intel's Processing Trace, delivers traversed branch offsets

Performance-related metrics, bring clues on the execution, grasping the interaction between the workload and the platform:

- Cache misses or hit

- Branch misprediction
- TLB misses
- Process CPU use rate

The methods to extract these metrics and generate the associated time series are detailed below. In this survey, we look at the main attributes of each classes of metrics, with the criteria of metric Definition and scope, ability to access the metric per process, performance penalty, workload preparation (i.e., if any) and required platform administrative rights (i.e., if any).

B.3.3 Control Graph inserted performance probes

B.3.3.1 Definition and scope

CG inserted performance probes correspond to our initial idea and are illustrated in Figure 30. Probes timestamp the entry and exit point of basic blocks, enabling to assess the basic block time to execute (ET) and call frequency (CF). Both metrics capture differently the speed of execution of the workload and both shall be de-noised and integrated:

- ET delivers a direct measure on a given basic block. However, as one basic block execution time is influenced by variables content and data stream, assessing the variation of the workload implies the integration of a representative set of basic block variation.
- CF is as well directly dependent on the ingested variables content and data stream. To illustrate that, for a network function, if the basic block is inside a main packet processing loop, it is directly influenced by the input traffic volume. If the traffic is very low, the main packet processing loop will be lowly called. At the network function saturation level, the main packet processing loop will be called at the max network function processing speed (ie, loop iteration frequency) whatever the incoming traffic (equal or above the network function throughput).

Both metrics capture an insight on the workload actual performance but require an averaging on several basic blocks for accuracy.

B.3.3.2 Per process isolation

Inserted in the process, the probes bring a per process monitoring.

B.3.3.3 Performance penalty occasioned by the monitoring

Probing each code block can induce major performance losses (ie, several 100%)- in hardly predictable manner according to the code structure. Hence, code block probe insertion shall be highly selective and limited “stable” code blocks on interest, through a hybrid analysis (ie, both static and dynamic) of the code. Being selectively inserted these probes are frugal and do not

affect the performance. In short, probes shall be inserted at a few and relevant places where they can bring direct business-related metrics (e.g., network function throughput).

B.3.3.4 Workload preparation, if required

Probe insertion requires pre-deployment executable code modification, a hard penalty on the network function DevSecOps.

B.3.3.5 Required platform administrative right, if required

No rights are required on the platform as not system calls are made to collect these probes.

B.3.3.6 Usability

Probe metrics provide exclusively payload performance. Turning a deviation of performance as resulting from an attack is a very risky stance. First, they are many causes of performance variations which all combine and aggregate into one performance value. Some grasp the interaction of the payload with the system (e.g., cache misses), based on the processed data content (e.g., parser rare case processing) and finally derived from exogeneous causes (e.g., a new workload on the shared resource). In dynamically orchestrated infrastructure, an heavy traffic may be split and directed to dynamically-instantiated platforms. In practice, visibility on performance does not deliver any kind of visibility on the root cause of the variation, being caused by a DoS attack or not.

The self-contained nature of the probes removes administrative right requirement, but the insertion is a complex pre-deployment operation, with modification of the binary file and a hybrid analysis of the workload including dynamic tracing. As stated above, for sustainability, a minimal quantity of such probes can be used. In such restricted usage, the probes cannot be used for control flow integrity as it requires a probe at each basic block but can deliver an indication of the network function performance. Reversely, DoS attacks do not impact a workload performance in a predictable manner, being able to accelerate, slow down or maintain speed.

Hence, we must be selective on a specific DoS attack with its known influence on performance and assess in which extent performance is sufficiently discriminative to align with the use case objective KPIs.

B.3.3.7 Metric discriminative strength

Table 37 brings the discrimination strength of the different metrics used for DoS detection.

Table 37. Performance metric by inserted probes discriminative power towards all types of DoS attacks.

DoS type	Influence on network function performance	Metric Discriminative power
Repeated packet attacks (eg, TCP Syn Flood, ICMP Echo flood, UDP flood, Control plane policy attacks, SYN cookie bypass attempts)	Acceleration thanks to higher cache hit rate, higher TLB and branch predictions.	The network function will run in saturated mode, with shortened packet processing cycles than legit traffic. The acceleration perdures all along the attack, hence over a relative long timing. False positives: with legit traffic, the operating system dispatching (i.e., time splitting) variations can result in the same acceleration over a long timing, notably at a CPU intensive sibling retirement. False negatives: CPU throttling (i.e., decrease of cpu clock caused by temperature elevation to be expected) can limit the acceleration during a short period
Randomly generated packet attacks (eg, TCP Syn with random packets, UDP flood with random destination ports, DNS random-subdomain attacks, fragmentation attacks with random fragment IDs, flow table exhaustion in SDN environments, IPS/IDS evasion flood with random payloads)	Possible deceleration caused by lower cache hits, TLB and branch prediction. This deceleration will be only caused if the randomized traffic shows a significant data distribution distortion compared to legit traffic.	Legit traffic and the attack traffic may feature the same data distribution, resulting in performance metric insufficiency. False positives: Deceleration can result of CPU throttling, an novel heavy and CPU-bound sibling process is instantiated on the same platform (i.e., core) and consumes a portion of the CPU cycles. False negatives: The deceleration caused by randomized traffic can be compensated by CPU boosting, the retiring of sibling processes on the process relaxing the processor.
All other DoS (as found in Annex A).	None	Performance metric does not characterize the attacks

B.3.4 Conclusion

Whatever the DoS types, measuring the performance of the network function appears to be insufficiently discriminative.

A workaround to augment the metric discerning power is to fully isolate the network function on the platform, removing interactions and influences from other processes. This can be done at the expenses of specific deployment first (i.e., restricting deploy ability) and done through administrative rights (i.e., in contradiction with the essence of self-contained probes). However, even though this workaround is considered, CPU throttling and boosting cannot be controlled, degrading behind the curtains the detection.

The use case KPIs cannot be met with these probes, whatever the capabilities of the AI used for their interpretation.

B.4 Hardware Performance Counters (HPC)

B.4.1 Definition and scope

HPC are standard and commonly used incremental counters, assessed on Linux by perf tool. The counters count for:

- Taken branches,
- Indirect branches,
- Retired instructions,
- Cache misses,
- Exceptions.

HPC delivers counts, no trace. By-default, HPC are delivered on one core basis, for all running processes. As discussed below, counts can be made on a per process basis through specific system parametrization.

Instruction Per Cycle (IPC), derived from HPC

IPC can be directly measured from HPC, being the number of retired instructions / CPU cycles and is a key performance indicator which grasps both workload and the platform. Platform micro-architectural elements (e.g., instruction decode buffer, branch predictor, micro-op cache, L1/L2/L3 cache size and access speed, out-of-order execution) and workload structural elements (e.g., instruction parallelism, cache locality, quantity of branches for good branch predictability) interact in forming the IPC. The workload time to execute comes from the multiplying IPC and the platform clock frequency. Factors that impede IPC are high rates of cache misses, TLB misses, branch mispredictions, CPU-ordered pipeline flushes, code serialization (result dependency

chain) and threads false sharing. Factors that augment IPC are good branch prediction, large and low latency caches, few workload branches, vectorized or compiler-optimized code.

B.4.2 Per process isolation

Leveraging so called Performance Monitoring Unit (PMU) virtualization, modern operating system uses their knowledge of context switches to produce per process, per thread or per cgroup counts. This can be done at user level without requiring administrative right.

Simultaneous Multi-Threading (aka SMT) breaks isolation as several logical cores share the same hardware resources. Two threads running on two sibling logical CPUs, may use the same cache memory. For isolating one process, SMT shall be disabled (or leave the other sibling SMT thread idle).

B.4.3 Performance penalty occasioned by the monitoring

Performance losses induced by PMU virtualization (for per process HPC isolation) is cheap, relying on counters that actually bring no cost. Performance losses comes from the PMU virtualization, inducing context switches. Collecting HPC with PMU virtualization enabled for process isolation induces performance penalty as defined below:

- Per context switch: latency in the range between 100-300 ns
- Global impact depends on quantity of context switches, a workload-dependent metrics defined as:
 - CPU-bound workload, below 10K switches, hence leading to performance loss in the range of 0,1%, not measurable.
 - Network functions: 10K-100K switches, which is a low value (i.e., relatively low value thanks to network function design employing user space networking, poll-mode and asynchronous operations), hence leading to performance loss in the range of 1%, fairly acceptable.
 - Syscall-heavy, event-driven applications: 100K-500k, hence leading to performance loss in the range of 5%.
 - Pathological cases (tiny tasks): 1M switches, hence leading to performance loss in the range of 10%

B.4.4 Workload preparation, if required

No workload preparation work is required, as HPC are hardware counters accessed by the kernel, operated for any workloads.

B.4.5 Required platform administrative right, if required

Kernel controls HPC registers. PMU virtualization requires administrative rights, for whatever per-process or kernel-generated counts. For perfect process isolation and collecting per process HPC values for accurate monitoring, SMT disablement and/or resource-exclusive process pinning must be carried out. In all cases, administrative rights on the platform are required

B.4.6 Usability

Two encouraging elements of HPC metrics is their variety on the one hand and the low CPU costs for their collection on the other hand. Our analysis of their relevance is given in Table 38.

We restrict our analysis of the usability on two attacks: HTTP GET flood (L7) and Algorithmic Complexity attacks, guessing in which extent these attacks expose network functions first and a comparison on the effectiveness of considered HPC-based detection with existing Intrusion Detection Systems (IDS).

B.4.6.1 HTTP GET Flood (Layer 7)

An HTTP GET Flood (L7 attack) is an application-layer denial-of-service attack where an attacker sends a very large number of legitimate-looking HTTP GET requests to a web server.

Unlike volumetric floods (e.g., UDP floods), this attack features

- Uses valid TCP connections
- Sends syntactically correct HTTP requests
- Often mimics normal browser behaviour
- Targets application resources (CPU, database, dynamic content)

Because the requests are valid, they bypass simple network-layer filtering.

Table 38. Relevance of HPC metrics for DoS detection

DoS Type	Attack Characteristics	Retired Instructions	Taken Branches	Indirect Branches	Reason for Metric Variation	Inference Feasibility
Volumetric Flood (UDP/ICMP flood)	High packet rate, minimal per-packet logic	Moderate – High increase	Moderate increase	No or slight increase	Fast-path packet processing loops execute repeatedly	Medium – similar to legitimate high traffic
TCP SYN Flood	Large number of incomplete handshakes	High increase	High increase	Moderate increase	Extra state handling, retransmission logic, timeout management	High – handshake-heavy control-flow signature
HTTP GET Flood (L7)	Application-layer request storm	Very high increase	Very high increase	High increase	Deep parsing, routing, logging, application logic	High – strong control-flow amplification
Slowloris	Many half-open HTTP connections	Low – Moderate increase	Moderate increase	Moderate increase	Timer checks, connection polling, indirect callbacks	Medium – High – idle-heavy branching pattern
Algorithmic Complexity Attack	Crafted input triggers worst-case path	Extremely high increase	Extremely high increase	High increase	Pathological loops and repeated conditional checks	Very high – abnormal instruction inflation
Reflection / Amplification	Traffic amplified via reflectors	Moderate increase	Moderate increase	Little or no change	Mostly packet filtering logic	Low – Medium – resembles volumetric flood
Malformed Packet Flood	Invalid headers trigger error paths	High increase	High increase	Moderate increase	Error-handling code paths with heavy branching	High – distinct error-path signature

B.4.6.2 Typical Characteristics

- High request rate per client or distributed clients (botnet)
- Requests to dynamic or expensive URLs
- Repeated access to the same endpoint
- Normal TCP handshake completion

The goal is to exhaust:

- CPU
- Thread pools
- Application workers
- Backend database connections

B.4.6.3 Detection by Classical IDS

Classical IDS [104][105] typically detect HTTP GET floods using signature-based detection (eg, excessive HTTP GET frequency, repeated identical URIs, known bot User-Agent strings). They use alternative or complementary detection methods as delivered in Table 39.

B.4.6.4 Relevance of HPC detection

HPC-based methods and classical IDS methods show differentiated abilities against DoS generated traffic as shown in Table 40.

Our analysis of the relevance of using HPC-based detection for HTTP Get flood follows.

Network functions are scarcely affected: HTTP GET flood attacks target L7 applications, with resource attrition. An HTTP GET flood is an application-layer (Layer 7) DDoS attack. Its primary targets are web servers or application-layer components (e.g., authentication, database backends with expensive queries). There, reverse proxies (e.g., Cloudflare [107], Akamai [108]) absorb these floods with per-client rate limits through WAF inspection engines. Network functions traversed on the path can be affected when they carry out functions such as TCP connection tracking, NAT tables, TLS handshake offload or Stateful firewall inspection, but the core objective is application resource exhaustion, not routing, switching, or packet-forwarding logic. Compared to SYN floods or UDP floods, HTTP GET floods are much less about network functions and much more about service-layer capacity.

Table 39. Methods for detecting HTTP GET flood DoS attacks by classical IDS

Detection method	Signature-based detection	Threshold/ rate-based detection	Behaviorioul anomaly detection
Metrics	Set rule triggers on: - excessive HTTP GET frequency - Detection of repeated identical URIs - Known bot User-Agent strings	Monitor: - Requests per second (RPS) - Concurrent sessions - Connection attempts - Trigger if exceeding predefined thresholds	Compare traffic patterns to historical baseline, looking for: - Uniform request intervals - Lack of JavaScript execution - No cookie reuse - Missing referrers
Limitation	Example logic and limitation: “Alert if more than X GET requests per second from same IP” - Fails if attacker distributes traffic across many IPs.	Hard to distinguish from flash crowds (e.g., Black Friday traffic).	Requires learning phase and tuning.

Table 40. Comparison of HTTP GET flood DoS detection abilities between classical IDS and HPC-based method

Criterion	Classical IDS	HPC-Based
High traffic rate	Easy to detect	Detected via instruction growth
Distributed botnet	Harder (per-IP threshold bypass)	Still visible as aggregated CPU inflation
Encrypted HTTPS	Limited payload visibility	Fully visible at execution level
Flash crowd differentiation	Difficult	Possible via branch/instruction ratios

Lack of HPC signal relevance: HTTP GET Flood are based on normal looking code paths, executed as expected by server routines, hence do not cause abnormal CPU patterns (e.g., elevation of cache misses, branch explosion, pathological algorithm behaviour) as ACA do. HPC signals extracted on the flow have statistical similarities with legit traffic. HPC metrics do not augment HTTP GET flood characterization.

Signal dilution: Virtualization, multi-tenancy opposes to network function isolation, required for accurate HPC metric monitoring.

High overhead: Monitoring HPCs at scale introduces kernel-level instrumentation, sampling overhead, interrupt handling and performance impact on production systems. At 100k+ req/s, fine-grained per-request HPC tracking is impractical.

Conclusion

HPC monitoring on HTTP Get flood is irrelevant, as it does not augment their detection while increase performance impact, obviously in contradiction with NATWORK security and performance reconciliation principle.

B.4.6.5 Algorithmic Complexity Attacks (ACA)

An Algorithmic Complexity Attack (also called a computational DoS attack) exploits worst-case behaviour of algorithms.

Instead of sending many requests, the attacker sends carefully crafted input that forces the application into elevated algorithmic patterns (e.g., worst-case parsing, pathological regular expression (regex) evaluation, hash collision, quadratic or exponential runtime behaviour. The attacker achieves high CPU consumption with very low traffic volume.

B.4.6.6 Detection by Classical IDS

These attacks are harder to detect because traffic volume is low.

Classical IDS detect them using several methods of Known Exploit signatures, payload inspection, Application layer log inspection as shown in Table 41.

Table 41. Application Complexity DoS attacks detection by classical IDS

Detection method	Known Exploit signatures	Payload inspection	Application layer log inspection
Metrics	Detect specific malicious payload patterns. Regex detection of: - Nested quantifiers - Known hash-collision patterns - XML bombs	Look for: - Extremely long parameters - Abnormally structured inputs - Suspicious repetition patterns	Look for: - High CPU usage with: - Low traffic rate - Few active connections - Correlate IDS with server logs

Detection method	Known Exploit signatures	Payload inspection	Application layer log inspection
Limitation	Works only for known vulnerabilities.	Limitation: Encrypted HTTPS hides payload from traditional IDS.	Limitation: Requires integration with host monitoring.

B.4.6.7 Relevance of HPC detection

Table 42. Comparison of Algorithmic Complexity DoS detection abilities between classical IDS and HPC-based method

Criterion	Classical IDS	HPC-Based
Low traffic volume	Often missed	Detected (high instructions per request)
Unknown exploit	Not detected	Detected as execution anomaly
Encrypted traffic	Blind	Not blind
Signature required	Yes	No

Algorithmic Complexity Attacks (ACA) are particularly suitable for HPC-based detection because they create computational amplification without traffic amplification. However, our analysis of the relevance and feasibility reveals strong obstacles to limit false positives and false negatives (as indicated by the use case KPIs):

- **False positives** (i.e., wrong attack alerts): With the considered time window, the augmentation of retired instructions or traversed branches may equally result from an algorithmic targeted attack or from legitimate traffic, background maintenance operations, software updates, activated garbage collection mechanisms or simply by legitimate shift in workload data (and distribution). Virtualization-caused context switches generate a significant noise of significant weight in the observed sampling time window. HPC are not self-sufficient and must be used in conjunction with other metrics. Typically, ingress traffic volume, could sort out if the HPC deviations come from a higher traffic.
- **False negatives** (i.e., undetected attacks): Nowadays, network functions are multi-thread, with threads dynamically distributed over several cores of the same platforms where SMT mode can be enabled for higher throughput, or over several distinct platforms in the continuum. ACA are targeted, short-lived and targeting a restricted number of these threads. For sustainability, the monitoring shall be restricted to a few threads, possibly out of the attack scoped threads. For sustainability again, the inspected sampling time

windows are larger than the ACA duration, hence integrating and diluting the HPC deviations, without detection thresholding passing. Last but not least, except in operationally hardly-met situations where the network function is isolated on a core, HPC in a given time window reflect all running processes there, where deviations become statically reduced.

- **Complex attack backtracking:** HPC deviations are accounted per core, backtracking the thread that caused the deviation, is impracticable.
- **Complex reference value generation:** Calibrating the HPC values with static reference values, hardly the real deployment conditions of the network function in operation, both in terms of collaborative platforms engaged and software workloads distributed over these platforms.
- **Specific training per deployment pattern:** ML-based detectors will similarly need to be defined for a specific implementation pattern. This is a major obstacle.
- **Heavy operational limitations caused by process isolation:** The isolation of the process on a core by process pinning is a requirement for leveraging HPC based detection. At the first place, it is worth noting that isolation is misaligned with the dynamic load balancing, shared processing, virtualization and hyper threading, which altogether suffice to abandon HPC-based detection (Table 42). More, isolation requires administrative rights which may simply not be delivered when the network function executes off-premises.

Conclusion

Traditional IDS focus on Network patterns, signatures and traffic thresholds where both HTTP Get flood and Algorithmic complexity attacks characterize as shown in Table 43.

Table 43. Comparison of HTTP GET flood and Algorithmic complexity attacks detection by classical IDS

Feature	HTTP GET Flood	Algorithmic Attack
Traffic volume	High	Low
Requests valid?	Yes	Yes
Detectable by rate limits?	Yes (sometimes)	No
Detectable by signatures?	Partially	Only if known exploit
Evasion difficulty	Medium	Low traffic makes it stealthier
IDS effectiveness	Moderate	Often low without deep inspection

Both selected attacks cause abnormal instruction inflation, increased branching density and high CPU cycles per request which makes them, a priori good candidates for behavioral detection using hardware performance counters (HPCs). A further analysis reveals that HPC do not elevate the detection of these two DoS types, looking at network functions.

At the first place, network functions are not the main targets of HTTP Get floods attacks. A common obstacle on both attack types is the signal dilution, as process isolation is hardly met in real situations.

B.5 Process CPU use rate

B.5.1 Definition and scope

For a given process, CPU use rate is defined as the time allotted by the processor on that process as a percentage in a given period of time.

The same CPU utilization level can arise from two distinct scenarios:

- The process is the lone workload of the processor. Once the process is terminated, the processor sets in idle mode.
- A time splitting distributes the processor activity to several workloads

B.5.2 Performance penalty occasioned by the monitoring

The performance penalty can be assessed at null as the CPU use rate is delivered by the scheduler natively, without asking it.

B.5.3 Workload preparation, if required

No workload (ie, the process) preparation is needed to collect the process CPU use rate, being directly delivered by Linux.

B.5.4 Required platform administrative

CPU use rate is accessible directly from user space through a Linux virtual file (i.e., /proc/<pid>/stat), without administrative right required.

B.5.5 Usability

Process CPU usage (%) gives very limited information about workload performance or speed. It tells you how busy the CPU is — not how fast the work is being completed. Assessing CPU attrition using the CPU use rate can be also misleading and it shall be completed by other collected metrics. To illustrate, a network function dealing with a low traffic puts the CPU in idle mode when the network function is terminated, leading to a low or very low CPU use rate. There is no attack-prone consuming workload on the processor while the CPU use rate diminishes. For this reason, process CPU use rate cannot be used alone. It can eventually bring an additional metric for a composite (i.e., multi metrics) detection.

B.6 Intel Processor Trace (IPT)

B.6.1 Definition and scope

IPT is an Intel proprietary security mechanism that traces and logs taken indirect branches, interruptions, exceptions and changes in the control flow. IPT logs addresses (i.e., memory offsets) not only counts, serving as the essential tool to assess the correct execution of an execution (i.e., Control Flow Integrity, exploit detection, runtime verification). A decoder (e.g., perf stat or perf report from Linux, Intel's libipt, Intel's VTune profiler, GDB support for IPT, Processor trace from QEMU) is used to reconstruct an accurate execution trace.

IPT logs "packets" known as:

- Packet Stream Boundary (PSB): checkpoint marker used for decoder synchronization
- Target IP Packet (TIP): delivers function pointer calls, returns, indirect jumps, context switches.
- Timestamp Counter (TSC): delivers timing information, measuring time between events, correlation.
- Taken/Not taken (TNT): records conditional branches decisions.
- Flow Update Packet (FUP): delivers precise IP information for certain events (i.e., interrupts, exceptions, asynchronous events).
- Core Bus Ratio (CBR): delivers CPU frequency ratio, used for higher accuracy in timing measurement.

B.6.2 Per process isolation

By default, IPT comes for all running processes (including the kernel) on a core. Process isolation can be delivered by modern kernels to isolate process against the kernel and between processes through context switches. Hence, per process processor trace can be delivered. On linux, perf tool is used in user space and enables to collect per process (i.e., PID) trace.

B.6.3 Performance penalty occasioned by the monitoring

IPT logging induces a performance penalty in the range of 3-5% for bare metal mounted network functions and 6-10% for virtualized network functions, depending on their structure and filtered packets. In both cases, performance impact is significantly affecting network function throughput in operation. Noticeably, decoding does not incur penalty as being processed off-line. IPT is to be viewed as a calibration tool to optimize a network function prior deployment.

Memory consumption can be very impactful, in the range between 1 Megabytes/ sec to several tenths of Megabytes/ sec per core, according the workload and selected packet types. Network

functions are generally branch-heavy and IPT comes in the higher range (i.e., several Megabytes/sec per core).

B.6.4 Workload preparation, if required

There is no workload preparation as IPT is a monitoring framework including the processor and the operating system.

B.6.5 Required platform administrative

For IPT, you need root privilege or `perf_event_paranoid` is set permissively. The latter is a kernel parametrization which defines which users can use `perf`.

B.6.6 Usability

B.6.6.1 IPT for Control Flow Integrity

IPT is a costly monitoring as it collects high frequency metadata from the execution trace, elevating security against control flow-based attacks (e.g., ROP, JOP). AI-powered decoding can detect anomalous traces [99][100][101][102][103] The delivered performance penalty is given in Table 44.

Table 44. Use of Intel Processing Trace for Control Flow Integrity verification

Academic paper	Performance impact	Technique
Griffin [99]	~11.9% overhead on SPECint, ~6.2% on SPECfp (comparable to software-only CFI)	CFI through shadow stack
HeNet[100]	<~4% worst, <~2% average overhead due to PT itself (tracing control flow)	PT tracing itself <i>without security logic</i>
DTrace[101]	<~4% worst, <~2% average overhead due to PT itself (tracing control flow)	Data and control trace with primitives (Nginx / microbenchmarks)
Exploitation Techniques and Defenses for Data-oriented[102]	~1.1–2.7% throughput drop (for specific primitives latency) and ~6.7% (emulated PTWRITE) with Redis; ~3.9% expected with optimized PTWRITE	CFI enforcement via PT trace and TIP graph matching
PT-CFI (TIP matching / trace analysis)[103]	21% average overhead in SPEC CPU2006 benchmarks; some cases up to ~65% on specific program	PT + trace processing pressures (alternative monitor evaluation)

In the above papers, CFI enforcement techniques vary from simple PT logging (i.e, inducing a low overhead of a few percent) to high real-time processing (i.e., inducing an overhead of 21%).

B.6.6.2 IPT For DoS detection

DoS attacks are diverse, and their effect on a network function’s control flow can vary widely as shown in .

Table 45.

Table 45. Influence on the control flow by DoS types

DoS attack type	Effect on the control flow	Notes and caveats
Volumetric floods (e.g., SYN flood, UDP flood)	Usually <i>minimal per-packet control-flow changes</i>	The network function may loop more often, but the <i>logic path per packet</i> is unchanged; mostly stress is on scheduling and queues.
Application-level floods (e.g., HTTP GET flood)	Sometimes changes control flow if different handlers are triggered	Can overload certain functions, but high packet similarity can make trace patterns look “normal.”
Protocol exploitation attacks (e.g., malformed TCP options)	May trigger exceptional branches or error-handling paths	Control-flow changes are detectable in PT traces.
Advanced stealthy DoS (e.g., low-rate “slowloris”)	Likely no significant control-flow deviation	Maintains normal request handling paths, only slows resource consumption — control flow remains similar to normal traffic
State exhaustion attacks (e.g., table/connection depletion)	May only increase loops or retries subtly	PT might see slightly higher loop frequency but <i>not a clear anomaly</i> unless baselines are very precise.

Only some DoS attacks produce detectable anomalies in control-flow — mostly when error-handling or unexpected execution paths are triggered. Advanced or stealthy attacks often leave the control-flow patterns indistinguishable from normal traffic.

PT is useful for detecting application-level DoS that triggers unexpected execution paths, such as:

- Malformed packets
- Unusual protocol features
- Exploit-driven floods

PT is less effective for volume-based or stealthy DoS, which mainly consume resources without changing the per-packet control flow.

Conclusion

IPT is proprietary to Intel, only enabled on Intel's processors, which constrains workload deployability. This is of course highly restrictive with low control on the host platform and infrastructures as encountered in networking.

IPT is used for post-mortem analysis, delivering a precise and deterministic execution trace, hence identifying which branches have been taken. This post execution vision complements what HPC reveals during execution. It is impracticable for early DoS detection in real time.

Another caveat is a higher performance impact as reflected here.

Last and not least, relying solely on PT for DoS is risky

- False negatives: Advanced attacks that mimic normal packet content and rate may consume resources without triggering abnormal branches.
- False positives: Legitimate bursts of traffic can create unusual branch density, appearing like DoS in PT traces.
- Scope limitation: PT only sees CPU execution paths, not queue lengths, buffer saturation, or network-level anomalies. Many DoS effects are purely resource-based (e.g., memory exhaustion) with minimal control-flow impact.
- DoS affects control flow is context-dependent. Only a subset of attacks produce detectable control-flow deviations. Relying solely on PT traces for DoS detection can miss advanced or stealthy attacks. PT should be complementary to traditional metrics (packet rate, queue length, CPU/memory saturation) for robust detection.

For these reasons, we believe IPT elements cannot be used for early DoS detection in real time.

B.7 Frequency-domain analysis

B.7.1 General

Frequency domain analysis is used to detect anomalous traffic, and also some longer-term research that combines graph representations with statistical or spectral methods for intrusion

and anomaly detection. Using frequency domain analysis of software control graph elements has not been discussed and explored yet.

B.7.2 Frequency-Domain Analysis for Early DoS Detection

There is clear academic work using frequency-domain transforms (e.g., Fourier, time–frequency analysis) on network traffic to detect DoS or malicious patterns early: Traffic Frequency-Domain Feature Extraction Methods transform traffic flows into the frequency domain (e.g., using FFT) and extract spectral characteristics that differentiate normal versus attack traffic. DoS attacks often exhibit distinct high-frequency or low-frequency signatures that are harder to spot in the time domain alone. One system (e.g., Whisper [106] uses frequency-domain features in a machine-learning detection pipeline to achieve both high accuracy and high throughput for malicious traffic detection — including DoS-like behaviors — with low latency.

B.7.3 Graph-Based Detection coupled with frequency analysis.

Graph-Based Detection Techniques While most research uses traffic flows as data and optionally graph structures, explicit work on graphs of code blocks in network functions is sparse. However, there are adjacent directions:

- **Intrusion Detection via Graph Representations:** Some studies frame network structure or interactions (e.g., hosts, connections) as graphs to improve detection by capturing relationships and topologies (e.g., GraphNeural Networks detecting anomalous flows). Graph-based alert correlation systems link alerts over time using graph structures to enhance detection context.
- **Graph + Frequency in a Different Context** Newer work (e.g., LogSHIELD [98]) applies frequency-domain analysis over graph samples (graph node features or embeddings) for real-time anomaly detection — though not specifically DoS on network function code blocks, but in broader provenance/host log graphs.

B.7.4 Benefits and costs of conversion to the frequency domain

B.7.4.1 Benefits

Frequency features can capture periodic floods, bursts, or recurring patterns that may not be obvious in raw time-series packet counts. Time-frequency analysis separates attacks by their

spectral patterns — e.g., burst DoS vs. covert low-rate attacks — giving a richer basis for classification than raw byte/packet counts.

Spectral graph analysis — transforms graph structure using eigen values/spectra (e.g., graph Fourier transforms) — could conceptually detect structural anomalies in function call graphs. Profiling combined with frequency transforms — one could extract function execution traces as time-series and then apply FFT or wavelets to see if DoS conditions induce spectral anomalies.

In the domain of network function control graph-based DoS detection, a similar frequency domain graph analysis can be considered.

B.7.4.2 Costs

Frequency-domain extraction combined with ML can increase compute overhead compared with simpler time-domain statistics because of transforms and feature extraction, but can still achieve real-time detection at line rates with optimized implementations.

For a window of size N : FFT complexity: $O(N \log N)$ shall be compared to simple thresholding: $O(N)$

Computing spectral features requires extra processing (FFT, feature transforms), and ML classification adds further cost — often mitigated by algorithmic optimization.

Graph Neural Networks and graph correlation layers add memory and compute overhead relative to simpler classifiers, which must be accounted for in high-throughput environments.

B.7.4.3 Relevance of frequency-domain analysis for early DoS detection

We explore how FFT can be beneficial (or not) for exploiting the control flow probe and HPC metrics.

B.7.4.4 FFT on control flow probe metrics

We analyse in Table 46 in which extent converting probe-based time series in the frequency domain can be useful.

Table 46. Usefulness of converting control flow probe time series in the frequency domain

DoS type	Attack periodicity	Relevance of frequency domain detection
Repeated packet attacks (eg, TCP Syn Flood, ICMP Echo flood, UDP flood, Control plane policy attacks, SYN cookie bypass attempts)	During the sliding window used for performance variation analysis, the attack is continuous, hence showing no periodicity.	None
Randomly generated packet attacks (eg, TCP Syn with random packets, UDP flood with random destination ports, DNS random-subdomain attacks, fragmentation attacks with random fragment IDs, flow table exhaustion in SDN environments, IPS/IDS evasion flood with random payloads)	During the sliding window used for performance variation analysis, the attack is continuous, hence showing no periodicity	None
All other DoS (as found in Annex A).	The performance metric does not characterize the attack	None

B.7.4.5 FFT on HTTP GET flood Attacks by HPC

An HTTP GET Flood is a high-rate request attack where the attacker overwhelms a web server with a large number of seemingly legitimate HTTP GET requests. It has been commonly observed against major web infrastructures, including services protected by companies like Cloudflare [107] and Akamai Technologies [108]. The attack properties are high request rate, periodic or semi periodic requests by botnets or scripted traffic, strong volume anomalies and possible burst patterns. Periodic request bursts are generally generated at fixed intervals, where FFT can reveal dominant frequencies (e.g., 10 Hz bursts). Synchronized bots amplify spectral spikes.

When attacks are randomized or mimic normal user distributions (e.g., Low-and-slow HTTP floods, highly adaptive botnets with jitter), spectral spikes fade or disappear. Table 47 gives a synthetic view of the benefits and costs of frequency-domain detection.

Table 47. FFT relevance for HTTP GET flood attacks

Factor	Relevance	Cost
Periodic bot floods	High	Moderate
Randomized floods	Low	Not justified

Factor	Relevance	Cost
High-speed inline detection	Limited	Potentially high
Offline forensic analysis	High value	Acceptable

B.7.4.6 Performance Costs

Computational Cost

For a window of size N :

FFT complexity: $O(N \log N)$ shall be compared to simple thresholding: $O(N)$

In high-speed networks (10–100 Gbps):

- Full packet FFT = too expensive.
- Flow-level aggregation + FFT = feasible.
- Often deployed in detection pipelines, not inline filtering.

Bufferisation

Implementing frequency-domain analysis forcibly requires traffic buffering over time windows, with the associated memory overhead for time-series storage.

Conclusion

For cost issues, classical reverse proxies moderately use Frequency-domain analysis for detecting structured HTTP GET floods. When used, it is typically supplementary to statistical and ML-based detectors. Applying FFT on per request HPC analysis is unsustainable and in conflict NATWORK security and performance reconciliation. Leveraging FFT can be used on flow level which does not consume HPC. Leveraging frequency-domain for HPC detection is not an option as it will only add performance costs.

B.7.4.7 Algorithmic Complexity attacks

There is no relevance of using frequency-domain analysis for ACA attacks, due to their sporadic, occasional occurrence. Frequency-domain analysis operates on time-series patterns. Algorithmic complexity attacks are about input structure, not frequency patterns. Applying frequency-domain to detect ACA has no technical relevance.

B.8 Conclusion of the feasibility study

B.8.1 Lesson learnt and discoveries

The main lessons learnt and conclusions of this feasibility are listed below:

DoS attack types are quite numerous, generating opposite effects on network functions performance.

Capturing the network function performance as do the control flow probes (i.e., resulting in ET and CF metrics), delivers insufficient DoS attack visibility. Moreover, these probes must be placed very moderately to avoid excessive penalty on the network function. The proposal stated technical risks of insufficient semantic visibility or excessive performance penalty is revealed and confirmed. The main interests of these probes relate to their self-containment and deep association with the process. The probes are carried with the monitored process itself, anywhere and their access does not depend on administrative rights on the platform. The probe signals are exclusive to the monitored process.

Probe-based performance metrics appear to be insufficiently discriminative to all forms of DoS attacks. An untaken workaround is to isolate the workload from sibling processes, at the costs of degrading the portability of the workload first with a novel uneasy administrative right required on the platform (ie, for off-premises operation). This would of course caused costs issues and degrade sustainability. Last but not least, this move may reveal to be insufficient as CPU throttling and boosting cannot be controlled, totally biasing the performance measurement, hence the detection ability.

The original idea of using control flow-based metrics, as defined in use case 4.6 was mis-leading.

Looking beyond these control-flow extracted metrics, HPC provide richer information content, notably for qualifying Algorithmic Complexity application-targeted Attacks (ACA). ACA are scarce with low prevalence below 5% as generally stated by DoS vendors. Setting up the right conditions for detection imposes a strong process isolation for which PMU virtualisation mode is required. PMU and system parametrization is needed with administrative right. Moreover, hyperthreading mode injects noises on HPC metrics in all conditions. Enforcing administrative right for process pinning and disabling hyper threading is a strong obstacle for adoption, breaking NATWORK sustainability and higher resource utilization. Nevertheless, we have considered two types of DoS attacks as potentially beneficiary on HPC monitoring as HTTP Get flood and Algorithmic Complexity. On both sides, our deep analysis reveals that a common obstacle of getting de-noised and process-isolated values is impractical in virtualized and multi-tenant operation. As a

consequence, signal dilution shall be expected with a massive level of false negatives and positives.

We had considered other metrics of process use rate and Intel Processing Trace, respectively semantically impoverished and requiring post execution decoding (i.e., an offline process).

We had explored how frequency-domain on control graph probes and on HPC-based detection can be beneficial, concluding that there is no detection enhancement while FFT processing costs will simply dig the deficit.

B.8.2 Feasibility study conclusion

This feasibility study concludes that our original approach—using control-flow-instrumented probes within network functions for early DoS detection—was not suitable for the intended purpose.

First, the performance of a network function, like that of any software system, is determined by multiple interacting factors that are difficult to distinguish and control, even with full administrative privileges over the execution platform. Performance metrics derived from control-flow instrumentation lack the discriminative power required to identify the underlying causes of performance variations.

Second, the diversity of DoS attacks further complicates detection. Different attack classes affect network function performance in fundamentally different ways: some degrade performance, others have little measurable impact, and some may even improve apparent execution speed by altering execution paths. This limitation became apparent early in our investigation.

More generally, performance metrics extracted from a network function's control-flow graph provide only limited semantic information for identifying DoS attacks. They are also heavily influenced by exogenous factors, including operating system activity, hardware resource contention, and co-located processes, which introduce significant bias and noise.

To address these limitations, we investigated the use of Hardware Performance Counters (HPCs), particularly cache miss rates. Obtaining reliable HPC measurements requires isolating the network function from other workloads to eliminate interference from sibling processes. Such isolation is difficult to achieve in practice, reduces deployment flexibility, and imposes platform-specific configuration requirements. Moreover, it conflicts with the NATWORK objective of maintaining efficient and transparent resource sharing.

Our study further shows that HPC-based metrics are primarily effective for detecting a narrow class of DoS attacks that exploit algorithmic complexity. These attacks represent only a small fraction of the overall DoS threat landscape, with an estimated prevalence below 5%.

Overall, we conclude that the original approach does not provide sufficient discriminative capability for reliable early DoS detection. Effective detection requires higher-level, semantically richer metrics that capture application and protocol behaviour rather than relying solely on low-level execution characteristics