Runtime Verification Across the Trust Boundary: Lessons from a Connected-Fleet Pipeline

Damian Navarro[1], Robert Abela[2], Christian Colombo[2], and Axel Curmi[2]

[1] Institute of ICT, Malta College of Arts, Science and Technology, Malta
damian.navarro.i63864@mcast.edu.mt
[2] Department of Computer Science, University of Malta, Malta
{robert.abela|christian.colombo|axel.curmi}@um.edu.mt

Abstract. Engineered autonomous and semi-autonomous vehicular systems act on streaming telemetry such as position, motion, and engine state, whose integrity directly affects safe behaviour. Runtime verification (RV) provides a means of checking the behavioural plausibility of such data at runtime, but in real deployments the monitor does not operate in isolation: it must coexist with the security mechanisms that protect the same data path, and with tight near-real-time latency budgets.
In this paper we report on the design tensions that arise when RV and hardware-assisted encryption share a single telemetry path, grounded in a concrete connected-fleet pipeline built around a Teltonika FMC650 telematics unit, an SEcube secure cryptographic co-processor used for hardware-assisted encryption, and monitors generated from a single LARVA specification deployed on both sides of the encryption boundary. We distil six lessons from the deployment. The resulting placement, integration, and consistency trade-offs are relevant to fleet, Internet of Things (IoT), and cyber-physical telemetry pipelines that combine behavioural monitoring with protected data transport.

Keywords: runtime verification · engineered autonomy · fleet telemetry · hardware-assisted security · LARVA

1 Introduction

Runtime verification (RV) has been applied where software correctness and security both matter, including avionics [6], financial systems [2], autonomous vehicles [14], and robotics [5]. In such systems, runtime monitoring must coexist with the mechanisms protecting the monitored data and infrastructure.

Encryption protects telemetry while it crosses the network, but an RV monitor can inspect the data only where it is available in plaintext. In the implemented pipeline this occurs at two points: after the edge component decodes a device record but before encryption, and at the server after decryption. The first point is closer to the source but runs in a less trusted environment; the second is more trusted but observes the record only after edge-side processing and

transmission. We examine monitors at both points and the resulting trade-offs in trust, consistency, and latency.

Our contributions are threefold: (i) we characterise the design space for placing runtime assurance relative to a hardware-enforced trust boundary on a vehicular telemetry path; (ii) we instantiate this design space through a concrete connected-fleet pipeline, reporting measured integration overhead and cross-layer consistency; and (iii) we distil six transferable lessons for building reliable runtime assurance into engineered autonomous systems.

Section 2 reviews related work; Sections 3 and 4 present the design and prototype; Section 5 reports the evaluation; Section 6 derives the lessons; and Section 7 concludes.

2 Background and Related Work

RV for autonomous and robotic systems. The application of RV to autonomous systems has a well-documented history. ROSMonitoring [5] and its successor [9] provide runtime verification frameworks for ROS-based robotic platforms, enabling the specification and checking of temporal properties over the publish-subscribe message flows that typify robotic architectures. ROSMonitoring was evaluated on a Mars Curiosity rover simulation, checking wheel-velocity commands against a safety threshold and withholding violating messages. This illustrates how RV can detect and react to violations during autonomous operation.

Monitor generation and deployment. LARVA [3] is a runtime verification tool that generates monitors from automata-based specifications.[3]

Its native deployment targets Java programs via AspectJ instrumentation, but the generated monitor classes can also be invoked through inter-process communication from non-Java environments (see for example [12])—a capability that proves essential in the heterogeneous pipeline we describe.

Where a monitor sits within a distributed system, and how it is packaged for deployment, has itself been considered in numerous papers (see the relevant section of the survey [10]). Focusing on two papers which are closer to this work, Tsigkanos et al. [11] present a service-based architecture for runtime verification of volatile IoT systems in which events are evaluated locally at edge nodes, avoiding the central or cloud-based collection that can add latency and communication overhead. Mehran and Ulus [8] instead concentrate on deployment packaging, running the monitor as a lightweight container alongside the publish–subscribe participants it observes and evaluating the approach on a software-defined-vehicle stack built around the Autoware autonomous-driving stack and the CARLA simulator. Both address monitor placement and integration overhead in distributed, telemetry-bearing systems; neither, however, contends with the constraint central to our setting, namely a hardware-enforced encryption boundary that determines where plaintext is available for a monitor to observe at all.

[3] Larva tool repository: `https://github.com/ccol002/larva-rv-tool`.

Securing the monitor and the monitored path. A complementary line of work asks how runtime verification and security mechanisms should be combined. RVsec [1] addresses the monitor itself, assembling a technology stack for secure monitor deployment that weighs several threat scenarios against the additional setup and runtime cost each mitigation entails, using a quantum-safe chat application as a case study. The RV-TEE line [12,4] instead builds a trusted execution environment out of runtime verification and a stock hardware security module (HSM), avoiding the need for specialised TEE hardware: the HSM supplies the trusted domain for cryptographic primitives and key storage, while two monitors execute in the untrusted domain, one verifying that the protocol implementation follows its design and one inspecting the data flows that cross the boundary to the HSM.

Our setting shares with RV-TEE the situation of a monitor operating around a hardware-enforced trust boundary, but the monitor is put to a different use. RV-TEE's boundary monitor observes the flows that cross into and out of the trusted domain and asks whether those flows are legitimate; the hardware protects keys and cryptographic execution. Here the encryption protects telemetry as it crosses the network, so the boundary determines where the record is legible to a monitor at all, and the properties concern the behavioural plausibility of the record rather than the legitimacy of the flow carrying it. This shifts the question from how to guard a trust boundary to where a behavioural check can be sited relative to one, with the architectural consequences we develop in Section 6.

3 Runtime Assurance Across the Trust Boundary

Hardware-assisted encryption creates two plaintext observation points with different trust characteristics. The edge monitor observes a record immediately after protocol decoding and before encryption; the server monitor observes the reconstructed record after decryption. The edge is closer to the source but more exposed, whereas the server provides a more controlled execution environment but sees the record only after edge-side parsing, serialisation, encryption, and transport.

Threat and trust assumptions. The network and edge host are treated as potentially exposed, while the central server and the SEcube cryptographic operation are trusted within the prototype scope. Telemetry may be faulty or behaviourally implausible before encryption. The prototype's Advanced Encryption Standard (AES) Electronic Codebook (ECB) configuration protects confidentiality only imperfectly and does not authenticate ciphertext or prevent replay. Authenticated encryption would detect modification after encryption, but would not establish that the original plaintext was physically plausible or produced by an uncompromised source; the RV properties address that separate question. A host-level compromise is out of scope: an attacker executing code on the edge host would control both the record and the flags it carries, defeating the edge monitor and the parity check alike.

Dual placement. Neither observation point subsumes the other. The edge monitor checks the representation closest to device decoding, whereas the server monitor checks the representation reconstructed after transport and decryption. We therefore deploy both monitors from one generated specification. Their agreement provides evidence that the monitored representation and property semantics remain consistent across the path; disagreement is treated as a diagnostic alarm rather than proof of a specific fault or attacker action.

The two placements do not merely duplicate one another. The edge instance evaluates values immediately after protocol decoding and unit normalisation, before JSON field mapping and cryptographic processing. The server-side instance evaluates the representation reconstructed after serialisation, encryption, transmission, decryption, and request handling. A unit-conversion error, omitted field, stale session state, or serialisation mismatch can therefore cause disagreement even when both monitors execute identical generated code. Conversely, perfect agreement does not prove that the source reading is truthful: a faulty or compromised source can emit plausible values that are preserved perfectly across the path. Dual placement is thus a consistency mechanism layered on top of, rather than a replacement for, authenticated transport, source attestation, and domain-specific plausibility checks.

Specification unity. One LARVA specification defines all monitored properties, and the generated monitor classes are deployed on both sides. This removes discrepancies caused by independently maintained property implementations. Deployment-specific differences may still arise from event normalisation, delivery, instrumentation, or session-state handling.

Cross-layer comparison. The two placements are compared using violation counts by type. A disagreement can expose an integration or state-management problem, but the aggregate comparison does not identify which monitor is authoritative or diagnose the cause by itself. Section 5 defines the parity score used in the evaluation.

4 A Connected-Fleet Telemetry Pipeline

We instantiate the design space described above through a concrete connected-fleet pipeline. The pipeline processes vehicular telemetry from a Teltonika FMC650 Global Positioning System (GPS) tracker through a C++ Transmission Control Protocol (TCP) ingestion layer, SEcube-backed AES encryption, and a Spring Java server for decryption, storage, and downstream processing. Figure 1 shows the two monitor locations relative to the cryptographic boundary.

Telemetry ingestion. The FMC650 telematics unit transmits AVL (Automatic Vehicle Location) records over TCP using the Teltonika Codec 8 protocol. The C++ layer parses the International Mobile Equipment Identity (IMEI) handshake, decodes individual AVL records into structured fields (timestamp, latitude, longitude, speed, engine state), and normalises units for downstream processing. For experimental repeatability, a replay client can substitute for the

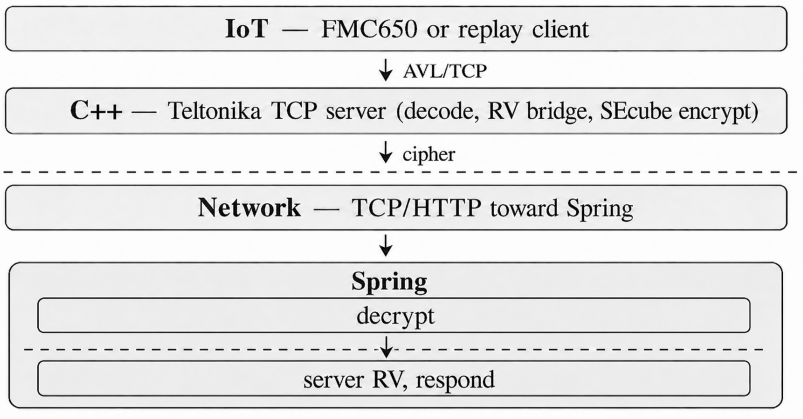


Fig. 1. The dashed lines bound the span over which the record is ciphertext and therefore not legible to a monitor. The edge monitor observes the decoded record before encryption and the server monitor the record reconstructed after decryption; both are generated from the same LARVA specification.

live device, streaming pre-recorded trajectory datasets through the same protocol path.

Hardware-assisted encryption. After parsing, each telemetry record is serialised to JSON and encrypted using the SEcube HSM [7] before transmission to the server.[4] The prototype uses AES-ECB to evaluate the integration path. AES-ECB leaks repeated block patterns and provides neither authentication nor replay protection. A production system would use authenticated encryption such as AES-GCM, together with nonce and replay management. Authentication would detect ciphertext modification after encryption, but would not determine whether the source telemetry was behaviourally plausible before encryption. Changing the mode would alter cryptographic cost and failure handling, but not the two plaintext monitor locations studied here.

Monitored properties. Five kinematic plausibility properties check the behavioural consistency of the telemetry stream. Table 1 summarises the thresholds decided on for the final runs. *Invalid GPS* is evaluated on the current record alone; *Time backwards*, *Teleport*, *Acceleration high*, and *Engine-off moving* compare the current record with a valid predecessor. *Teleport* and *Acceleration high* share GPS-derived state. *Engine-off moving* additionally reads the vehicle's engine-state field, a Boolean telemetry value that is true when the engine is reported as on and false when it is reported as off.

The WGS-84 latitude and longitude limits are definitional validity bounds. The remaining thresholds were selected as conservative engineering plausibility limits during prototype development: 80 m/s rejects an inter-record speed well above ordinary road travel, 6 m/s^2 bounds abrupt changes in implied speed, and 5 m distinguishes meaningful movement from small GPS variation while the

[4] SEcube-SDK repository: `https://github.com/SEcube-Project/SEcube-SDK`.

engine is reported off. These values were decided on before the final runs, were not tuned against their results, and were shared by both generated deployments.

Specification and deployment. All five properties are defined in a single LARVA specification file. On the server side, AspectJ instrumentation invokes the generated monitor when the Spring server's telemetry-receiving `onSample` method receives a decrypted sample. On the transport side, the C++ process writes one normalised event per decoded AVL record to a child Java process, conceptually `SAMPLE(ts, lat, lon, speed, engineOn)`. The Java process applies the same generated monitor and returns violation flags through standard output. C++ attaches the transport-side flag to the outbound JSON so that it can be recorded alongside the independently produced server-side flag. Parity is then computed offline after each run from aggregate per-property counts extracted from the paired TCP and Spring diagnostic logs. The first record on each TCP connection is skipped because pairwise properties require a predecessor, and a `RESET` command prevents state from leaking between sessions.

Prototype scope. The system is a working integration, not a production-grade fleet platform. Server-side key material is configured in code; fleet-scale multi-device orchestration and cloud-native deployment are out of scope. Evaluation uses deterministic dataset replay from the T-Drive taxi trajectory collection [13], which contains real taxi GPS traces with changing speeds and stop–start behaviour. Controlled faults are injected by modifying existing records at reproducible, spaced positions: each single-property run targets approximately 20% of records using a fixed seed, while the mixed run rotates the five fault types over the same index pattern. The exact field mutation depends on the property, for example an out-of-range coordinate for *Invalid GPS*, a reduced timestamp for *Time backwards*, or a large coordinate jump for *Teleport*. The implemented temporal properties use timestamps carried inside each telemetry record. They do not use arrival time, meaning the wall-clock time at which a record reaches the edge or server monitor. Bridge, encryption, and network delay therefore affect how quickly a verdict is produced, but not the truth of the evaluated properties. Properties defined over arrival time would instead require synchronised clocks and explicit treatment of transmission delay.

5 Empirical Evidence

The final set of runs was executed on a Windows version 25H2 development host with twelve logical processors, using JDK 21 for the generated monitor process and Spring application. The SEcube device was physically connected and

Table 1. Monitor thresholds decided on before the final runs.

Property	Threshold	Interpretation
Invalid GPS	$\lvert lat\rvert > 90$ or $\lvert lon\rvert > 180$	Outside WGS-84 coordinate domain
Teleport	implied speed $> 80\,\mathrm{m/s}$	Implausible inter-record displacement
Acceleration high	$\lvert \Delta v/\Delta t\rvert > 6\,\mathrm{m/s^2}$	Excessive change in implied speed
Engine-off moving	displacement $> 5\,\mathrm{m}$ *and* engine off	Movement but engine-state reports off
Time backwards	$t_i < t_{i-1}$	Non-monotonic source timestamp

invoked through its vendor SDK. Each accepted run produced per-record timing diagnostics from the C++ pipeline, paired server-side logs, and operating-system samples of Spring and native-process working set and machine-scoped CPU every 500 ms. End-to-end latency was measured around the record-processing path in the C++ pipeline, while separate fields recorded Java-bridge and encryption durations. Consequently, the reported end-to-end figures include integration and scheduling effects rather than only the execution time of the generated property logic.

Three cleaned T-Drive trajectory traces were selected to represent shortest, median, and longest journeys. Candidate files that were empty, too short, internally spliced—that is, containing segments from different journeys joined at a file boundary, visible as discontinuous timestamps or implausible coordinate jumps—or inconsistent with their source metadata were rejected rather than manually repaired. The anomaly generator mutates existing rows so that injected and clean traces retain the same number of records. Single-anomaly runs target approximately 20% of rows at spaced indices using a fixed seed, and mixed runs rotate the five anomaly classes over the same index pattern. This preserves repeatability while allowing each property to be studied independently before interactions are introduced.

For each shortest and median workload, the clean RV-off baseline, clean RV-on baseline, five single-anomaly scenarios, and one mixed scenario were each repeated three times. Corresponding longest scenarios were executed once after the repeated tiers showed low variability: average p50 coefficients of variation (CV) were 0.141% and 0.613%, with maxima of 0.277% and 0.945% for shortest and median respectively. CV is the standard deviation divided by the mean, expressed as a percentage. Longest-tier values are therefore point estimates and are not treated as having the same statistical strength as the repeated tiers. The replay client waited 50 ms between transmitted Teltonika records. After a record entered the C++ pipeline, a separate 10 ms pacing delay controlled internal forwarding so that all compared runs used the same input rate. These are deliberate replay delays rather than measured processing latencies. A run was accepted only when startup completed, the intended dataset was recorded in its manifest, all diagnostic outputs were present, and both processes exited cleanly; otherwise the complete run was repeated.[5]

Violation behaviour. Table 2 reports violation rate, defined as the number of processed records that emitted a given violation flag divided by the total number of processed records. This is compared with the injector's target of approximately 20% to check whether the observed flag frequency follows the constructed input, but it is not precision or recall because a single injected record can influence more than one stateful comparison. *Acceleration high* ranges from 0.1631 to 0.1950 because resetting its predecessor state after *Teleport* creates the bounded gap explained in Section 6. *Engine-off moving* ranges from 0.0730 to 0.1946 because the engine-state field, displacement, and timing conditions must

[5] Repository: `https://github.com/Naronitsu/Damian-Navarro-Thesis-2025-2026`.

all hold. Mixed-run rates also include interactions between properties, so they are not interpreted as a direct missed-detection measure.

Overhead and resources. On the repeated workloads, RV adds 30.20–30.58 ms to median per-record latency. The similarity of this increment across differently sized traces, together with the CV values in Table 3, supports interpreting it as a per-record integration cost at the evaluated pacing rather than a cost that grows with trace length. The longest clean run shows a smaller 16.90 ms median increment, but because it was executed once and anomaly scenarios on the same trace reached the low-60 ms range, it is reported as a point estimate rather than evidence of improved scaling.

The C++–Java bridge accounts for most of the measured latency increase; server-side LARVA monitoring is below 0.1 ms. The integrated encryption step records approximately 2.6 ms per record, but the experiment lacks an otherwise identical software-only cryptography baseline, so this value is not the isolated marginal cost of SEcube. Table 3 shows that Spring uses most of the measured process memory, while the native C++ process remains below 18 MB. Mean CPU utilisation is below 0.6% for each process, measured as a share of the whole twelve-logical-processor machine; the small value is plausible because records are deliberately replayed at tens-of-milliseconds intervals rather than processed at maximum throughput. The CPU result therefore describes this paced experiment and should not be read as the utilisation expected under an unthrottled fleet load. The median (p50) describes a typical record, whereas tail latency describes slower records near the upper end of the distribution. The 95th percentile (p95) is the latency below which 95% of records fall. In the longest *Acceleration high* run, p95 reached approximately 62.6 ms even though the median was lower, showing that occasional slower records would be hidden by reporting p50 alone.

Cross-layer parity. For each run, consistency is measured from aggregate violation counts by type:

$$P = \begin{cases} 1, & \text{if } \sum_t \max(N_t^{\text{tcp}}, N_t^{\text{server}}) = 0, \\ \dfrac{\sum_t \min(N_t^{\text{tcp}}, N_t^{\text{server}})}{\sum_t \max(N_t^{\text{tcp}}, N_t^{\text{server}})}, & \text{otherwise.} \end{cases}$$

Table 2. Violation rates (fraction of processed records) and clean-baseline median latency (ms). Shortest and median values are means across three runs; longest values are single-run estimates.

Scenario	Shortest	Median	Longest
Invalid GPS	0.2002	0.2000	0.2004
Teleport	0.1992	0.1996	0.2006
Acceleration high	0.1631	0.1950	0.1899
Engine-off moving	0.1878	0.0730	0.1946
Time backwards	0.1992	0.1996	0.2000
Mixed anomalies	0.1889	0.1745	0.1971
RV-off p50	31.00	30.94	30.57
RV-on p50	61.58	61.14	47.47
RV increment	30.58	30.20	16.90

Here t ranges over violation-flag types. The numerator sums the count shared by both placements for each type, while the denominator sums their per-type union. Counts from different flag types cannot cancel because the minimum and maximum are calculated separately before summation. A score of 1 denotes identical aggregate violation counts across all types, while anything less signals a disagreement and triggers an investigation. In the evaluated runs, both placements produced identical aggregate per-property violation counts, yielding P=1 throughout.

Dataset governance and validity. An initial set of runs used replay files containing spliced journeys, meaning that separate trip segments had been joined into one trace and produced discontinuous timestamps or implausible coordinate jumps at their boundaries, fields inconsistent with source metadata, and injections over already-corrupted records. Those runs were excluded; replacement baselines were selected using fixed acceptance checks, all scenarios were regenerated, and the complete set of runs was repeated. No coordinates or timestamps from rejected traces were manually corrected. This was an evaluation-data issue rather than a failure of the live monitoring architecture.

The measurements have four main limitations. Violation rate is not event-level precision or recall where one injected fault affects later stateful comparisons. Latency also includes operating-system scheduling, replay pacing, and child-process communication, although deterministic inputs and repeated runs reduce variation. Results from one tracker, one host arrangement, five properties, prototype key management, and replayed taxi traces may not generalise to other deployments. Finally, longest-workload scenarios were run once, so those estimates have lower statistical confidence. The results therefore characterise this prototype rather than universal fleet behaviour.

6 Lessons from the Deployment

This section presents six lessons learnt with the first two concerning the trust-boundary architecture, while the rest are broader RV engineering lessons exposed by the implementation.

Trust Architecture Determines Observation Points. Monitoring before encryption provides the closest observation of decoded device input, whereas monitoring after decryption places the monitor in the more controlled server environment. Monitor placement is therefore a trust decision rather than merely a performance decision. An edge-only deployment accepts that the monitor

Table 3. Repeated-workload stability and clean RV-on resource use. For each scenario CV is calculated across the three run-level p50 latencies; the table reports the average and maximum scenario CV for each workload. Memory is mean working set.

Workload	Avg. CV	Max. CV	Spring (MB)	C++ (MB)
Shortest	0.141%	0.277%	187.6	14.1
Median	0.613%	0.945%	197.3	16.2
Longest*	–	–	194.1	17.8

*Single-run point estimate; no repeated-run CV is available.

shares the exposed host environment, while a server-only deployment accepts that records are observed only after edge-side parsing and transformation.

Dual Placement Provides Cross-Layer Consistency Checking. Placing monitors on both sides of the trust boundary provides a cross-layer consistency alarm without replacing authenticated transport or source attestation. Agreement between the two monitors increases confidence that the monitored record was preserved across the intervening path. A disagreement does not identify the precise origin of the fault, but it narrows the fault domain to the processing and communication path between the two observation points.

Integration Dominated Monitoring Cost in the Prototype. For the selected properties and implementation, the C++–Java bridge contributed substantially more latency than monitor evaluation. The properties themselves perform bounded arithmetic and state updates, whereas every edge event crosses a process boundary, is formatted, scheduled by the operating system, and waits for any returned flag. This distinction matters because optimising the property automata would not address the dominant measured cost.

This result is prototype-specific; richer temporal, aggregate, or learned monitors could reverse the balance. Replacing the bridge with a native generated monitor, or a persistent binary IPC protocol, is therefore the most direct next performance experiment.

Dependent Properties Can Interfere at the Verdict Level. *Teleport* and *Acceleration high* both derive state from GPS displacement. Suppose record r_i introduces an implausibly large position jump. The teleport property fires because the implied speed between r_{i-1} and r_i exceeds its threshold. That same jump also establishes an extreme velocity baseline, so the transition from r_i to the next ordinary record r_{i+1} appears to contain an excessive acceleration or deceleration. A second flag is therefore produced even though both verdicts originate from one discontinuity.

Resetting acceleration state after a teleport removes this duplicate interpretation, but the reset has a cost: acceleration cannot be evaluated at the immediately following boundary because no valid predecessor remains. The resulting one-record grey zone shows that properties sharing derived inputs must be analysed together rather than validated only in isolation. This issue is independent of dual placement; using the same generated monitor confirmed that the interaction was reproduced on both sides.

Evaluation Data Is Part of the Assurance Argument. Monitor verdicts depend on the provenance of the trace and cannot be interpreted in isolation. If a splice is correctly flagged in a trace assumed to be clean, the experiment is no longer measuring false positives or controlled injection density. Conversely, introducing additional anomalies into already corrupted segments obscures results.

For this reason, deterministic replay, documented acceptance checks, regenerated injections, and auditable run selection are treated as part of the evidence itself. While all raw runs were retained, only complete and accepted sets were

included in aggregation. Assurance tooling does not compensate for weak data governance; it makes its consequences visible.

One Generated Specification Avoids Reimplementation Drift. Earlier handwritten C++ and Java properties diverged on threshold rounding, missing predecessors, and reset behaviour, producing the same cross-layer symptom as secured-path faults. Generating both deployments from one LARVA specification removed disagreements caused by separately maintained property logic.

The deployments can still differ in event delivery, normalisation, instrumentation, process lifetime, and state management. One specification reduces reimplementation drift, while parity checking exposes remaining integration differences.

7 Conclusion and Future Work

This paper shows how an encryption boundary shapes plaintext observability, monitor placement, integration cost, and cross-layer consistency in a connected-fleet prototype. Dual monitoring evaluated the pre-encryption and post-decryption representations of the telemetry stream, while one generated specification avoided reimplementation drift. The principal measured RV cost came from the transport-side bridge; property interactions and trace provenance also affected interpretation. These are architecture and integration results, not universal performance or security guarantees.

Future work will replace the C++–Java bridge with a native monitor; if the bridge is retained, it should use versioned framing, heartbeats, bounded retries, and binary IPC profiling. Security hardening will add production key lifecycle management (provisioning, rotation, hardened storage, and compromise response), authenticated encryption with nonce and replay protection, and cryptographic binding of edge-side flags and sequence identifiers to ciphertext, potentially supported by signed verdicts or remote attestation. Evaluation will extend to multi-device concurrency, longer mixed-traffic operation, live control-loop telemetry, arrival-time properties and clock skew, constrained hardware, and selective repetition of the longest or highest-variance runs. The monitored property set should also expand to sequence integrity, geofence-context rules, and multi-sensor temporal correlation, with deployment policy defining fail-open or fail-closed behaviour after an initial log-only phase.

Acknowledgments. The authors thank Andrew Caruana for supervision and feedback, and Alfred Fenech and Tektraco for loaning the Teltonika FMC650 used during prototype integration. We also thank colleagues at MCAST and the University of Malta for technical feedback on the prototype and write-up.

References

1. Colombo, C., Curmi, A., Abela, R.: RVsec: Towards a comprehensive technology stack for secure deployment of software monitors. In: Verification and Monitoring at Runtime Execution (VORTEX). pp. 13–18. ACM (2024). https://doi.org/10.1145/3679008.3685542

2. Colombo, C., Pace, G.J.: Industrial experiences with runtime verification of financial transaction systems: Lessons learnt and standing challenges. In: Lectures on Runtime Verification, LNCS, vol. 10457, pp. 211–232. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_7
3. Colombo, C., Pace, G.J.: Runtime Verification — A Hands-On Approach in Java. Springer (2022). https://doi.org/10.1007/978-3-031-09268-8
4. Curmi, A., Colombo, C., Vella, M.: RV-TEE-based trustworthy secure shell deployment: An empirical evaluation. J. Object Technol. **21**(2), 2:1–2:15 (2022). https://doi.org/10.5381/jot.2022.21.2.a4
5. Ferrando, A., Cardoso, R.C., Fisher, M., Ancona, D., Franceschini, L., Mascardi, V.: ROSMonitoring: A runtime verification framework for ROS. In: Towards Autonomous Robotic Systems. pp. 387–399. LNCS, Springer (2020). https://doi.org/10.1007/978-3-030-63486-5_40
6. Goodloe, A.E., Havelund, K.: High-integrity runtime verification. Computer **57**(4), 37–45 (2024). https://doi.org/10.1109/MC.2023.3322902
7. Gossen, F., Neubauer, J., Steffen, B.: Securing C/C++ applications with a SEcube-based model-driven approach. In: Design and Technology of Integrated Systems in Nanoscale Era (DTIS). IEEE (2017). https://doi.org/10.1109/DTIS.2017.7930157
8. Mehran, A., Ulus, D.: Runtime verification containers for publish/subscribe networks. arXiv:2408.06380 (2024)
9. Saadat, M.G., Ferrando, A., Dennis, L.A., Fisher, M.: ROSMonitoring 2.0: Extending ROS runtime verification to services and ordered topics. In: Formal Methods for Autonomous Systems (FMAS). EPTCS, vol. 411, pp. 38–55 (2024). https://doi.org/10.4204/EPTCS.411.3
10. Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstić, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). https://doi.org/10.1007/s10703-019-00337-w
11. Tsigkanos, C., Bersani, M.M., Frangoudis, P.A., Dustdar, S.: Edge-based runtime verification for the Internet of Things. IEEE Trans. Serv. Comput. **15**(5), 2713–2727 (2022). https://doi.org/10.1109/TSC.2021.3074956
12. Vella, M., Colombo, C., Abela, R., Špaček, P.: RV-TEE: Secure cryptographic protocol execution based on runtime verification. J. Comput. Virol. Hacking Tech. **17**(3), 229–248 (2021). https://doi.org/10.1007/s11416-021-00391-1
13. Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X., Sun, G., Huang, Y.: T-Drive: Driving directions based on taxi trajectories. In: Advances in Geographic Information Systems (SIGSPATIAL). pp. 99–108. ACM (2010). https://doi.org/10.1145/1869790.1869807
14. Zapridou, E., Bartocci, E., Katsaros, P.: Runtime verification of autonomous driving systems in CARLA. In: Runtime Verification. pp. 172–183. LNCS, Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_9