Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Policy-Gated Firmware Trust and Anti-Rollback Resilience for Autonomous UAV Edge Platforms: A Formal Security and Software Validation

Md Shahanur Islam Shagor

shagormd.s.i@iunp.edu.rs

Voronezh State University of Forestry and Technologies https://orcid.org/0009-0003-7730-3202

Research Article

Keywords: UAV cybersecurity, firmware trust, anti-rollback, secure boot, software supply chain, remote attestation, autonomous systems, fail-closed security

Posted Date: August 26th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10812592/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Policy-Gated Firmware Trust and Anti-Rollback Resilience for Autonomous UAV Edge Platforms: A Formal Security and Software Validation

Md Shahanur Islam Shagor
Independent Researcher, Voronezh State University of Forestry and Technologies, Voronezh, Russia
Corresponding author: shagormd.s.i@iunp.edu.rs

Abstract

Autonomous unmanned aerial vehicles (UAVs) increasingly rely on software-defined sensing, estimation, networking, and decision pipelines, making firmware integrity a prerequisite for trustworthy autonomy. A compromised or downgraded firmware image can silently alter safety constraints, navigation logic, command authorization, or telemetry behavior even when higher-layer cryptographic communication remains intact. This paper studies a policy-gated firmware trust mechanism implemented in an open GPS-denied UAV swarm research platform and develops a formal model for its anti-rollback and fail-closed properties. The implementation represents each firmware image through a manifest containing semantic version, measurement, signer, keyed-digest field, secure-boot claim, bootloader-lock claim, and monotonic rollback counter. Field and production profiles require secure-boot and bootloader predicates, non-regression of both version and rollback counter, trusted signer membership, keyed manifest integrity, and an admissible firmware measurement before node startup is permitted. Accepted state is persisted only after validation. The paper formalizes these predicates, derives monotonicity and rejection properties, maps threat classes to enforcement paths, and distinguishes source-verified behavior from assumptions that require a hardware root of trust. Repository evidence includes a registered native boot-trust test target with cases for trusted production acceptance, rollback rejection, and unauthorized maintenance rejection; the repository validation report records the native test suite as green at the studied commit. The analysis also exposes two important trust-boundary conditions: rollback protection depends on integrity of the persistent state record, and current secure-boot/bootloader values are externally supplied claims rather than independently measured hardware evidence. The result is a reproducible software-level firmware resilience baseline and a precise roadmap toward hardware-backed attestation.

Keywords: UAV cybersecurity; firmware trust; anti-rollback; secure boot; software supply chain; remote attestation; autonomous systems; fail-closed security

1 Introduction

Autonomous UAVs are no longer simple radio-controlled vehicles. Contemporary platforms execute perception, state estimation, navigation, mapping, swarm coordination, telemetry, and safety logic on heterogeneous embedded computers. This software concentration makes the firmware and early-boot chain a security dependency for nearly every subsequent control decision. If an attacker can replace or downgrade a privileged image, then secure communication at the application layer is insufficient: the compromised firmware can simply misuse valid keys, falsify sensor data, weaken authorization, suppress alerts, or modify recovery behavior.

Firmware resilience is therefore a distinct problem from packet authentication or encrypted telemetry. NIST SP 800-193 organizes platform firmware resilience around protection, detection, and recovery [1]. The IETF Software Updates for IoT (SUIT) architecture similarly treats a protected firmware manifest and trustworthy update process as fundamental to secure lifecycle management on constrained devices [2]. Remote attestation standards such as the IETF RATS architecture define how evidence about device state can be conveyed and appraised, while emphasizing that the trustworthiness of evidence depends on protected attestation keys and an isolated attesting environment [3]. These principles are directly relevant to UAVs because a flight computer is simultaneously a network node, an embedded controller, and a safety-critical decision point.

UAV-specific security research has concentrated heavily on wireless authentication, swarm membership, privacy, and command-channel protection. Bansal and Sikdar studied secure attestation for UAV fleets [7], while SHOTS combined authentication and attestation for scalable UAV swarms [8]. Broader UAV cybersecurity reviews identify firmware, software supply chains, command interfaces, and update mechanisms as persistent attack surfaces [10,11]. Recent consumer-drone firmware studies further show that practical firmware extraction can reveal outdated components and weak binary hardening, reinforcing the need for verifiable boot and update trust [14].

Despite this literature, a gap remains between high-level secure-boot requirements and a transparent, control-oriented policy that a UAV node can evaluate at startup. In particular, a practical research platform needs explicit answers to five questions. What constitutes a firmware identity? Which fields must be monotonic? How are maintenance windows authorized without silently weakening production policy? Which checks are software-enforced, and which depend on external hardware trust? What state must be persisted so that a previously valid but vulnerable image cannot be reintroduced after a successful upgrade?

This paper addresses those questions using the firmware-trust path of an open GPS-denied UAV swarm research platform at commit `c040073969a6111f4c652d08e8de8629d70b2317`. The implementation defines a `FirmwareManifest`, `FirmwareTrustPolicy`, persistent `FirmwareTrustRecord`, and a deterministic validator. The application loads firmware evidence, applies profile-dependent checks, and terminates startup in hardened field or production profiles when trust validation fails. The boot-trust target is registered in the native CTest suite, and the repository validation report records the native test suite as green at the studied commit.

The contribution is not a claim of a new cryptographic signature primitive or a hardware secure-boot chain. The current code uses a keyed SHA3-derived manifest digest when a signing secret is configured, and it consumes `secure_boot_attested` and `bootloader_locked` as externally provided claims. This paper intentionally preserves that evidence boundary. The contribution is instead a formalization and security analysis of the implemented policy gate, including its anti-rollback logic, maintenance authorization, persistence rule, fail-closed startup behavior, and the assumptions required to elevate the software baseline into hardware-backed attestation.

The paper makes six contributions. First, it defines a compact mathematical model for firmware identity, policy, persistent state, and profile-dependent acceptance. Second, it derives a two-dimensional anti-rollback rule combining semantic version order with a monotonic counter. Third, it proves rejection properties for rollback, unauthorized maintenance, untrusted signer, invalid keyed digest, insecure boot claims, and disallowed measurements under a protected-state assumption. Fourth, it maps the formal rules to the C++ implementation and startup sequence. Fifth, it separates executed repository evidence from source-path analysis and from future hardware-backed claims. Sixth, it identifies implementation-hardening requirements that are easy to miss in a purely descriptive treatment, notably protected persistence and mandatory cryptographic secret configuration for hardened profiles.

2 Background and Related Work

2.1 Firmware resilience and secure update architecture

NIST SP 800-193 treats platform firmware as a privileged security layer and recommends mechanisms to protect firmware from unauthorized modification, detect corruption, and recover to a known-good state [1]. For autonomous UAVs, the same principles apply to bootloaders, flight-control binaries, sensor-interface firmware, and the operating environment that hosts autonomy software. A firmware compromise can persist below application-level defenses and can therefore invalidate assumptions made by higher layers.

The IETF SUIT architecture addresses secure firmware updates for constrained IoT systems and emphasizes protected manifests, authenticated metadata, version handling, and transport independence [2]. Although the implementation studied here is not a SUIT implementation, the manifest-oriented design is conceptually aligned: update identity and trust are expressed through metadata that can be checked before activation. Uptane provides a more elaborate update framework for vehicles and explicitly models rollback, freeze, mix-and-match, and arbitrary-software attacks [5]. These attack classes transfer naturally to aerial edge platforms, where software revisions may be distributed over wireless or intermittently connected infrastructure.

Falas et al. proposed a modular end-to-end secure firmware-update framework for embedded systems using cryptographic and hardware primitives [6]. Their work highlights an important design tension: stronger security mechanisms must remain feasible for resource-constrained devices. The present study occupies a different point in the design space. Rather than proposing a complete update-distribution protocol, it focuses on the local acceptance boundary immediately before a UAV node starts trusted operation.

2.2 Attestation and UAV trust establishment

Remote attestation allows a verifier to reason about the software or firmware state of a prover. The IETF RATS architecture separates the roles of attester, verifier, relying party, evidence, endorsements, and appraisal policy [3]. RFC 9683 further describes remote integrity verification for devices containing TPMs or equivalent protected capabilities [4]. These standards are important because they clarify that a Boolean statement such as “secure boot passed” is not, by itself, a trustworthy attestation. The claim becomes meaningful only when it is produced and protected by a trusted attesting environment and evaluated against known reference values.

UAV research has explored lightweight authentication-attestation protocols because a swarm can contain many resource-constrained nodes. Bansal and Sikdar proposed a secure and trusted attestation protocol for UAV fleets [7]. SHOTS used an optimal communication trajectory and physical unclonable functions to establish trust in UAV swarms [8]. Scalable topologies for time-optimal authentication further demonstrate the importance of minimizing trust-establishment delay as swarm size grows [9]. These works generally focus on distributed or remote trust establishment. The present paper instead studies the local firmware admission policy that must hold even before a node is considered eligible to participate in such protocols.

2.3 UAV firmware attack surface

UAV cybersecurity surveys identify firmware modification, insecure update paths, weak authentication, and compromised communication interfaces as recurring risks [10,11]. A 2026 firmware-acquisition study on consumer drones demonstrated practical extraction and validation of firmware images and found identifiable software components and limited binary hardening in examined devices [14]. That type of evidence illustrates why boot-time trust must not be reduced to a version string or vendor label.

Firmware security has also been studied in UAV-assisted update systems. Seo et al. proposed a blockchain-based secure firmware update approach using a UAV in IoT environments [15]. The goal differs from the current work, but both studies emphasize that update provenance and update state are part of the trust chain. For swarms, recent work on firmware attestation in IoT swarms uses graph-based models and volatile memory to detect anomalous firmware behavior [12,13]. Such runtime techniques complement, rather than replace, boot-time admission checks.

2.4 Research gap and scope

Existing standards define strong goals, and remote-attestation research provides cryptographic methods for trustworthy evidence. What is less frequently exposed in detail is the exact local policy logic that decides whether an autonomous node should boot, enter maintenance, or terminate before mission software starts. This paper focuses on that enforcement boundary.

The studied implementation is deliberately modest in cryptographic scope. It does not claim a TPM, TEE, immutable boot ROM, hardware monotonic counter, asymmetric code signing, or remote verifier. Instead, it implements a software policy that can consume those claims when available. This distinction permits a more useful security analysis because it makes the minimum assumptions explicit.

3 System Context and Evidence Boundary

3.1 Firmware manifest

A candidate firmware image is represented by the manifest

$$M = (v, \mu, s, \sigma, a, b, r),$$

where v is the semantic version string, μ is the firmware measurement, s is the signer identifier, σ is the manifest integrity field, $a \in \{0,1\}$ indicates a secure-boot attestation claim, $b \in \{0,1\}$ indicates a bootloader-lock claim, and $r \in \mathbb{N}_0$ is the rollback counter.

The implementation initializes default values suitable for laboratory execution and allows the manifest to be loaded from a file or overridden by environment configuration. In field or production profiles, an unspecified measurement is converted to an explicitly untrusted marker (`unsigned-local-build`), which prevents a hardened node from silently accepting a local development image.

Table 1 summarizes the manifest fields and the role each one plays.

Field	Symbol	Security role	Current evidence source
Firmware version	v	Semantic anti-rollback ordering	Manifest / environment
Measurement	μ	Firmware identity and admissibility	Manifest / environment
Signer	s	Provenance allowlist	Manifest / environment
Integrity field	σ	Keyed manifest integrity when secret configured	Manifest / environment
Secure-boot claim	a	Boot-chain predicate	External Boolean claim
Bootloader-lock claim	b	Bootloader state predicate	External Boolean claim
Rollback counter	r	Monotonic anti-rollback state	Manifest / environment

3.2 Trust policy and persistent state

The policy is modeled as

$$P = (p, S, k, m, \tau, f),$$

where p is the security profile, S is the trusted signer set, k is the configured manifest-integrity secret, $m \in \{0,1\}$ denotes maintenance mode, τ is the maintenance approval token, and f identifies the persistent state record.

The stored trust record is

$$R^\star = (r^\star, v^\star, \mu^\star),$$

where $r^\star$ and $v^\star$ are the most recently accepted rollback counter and firmware version, and $\mu^\star$ is the corresponding measurement.

The current implementation writes the accepted record to a filesystem path after successful hardened validation. Therefore, $R^\star$ is a security-critical object. Anti-rollback guarantees hold only if an attacker cannot arbitrarily restore or modify this state.

3.3 Security profiles

Let the profile indicator be

$$H(p) = \begin{cases} 1, & p \in \{\text{field,production}\}, \\ 0, & \text{otherwise.} \end{cases}$$

The implementation uses laboratory behavior when $H(p) = 0$. In laboratory mode, the validator records the available evidence but does not require the hardened trust chain. In hardened profiles, failed trust validation terminates node startup. This distinction prevents development convenience from being mistaken for production assurance.

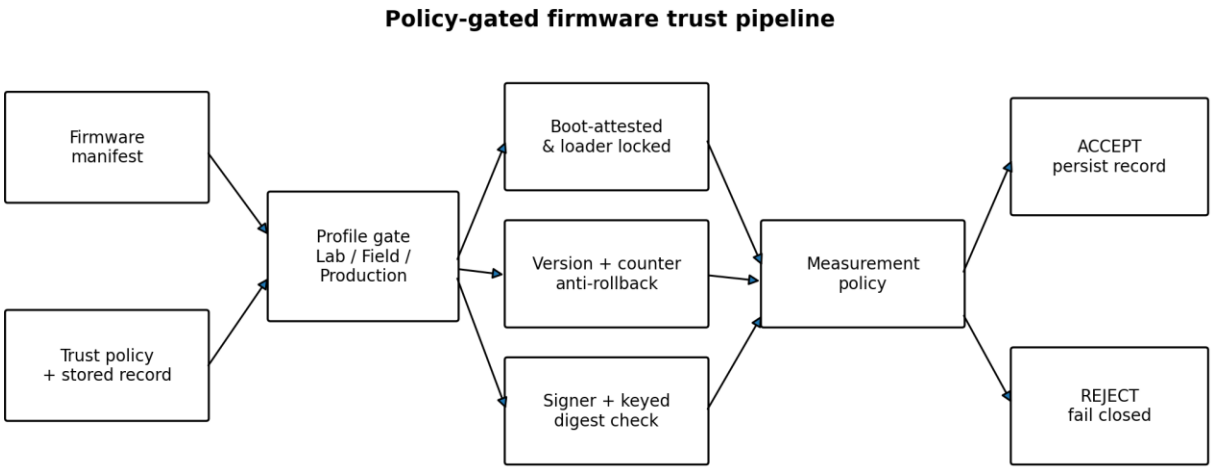


Figure 1. Policy-gated firmware trust pipeline. The hardened path evaluates boot claims, anti-rollback state, signer/integrity evidence, and measurement policy before persistence and startup authorization.

3.4 Evidence boundary

A central methodological point is that the current software does not independently establish a hardware root of trust. The values a and b are supplied to the process, and the measurement μ is a string value rather than a measurement read from a protected PCR or secure element. Consequently, the implemented validator is best understood as an **attestation-aware policy consumer**, not a complete attester.

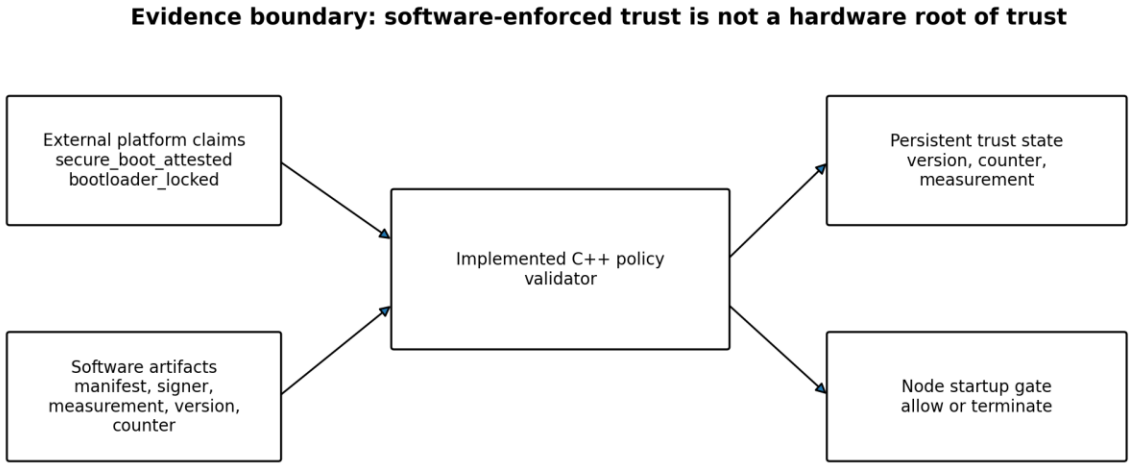


Figure 2. Evidence boundary. The C++ policy engine consumes external platform claims and software artifacts, enforces a deterministic policy, persists accepted state, and gates node startup. Hardware-backed measurement and protected attestation are outside the current implementation.

This boundary is consistent with the RATS view that trustworthy evidence depends on a protected attesting environment [3]. It also provides a clear migration path: external Booleans can later be replaced by verified TPM/TEE evidence without changing the higher-level acceptance logic.

4 Threat Model

4.1 Adversary capabilities

The analysis considers an attacker who may obtain or replay previously valid firmware metadata, modify an unprotected manifest, propose an unauthorized signer, submit an invalid integrity field, attempt maintenance without approval, or request execution of a lower firmware version. The attacker may also attempt to exploit configuration mistakes that weaken the trust gate.

The baseline threat set is

$$\mathcal{T} = \{T_{\text{rollback}}, T_{\text{freeze}}, T_{\text{signer}}, T_{\text{manifest}}, T_{\text{boot}}, T_{\text{measure}}, T_{\text{maint}}, T_{\text{state}}\}.$$

`T_rollback` represents a direct downgrade to an older version or lower rollback counter. `T_freeze` represents continued use of a valid but stale firmware image. `T_signer` represents provenance from an unauthorized signer. `T_manifest` represents modification of manifest fields. `T_boot` represents false or absent secure-boot and bootloader-lock evidence. `T_measure` represents a missing or development-only measurement in hardened mode. `T_maint` represents unauthorized entry into the maintenance workflow. `T_state` represents rollback or modification of the local persistent trust record.

The first seven threats are directly addressed by the software policy under appropriate configuration. `T_state` is an explicit boundary condition because the current state record is a filesystem object, not a hardware monotonic counter.

4.2 Trusted computing assumptions

The strongest properties in this paper assume

$$A_1 : k \text{ remains secret and non-empty in hardened profiles,}$$

$$A_2 : R^\star \text{ cannot be rolled back or modified by the firmware-update adversary,}$$

and

$$A_3 : a, b, \mu \text{ faithfully represent platform evidence.}$$

Assumption A_1 is particularly important because the current source code treats any non-empty `signature` field as acceptable when no signing secret is configured. This behavior is convenient for non-hardened experimentation but should not be considered a cryptographic trust check in production. A secure deployment therefore needs an external configuration invariant requiring a non-empty protected secret, or preferably a migration to asymmetric signatures verified against an embedded public-key root.

Assumption A_2 is needed for durable anti-rollback. If the adversary can replace `firmware_trust.state` with an older copy, then both $v^\star$ and $r^\star$ can regress, defeating the local monotonicity check. A TPM NV counter, secure element, RPMB-backed storage, or another rollback-resistant state mechanism would strengthen this boundary.

Assumption A_3 separates policy evaluation from evidence production. The implementation can reject `secure_boot_attested=false`, but it cannot prove that a `true` value originated from immutable boot ROM or a trusted attester.

4.3 Threat-to-control map

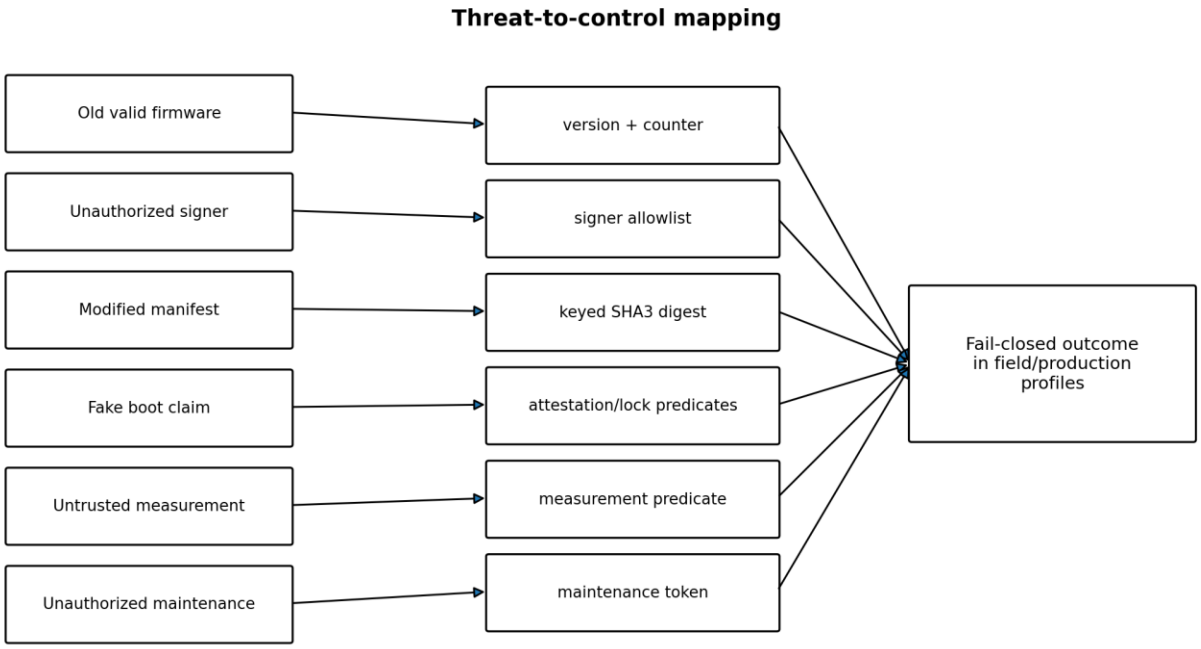


Figure 3. Threat-to-control mapping. Each threat is paired with a local policy predicate; field and production profiles terminate startup when the resulting trust report is rejected.

Threat	Policy control	Source-level outcome
Old valid firmware	Version + rollback counter	ROLLBACK_REJECTED
Unauthorized signer	Signer allowlist	SIGNER_REJECTED
Modified manifest	Keyed SHA3-derived digest when $k \neq \emptyset$	SIGNATURE_INVALID
Missing boot trust	Secure-boot and bootloader predicates	SECURE_BOOT_FAILED
Development/empty measurement	Measurement admissibility	MEASUREMENT_UNTRUSTED
Maintenance without approval	Maintenance token predicate	MAINTENANCE_UNAUTHORIZED
State-file rollback	Not fully prevented by current software	Requires protected persistence

5 Formal Firmware Trust Model

5.1 Semantic version order

Let $\preccurlyeq_v$ denote the component-wise semantic-version order implemented by parsing dot-separated integer components and padding to at least three fields. A candidate version passes the version check if

$$V(M, R^\star) = \mathbf{1}[v^\star \preccurlyeq_v v].$$

The implementation compares each integer component in order. Therefore, `2.1.0` is newer than `2.0.5`, while equal versions are permitted if the rollback counter does not regress.

5.2 Monotonic rollback counter

The counter predicate is

$$C(M, R^\star) = \mathbf{1}[r \geq r^\star].$$

The combined anti-rollback predicate is

$$R(M, R^\star) = V(M, R^\star) \wedge C(M, R^\star).$$

A candidate is rejected if either dimension regresses. This dual check reduces dependence on semantic version naming alone. A project can increment r even when version semantics are complex, while v provides a human-readable ordering check.

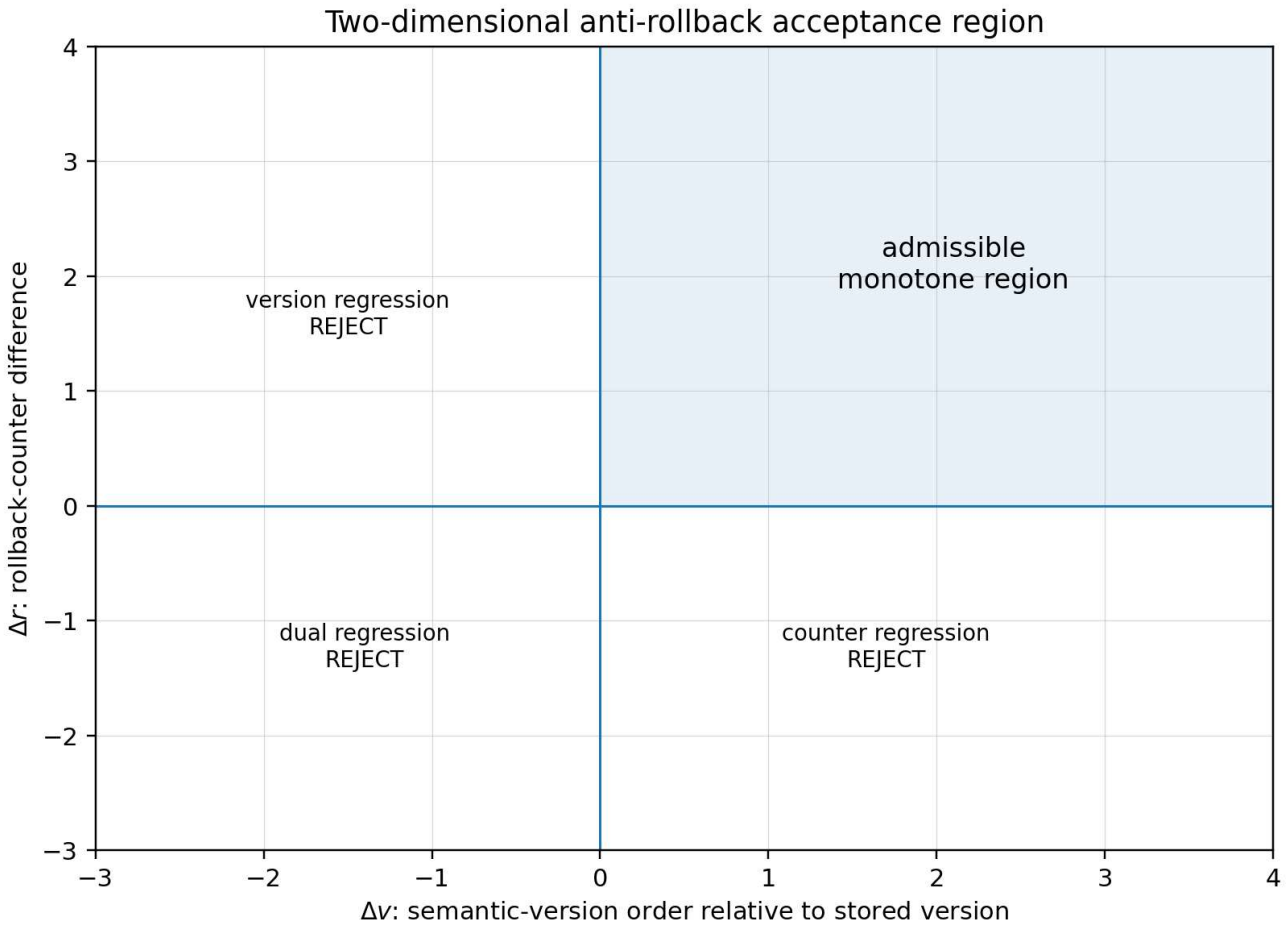


Figure 4. Two-dimensional anti-rollback acceptance region. Only candidates that are non-decreasing in both semantic version order and rollback counter lie in the admissible region.

5.3 Signer predicate

Let S be the configured signer allowlist. The implementation treats an empty list as unrestricted, otherwise the signer must belong to S. Formally,

$$G_s(M, P) = \mathbf{1}[S = \varnothing \vee s \in S].$$

For hardened deployment, the recommended invariant is

$$H(p) = 1 \Rightarrow S \neq \varnothing.$$

The main application configures `release-ca,maint-ca` by default, which satisfies this invariant unless operators override the setting.

5.4 Keyed manifest-integrity field

The implementation computes a SHA3-256 digest over a canonical concatenation of the secret and manifest fields:

$$d = \mathrm{SHA3}_{256}(k \parallel \mathrm{LF} \parallel v \parallel \mathrm{LF} \parallel \mu \parallel \mathrm{LF} \parallel s \parallel \mathrm{LF} \parallel a \parallel \mathrm{LF} \parallel b \parallel \mathrm{LF} \parallel r).$$

When $k \neq \emptyset$, the integrity predicate is

$$G_\sigma(M, P) = \mathbf{1}[\sigma = d].$$

This construction is a keyed digest prototype. It should not be described as a public-key digital signature, and it is not implemented using the standardized HMAC transform. A production hardening path should replace it with a standard MAC or, preferably for software supply-chain provenance, an asymmetric code-signing scheme.

The current source also contains a fallback:

$$k = \emptyset \Rightarrow G_\sigma = \mathbf{1}[\sigma \neq \emptyset].$$

Equation (16) is not a cryptographic authenticity guarantee. Therefore, the formal hardened model strengthens the implementation with the configuration requirement

$$H(p) = 1 \Rightarrow k \neq \emptyset.$$

This is one of the principal findings of the source-level security analysis.

5.5 Secure-boot and bootloader predicates

The boot predicate is

$$B(M) = a \wedge b.$$

A hardened candidate with either missing secure-boot attestation or an unlocked bootloader is rejected. In the current system this is a policy check over supplied evidence. A hardware-backed version would replace a and b with verifier outputs derived from signed evidence or measured boot.

5.6 Measurement admissibility

Let $\mathcal{U}$ be the set of disallowed measurement patterns in the current implementation:

$$\mathcal{U} = \{\mu\text{: contains “unsigned”}\} \cup \{\mu\text{: contains “lab-local”}\} \cup \{\epsilon\}.$$

Then

$$Q(M) = \mathbf{1}[\mu \notin \mathcal{U}].$$

This is intentionally a lightweight admission rule, not a cryptographic reference-value database. It prevents accidental use of obvious development markers in hardened operation, but it does not prove that an arbitrary non-empty measurement corresponds to a known-good binary. A mature implementation should compare a cryptographic measurement against signed reference values.

5.7 Maintenance authorization

Let m denote maintenance mode and τ the approval token. The implemented maintenance predicate is

$$A_m(P) = \neg m \vee \mathbf{1}[\tau \neq \epsilon].$$

If maintenance mode is active without a token, validation terminates with `MAINTENANCE_UNAUTHORIZED`. With a token, validation continues through the normal hardened boot, rollback, signer, integrity, and measurement checks. This composition is important: maintenance authorization does not bypass the entire trust gate.

5.8 Hardened acceptance function

Combining the predicates, the strengthened hardened acceptance model is

$$\mathcal{A}_H(M, P, R^\star) = A_m(P) \wedge B(M) \wedge R(M, R^\star) \wedge G_s(M, P) \wedge G_\sigma(M, P) \wedge Q(M) \wedge \mathbf{1}[k \neq \emptyset].$$

The implemented acceptance function is nearly identical except for the final non-empty-secret requirement, which is currently a deployment invariant rather than an in-code rejection branch.

For laboratory profiles,

$$\mathcal{A}_L(M, P, R^\star) = A_m(P),$$

because the validator intentionally bypasses hardened boot trust after maintenance-token handling. This mode is suitable for simulation and development but must not be interpreted as production firmware assurance.

5.9 Persistent-state commit

Let $\mathrm{commit}(M)$ update the stored record to the candidate values:

$$R^\star_{t+1} = (r_t, v_t, \mu_t).$$

The implementation performs this commit only after all hardened checks pass. Thus,

$$\mathcal{A}_H = 1 \Rightarrow R^\star_{t+1} = R(M_t),$$

while

$$\mathcal{A}_H = 0 \Rightarrow R^\star_{t+1} = R^\star_t.$$

This "validate before persist" ordering prevents a rejected candidate from poisoning the anti-rollback reference state.

6 Security Properties

6.1 Property 1: rollback rejection under protected state

Proposition 1. Assume A_2. If a candidate firmware has either $v \prec_v v^\star$ or $r < r^\star$, then $\mathcal{A}_H = 0$.

Argument. By Eq. (11), $R = V \wedge C$. A regression in either version or counter makes one conjunct zero, therefore the complete hardened conjunction in Eq. (22) is zero. The source maps this outcome to `ROLLBACK_REJECTED`.

This property blocks both direct version downgrade and counter regression. It also limits freeze attacks after an accepted upgrade, provided the attacker cannot roll back the stored trust record.

6.2 Property 2: provenance rejection

Proposition 2. If $S \neq \varnothing$ and $s \notin S$, then $\mathcal{A}_H = 0$.

This follows directly from Eq. (12). The property is independent of the manifest integrity field; even a correctly keyed digest from an untrusted signer identifier is rejected.

6.3 Property 3: keyed manifest tamper detection

Proposition 3. Assume A_1 and collision resistance of SHA3-256. If an adversary changes any bound manifest field without knowledge of k, then the probability of preserving Eq. (15) is negligible under the digest model.

The bound fields include version, measurement, signer, boot claims, and rollback counter. Consequently, a correctly configured keyed digest makes these fields mutually integrity-coupled. The implementation should still migrate to a standard authenticated construction for production use.

6.4 Property 4: maintenance non-bypass

Proposition 4. In maintenance mode, a non-empty token is necessary but not sufficient for hardened acceptance.

From Eq. (21), $\tau \neq \epsilon$ permits the validator to continue. Equations (18), (11), (12), (15), and (20) must still hold. Therefore an operator cannot authorize an insecure-boot, rollback, untrusted-signer, invalid-integrity, or untrusted-measurement image merely by opening a maintenance window.

6.5 Property 5: fail-closed startup

Let N denote entry into normal node initialization. The main application enforces

$$H(p) = 1 \wedge \mathcal{A}_H = 0 \Rightarrow N = 0.$$

The process logs the firmware trust failure and returns before sensors, VIO, swarm networking, and other mission subsystems are started. This is a stronger control than merely marking the node degraded after it has joined the swarm.

6.6 Property 6: monotonic accepted history

Under A_2, every accepted hardened transition satisfies

$$v^\star_{t+1} \succcurlyeq_v v^\star_t$$

and

$$r^\star_{t+1} \geq r^\star_t.$$

Therefore the sequence of accepted states forms a partial monotone history in the (v, r) plane.

6.7 State-file attack limitation

If A_2 is removed, Proposition 1 no longer provides durable rollback protection. Suppose an attacker restores a prior record

$$\tilde{R}^\star = (\tilde{r}^\star, \tilde{v}^\star, \tilde{\mu}^\star)$$

with

$$\tilde{r}^\star < r^\star, \qquad \tilde{v}^\star \prec_v v^\star.$$

A firmware image that is globally stale but newer than $\tilde{R}^{\star}$ may then satisfy Eq. (11). This shows why anti-rollback state must itself be rollback resistant. The current filesystem record is sufficient for software-level experiments but not equivalent to a hardware monotonic counter.

7 Implementation Mapping

7.1 C++ trust structures

The implementation is concentrated in `include/security/FirmwareTrust.hpp`. Four structures define the data model: `FirmwareManifest`, `FirmwareTrustPolicy`, `FirmwareTrustRecord`, and `FirmwareTrustReport`. The validator is deterministic and side-effect free until the final accepted-state persistence step.

The manifest signature helper serializes the configured secret and manifest fields into a newline-delimited payload and computes a 256-bit SHA3 digest. On Windows, broader project cryptography uses native platform APIs in other security components, but the firmware-trust helper itself uses the bundled SHA3 implementation. The function name `sign_firmware_manifest` reflects project semantics; cryptographically, the current operation is more accurately described as a keyed digest.

7.2 Startup integration

The application startup sequence performs the following operations before mission subsystems are brought online:

1. normalize the security profile;
2. load a manifest file when configured;
3. apply environment overrides for version, measurement, signer, integrity field, rollback counter, secure-boot claim, and bootloader-lock claim;
4. construct the trust policy, including allowed signers, state-file path, signing secret, maintenance mode, and approval token;
5. call `validate_firmware_trust`;
6. terminate startup when validation fails under `field` or `production`;
7. instantiate the runtime security monitor only after firmware trust is resolved.

This ordering is security-relevant because a rejected production node does not initialize the sensor, VIO, swarm, or autonomy subsystems.

7.3 Profile-safe defaults

In hardened profiles, if no firmware measurement is supplied, the application sets

$$\mu = \text{“unsigned-local-build”}.$$

Because this string belongs to $\mathcal{U}$, Eq. (20) rejects it. This is a useful fail-safe default: forgetting to configure a production measurement results in denial rather than implicit trust.

The allowed signer list defaults to `release-ca,maint-ca`, making the provenance gate non-empty under standard startup configuration. Conversely, the signing secret defaults to empty unless explicitly supplied, which motivates the hardened configuration invariant in Eq. (17).

7.4 Maintenance workflow

Maintenance mode changes the report state to `maintenance-window-open` and requires a non-empty approval token. If authorized, the update state becomes `maintenance-authorized`, after which normal hardened validation continues. On success, the boot state is `MAINTENANCE_TRUSTED`; otherwise the relevant failure state is returned.

This structure separates **maintenance authorization** from **firmware authorization**. The former establishes that a maintenance operation is permitted; the latter still establishes that the candidate firmware is trusted.

8 Validation Methodology

8.1 Evidence classes

To avoid overstating the implementation, results are divided into three evidence classes.

E1 — source-verified behavior. The acceptance branches and startup integration are directly visible in the C++ source at the studied commit.

E2 — test-asserted behavior. `tests/test_boot_trust.cpp` defines explicit unit cases and expected outcomes.

E3 — repository validation status. The project validation report records `ctest` among the validation commands and reports Phase 1–9 regression preservation with native tests remaining green. The CMake test configuration registers `test_boot_trust` as a native security unit target.

This paper does not infer physical-flight validation, secure-element measurements, TPM evidence, or production OTA behavior from those software tests.

8.2 Registered boot-trust tests

The source includes three boot-trust unit tests:

- **Trusted production manifest.** Version `2.1.0`, secure measurement, signer `release-ca`, secure-boot attestation, locked bootloader, rollback counter 4, trusted signer list, and keyed digest are expected to produce `SECURE_BOOT_TRUSTED`.
- **Rollback regression.** After accepting `2.1.0` with counter 4, the candidate `2.0.5` with counter 3 is expected to produce `ROLLBACK_REJECTED`.
- **Unauthorized maintenance.** Field profile with maintenance mode enabled and no approval token is expected to produce `MAINTENANCE_UNAUTHORIZED`.

Table 3 records these scenarios without adding unmeasured latency or throughput claims.

Scenario	Candidate condition	Expected decision in test source	Evidence class
Trusted production boot	All hardened predicates satisfied	Accept / SECURE_BOOT_TRUSTED	E2 + E3
Dual rollback	$v \prec v^{\star}$ and $r < r^{\star}$	Reject / ROLLBACK_REJECTED	E2 + E3
Unauthorized maintenance	$m = 1, \tau = \epsilon$	Reject / MAINTENANCE_UNAUTHORIZED	E2 + E3

8.3 Source-path branch analysis

Several additional rejection paths are present in the validator even though dedicated test cases are not shown in `test_boot_trust.cpp`. These include missing secure boot or bootloader lock, untrusted signer, invalid keyed digest, and untrusted measurement. Their existence is source verified, but this paper labels them E1 rather than claiming executed unit coverage.

Rejection state	Trigger	Evidence
SECURE_BOOT_FAILED	$a = 0$ or $b = 0$ in hardened profile	E1
ROLLBACK_REJECTED	version or counter regression	E1, E2, E3
SIGNER_REJECTED	signer not allowed	E1
SIGNATURE_INVALID	keyed digest mismatch	E1
MEASUREMENT_UNTRUSTED	empty, unsigned, or lab-local measurement	E1
MAINTENANCE_UNAUTHORIZED	maintenance enabled without token	E1, E2, E3

8.4 Reproducibility

The studied implementation is available in the public repository `smshagor-dev/UVA-GPS-Denied-Navigation-in-Dynamic-Environments`. The evidence boundary in this paper corresponds to commit `c040073969a6111f4c652d08e8de8629d70b2317`. Relevant files are:

- `include/security/FirmwareTrust.hpp`;
- `tests/test_boot_trust.cpp`;
- `tests/CMakeLists.txt`;
- `src/main.cpp`;
- `docs/enterprise-deployment-security-readiness/VALIDATION_REPORT.md`.

These paths make the policy rules and claimed test evidence independently inspectable.

9 Results

9.1 Acceptance and rejection semantics

The software implements a clear fail-closed hierarchy for hardened profiles. The primary result is not a benchmark number but an authority boundary: a failed firmware report prevents the UAV node from continuing into mission initialization. This property reduces the chance that a node with untrusted firmware can join a swarm and then rely on higher-level security to compensate.

The accepted production path requires all implemented hardened predicates. When accepted outside maintenance, the report state is `SECURE_BOOT_TRUSTED`; during an authorized maintenance workflow, it is `MAINTENANCE_TRUSTED`. Rejection states are specific enough to support operator diagnosis without conflating all failures into one generic error.

9.2 Anti-rollback behavior

The explicit unit test demonstrates a stateful anti-rollback sequence. First, version `2.1.0` with rollback counter 4 is accepted and persisted. A subsequent candidate with version `2.0.5` and counter 3 is rejected. This test exercises both dimensions simultaneously.

The formal model shows that the implementation is stronger than a counter-only or version-only scheme under the protected-state assumption. Let

$$\Delta_v = \mathrm{cmp}(v, v^\star), \qquad \Delta_r = r - r^\star.$$

The candidate is anti-rollback admissible iff

$$\Delta_v \geq 0 \wedge \Delta_r \geq 0.$$

Thus the accepted region is the non-negative quadrant shown in Figure 4. This provides a compact method for reasoning about update histories and test generation.

9.3 Maintenance behavior

The maintenance test confirms that simply setting maintenance mode does not open the trust gate. Without an approval token, the report is rejected before hardened validation can complete. Source analysis further shows that with a token, the candidate must still pass the secure-boot, anti-rollback, signer, integrity, and measurement checks.

This prevents a common policy error in which "maintenance mode" becomes an unrestricted bypass path.

9.4 Configuration-sensitive integrity

The source analysis reveals that manifest integrity has two distinct operational regimes:

$$G_\sigma = \begin{cases} \mathbf{1}[\sigma = d], & k \neq \emptyset, \\ \mathbf{1}[\sigma \neq \emptyset], & k = \emptyset. \end{cases}$$

The first regime provides keyed tamper detection under the stated assumptions. The second provides only syntactic presence. Therefore, a hardened deployment that fails to configure `DRONE_FIRMWARE_SIGNING_SECRET` does not obtain the same integrity property as the unit-tested production example.

This finding is operationally important because the test fixture explicitly configures a secret, whereas the main application's default is empty. The paper therefore recommends treating Eq. (17) as a mandatory production configuration rule or moving the requirement into the validator itself.

9.5 Coverage against firmware threats

A qualitative coverage score can be defined over the seven software-addressable threats:

$$C_{\text{soft}} = \frac{|\{T_i \in \mathcal{T}_{\text{soft}} : \text{explicit reject path exists}\}|}{|\mathcal{T}_{\text{soft}}|}.$$

For

$$\mathcal{T}_{\text{soft}} = \{T_{\text{rollback}}, T_{\text{signer}}, T_{\text{manifest}}, T_{\text{boot}}, T_{\text{measure}}, T_{\text{maint}}\},$$

the source contains an explicit policy branch for each class. Therefore the structural path coverage is complete for this selected threat set. This is **not** a measured attack-detection rate; it only states that each modeled threat has a corresponding source-level decision branch.

9.6 Standards alignment

Table 5 positions the implementation against broader firmware-trust principles.

Principle	NIST/SUIT/RATS/Uptane expectation	Current software baseline	Gap
Firmware authenticity	Cryptographic provenance	Signer allowlist + keyed SHA3 digest when configured	Prefer asymmetric code signing
Integrity	Protected manifest / measured firmware	Manifest-bound digest + measurement string	Measurement not independently recomputed

Anti-rollback	Version/freshness protection	Version + persistent counter	State file not hardware protected
Secure boot	Rooted boot-chain verification	Boolean predicate consumed	External claim, not hardware attestation
Maintenance	Authorized recovery/update path	Token-gated maintenance plus normal checks	Token source not hardware protected
Recovery	Restore known-good firmware	Not implemented as recovery engine	Future A/B image or recovery partition
Remote attestation	Protected evidence to verifier	Not implemented	Future TPM/TEE/RATS integration

10 Discussion

10.1 Why firmware trust is separate from communication trust

A UAV can use strong packet authentication and still be unsafe if the firmware that verifies those packets is compromised. Conversely, a securely booted node can still be attacked over the network if command authorization is weak. Firmware trust and communication trust therefore form complementary layers.

The runtime cyber-resilience layer previously studied in the same platform controls command authority after startup. The present paper addresses an earlier boundary: whether the node should be trusted enough to start mission software at all. Keeping these questions separate avoids overloading one paper with unrelated contributions and makes the security argument easier to audit.

10.2 Dual monotonicity as a practical design choice

Semantic versions are meaningful to operators but can be ambiguous across release branches, prereleases, or vendor conventions. A monotonic counter is easy to compare but carries little semantic information. Combining both creates a simple two-dimensional guard.

The model can be extended to branch-aware versions by replacing $\preccurlyeq_v$ with an application-specific partial order. The essential property is that the accepted history must never move "backward" under the selected order while the rollback counter remains protected.

10.3 Need for protected persistent state

The strongest limitation is not the comparison logic; it is storage. Software anti-rollback is only as durable as the state that records prior acceptance. An attacker with sufficient privilege to restore the state file can create an artificial past in which older firmware again appears admissible.

This is precisely where platform firmware resilience transitions from software policy into hardware architecture. A production implementation should bind the accepted rollback counter to a protected monotonic store. Candidate mechanisms include TPM NV counters, secure-element monotonic counters, eMMC RPMB, or another authenticated non-rollbackable storage primitive.

10.4 From keyed digest to code signing

The current keyed SHA3 construction is useful for controlled laboratory validation because it binds all manifest fields to a secret. However, a shared secret is not ideal for a software supply chain. If the verification secret must exist on every UAV, compromise of one node can expose material that may permit forging manifests for others.

Asymmetric code signing provides a better separation: signing keys remain offline or in a protected release service, while UAV nodes contain only public verification keys. A future implementation should bind the same fields in Eq. (14) to an Ed25519, ECDSA, or post-quantum signature, depending on platform constraints and lifecycle requirements.

10.5 From Boolean boot claims to attestation evidence

The variables a and b are policy inputs, not cryptographic proofs. A complete attestation design would define evidence

$$E = (\text{measurement log,boot state,device identity,nonce,signature})$$

and a verifier function

$$\mathcal{V}(E, \mathcal{R}) \rightarrow \{0,1\},$$

where $\mathcal{R}$ contains endorsements and reference values. The policy would then set

$$a = \mathcal{V}(E, \mathcal{R})$$

rather than reading a plain environment value. This mapping would align the local trust gate with RATS terminology and permit remote fleet admission based on the same verified result.

10.6 Recovery is the missing third leg

The present mechanism covers **protection** through admission policy and some **detection** through rejection states. It does not implement a complete recovery process. NIST SP 800-193 explicitly treats recovery as a separate resilience objective [1].

A practical UAV recovery design could maintain an A/B firmware layout with a known-good recovery image and require that recovery itself satisfy an authenticated manifest policy. The state machine would then become

$$\text{candidate} \rightarrow \begin{cases} \text{boot trusted}, & \mathcal{A}_H = 1, \\ \text{recover known-good}, & \mathcal{A}_H = 0 \wedge \mathcal{R}_{kg} = 1, \\ \text{remain offline}, & \text{otherwise.} \end{cases}$$

Such a design would extend the current fail-closed behavior without allowing an untrusted image to execute.

11 Limitations and Future Work

The current study has several deliberate limitations.

First, the secure-boot and bootloader-lock fields are external claims. The study does not demonstrate an immutable boot ROM, TPM, TEE, secure element, or protected measurement register. The paper therefore does not claim hardware-rooted attestation.

Second, the firmware measurement is not recomputed from a binary within `FirmwareTrust.hpp`. It is supplied as metadata and filtered against obvious development markers. Future work should hash the candidate image and compare that digest against signed reference values.

Third, the persistent trust record is a normal file. Durable anti-rollback requires protected storage. A future hardware-in-the-loop implementation should evaluate TPM NV, RPMB, or secure-element counters and include power-failure and state-restoration tests.

Fourth, the keyed SHA3 construction is a prototype integrity mechanism rather than standardized code signing. Hardened deployment requires a non-empty secret, but the current validator does not itself reject an empty secret if a non-empty signature field is present. This should be fixed in code or replaced with public-key signature verification.

Fifth, the current boot-trust test file contains three direct cases. Additional dedicated tests should cover signer rejection, digest mismatch, missing secure-boot claim, unlocked bootloader, measurement rejection, version-only rollback, counter-only rollback, equal-version counter advance, maintenance success, persistence failure, and state corruption.

Sixth, no firmware-update transport, delta update, A/B partition recovery, power-loss robustness, remote attestation protocol, or fleet-wide rollout mechanism is evaluated here.

Seventh, the validation is software-oriented. No physical UAV free flight, production bootloader chain, secure-element key storage, RF-adversarial update transport, or airworthiness qualification is claimed.

Future work will therefore proceed along three tracks: (i) move integrity verification to asymmetric signed manifests; (ii) replace external boot claims and file-based counters with hardware-backed evidence and monotonic state; and (iii) integrate recovery and remote attestation so that a swarm can exclude nodes whose boot evidence no longer satisfies mission policy.

12 Conclusion

This paper presented a formal and source-grounded study of firmware trust and anti-rollback enforcement for autonomous UAV edge platforms. The studied implementation models firmware through version, measurement, signer, integrity field, boot-state claims, and a rollback counter; combines these with profile, signer, maintenance, and persistent-state policy; and refuses hardened node startup when trust validation fails.

The formal model shows that the current design provides useful software-level properties: two-dimensional rollback rejection, signer gating, maintenance non-bypass, manifest-field integrity when a secret is configured, measurement filtering, validate-before-persist ordering, and fail-closed startup. Repository evidence includes native boot-trust tests for trusted production acceptance, rollback rejection, and unauthorized maintenance rejection, together with a project validation report indicating the native regression suite remained green at the studied commit.

The analysis also identifies the precise boundary between software policy and platform trust. Anti-rollback is not durable if the persistent state can itself be rolled back. Secure-boot and bootloader values are not hardware evidence unless produced by a protected attester. The keyed digest is not equivalent to public-key code signing, and hardened operation should require a non-empty protected verification secret until that migration is complete.

These constraints do not negate the value of the current mechanism; they make its assurance level explicit. The resulting framework provides a reproducible firmware-admission baseline for UAV research and a mathematically clear path toward hardware-backed, remotely attestable, rollback-resistant autonomous flight platforms.

Declarations

Funding. No external funding was received for this study.

Conflict of interest. The author declares no conflict of interest.

Ethics approval. Not applicable.

Consent to participate. Not applicable.

Consent for publication. Not applicable.

Data and code availability. The software evidence analyzed in this study is available in the public GitHub repository `https://github.com/smshagor-dev/UVA-GPS-Denied-Navigation-in-Dynamic-Environments` at commit `c040073969a6111f4c652d08e8de8629d70b2317`. The principal evidence files are identified in Section 8.4.

Author contributions. M.S.I.S. conceived and designed the study, developed the research methodology and formal security model, performed the software and source-evidence analysis, interpreted the results, prepared the figures and tables, and wrote, reviewed, and approved the final manuscript.

Generative AI statement. During preparation of this manuscript, the author used OpenAI ChatGPT to assist with language drafting, mathematical presentation, and document formatting. All technical claims, source mappings, interpretations, and final text were reviewed and remain the responsibility of the author.

References

1. Regenscheid A. Platform Firmware Resiliency Guidelines. NIST Special Publication 800-193. National Institute of Standards and Technology; 2018. https://doi.org/10.6028/NIST.SP.800-193.
2. Moran B, Tschofenig H, Brown D, Meriac M. A Firmware Update Architecture for Internet of Things. RFC 9019. IETF; 2021.
3. Birkholz H, Thaler D, Richardson M, Smith N, Pan W. Remote ATtestation procedureS (RATS) Architecture. RFC 9334. IETF; 2023.
4. Fedorkow GC, Voit E, Fitzgerald-McKay J. Remote Integrity Verification of Network Devices Containing Trusted Platform Modules. RFC 9683. IETF; 2024.
5. Uptane. Uptane Standard for Design and Implementation, Version 2.1.0. Uptane Alliance; 2025.
6. Falas S, Konstantinou C, Michael MK. A Modular End-to-End Framework for Secure Firmware Updates on Embedded Systems. ACM Journal on Emerging Technologies in Computing Systems. 2022;18(1):1-19. https://doi.org/10.1145/3460234.
7. Bansal G, Sikdar B. Secure and Trusted Attestation Protocol for UAV Fleets. In: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops. 2022:1-6.
8. Bansal G, Naren N, Chamola V, Sikdar B. SHOTS: Scalable Secure Authentication-Attestation Protocol Using Optimal Trajectory in UAV Swarms. IEEE Transactions on Vehicular Technology. 2022;71(6):5827-5836. https://doi.org/10.1109/TVT.2022.3162226.
9. Bansal G, Chamola V, Ansari N, Sikdar B. Scalable Topologies for Time-Optimal Authentication of UAV Swarms. IEEE Network. 2022;36(6):126-132. https://doi.org/10.1109/MNET.008.2100472.
10. Kumar N, Chaudhary A. Surveying cybersecurity vulnerabilities and countermeasures for enhancing UAV security. Computer Networks. 2024. https://doi.org/10.1016/j.comnet.2024.110695.
11. Toward secure complex UAV cyber-physical systems: A unified threat taxonomy and cross-layer survey of cybersecurity challenges. Internet of Things. 2026;37:101902. https://doi.org/10.1016/j.iot.2026.101902.
12. Rouagubi A, El Youssofi C, Chougdali K. Firmware Attestation in IoT Swarms Using Relational Graph Neural Networks and Static Random Access Memory. AI. 2025;6(7):161. https://doi.org/10.3390/ai6070161.
13. Kohli V, Kohli B, Aman MN, Sikdar B. Swarm-Net: Firmware Attestation in IoT Swarms using Graph Neural Networks and Volatile Memory. arXiv:2408.05680; 2024.
14. More S, Sudhakaran S, Carvalho M. A Multi-Interface Firmware Acquisition and Validation Methodology for Low-Cost Consumer Drones: A Case Study on Three Holy Stone Platforms. arXiv:2605.11040; 2026.
15. Seo JW, Islam A, Masuduzzaman M, Shin SY. Blockchain-Based Secure Firmware Update Using an UAV. Electronics. 2023;12(10):2189. https://doi.org/10.3390/electronics12102189.
16. Banks AS, Kisiel M, Korsholm P. Remote Attestation: A Literature Review. arXiv:2105.02466; 2021.
17. MITRE. EMB3D MID-030: Firmware Rollback Protections. MITRE Embedded Device Threat Matrix; 2026.
18. Shagor MSI. UVA GPS Denied Navigation in Dynamic Environments. GitHub repository, commit c040073969a6111f4c652d08e8de8629d70b2317; 2026.