Typed Non-Laundering Authorization for Signed Deterministic-Program Envelopes

Anonymous Author(s)

Abstract

We study authenticated recovery of signed deterministic programs over channels that do not preserve bytes. We separate signed programs from signed execution envelopes and define a typed non-laundering authorization design over complete current report sets from a correctly curated verifier store. This prevents consumer-side report subset selection, not administrative omission or silent supersession. The executable model is exhaustively tested over its finite policy language. Policy can grant reasoned origin-only use, but cannot rename evidence or derive execution-agreement authority from a non-agreement aggregate. Independence policy receives only an eight-bit verified projection; a `Distinct` axis qualifies only with registry-verified, externally attested, or verifier-established provenance and its required evidence reference.

On a bounded surface of the real deterministic DSL L, a 160-cell oracle separates program-only A0, authentication-only A1, final-result re-execution B, and extended-observable transcript scope C. The implementation matched every defined cell oracle. Observation-divergent stale-3 is unconstructible at A0, missed at A1, and detected throughout B/C; one controlled Rust-runtime fault is detected only across a distinct backend. Separately, a test-only bridge fed harness-emitted report JSON through the existing finite store and authorization APIs, where four valid sequences matched their expected outcomes and one malformed agreement was rejected before aggregation. This is not a production end-to-end path or conformance result. These are bounded mechanism results, not publisher-execution evidence, whole-language validation, or source-independent replication.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

authorization, attestation, deterministic re-execution, provenance envelopes, non-laundering

1 Introduction

Authenticated origin must be separated from decision authority. For signed deterministic programs, a recipient must distinguish the exact program and context, an attributed execution claim, local deterministic agreement with that claim, and authority for an action. A signature authenticates the claim rather than proving that the computation occurred or was correct. Re-execution supplies only case-specific agreement, and shared faults or unsupported profiles remain possible. Treating these states as interchangeable launders authority.

DSSE, in-toto, SLSA, and Sigstore establish exact-message or provenance facts [7, 11, 12, 14], but not verifier-side execution agreement or the later authorization decision. We encode that downstream non-laundering boundary in a typed state model and test the incremental detection value of re-execution on real engines.

Consider a package registry that receives a signed migration program plus a publisher-attributed execution result. After authentication, the registry may display that result as a publisher statement, but production rollout requires verifier re-execution under a pinned profile and the policy's required verified independence axes. If the verifier engine is unavailable, a quarantined low-risk dry-run workflow may proceed only with an explicit `origin_only(engine_unavailable)` basis. If an unsuperseded later report disagrees, the next aggregate from that correctly curated store has no grant, so the action layer must withdraw that fallback before rollout.

`SignedProgramV1` binds canonical L IR, semantics, and context. `SignedExecutionEnvelopeV1` adds a versioned engine-attributed claim. `VerificationStatus` records terminal execution evidence and `AuthorizationBasis` a later grant without rewriting it. Under a correctly curated verifier store, authorization consumes the complete current-report aggregate for one envelope, context, scope, and epoch, preventing consumer-side report subset selection. A store administrator who silently supersedes a disagreement can achieve the same effect as cherry-picking; store curation remains outside the model. Theorem 1 states the resulting transition-level authorization invariant. The executable abstraction matched every defined oracle in its finite conjunctive-policy domain and rejected three laundering mutations.

The real-engine matrix spans four families and five paths on a bounded L surface. I3 compares the native reference with a cross-language transliteration that shares source, parser, and semantic-core lineage but separates runtime backend and build toolchain. The implementation matched all defined condition oracles.

1.1 Contributions

This paper makes two contributions in claim order:

(1) **A typed non-laundering authorization design.** Split signed types, verifier-owned joins, typed fallback reasons, and an eight-axis verified projection constrain policy to status-preserving transitions. Theorem 1 records the type-safe invariant; the executable abstraction matched every defined oracle over its finite policy, join, provenance, and report-metadata domains and rejected three mutations.
(2) **A bounded real-engine mechanism result.** The implementation matched every oracle in a defined 160-cell matrix, and four programs across five paths yielded byte-identical canonical transcripts on rerun. A0/A1/B/C isolate claim existence, binding, and re-execution.

2 System Model and Verification Pipeline

2.1 Roles and data flow

In the normative design, the publisher signs a canonical `SignedProgramV1` for A0 or adds profiles and an execution claim in `SignedExecutionEnvelopeV1` for A1/B/C. A1 disables re-execution after context checking and records a typed policy-skip reason. B/C perform verifier re-execution under final-result and extended-observable commitments. After channel recovery, the verifier parses one canonical variant, authenticates it under external trust policy, checks bootstrap hints against signed values and signed context, conditionally re-executes L, aggregates every current report from its correctly curated store, and only then applies promotion policy.

The decomposition is:

```
Recover(y) -> m | bottom carrier layer
Authenticate(m, TrustPolicy) origin layer
CheckContext(e, ContextPolicy) context/freshness layer
CheckExecution(e, EnginePolicy) execution-agreement layer
Authorize(Aggregate(envelope_id, context_id,
        verifier_scope, all_current_reports),
        PromotionPolicy) authorization layer
```

No layer inherits the authority of the next. Recovery or decoder confidence is not authentication. Authentication is not execution agreement. Execution agreement is not authorization. Any outer carrier label and embedded key identifier are bootstrap hints only; neither can create a trust root.

2.2 Why determinism is load-bearing

Let $\mathbb{B}^*$ be finite byte strings, $I \subseteq \mathbb{B}^*$ canonical L intermediate representations, S semantics profiles, G execution profiles, and $\mathcal{T}$ finite execution transcripts. For $s \in S$ and $g \in G$, the reference semantics is a partial function

$$\mathrm{Exec}_{s,g} : I \rightharpoonup \mathcal{T}.$$

Partiality represents unsupported profiles, exhausted local limits, or other conditions under which the verifier cannot make the signed claim comparable. It does not introduce nondeterministic outcomes.

Assumption D (determinism of L). For all $i \in I$, $s \in S$, and $g \in G$, if $\mathrm{Exec}_{s,g}(i)$ is defined, then it has a unique value. Equivalently, two conforming evaluations under the same canonical IR, semantics profile, execution profile, and modeled inputs produce the same finite transcript.

Let q be a versioned commitment scope and $\mathrm{Project}_q : \mathcal{T} \to \mathbb{B}^*$ its deterministic canonical projection. A scope names its observation axes—for example, selected values, standard output, diagnostics, termination status, warning order, or resource reports—and makes no statement about axes outside that set. For transcript τ, define

$$\begin{aligned} \mathrm{td}_q(\tau) &:= H(\mathrm{Project}_q(\tau)), \\ \mathrm{od}(\tau) &:= \mathrm{Outcome}(\tau), \\ \mathrm{dd}(\tau) &:= H(\mathrm{DetEnc}(\mathrm{Diagnostics}(\tau))). \end{aligned}$$

Execution agreement holds only when local profiles and transcript schema are comparable, local execution is defined, and these recomputed values equal the signed `transcript_digest`, `claimed_outcome`, and `diagnostics_digest`. Agreement is therefore a case-specific equality test against a signed claim. Determinism makes it reproducible. It does not establish that the attributed publisher-side engine ran, when it ran, or that either implementation is defect-free.

3 Authoritative Signed Program and Execution Types

3.1 Typed envelope

All strings and tagged unions have one deterministic byte encoding. Digests carry algorithm identifiers, and profile identifiers resolve only through verifier-approved registries or local policy.

```
CanonicalProgramIR {
  source: Bytes;
  normalized_ir: Bytes;
}

ExecutionProfile {
  modeled_inputs: [CanonicalIRBinding | DigestReference];
  initial_state_digest: Optional<Digest>;
  environment_profile_digest: Digest;
  resource_policy_digest: Digest;
}

EngineDescriptor {
  implementation_name: Identifier;
  source_revision: Identifier;
  executable_or_module_digest: Digest;
  build_profile_digest: Digest;
  dependency_lock_digest: Digest;
  runtime_target: Identifier;
  semantics_profile_digest: Digest;
}

ExecutionClaim {
  transcript_schema: Identifier;
  commitment_scope: CommitmentScope;
  transcript_digest: Digest;
  claimed_outcome: Outcome;
  diagnostics_digest: Digest;
  execution_profile: ExecutionProfile;
  attributed_engine: EngineDescriptor;
}

EnvelopeContext {
  purpose: Identifier;
  subject: Identifier;
  issuer: Identifier;
  nonce: Nonce;
  issued_at: Timestamp;
  expires_at: Timestamp;
}

SignedProgramV1 {
  canonical_ir: CanonicalProgramIR;
  language_profile: Identifier;
  semantics_digest: Digest;
  context: EnvelopeContext;
}

SignedExecutionEnvelopeV1 {
  program: SignedProgramV1;
  execution_claim: ExecutionClaim;
}

SignedObjectV1 {
  domain: Identifier;
  object_version: Version;
  codec_profile: Identifier;
  payload_tag: Program | Execution;
  payload: SignedProgramV1 | SignedExecutionEnvelopeV1;
}
```

In the normative design, scope A0's payload is exactly `SignedProgramV1`. Scopes A1, B, and C use `SignedExecutionEnvelopeV1` with a nonempty versioned commitment scope; A1 and B use the same final-result claim shape, while only B enables verifier re-execution. `payload_tag` must match the selected payload type. The source is not a missing top-level field: it is the `source` member of `canonical_ir`, beside the normalized IR bytes, and both are covered by the signed deterministic encoding. Well-formedness for either payload requires the fixed V1 framing, an allowed L language profile, canonical L IR, internally consistent semantics digests, and ordered context timestamps. A well-formed execution-bearing envelope binds every entry of `execution_profile.modeled_inputs`: a `CanonicalIRBinding` identifies bytes embedded in `canonical_ir`, while a `DigestReference` binds external bytes that the verifier must resolve and hash before execution; dependence on any unsigned external input yields `not_comparable`, not agreement. The RQ3 harness instead uses a reduced experimental envelope that omits `execution_profile`. Its programs are closed, so no unsigned program input contributes, but RQ3 neither instantiates the full design type nor tests digest-referenced external-input binding. Only the normative execution-bearing type requires an execution claim and nonempty commitment scope. External policy may be stricter.

In the registry example, `canonical_ir` holds the migration program, `context` binds its target package and purpose, and `execution_claim` binds the publisher-attributed result, profile, and engine descriptor. Authentication permits the registry to label and display that result as publisher-attributed; it does not satisfy the production-rollout policy. That decision requires a later verifier report under the pinned scope and required independence projection.

Let `DetEnc` be injective and deterministic over these typed values and PAE injectively domain-separate the message. For payload e,

$$\begin{aligned} \mu(e) &:= \mathrm{PAE}(\mathrm{DOMAIN}_{V1}, \mathrm{DetEnc}(e)), \\ \sigma &:= \mathrm{Sign}(sk, \mu(e)), \\ m &:= \mathrm{DetEnc}(e, \sigma). \end{aligned}$$

Here e is the complete `SignedObjectV1` and DOMAIN_{V1} is a fixed public domain-separation string whose concrete prototype spelling is blinded. Every listed framing and payload field is under one signature. Signing only `canonical_ir`, or signing separately replaceable blocks, is nonconforming because it reopens version mixing, profile substitution, stale-result splicing, and status laundering.

The construction is signature-scheme-agnostic: an approved scheme must provide EUF-CMA security under the verifier's trust policy. Ed25519 is the intended deployment choice. The dependency-free RQ3 mechanism harness instead used HMAC-SHA256 as a stand-in for producing and checking a single authenticator over the same domain-separated bytes. No security claim or comparative result rests on HMAC behaving as a public-key signature or on the choice of a particular primitive.

3.2 Meaning of the execution claim

An **engine-attributed execution claim** is a publisher-signed statement that a particular observation tuple is attributed to the implementation described by `attributed_engine`. **Verified execution agreement** is the verifier's finding that its deterministic re-execution yields the same committed observation. An **execution proof** would verify computation correctness without re-execution. This construction supplies the first notion and can derive the second; it supplies no execution proof.

The signature makes the attribution immutable as part of e. It does not attest the occurrence of publisher-side execution. Operationally, the commitment is a promotion contract: execution-agreement authority is unavailable unless a comparable verifier-side re-execution agrees with the signed observation. The comparison can expose version drift, a newly signed stale claim, or implementation divergence, while remaining agnostic about whether the publisher obtained the observation by execution, prediction, or another process.

`attributed_engine` has exactly two meanings: the signer attributes the signed execution claim to the identified implementation, and the descriptor supplies publisher-asserted inputs for constructing an `IndependenceProfile`. Its signature proves only that those inputs are unchanged. It does not prove their independence assertions, attest that the described binary existed at a place or time, or show that it was loaded, executed this IR, produced the claim, had a trustworthy build, or returned a correct result. Section 4.1 therefore requires independent provenance before an axis can support high-authority promotion.

3.3 Stale claims and bound fields

For `SignedExecutionEnvelopeV1`, the whole-object signature distinguishes three stale cases:

(1) **stale-1, cross-envelope result splicing:** attaching a result digest from a different envelope to the current IR, or otherwise mixing an IR and result from different envelopes, changes signed bytes and fails authentication;
(2) **stale-2, valid whole-envelope replay:** replaying an old but valid whole envelope does not forge a signature and is addressed, when required, by signed purpose, subject, nonce, expiry, and freshness policy; and
(3) **stale-3, freshly re-signed stale observation:** signing an old result as a new claim for the current IR can pass authentication and freshness. Re-execution detects it only when its committed observation differs from the current deterministic execution; an observation-equal stale claim remains undetected.

Full intermediate-trace differences are detectable only when the versioned `commitment_scope` includes those observation axes. A final-result commitment makes no completeness claim about uncommitted intermediate behavior.

4 Typed Verification and Authorization

4.1 Independence is a vector

An independence profile records relationships between the publisher-attributed and verifier-side execution paths:

```
Relation := Shared | Partial | Distinct | Unknown
Provenance := PublisherAsserted | RegistryVerified |
              ExternallyAttested | VerifierEstablished

AxisEvidence {
  relation: Relation;
  provenance: Provenance;
  evidence_ref: Digest if provenance != PublisherAsserted
                else Optional<Digest>;
}

IndependenceProfile {
  source_lineage: AxisEvidence;
  parser_lineage: AxisEvidence;
  semantics_core_lineage: AxisEvidence;
  build_toolchain: AxisEvidence;
  runtime_backend: AxisEvidence;
  dependency_set: AxisEvidence;
  hardware_architecture: AxisEvidence;
  operator_or_organization: AxisEvidence;
}
```

`RegistryVerified` requires a registry-entry digest, `ExternallyAttested` an attestation digest, and `VerifierEstablished` a local evidence-record or derivation identifier. Only `PublisherAsserted` may omit `evidence_ref`. Provenance validation remains external to `AxisEvidence`; a missing or invalid required reference makes the axis ill-formed before projection, and `PublisherAsserted` never satisfies high-authority independence.

The high-authority independence predicate for one axis is

$$\text{Independent}(a) \iff a.\mathit{relation} = \text{Distinct} \wedge a.\mathit{provenance} \in \text{VerifiedProvenance},$$

where `VerifiedProvenance` is exactly {`RegistryVerified`, `ExternallyAttested`, `VerifierEstablished`} and well-formedness already requires the corresponding `evidence_ref`. `PublisherAsserted` is insufficient even when the signed relation says `Distinct`; the signature authenticates the assertion but does not make it true. `Partial`, `Unknown`, and `Shared` also project to false. We do not assume an honest publisher for independence metadata.

Policy cannot inspect raw `AxisEvidence`. Use the eight-axis declaration order above and define

$$\mathcal{I}(P) := (\text{Independent}(P_1), \ldots, \text{Independent}(P_8)) \in \{0,1\}^8,$$
$$\phi : \{0,1\}^8 \to \{0,1\}.$$

The execution-agreement guard is exactly $\phi(J)$, where J is the aggregate projection defined below; policy never receives P. Thus `PublisherAsserted + Distinct` becomes zero before policy sees it and cannot be granted by a raw-relation predicate. A conforming promotion policy is any total predicate over this verified Boolean projection that also obeys the status-preserving transitions and the pinned verifier scope below; it may be disjunctive or non-monotone over the eight bits. Labels I0–I3 may summarize profiles for exposition, but they do not form a total order. Agreement raises differential detection sensitivity only for fault domains actually separated by the concrete supporting profiles. It does not exclude shared defects, coincident defects, common specification ambiguity, or a fault inactive on the tested input.

This projection retains neither witness identities and counts nor pairwise relations among verifiers, and intersecting several agreement reports does not recover them. A policy requiring reports from two distinct verifier organizations or completion by every designated verifier is therefore inexpressible in $\phi : \{0,1\}^8 \to \{0,1\}$; it needs a richer multi-witness provenance model.

4.2 Separate evidence from authority

Let the report-level evidence-derived status be

$$\text{VS} := \{\text{execution\_agreed}(J), \text{execution\_disagreed}, \text{not\_reexecuted}(R_{nr}), \text{not\_comparable}(R_{nc})\}.$$

Here $J \in \{0,1\}^8$. An agreement report carries only $J_i = \mathcal{I}(P_i)$; its concrete P_i and axis references live in the supporting-evidence record committed by `supporting_evidence_digest`. Thus report-level `VerificationStatus` has exactly four variants; `authenticated` is a pipeline state. R_{nr} is a nonempty set over `skipped_by_policy`, `engine_unavailable`, `unsupported_profile`, and `resource_budget_declined`. R_{nc} is a nonempty set of typed comparison failures such as profile mismatch, unsupported scope, schema mismatch, or undefined execution. Every report is keyed by (`envelope_id`, `context_id`, `verifier_scope`), has an immutable report identifier and creation epoch, optional supersession epoch, supporting-evidence digest, and exactly one status. For a key and epoch, `current` means the complete set of matching reports created by then and not superseded at or before it; caller-supplied subsets are not current sets.

Authorization never consumes one caller-selected report. The verifier computes

```
Aggregate(envelope_id, context_id, verifier_scope,
        aggregation_epoch, verifier_store)
```

The verifier, not the consumer, selects from its owned store every matching report created no later than the epoch and not superseded at or before it. This internally derived complete current set includes older unsuperseded reports; the aggregation API accepts no caller-supplied report iterable. Write D for disagreement, $A_I(J) := \text{execution\_agreed}(J)$ for agreement, $C(R)$ for incomparability, and $N(R)$ for non-execution. Each agreement report exposes $A_I(J_i)$. Aggregate status folds one binary join $\sqcup$ with identity $\bot$; when agreement wins, $J = \bigcap_i J_i$. The precedence is

$$D > A_I(J) > C(R) > N(R) > \bot.$$

For unequal kinds, the higher kind wins. For equal kinds,

$$A_I(J) \sqcup A_I(K) = A_I(J \cap K),$$
$$C(R) \sqcup C(S) = C(R \cup S),$$
$$N(R) \sqcup N(S) = N(R \cup S).$$

Disagreement is absorbing. This single algebra has identity, associativity, commutativity, and idempotence; multiple agreements intersect high-authority axes without constructing new provenance evidence, while like-kind fallback reports union reason sets. Each underlying verifier-store report commits its concrete profile and evidence references through `supporting_evidence_digest`. An unsuperseded later disagreement therefore dominates the next

aggregate and invalidates an earlier non-execution fallback regardless of report order. The consumer receives only the verifier's aggregate and cannot cherry-pick a member report.

An aggregate `execution_agreed(J)` means that at least one current report agreed and no current report disagreed; it does not imply that every current report was comparable, re-executed, or agreeing. Requiring completion by every designated verifier needs the richer multi-witness aggregate identified above, which the eight-bit projection cannot express.

Let

$$\mathsf{AB} := \{\mathsf{origin\_only}(r), \mathsf{execution\_agreement}\},$$

where the typed fallback reason r contains the complete `not_reexecuted` or `not_comparable` reason set that authorized origin-only use. An authorization state is $(s, b) \in \mathsf{VS} \times \mathsf{Option}(\mathsf{AB})$. None means no authorization; it is not a third authorization basis. For promotion policy Π_P, the grant transitions from the verifier-computed aggregate are

$$(\mathsf{not\_reexecuted}(R_{nr}), \mathsf{None}) \to_{\Pi_P} (\mathsf{not\_reexecuted}(R_{nr}), \mathsf{origin\_only}(R_{nr}))$$
if origin-only use is explicitly allowed,

$$(\mathsf{not\_comparable}(R_{nc}), \mathsf{None}) \to_{\Pi_P} (\mathsf{not\_comparable}(R_{nc}), \mathsf{origin\_only}(R_{nc}))$$
if origin-only use is explicitly allowed,

$$(A_I(J), \mathsf{None}) \to_{\Pi_P} (A_I(J), \mathsf{execution\_agreement})$$
if execution use is allowed and $\phi_{\Pi_P}(J)$.

`not_comparable(reason) -> origin_only(reason)` does not relabel incomparability as agreement. This removes only one of two skip incentives: attempting and failing into `not_comparable` no longer loses to skipping, but a party that anticipates disagreement can still skip to obtain `origin_only` rather than attempt and receive `execution_disagreed`; `origin_only` is the honest weaker floor, never `execution_agreement`, and an unsuperseded later disagreement dominates the next aggregate. Disagreement has no grant, and neither fallback status has an execution-agreement grant. Every authorization output records its exact basis and typed reason plus `verifier_scope_digest`, `aggregation_epoch`, `report_set_digest`, and `promotion_policy_digest`. `report_set_digest` uses a canonical typed encoding of each current report's globally unique identifier, status, creation epoch, and supporting-evidence digest, not merely the final join. Current-set membership is evaluated at `aggregation_epoch`: a report exists by that epoch and has not been superseded at or before it, while later reports and future supersessions cannot alter the historical snapshot. The action policy must pin an allowed `verifier_scope_digest`; a mismatched scope yields no grant, so a consumer cannot request a more favorable scope. Execution agreement records the intersected verified projection; its concrete supporting profiles and axis evidence remain committed through the report-set digest.

For the registry, `origin_only(r)` may authorize two named low-risk actions without accepting the claimed result as correct: display it explicitly as publisher-attributed, or admit the package to an isolated dry-run channel when independent execution is unavailable. Production rollout still requires `execution_agreed(J)` under the pinned scope. If a later unsuperseded report disagrees, the next aggregate is `execution_disagreed` and yields no grant; undoing an already performed external action or delivering revocation is outside this transition model.

4.3 Non-laundering invariant

The first invariant says policy may attach authority but cannot rewrite evidence:

$$\boxed{q \to_{\Pi_P} q' \implies \mathsf{status}(q') = \mathsf{status}(q).} \tag{NL-1}$$

Let B contain `execution_disagreed`, every `not_reexecuted(reason)`, and every `not_comparable(reason)`. The central safety property is

$$\boxed{s \in B \implies \neg\exists q'.\ (s, \mathsf{None}) \to^*_{\Pi_P} q' \wedge \mathsf{basis}(q') = \mathsf{execution\_agreement}.} \tag{NL-2}$$

Policy may decide that already-established origin evidence is sufficient for a named low-risk use and must record why. It may not relabel that evidence or report that execution agreement occurred.

4.4 Safe promotion and finite-model safety check

Theorem 1 (type-safe authorization invariant). Within the specified transition relation, assuming a conforming verifier-owned complete current set, aggregation is report-order independent, every conforming policy transition preserves the aggregate evidence-derived `VerificationStatus`, and no aggregate with status `execution_disagreed`, `not_reexecuted(reason)`, or `not_comparable(reason)` can acquire an `execution_agreement` basis. A `Distinct` axis projects as independent only with registry-verified, externally attested, or verifier-established provenance and its required evidence reference.

Argument. `Aggregate` folds the associative and commutative join over the complete current report set, with disagreement dominant; thus adding disagreement replaces any prior non-execution aggregate regardless of report order. The only rule emitting basis `execution_agreement` has source $A_I(J)$ and guard $\phi_{\Pi_P}(J)$. Every grant rule preserves the complete projected status or typed fallback reason; the report-set digest separately preserves the supporting-evidence commitments. The three prohibited source classes have no transition to execution-agreement authority. NL-1 follows inductively over finite policy paths; NL-2 follows by reachability induction. Required evidence references are checked before $\mathcal{I}$, and policy never receives raw P_i, so a valid signer cannot create high-authority independence merely by asserting `Distinct`. Scope pinning independently prevents selection of a different verifier aggregate. This is a syntactic authorization invariant: it does not establish that a stored `execution_agreed` status was correctly derived, only that the specified promotion relation cannot manufacture that status from another variant.

This argument does not require ϕ to be monotone, conjunctive, or compositional over the eight projected bits. For any predicate

shape—including disjunctions and non-monotone acceptance sets—$\phi(J)$ is only a guard on the edge from an already established aggregate $A_I(J)$. Changing that guard can add or remove authorizations among agreed projections, but it cannot create an execution-agreement edge from a prohibited status or change the source status. Non-laundering is therefore a structural property of the transition relation, independent of the projected predicate's shape.

Authority need not be monotone under report addition for arbitrary ϕ. Let $\phi(J) = \neg J_{runtime}$. A first agreement report with $J_{runtime} = 1$ is refused. Adding a second, weaker agreement report with the same other bits and $J_{runtime} = 0$ clears that bit in the intersection and produces a grant. Evidence weakened while authority strengthened. Both aggregates remain `execution_agreed`, so this does not violate Theorem 1.

Corollary (monotone aggregate authority). Fix the report key, pinned scope, and policy. Let a correctly curated complete current set R aggregate to $A_I(J)$, and let another correctly curated complete current set $R' \supseteq R$ be a pure extension with no removal or supersession. If ϕ is componentwise monotone, then R' cannot receive an execution-agreement grant refused to R. A new disagreement yields no grant; otherwise the aggregate remains agreement with $J' \preceq J$ because added agreement projections are intersected and lower-precedence reports leave J unchanged, so $\phi(J') \leq \phi(J)$. We therefore recommend componentwise-monotone guards for high-authority production policies.

Executable-model evidence. `non_laundering_model.py` matched every defined oracle over all 4^8 relation vectors, the per-axis relation/provenance/reference cases, the exact projected equivalence classes, all 1,024 conjunctive policies, typed fallback sets, and singleton, pair, and triple joins. It also matched the defined scope-pin and current-set metadata oracles. This is exhaustive testing of the executable model's finite conjunctive-monotone policy language; arbitrary projected predicates rely on the structural argument above, not enumeration. The enumerated guards are componentwise monotone, but the checker does not compare decisions before and after report-set extension; the corollary is the order argument above, not a machine-covered claim.

Three negative-control mutations attempt status relabeling, a corrupted publisher-asserted projection, and policy access to raw evidence. The checker rejects all three against the defined oracles; it is not a proof that an arbitrary implementation conforms to the model.

5 Staged Verification Semantics

For fixed external policies $\Pi = (\Pi_T, \Pi_C, \Pi_E, \Pi_P, \Pi_F)$, where Π_F includes allowed formats and profiles, local resource limits, and freshness requirements, define:

$$
\begin{aligned}
\text{PipelineState} := \cdots \\
&\mid \text{authenticated}(\ldots) \mid \text{context\_checked}(\ldots) \\
&\mid \text{verified}(e, r, s) \mid \text{aggregated}(e, c, v, R, s) \\
&\mid \text{authorized}(e, c, v, R, s, b) \\
&\mid \text{nonpromoted}(e, c, v, R, s) \mid \text{reject}(\rho).
\end{aligned}
$$

Here $r \in \{\text{exact}, \text{robust}\}$, c is the context identifier, v the verifier scope, R the complete current report set, $s \in \text{VS}$, $b \in \text{AB}$, and ρ is a typed rejection reason. `authenticated(...)` and `context_checked(...)` are pipeline variants, never report-level statuses. The least transition relation follows nine stages:

(1) **Bootstrap.** A fixed, non-extensible grammar parses an untrusted candidate. A decoder profile is selected only from the external allowlist. Unknown profiles or failure reject.
(2) **Recovery.** The decoder returns candidate bytes or failure. The report records `exact` if `exactRecover` holds and otherwise `robust` if `robustRecover` holds. Confidence alone creates no successor.
(3) **Structural validation.** The candidate must parse as exactly one canonical (e, σ), satisfy V1 types and bounds, and re-encode byte-for-byte to m.
(4) **Authentication.** Candidate keys come only from external trust policy with issuer and purpose constraints. An embedded key cannot add itself to that policy.
(5) **Bootstrap recheck.** Every security-relevant outer value used for decoding is compared with its authenticated internal counterpart. A mismatch rejects; an outer label never overrides a signed value.
(6) **Context.** Signed purpose, subject, issuer, nonce, and timestamps are checked. Success gives a context-checked object, not authorization.
(7) **Execution classification.** Exactly one report-level status is recorded: `not_reexecuted(reason)`; `not_comparable(reason)` when comparison cannot complete; `execution_disagreed`; or `execution_agreed(J_i)`, where $J_i = \bar{I}(P_i)$ and `supporting_evidence_digest` commits P_i and its evidence references. A later execution adds a new immutable report rather than mutating the prior report.
(8) **Aggregation.** The verifier, not the consumer, joins all current reports for the envelope, context, and verifier scope. Disagreement dominates and invalidates any earlier origin-only fallback in that context; multiple agreement reports yield $A_I(J)$ with $J = \bigcap_i J_i$.
(9) **Authorization.** Policy consumes only the aggregate. Origin-only grants preserve their typed fallback reason; execution authority requires an already established $A_I(J)$ satisfying $\phi_{\Pi_P}(J)$. All other results are typed nonpromotion.

Every reachable configuration satisfies stage monotonicity, signed-value authority, complete-set aggregation, status immutability under policy, basis provenance, and typed terminal failure. Earlier-stage success never implies later-stage authority. `non_laundering_model.py` covers aggregate order and disagreement dominance, status immutability, basis provenance, provenance-gated independence, typed fallback reasons, and typed terminal failure; it does not model stage monotonicity or signed-value authority.

6 Threat Model and Security Games

6.1 Adversary and assumptions

The adversary may submit, edit, transform, resize, recompress, photograph, or reassemble arbitrary visual objects; replace labels and format blocks; splice regions from valid objects; replay old valid objects; mix versions and profiles; alter envelope bytes; reconstruct capability-shaped objects; and copy or substitute any previously issued valid envelope. For signature-forgery games preserved in the artifact, it may query a sealing oracle but does not obtain an approved signing key unless compromise is explicitly modeled. Independence authorization does not rely on that restriction: an approved-key publisher may over-assert `Distinct`, but the assertion remains `PublisherAsserted` and cannot satisfy the high-authority provenance gate without external registry verification, attestation, or verifier establishment.

The games allow the adversary to choose any received object y. Their cryptographic conclusions therefore do not depend on a bound on the decoder's raw wrong-output rate. Correct canonical parsing, injectivity of `DetEnc` and `PAE`, and conformance to the transition relation define the modeled verifier.

We assume: (1) collision resistance for every digest algorithm admitted by policy; (2) existential signature unforgeability under adaptive chosen-message attack for approved, uncompromised keys; and (3) Assumption D for every profile on which the verifier reports agreement. If an approved key is exposed, an admitted digest is collision-broken, or L is nondeterministic under supposedly identical modeled inputs, the corresponding property is not claimed.

Report lifecycle is a trust boundary. Operationally, only writers admitted by external verifier-store policy may append reports, but the abstraction neither represents writer identities nor authenticates that admission. It also does not specify which principal may assign a supersession epoch, under what conditions, or for what reason, and it does not forbid silently superseding an `execution_disagreed` report. Disagreement is therefore absorbing only within a correctly curated complete current set; unauthorized creation or supersession is an evidence-set curation risk outside Theorem 1. Likewise, `VerifierEstablished` requires a referenced derivation, but the model does not validate that derivation's correctness; a mis-derived `Distinct` relation that passes the external gate is out of scope.

Properties 1–3 are DSSE-class carrier-transparency results, not new hardness results. Under canonical encoding, EUF-CMA security, and admitted-digest collision resistance, a novel decoded message reduces to envelope forgery or binding failure; a region mixture instead yielding an existing envelope is substitution or replay. The artifact preserves their games and proof sketches, while the body retains Property 4 because safe promotion is the authorization delta.

6.2 Property 4: safe-promotion game

Game G_{promote}. The adversary chooses a received object and any promotion policy expressible by Section 4.2. After complete-set aggregation it may take any finite sequence of policy transitions. It wins if a policy transition changes the evidence-derived status, or if `execution_disagreed`, `not_reexecuted(reason)`, or `not_comparable(reason)` reaches basis `execution_agreement`; omitting a current report is verifier nonconformance, not a consumer strategy in the game.

By Theorem 1,

$$\Pr[G^{\mathcal{A}}_{\text{promote}} = 1] = 0.$$

Within the specified authorization relation, the game has no winning transition. The statement assumes a conforming current set, classifier, and supporting-evidence record; it is not report-store security or an implementation-conformance result. The semantic meaning of `execution_agreed(J)` additionally depends on L determinism.

6.3 Decoder outcomes and replay boundary

For intended envelope m, evaluation records three separate events:

If freshness is required, purpose, subject, nonce, issue time, and expiry are signed and checked against external state or policy. If freshness is intentionally out of scope, replay of an old valid object in the same context is not prevented. A newly signed stale execution claim is stale-3, not replay; re-execution detects it only when the committed observation differs from current deterministic execution.

7 Evaluation

7.1 Research questions

- **RQ1—binding soundness:** Does one signed typed object prevent accepted cross-profile substitution, IR/claim mix-and-match, payload-kind confusion, and bootstrap override?
- **RQ2—safe promotion:** Can an authenticated artifact lacking execution agreement acquire execution-agreement authority?
- **RQ3—incremental value of claims and re-execution:** Compared with A0 and A1, which drift, divergence, stale-3, and laundering failures become detectable, and how does the pattern depend on independence?

7.2 Executable-model evaluation

RQ1's carrier-transparency reductions are preserved in the artifact. RQ2 is the type-safe design invariant of Theorem 1, with executable evidence from the finite model. The implementation matched every defined oracle over the enumerated conjunctive-policy, join, reason, provenance, scope-pin, and report-metadata domains and rejected all three negative-control mutations. It did not enumerate disjunctive or non-monotone predicates, which rely only on the structural argument about the guard over J. It also does not check stage monotonicity, signed-value authority, report-writer or supersession authorization, derivation correctness, parser or cryptographic implementation safety, arbitrary application policy, or implementation-to-model correspondence. The anonymous artifact at `https://anonymous.4open.science/r/sealed2026-artifact` reproduces these checks and the RQ3 outputs.

Table 1: Decoder and replay outcomes.

Event	Definition	Security interpretation
decoder wrong-output	$D_v(y) = m' \neq m$	carrier-quality failure; need not be zero for security
authenticated false accept	$D_v(y) = m' \neq m$ and origin/context checks accept	cryptographic false acceptance; target is observed zero plus cryptographic bound
valid substitution/replay	$D_v(y) = m'$, where m' is another or older valid envelope	not a forgery; controlled by signed context and freshness policy

7.3 Executed RQ3 failure-injection design

RQ3 uses the real deterministic DSL L and four engine families. E1 is the native Rust reference interpreter. E2 is a self-hosted L runner with its own guest tokenizer, reader, normalizer, and machine. E3 is the native Rust product runner through the observation protocol. E4 is an engineered cross-language transliteration of E1 run on a distinct interpreter backend; a generated-native host for the same E4 transliteration is a fifth execution path, not a fifth engine lineage. E4 shares E1's source and semantic-core lineage by construction even though its runtime backend and build toolchain differ. All are real repository implementations rather than simulations of engine separation.

The clean corpus contains only four programs exercising arithmetic, pairs/lists, non-tail recursion, and branching. This is a bounded mechanism surface, not a conformance suite or a program-diversity sample. It stays within the **84/205** primitive-row intersection advertised by the hosted engines. Here 205 is the full product capability registry and 84 is the subset supported by both hosted families; separately, **34/144** is the repositories' historical count of all-family agreeing cases in a broader cross-family corpus, not a primitive-coverage fraction. RQ3 claims use the 84/205 intersection because every program and controlled primitive fault in this matrix stays within it, while 34/144 is retained as broader conformance context rather than converted into a coverage claim. Each program ran through E1, E2, E3, E4 on its cross-language interpreter host, and E4 on its generated-native host. `SignedProgramV1.canonical_ir.source` holds L's source bytes and `normalized_ir` holds the real canonical program bytes; the verifier re-canonicalizes source before checking the pair. Raw observations are retained, but comparison uses a documented canonical language-level transcript projection.

RQ3 compares four verifier-policy scopes. A0 admits only the signed `Program` payload tag and has no execution claim; source is inside `canonical_ir` as specified above. A1 admits the signed `Execution` tag and the same final-result claim bytes used by B, but verifier re-execution is disabled, so applicable divergences remain `not_reexecuted(skipped_by_policy)`. B enables verifier re-execution over stdout and final values, with separately bound outcome and diagnostics. Extended observable transcript scope C compares the full common canonical observable projection, including root/termination events and an explicit `step_trace: unavailable` marker. It has no comparable instruction- or reduction-level trace and supports no step-by-step equivalence claim. The harness derives payload kind from the signed tag and commitment shape; A1 versus B is an externally fixed verifier policy, not an unsigned payload reinterpretation.

Each scope is evaluated under four recorded profiles. `SHARED` denotes a common lineage or fault domain, `DISTINCT` a separated domain, `PARTIAL` a mixed boundary, and `UNKNOWN` unavailable relation evidence. Only `DISTINCT` can satisfy independence. For this experiment the verifier records the displayed relations as `VerifierEstablished`; every axis therefore carries a required local derivation identifier, and `independence-profiles.json` retains the canonical derivation preimage and source/artifact digests named by that reference. These experiment-local annotations support attribution analysis; the RQ3 mechanism harness does not consume them in an authorization promotion decision, and they are neither registry verification nor third-party attestation:

The labels I0–I3 name evaluated profiles; they are not ranks. I2 and I3 separate complementary domains. In particular, E1→E4 is a distinct-runtime-backend and distinct-build-toolchain pairing, not a source-independent pairing. The transliteration inherits E1's semantic decisions and can inherit defects in that logic. No evaluated profile combines independently authored source and semantic-core lineage with a distinct runtime backend. `Partial` is non-independent under promotion, exactly like `Unknown`; neither is silently rounded up to `Distinct`.

The ten conditions are a clean control plus stale-1, stale-2, stale-3, semantics-version drift, a single-operation backend fault, equal final results with divergent observable traces, normal/error outcome confusion, diagnostics promoted as success, and a common-mode Rust-runtime-backend fault. The matrix constructs them from six fixed fixture programs, separately from the four-program clean feature corpus. All stale-3 injections are observation-divergent, so the experiment does not cover observation-equal stale claims. In the last condition, fault-injected E1/E2/E3 artifacts introduce a one-input off-by-one in a supported exact-addition host primitive while E4 is unmodified. The resulting $4 \times 4 \times 10 = 160$ condition-combination cells are oracle cases, not a program-diversity sample. Ground-truth injection class is separate from verifier output; fresh-process reruns test deterministic canonical and raw observations per path, while cross-engine equality applies only to the canonical projection.

8 Results

8.1 RQ3: re-execution adds detection that signed IR alone lacks

The implementation matched every defined oracle in the 160-cell matrix. A second fresh-process run reproduced all six result artifacts byte-for-byte, including the common-mode Rust-runtime-backend cases. Four real programs across five execution paths produced byte-identical canonical transcripts, while raw observations reproduced per path but intentionally differed across engines.

Table cells name the detecting layer. `PASS` is a clean verification control. Under A0, `N/A` marks an execution-claim attack

Table 2: Recorded eight-axis RQ3 independence vectors.

Axis	I0: E3 → E3	I1: E3 → E1	I2: E3 → E2	I3: E1 → E4
source_lineage	SHARED	SHARED (common Rust lineage; distinct revision)	DISTINCT (guest L source)	SHARED (engineered transliteration)
parser_lineage	SHARED	SHARED (common lineage)	DISTINCT (guest tokenizer/reader)	SHARED (transliterated frontend)
semantics_core_lineage	SHARED	SHARED (common lineage)	DISTINCT (guest machine source)	SHARED (transliterated semantic choices)
build_toolchain	SHARED	SHARED (Cargo/rustc)	SHARED (Cargo wrapper)	DISTINCT (Rust/Cargo vs cross-language artifact)
runtime_backend	SHARED	SHARED (native Rust)	PARTIAL (guest L machine; shared Rust host)	DISTINCT (native Rust vs cross-language interpreter)
dependency_set	SHARED	PARTIAL (substantial overlap)	PARTIAL (shared host/contracts)	PARTIAL (separate engine trees; shared OS/contracts)
hardware_architecture	SHARED	SHARED	SHARED	SHARED
operator_or_organization	SHARED	SHARED	SHARED	SHARED

that the signed `Program` payload structurally cannot contain. Under A1, `MISS` marks an authenticated signed claim whose divergence remains undetected because re-execution is disabled. The real-L execution-classification path stops at per-condition `VerificationStatus`: it does not place reports in the verifier store, exercise supersession or complete-set aggregation, consume experiment-local independence profiles in promotion, or issue an action decision.

`SIG`, `FRESH`, and `REEXEC` are verifier-reported detecting layers: signature, nonce/freshness, and deterministic re-execution agreement. The fault names are controlled harness ground truth, not diagnoses emitted by the verifier.

The load-bearing comparison is stale-3. A0 has no execution claim, so stale-3 is structurally unconstructible and correctly `N/A`. A1 authenticates the signed stale claim but does not re-execute, so every A1 stale-3 cell is a valid `MISS`. B and C caught every injected stale-3 case because each committed observation differed from current deterministic execution; observation-equal stale claims remain undetected and were not injected.

Independence changes fault visibility only under re-execution. B/C missed the controlled backend fault at same-artifact I0 and detected it at I1–I3. For the Rust-host common-mode mutation, E1/E2/E3 agreed on the same wrong result, so B/C missed it at I1/I2; E4's distinct backend disagreed at I3. This is one controlled backend-separation result on the evaluated surface.

Commitment scope is also material. B cannot observe an equal-result divergence confined to the available canonical event trace; C caught that mutation at I0–I3. Because C is not a language-level step trace, the claim is specifically about the committed observable root/termination events, not intermediate reduction equivalence.

The artifact retains the prior toy-L matrix only as a controlled commitment-scope check; the real-L matrix is the primary mechanism evidence.

8.2 Test-only JSON-to-model bridge

A test-only harness adapter emitted report JSON at the RQ3 `VerificationStatus` boundary, including a deliberately induced `engine_unavailable` fixture, and fed it into the existing finite model's `VerifierStore`, `aggregate`, and `authorize` APIs. The four valid sequences and one malformed-input case produced the outcomes below; both reports remained current in the third sequence. The malformed case removed `profile_id` from an `execution_agreed` record, and the existing parser rejected it before aggregation with `execution_agreed requires exactly profile_id`. The real-L execution-classification path itself still ends at `VerificationStatus`. This is integration wiring between two existing components, not a new production aggregation/promotion implementation or implementation-to-model conformance evidence.

9 Threat-Model Boundaries and Non-Claims

The following boundaries are the authoritative non-claims for the paper:

- **Execution assurance.** The envelope supports an attributed claim and verifier agreement, not proof that the engine ran, an execution or historical proof such as a SNARK, STARK, or TEE attestation, or defect exclusion. Agreement only adds differential sensitivity. The games do not estimate attack prevalence.
- **Authorization model.** Theorem 1 is a type-safe invariant of the specified transition relation, conditional on a correctly curated complete current set. The model does not authenticate report writers, govern supersession, prevent silent supersession of disagreement, validate `Verifier Established` derivations, prove arbitrary code conforms, or express multi-witness requirements such as two distinct verifier organizations. It does not guarantee authority monotonicity for arbitrary non-monotone guards, report removal or supersession, or policy or scope changes.
- **Evaluation path.** The real-L harness uses a reduced experimental envelope that omits `execution_profile`, and the real-L execution-classification path itself still ends at per-condition `VerificationStatus`; it neither instantiates the full normative design type nor tests digest-referenced external-input binding. A test-only bridge parses emitted report JSON into the existing finite abstraction's store and authorization APIs. Aggregation and promotion remain

Table 3: RQ3 detecting layer and harness ground-truth injection class.

Injection	A0/I0	A0/I1	A0/I2	A0/I3	A1/I0	A1/I1	A1/I2	A1/I3	B/I0	B/I1	B/I2	B/I3	C/I0	C/I1	C/I2	C/I3
clean	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
stale-1	N/A	N/A	N/A	N/A	SIG	SIG	SIG	SIG	SIG	SIG	SIG	SIG	SIG	SIG	SIG	SIG
stale-2	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH	FRESH
stale-3	N/A	N/A	N/A	N/A	MISS	MISS	MISS	MISS	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC
semantics-version-drift	N/A	N/A	N/A	N/A	MISS	MISS	MISS	MISS	MISS	REEXEC	REEXEC	REEXEC	MISS	REEXEC	REEXEC	REEXEC
single-op backend fault	N/A	N/A	N/A	N/A	MISS	MISS	MISS	MISS	MISS	REEXEC	REEXEC	REEXEC	MISS	REEXEC	REEXEC	REEXEC
equal-result trace divergence	N/A	N/A	N/A	N/A	MISS	MISS	MISS	MISS	MISS	MISS	MISS	MISS	REEXEC	REEXEC	REEXEC	REEXEC
outcome confusion	N/A	N/A	N/A	N/A	MISS	MISS	MISS	MISS	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC
diagnostics as success	N/A	N/A	N/A	N/A	MISS	MISS	MISS	MISS	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC	REEXEC
Rust-backend common-mode	N/A	N/A	N/A	N/A	MISS	MISS	MISS	MISS	MISS	MISS	MISS	REEXEC	MISS	MISS	MISS	REEXEC

Table 4: RQ3 JSON-to-model bridge outcomes.

Bridge sequence	Aggregate or check	Basis
clean B/I3 agreement, `J=(0,0,0,1,1,0,0,0)`	`execution_agreed`	`execution_agreement`
deliberate `engine_unavailable` fixture	`not_reexecuted`	`origin_only`
preceding fallback plus current B/I3 stale-3 disagreement	`execution_disagreed`	none
agreement with mismatched `verifier_scope` pin	scope pin rejects	no grant
`execution_agreed` omitting `profile_id`	rejected before aggregation	no grant

implemented only there. The bridge is not an end-to-end production-engine path, implementation-to-model conformance evidence, a store-curation test, or a test of arbitrary report mappings. The executable model matched every defined finite-domain oracle, but arbitrary projected predicates rest on the structural argument rather than enumeration.

- **Independence and semantic scope.** I3 separates runtime backend and toolchain but shares transliterated source, parser, and semantic core; all paths share operator, organization, machine, and specification. Four programs and six fixtures are restricted to the supported 84/205 hosted-row intersection; the 160 cells are conditions, and 34/144 is only broader historical agreement. This is not conformance, whole-language, source-independent, organizationally independent, or exhaustion-equivalence evidence. Scope C compares extended observable events, not language-level steps; canonical equality is neither raw equality nor completeness for omitted axes.
- **Trust infrastructure.** An embedded key is not a trust root. External trust, profile, semantics, decoder, and verifier policies are assumed correct. We do not solve policy correctness, key custody, or revocation. Replay depends on external enforcement of signed freshness. The system supplies no log or trusted timestamp. Offline means no online manifest or sidecar, not no signature infrastructure.
- **Carrier.** Robust mode authenticates recovered envelope bytes, not received pixels.

10 Related Work

Exact-byte provenance and supply chains. DSSE authenticates serialized payload bytes, in-toto binds supply-chain steps and artifacts, SLSA standardizes provenance, and Sigstore makes software-signing identities and events auditable [7, 11, 12, 14]. We use that evidence substrate for a downstream typed authorization machine and do not replace its trust, identity, revocation, or transparency functions. The comparison table summarizes the architectural distinctions.

Evidence-authority separation is not itself new: RATS separates appraisal from relying-party decisions, and Macaron separates evidence-gathering checks from consumer policy. For signed deterministic-program claims, our claimed delta is the combination of immutable execution-comparison status, complete-current-set aggregation with disagreement dominance, and provenance-gated independence projection before promotion policy. This is an authorization-design invariant, not occurrence evidence or general policy correctness; it assumes correct current-set curation and external validation of `VerifierEstablished` provenance.

Information flow, typestate, and attestation. Information-flow lattices constrain flows between labels, typestate permits operations only from established states, and RATS separates attestation evidence and appraisal from relying-party authorization [1, 3, 5, 13]. We specialize that boundary to deterministic claim comparison and status-preserving promotion.

Proof carrying and translation validation. Proof-carrying code attaches a machine-checkable safety proof, which our execution-claim-carrying envelope does not provide [6]. Translation validation checks a particular output against source semantics, while our case-specific comparison checks a verifier observation against a signed engine-attributed observation and does not validate any source-to-L translation [9].

Carrier-adjacent work. C2PA separates cryptographic hard binding from soft fingerprints or watermarks, while QRscript shows that executable bytecode can be carried by standard visual symbols [2, 10]. We require exact envelope recovery followed by signature verification and make no carrier novelty claim.

Table 5: Architectural comparison with the closest evidence and policy families.

System family	Occurrence evidence	Verifier re-exec.	Evidence/authority split	Conflict rule	Independence provenance
DSSE + in-toto/SLSA [11, 12, 14]	Attested step/build claim	Optional inspections	Layout or policy-defined	Threshold/artifact rules	Functionary/build identity
RATS [1]	Attested target state	Not required by architecture	Architectural split	Policy/protocol-defined	Endorsement data
Macaron + OPA/Rego [4, 8]	Checker/input facts	No dedicated deterministic-claim type	Architectural split	Policy-defined	Application/policy-defined
Ours	No publisher-run proof	Required for agreement	Immutable status/basis	Complete set; disagreement dominates	Eight referenced, verified axes

11 Discussion

11.1 What the execution commitment adds

A0/A1 separate claim absence from authenticated publisher attribution; B/C add verifier-side deterministic comparison. The controlled faults show that detection follows the separated domains recorded in the evaluated profiles.

11.2 Policy usability

The registry example makes fallback auditable: low-risk display or isolated dry-run actions retain their exact `origin_only` reason, while production rollout requires agreement. This avoids encoding an operational exception as stronger evidence.

The Boolean projection also collapses the verified provenance classes, so a policy that distinguishes local verifier derivation from third-party attestation needs a richer verified projection rather than access to raw publisher assertions.

11.3 Channel and carrier independence

We evaluated a representative lossy visual carrier, rejected a custom design against a strong standard mosaic, and therefore use standard symbols with sharding and outer erasure correction. Full definitions and results are in the artifact.

11.4 From model to implementation

An implementation must centralize complete-set aggregation, preserve immutable typed status and reasons, authenticate report lifecycle operations, require and validate independence provenance, and pin verifier scope. Implementation conformance and store security remain separate obligations.

12 Conclusion

We separate signed publisher attribution, verifier-derived execution status, and later authority with split types and verifier-owned aggregation. The type-safe transition invariant makes disagreement dominant and gates independence before policy; the executable abstraction matched every defined finite-domain oracle and rejected three laundering mutations.

The real-L harness likewise matched every defined condition oracle and isolated the incremental value of deterministic re-execution and measured backend separation on this bounded surface.

References

[1] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. 2023. *Remote ATtestation procedureS (RATS) Architecture.* Technical Report RFC 9334. Internet Engineering Task Force. doi:10.17487/RFC9334

[2] Coalition for Content Provenance and Authenticity. [n. d.]. C2PA Technical Specification 2.4: Binding to Content. https://spec.c2pa.org/specifications/specifications/2.4/specs/ContentCredentials.html.

[3] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243. doi:10.1145/360051.360056

[4] Behnaz Hassanshahi, Trong Nhan Mai, Alistair Michael, Benjamin Selwyn-Smith, Sophie Bates, and Padmanabhan Krishnan. 2023. Macaron: A Logic-based Framework for Software Supply Chain Security Assurance. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. 29–37. doi:10.1145/3605770.3625213

[5] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. 129–142. doi:10.1145/268998.266669

[6] George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 106–119. doi:10.1145/263699.263712

[7] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software Signing for Everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2353–2367. doi:10.1145/3548606.3560596

[8] Open Policy Agent. 2026. Open Policy Agent Documentation. https://www.openpolicyagent.org/docs/. Accessed 2026-07-20.

[9] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 1384)*. 151–166.

[10] Stefano Scanzio, Gianluca Cena, and Adriano Valenzano. 2024. QRscript: Embedding a Programming Language in QR Codes to Support Decision and Management. arXiv:2404.05073 [cs.PL] https://arxiv.org/abs/2404.05073

[11] Secure Systems Lab. 2024. Dead Simple Signing Envelope (DSSE), Protocol v1.0.2. https://github.com/secure-systems-lab/dsse/blob/master/protocol.md.

[12] SLSA Community. [n. d.]. SLSA v1.2: Provenance. https://slsa.dev/spec/v1.2/provenance.

[13] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 157–171. doi:10.1109/TSE.1986.6312929

[14] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing Farm-to-Table Guarantees for Bits and Bytes. In *28th USENIX Security Symposium (USENIX Security 19)*. 1393–1410.