Proof-Carrying State Transitions for Reliable AI Systems

Adrian Diamond
AADSystems Research
adrian@AADSystems.com

November 6, 2026

Abstract

What can an independent observer establish about a state change proposed by an AI system? An authorization certificate can show that an update satisfies a formal policy. It does not, by itself, show that the update was committed. The distinction matters whenever the authority to change persistent state lies outside the proposing agent.

We develop a proof-carrying transition architecture in which a source state, action, successor state, and policy are accompanied by finite evidence checked by a separate verifier. We establish conditional invariant preservation, distinguish authorization from authenticated execution evidence, and prove that authorization-only observations cannot determine whether an authorized transition actually occurred.

An executable Python reference construction uses Merkle inclusion witnesses and a quota-and-permission policy to export certificates and transition histories for offline audit without distributing the full authorization list. All 18 automated tests pass, and an independent audit verifies two accepted transitions. The experiments also expose a precise boundary: membership evidence neither authenticates the requesting principal nor binds the principal to a particular action.

The contribution is a formal and reproducible account of three distinct evidentiary claims—policy admissibility, consistency of a recorded transition history, and authenticated commit reporting. Situated within proof-carrying computation and recent AI-agent authorization research, the construction identifies which claims portable certificates support and the trust assumptions required to substantiate stronger ones.

Keywords: proof-carrying state transitions; AI agents; runtime verification; authorization certificates; execution evidence; independent audit; formal state.

1 Introduction

AI agents can draft plans, call tools, and propose changes to a system's persistent state. A plausible natural-language explanation of such a proposal is not an authorization decision. Even an authoritative monitor that accepts or rejects an action need not provide a portable justification that a third party can recheck. The problem is not solely whether the state is protected at the instant of execution; it is also what an external auditor can establish afterwards, given an evidence object separated from the original decision procedure.

The companion work on formal-state-augmented language models places consequential updates under an external deterministic executor with explicit guards, partial transition semantics, and state invariants (Diamond, 2026). Under its assumptions, monitoring preserves invariants for arbitrary sequences of model proposals. The present study addresses a complementary question: *what verifiable evidence can accompany a particular claimed transition, and what is the logical strength of that evidence?* The answer matters even when the proposer is not a language model. A script, a planning agent, or another untrusted component can supply the same kind of claim.

Proofs attached to computational artifacts have a long history. Hoare-style reasoning supplies the logical foundation (Hoare, 1969); proof-carrying code attaches checkable safety evidence to software (Necula, 1997); and proof-carrying data considers evidence about computed values and their histories (Chiesa and Tromer, 2010; Chong et al., 2013). More recently, proof-carrying agent actions (PCAA) have been implemented with typed action certificates and an independent checker (Kadyrbek and Mansurova, 2026), while cryptographically verifiable authorization (CVA) has modeled request-, principal-, and policy-binding requirements for autonomous agents (Llambí-Morillas and Fernández-Fernández, 2026). The approach builds on this established literature. We make its guarantees and trust boundaries explicit through a state-transition semantics and an inspectable executable reference construction.

The contribution has three parts. First, we separate three kinds of claims: *policy admissibility*, *consistency of a claimed transition history*, and *actual occurrence of a commit*. Second, we give a concise semantics and conditional results that prevent these claims from being conflated. Third, we report a reproducible reference implementation with bounded tests, certificate sizes, and explicit counterexamples to overly strong interpretations. The result is a transparent reference construction and a formal account of the exact claims supported by each level of evidence. The cryptographic techniques and elementary proof principles are established methods, applied here to a clearly specified verification boundary.

2 Related Work and Scope of the Contribution

2.1 From program proofs to history-sensitive evidence

Hoare logic and model checking study correctness with respect to formal specifications (Hoare, 1969; Baier and Katoen, 2008). Proof-carrying code makes a consumer-side checker central to a software safety decision (Necula, 1997). Proof-carrying data (PCD) goes further by attaching evidence to values whose computational histories must obey a predicate, including in distributed settings (Chiesa and Tromer, 2010; Chong et al., 2013). The certificate sequence studied here differs from succinct PCD: verification of n steps requires inspection of n certificates. The earlier work provides the foundations for history-sensitive evidence checking.

2.2 Proof-carrying authorization for AI agents

PCAA separates a planner from a checker that evaluates typed certificates for exact tool invocations. Its authors demonstrate that a stateful guard with the same certified-state predicate can match certificate checking on safety, while the certificate provides a typed, replayable artifact (Kadyrbek and Mansurova, 2026). CVA studies request-bound cryptographic authorization, including principal, context, policy, and replay concerns; it explicitly distinguishes authorization from later execution (Llambí-Morillas and Fernández-Fernández, 2026). These studies define the closest agent-specific context. Our membership witness supplies policy-list membership rather than CVA's principal- and request-bound authorization; the present checker uses direct arithmetic rules rather than PCAA's typed proof system.

2.3 Relationship to the companion state monitor

The companion monitor decides whether an action can change authoritative state; this work exports checkable evidence of a *claimed* decision and successor. The companion architecture already defines traces and conditions for atomic commits. Here we specify a concrete serialization, membership

witness, and independent offline auditor. These mechanisms are compatible with the companion monitor, while providing a distinct evidence-distribution and audit interface. Both enforce the same local invariant in the reference comparison. Table 1 places the construction in its research context.

Table 1: Related approaches and the evidence they primarily address. The comparison distinguishes evidence capabilities rather than ranking overall system security.

Approach	Relevant capability	Relationship to this work
Companion state monitor (Diamond, 2026)	Independent guard checks and invariant-preserving commits under stated assumptions	Adds portable certificates and history objects
Proof-carrying data (Chiesa and Tromer, 2010; Chong et al., 2013)	Evidence about computation histories and semantic compliance	History evidence precedent; simpler, nonsuccinct construction
PCAA (Kadyrbek and Mansurova, 2026)	Planner-independent typed certificates for tool actions	Direct overlap in authorization evidence
CVA (Llambí-Morillas and Fernández-Fernández, 2026)	Cryptographic principal, request, and policy binding	Supplies principal and request binding absent here
Authenticated logs (Laurie et al., 2021)	Commitments, inclusion/consistency checking, externally witnessed log state	Precedent for log auditing; no automatic truth of logged events

3 State Transitions and Certificates

3.1 Authoritative state and policy

Definition 3.1 (Transition specification)**.** A transition specification is a tuple $\mathcal{F} = (S, A, \Pi, T, I)$, where S is a set of authoritative states, A a set of structured actions, Π a family of policies, $T_\pi \subseteq S \times A \times S$ the permitted transition relation under policy $\pi \in \Pi$, and $I : S \to \{\mathsf{true}, \mathsf{false}\}$ a designated state invariant. A proposed transition is (s, a, t, π), asserting $(s, a, t) \in T_\pi$.

The relation is intentionally not defined by a model's verbal explanation. Its meaning is fixed by the system specification. A policy identifier $\mathsf{pid}(\pi)$ and, where needed, a commitment $\rho(\pi)$ represent an externally anchored policy version. These fields do not by themselves show that the policy root came from a trusted source; that provenance is a separate assumption.

Definition 3.2 (Certificate and verifier)**.** Let P be a set of finite evidence objects. A certificate verifier is a computable predicate

$$V_\pi : S \times A \times S \times P \longrightarrow \{0, 1\}.$$

It is *transition-sound* for T_π when

$$\forall s, a, t, p : \qquad V_\pi(s, a, t, p) = 1 \implies (s, a, t) \in T_\pi. \tag{1}$$

Soundness is relative to the intended policy semantics and correctly identified authoritative inputs. It does not follow merely from the fact that a checker terminates or has passing tests.

A conforming executor first obtains the actual current state s, verifies the certificate against that state and the trusted policy, and then either commits precisely t or rejects without changing

s. The single-threaded reference construction implements this discipline within one Python object. In concurrent settings, correctness requires a linearizable or otherwise atomic check-and-commit mechanism; a version number without an atomic commit operation is insufficient (Herlihy and Wing, 1990).

3.2 Safety and its assumptions

Theorem 3.3 (Conditional invariant preservation)**.** *Let $s_0 \in S$ satisfy $I(s_0) = \mathsf{true}$. Suppose (i) V_π is transition-sound for T_π; (ii) T_π preserves I, meaning $I(s) \wedge (s, a, t) \in T_\pi \Rightarrow I(t)$; and (iii) the executor can commit only exactly those successor states whose certificates were checked against the authoritative current state. Then every state reachable through a finite sequence of accepted commits satisfies I.*

Proof. Induct on the number of committed steps. The base case is the assumption on s_0. For a step from s_i to s_{i+1}, verifier acceptance and soundness give $(s_i, a_i, s_{i+1}) \in T_\pi$. Invariant preservation gives $I(s_{i+1})$. Rejected proposals leave the authoritative state unchanged. □

The theorem isolates the soundness, invariant-preservation, and exclusive-commit assumptions of the architecture using standard inductive reasoning. Safety can hold even for a checker that rejects every proposal; progress and reachability require separate analysis and do not follow from Theorem 3.3.

3.3 Admissibility is not attribution

A proof of allowlist membership is evidence about a *string in a committed set*, not proof that the named principal submitted a particular request. For example, let a public membership witness establish that the label `agent-0042` belongs to an authorized set. A third party can reuse this witness on a different otherwise permitted quota update unless the protocol separately authenticates the requester and binds the request to the authorization. Whether the new action is *policy admissible* remains a distinct question. Such substitution does not falsify Theorem 3.3; it shows that the invariant theorem was never an authentication theorem. Stronger principal and request binding are explicitly pursued in CVA and PCAA (Llambí-Morillas and Fernández-Fernández, 2026; Kadyrbek and Mansurova, 2026).

4 Three Levels of Transition Evidence

4.1 Authorization, claimed history, and occurrence

Let $E_{\text{auth}} = (s, a, t, \pi, p)$ be the evidence visible to a static certificate checker. Let a claimed history be a sequence $h = (E_1, \ldots, E_n)$ whose successor states agree with the following source states. Finally, let $\mathsf{Occurred}(w, s, a, t)$ mean that an execution world w actually contains a committed transition with that source, action, and target. A verifier receives artifacts, not an oracle for w.

Proposition 4.1 (Authorization-only evidence cannot decide occurrence)**.** *Suppose there are two execution worlds w_1, w_0 presenting the same authorization evidence E_{auth} to an auditor, where $\mathsf{Occurred}(w_1, s, a, t)$ holds and $\mathsf{Occurred}(w_0, s, a, t)$ does not. No deterministic auditor whose only input is E_{auth} can correctly decide occurrence in both worlds.*

Table 2: Evidence hierarchy. The rightmost column is a conditional statement about what acceptance supports, not an absolute security guarantee.

Evidence object	Required independent information	Claim supported on successful verification
Transition certificate	Trusted policy semantics and commitment	Proposed transition is admissible under that policy
Certificate history	Trusted genesis and, for a fixed claimed history, externally obtained final head	Presented sequence is a consistent, policy-admissible claimed path
Authenticated commit receipt	Pinned committer identity/key and truthful receipt-after-commit discipline	Trusted committer attested a specific commit; occurrence conditional on committer honesty
External effect verification	Independent observation or stronger environment-specific attestation	Selected real-world effects occurred under the observation model

Proof. The auditor is a function of its input. Because the input is identical in the two worlds, its output is identical. The correct occurrence answers differ; hence it must be wrong in at least one world. □

For a randomized auditor, the output distribution is likewise identical across the two worlds, so the input supplies no distinguishing advantage. This elementary observation is not an impossibility result about all possible forms of execution evidence. It establishes the limitation of *authorization-only* observations. The execution-binding gap is also discussed directly by Llambí-Morillas and Fernández-Fernández (2026).

4.2 Certificate histories and external commitments

A claimed history is *locally auditable* if each step passes V_π with its stated predecessor, and the chain begins at a separately trusted genesis state. For tamper evidence, define a public hash function H on canonical byte encodings and a chain

$$\ell_0 = 0^{256}, \qquad \ell_i = \mathsf{H}(\mathsf{Enc}(\ell_{i-1}, E_i)) \quad (i \geq 1). \tag{2}$$

An auditor may compare ℓ_n with an independently obtained expected head. Collision resistance makes many after-the-fact modifications detectable relative to that fixed external commitment. A *self-supplied* head offers no equivalent assurance: an adversary may rewrite the events and recompute their hash chain, provided each replacement certificate remains admissible. An externally trusted head also does not guarantee that every real-world commit was logged, or that the log producer reported events truthfully.

Proposition 4.2 (Fixed-head binding)**.** *Assume canonical unambiguous encoding, a collision-resistant hash, and resistance to finding a nonempty chain returning the distinguished genesis head. Given a fixed, independently trusted genesis and final chain head, it is computationally infeasible to replace the presented certificate history with a distinct well-formed history that produces the same head, except with negligible probability under these assumptions.*

Proof. If the substituted history has the same final head, take the latest position at which the hash-chain inputs differ while their output heads agree. This yields a collision in the hash-chain computation. If one history is empty, matching the genesis head with a nonempty history contradicts

the separate distinguished-head assumption. The argument depends on canonical encoding and the stated cryptographic assumptions. □

This is a standard cryptographic binding observation, not an assertion of append-only availability or a proof that all commits occurred. Stronger log-consistency and externally witnessed checkpoints are established topics in certificate-transparency work (Laurie et al., 2021).

4.3 Execution receipts require an execution trust anchor

For comparison, an authenticated execution receipt has the following schematic form

$$\begin{aligned} R_i &= (m_i, \mathsf{Sig}_{sk_C}(\mathsf{domain} \parallel \mathsf{Enc}(m_i))), \\ m_i &= (\mathsf{store}, i, \mathsf{H}(E_i), s_i, t_i, \ell_{i-1}, \mathsf{pid}(\pi)). \end{aligned} \tag{3}$$

A verifier with a securely pinned public key can establish that a signer attributed the record to a particular store and step. It cannot establish actual commit occurrence unless the signing and release protocol is itself tied to a successful authoritative commit.

Proposition 4.3 (Conditional receipt interpretation)**.** *Suppose a trusted committer controls the receipt signing key, never exports that key, and publishes a correctly bound receipt only after a successful atomic transition commit; suppose further that the signature scheme resists forgery and the auditor has the correct trusted verification key. Then acceptance of a fresh, correctly bound receipt supports the statement that the trusted committer recorded a successful commit of that transition, subject to the committer's stated correctness and honesty assumptions.*

Proof. Signature verification attributes the receipt to the committer, except in the forgery event. The trusted receipt-emission rule admits publication only following a successful commit of its bound transition. The conclusion follows conditionally from that rule and the trusted execution boundary. □

The reference implementation does not implement eq. (3)*.* It has no authenticated committer or independently pinned signing key. Even a correctly authenticated receipt supports claims about the trusted store, not arbitrary external effects such as physical device movement or payment settlement.

5 Executable Reference Construction

5.1 A minimal quota-and-permission specification

The prototype is a dependency-free Python 3.10+ program. It uses a synthetic authorization set of 256 labels `agent-0000`, . . . , `agent-0255`. Its authoritative state is

$$s = (q, v, \rho), \quad q \in \{0, 1, 2, 3\}, \quad v \in \mathbb{N},$$

where q is a quota, v a monotonically increasing version, and ρ the SHA-256 Merkle root of the committed authorization list. An action is a pair $a = (\mathsf{actor}, \delta)$ with $\delta \in \{-1, +1\}$. A proposed successor $t = (q', v', \rho')$ is accepted precisely when the current and claimed source states coincide, the actor label has a valid inclusion witness against trusted ρ, the policy identifier is correct, and

$$q' = q + \delta, \qquad 0 \le q' \le 3, \qquad v' = v + 1, \qquad \rho' = \rho. \tag{4}$$

A proposal that fails any check is rejected without changing the reference executor's state.

The membership tree uses domain-separated hashes for leaves and internal nodes. A tree of 2^8 labels gives an eight-sibling inclusion path. Certificates include the entire claimed source and target, action, policy identifier, and membership witness. The checker does not rely on a sender-supplied Boolean asserting that verification succeeded.

5.2 Independent verification and evidence export

The executor records each accepted certificate in a hash-linked event list. It exports a JSON history bundle plus a separate JSON genesis/head anchor. The independent auditing command consumes the bundle, trusted genesis, and expected head; it recomputes the hash links and evaluates each certificate under the prior claimed successor. It does not access the original executor object or receive the complete allowlist. Trust in the root and final head must come from outside the untrusted evidence bundle in a real application. The demonstration generates both files itself and therefore does not establish independent institutional anchoring.

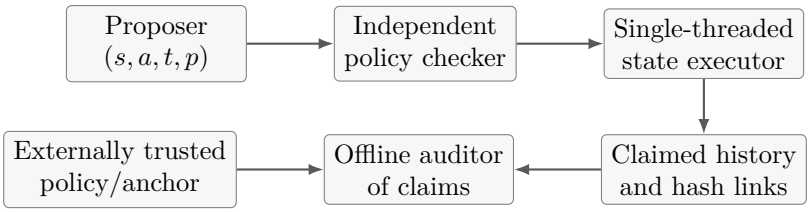


Figure 1: Reference construction. The audit establishes consistency of a claimed, policy-valid history. It does not authenticate an external commit or the identity of the actor label.

The reference implementation isolates the certificate and audit semantics. It does not include a network service, model API, production authorization system, public-key identity layer, signing authority, or secure hardware. The verifier operates on one explicit arithmetic policy, permitting direct inspection of its tests and verification rules.

6 Reproducible Evaluation

6.1 Test procedure and observed trace

The automated suite was executed using Python's `unittest` discovery. All 18 tests passed on the supplied reference artifact. These results verify the implemented test cases, not correctness over all possible inputs or attack models. The example begins at quota $q = 2$, version $v = 0$, and submits three successive proposals attributed to `agent-0042`. Table 3 reports the actual reference output.

Table 3: Reference execution. Only accepted proposals advance the version or alter the quota.

Attempt	Claimed action	Pre-quota	Decision	Post-quota	Post-version
1	$\delta = +1$	2	accept	3	1
2	$\delta = +1$	3	reject (over cap)	3	1
3	$\delta = -1$	3	accept	2	2

The exported history contains two accepted transitions. A separate offline audit verified those two claims and returned the same final state $(q, v) = (2, 2)$. This is a *certificate-history verification*; the auditor does not independently observe an external state-store commit.

6.2 Bounded test coverage

Table 4 groups the suite by property. It checks malformed or substituted witness data, stale versions, impossible quota successors, wrong policy roots and identifiers, malformed scalar values, reordered or modified histories, and incorrect trusted genesis. In addition, a test confirms that two alternative yet policy-valid histories cannot be distinguished by a head value supplied solely by the party supplying the history. Another test confirms that an R15-like direct guard can enforce the *same toy safety policy* without exported certificates.

Table 4: Reproduced test suite; grouping is descriptive, not a statistical coverage claim.

Test family	Cases covered	Count
Valid execution and matching policy guard	Positive certificate/audit; direct-guard comparison	2
State, arithmetic, and representation	Quota overflow; stale state; incorrect target; incorrect version; Boolean-as-integer input	5
Policy/witness binding	Unknown label with stolen path; changed path; wrong root; wrong policy ID; substituted actor label	5
History and external anchors	Reordered and modified history; alternative valid head; independence from full list; modified certificate with recomputed chain; wrong genesis	6
Total	All named tests completed successfully	**18**

A further small diagnostic, performed separately from the 18-test suite, kept the same actor label and membership path but changed the proposed action and recomputed a valid successor: both $\delta = +1$ and $\delta = -1$ were accepted from quota 2. This is not a failure of the quota invariant. It illustrates that the witness establishes membership of `agent-0042`, not authorization of a particular action by an authenticated principal.

6.3 Artifact sizes and costs

Table 5: Measured canonical compact-JSON sizes from the provided demonstration. These are data sizes, not throughput or latency benchmarks.

Artifact	Size
Complete synthetic allowlist (256 labels)	3,329 bytes
One Merkle inclusion witness	590 bytes
One source/action/target certificate	902 bytes
Two-transition history bundle	2,322 bytes
Merkle path of eight SHA-256 digests	256 raw bytes

The eight-digest path illustrates portability relative to shipping a full membership list with *one* decision. It does not establish an end-to-end storage advantage: repeated certificates may cost more than sharing a list once, and the JSON encoding adds overhead. No reliable latency or asymptotic-scaling measurements were made. Accordingly, the reported size measurements are not performance benchmarks.

7 Scope and Guarantees of Independent Verification

7.1 Portable verification beyond local monitoring

The companion reference monitor checks authorization using locally held policy data. The evidence exporter developed here allows a recipient to check a membership witness against a trusted root without receiving the complete policy list. This adds a *distribution and auditing interface* to local monitoring. Both reference implementations enforce the same quota invariant; the contribution is independently checkable evidence portability, not a stronger invariant guarantee.

7.2 The trust boundary cannot be hidden in the certificate

The checker must trust its own implementation, policy semantics, and input provenance. In the reference implementation, the membership root must be trusted, the genesis state must be fixed, and the final head must have an independent origin to detect history substitution. The entity identified by a certificate is merely an allowlisted label; neither possession of a Merkle path nor a correct quota update proves ownership of an agent identity. An authenticated signer is not part of the design.

Similarly, the reference executor's local assignment to a Python dictionary is not an independently attested commit in an external database. Logging a claimed successor after that assignment allows internal reproducibility but does not provide an independent witness to a state-store write. Cryptographic integrity of a *record* and correctness of the recorded *event* are different propositions.

7.3 Concurrency, rollback, and external side effects

The single-threaded reference code has no concurrency control beyond sequential operation. A parallel deployment must prevent time-of-check/time-of-use errors by binding verification to a linearizable commit point or an appropriate transactional abstraction (Herlihy and Wing, 1990). Version fields are useful for detecting stale certificates, but without atomic update semantics they do not close races. Rollback raises separate questions about whether a certified inverse transition is permitted, whether previously issued receipts can be invalidated, and how observers learn which history is authoritative. External side effects require their own evidence sources; certification of a requested payment, file write, or device command is not certification of its completion.

8 Applications and Engineering Implications

The reference construction supplies an inspectable framework for verifying portable transition evidence. A recipient can check a policy-list membership witness and audit a certificate history without receiving the complete authorization database or relying on the original proposer's decision. This separation is relevant to delegated administrative workflows, resource allocation, and automated change control whenever parties must examine the justification for a proposed state change across an organizational boundary.

Deployment requirements follow from the distinction among evidence levels. Identity-sensitive decisions require authenticated principals and request-bound credentials. Claims about completed state changes require receipts issued under a trustworthy commit protocol. Long-lived audit histories require policy-version management and externally anchored checkpoints. Established research on proof-carrying agent actions, cryptographic authorization, authenticated logs, and linearizable transactions provides the relevant mechanisms and constraints (Kadyrbek and Mansurova, 2026; Llambí-Morillas and Fernández-Fernández, 2026; Laurie et al., 2021; Herlihy and Wing, 1990).

The evaluation covers one arithmetic policy, one local executor, one independent auditor, and 18 automated tests. It does not include a machine-checked proof of the Python implementation, exhaustive security testing, principal authentication, signed execution receipts, performance benchmarking, or a connected language-model evaluation. These are distinct system properties and are not implied by the conditional safety theorem. In particular, checker soundness is a mathematical assumption, not a consequence of passing the test suite.

The resulting framework provides a reproducible basis for separating policy admissibility, audit consistency, and authenticated execution. It is suitable for examining the strengths and limits of independently checkable transition evidence without treating those distinct guarantees as interchangeable.

9 Conclusion

We have developed a formal and executable account of proof-carrying state transitions for AI-mediated updates. The transition semantics and conditional safety result establish how independently checked certificates support policy-admissibility claims. The indistinguishability argument shows that authorization evidence alone cannot establish the occurrence of a commit, while the history and receipt analyses identify the additional commitments and trust assumptions required for stronger claims.

The quota-and-permission reference implementation exports portable certificates and a hash-linked transition history that an offline auditor verifies against a trusted policy commitment. Its 18 passing tests and independently checked two-transition history provide reproducible evidence for those specified capabilities. The construction also makes its security boundary explicit: a membership witness does not authenticate an actor, and a consistent history is not an attestation of external execution.

The central contribution is a precise distinction between what an AI system is permitted to do, what its recorded history consistently claims, and what a trusted execution mechanism can substantiate as having occurred. Making these evidentiary levels explicit provides a practical foundation for evaluating and designing reliable AI state-transition systems.

Reproducibility and Reference Artifact

The accompanying implementation requires Python 3.10 or later and no third-party Python packages. From its directory, run:

```
python r16_poc.py demo -out .
python r16_poc.py verify -bundle sample_bundle.json -anchor trusted_anchor.json
python -m unittest discover -s .  -p 'test_*.py' -v
```

The accompanying source, tests, and example artifacts reproduce the certificate checks, bounded execution trace, and offline history audit reported in this paper.

References

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. *Innovations in Computer Science*, 2010. https://people.csail.mit.edu/alexch/research/pcd/pcd-ics.pdf.

Stephen Chong, Eran Tromer, and Jeffrey A. Vaughan. Enforcing language semantics using proof-carrying data. *IACR Cryptology ePrint Archive*, Report 2013/513, 2013. `https://eprint.iacr.org/2013/513`.

Adrian Diamond. Formal-state-augmented language models: Invariant-preserving execution with external formal state. AADSystems Research, November 1, 2026.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

Nurgali Kadyrbek and Madina Mansurova. Proof carrying agent actions enable verifiable licensing of language model tool use. *Scientific Reports*, October 13, 2026. doi:10.1038/s41598-026-74900-x.

Ben Laurie, Eran Messeri, and Rob Stradling. Certificate transparency version 2.0. *RFC 9162* (Experimental), December 2021. doi:10.17487/RFC9162.

Mar Llambí-Morillas and Daniel Fernández-Fernández. Cryptographically verifiable authorization for autonomous AI agents: A falsifiable hypothesis and proof of concept. *Frontiers in Computer Science*, 8, September 24, 2026. doi:10.3389/fcomp.2026.1966725.

George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997. doi:10.1145/263699.263712.