Verifiable Runtime Governance for Autonomous AI Agents via Cryptographic Receipts

Jack Brennan
Attested Intelligence
Illinois, United States
admin@attestedintelligence.com

***Abstract*—Autonomous AI agents invoke external tools through protocols such as the Model Context Protocol (MCP). Existing systems either evaluate policy without producing cryptographically verifiable records or provide provenance guarantees without capturing runtime authorization decisions. As a result, there is no standard mechanism for producing tamper-evident, independently verifiable evidence of agent behavior.**

We define and evaluate a governance receipt system that generates signed, hash-chained, and Merkle-proven records of authorization decisions at a gateway boundary. The system uses Ed25519 signatures, SHA-256 hashing, and RFC 8785 canonical JSON, with all signing keys held outside the agent.

We validate the design through four independent implementations against 37 shared test vectors and evaluate it against five adversarial scenarios, detecting four and explicitly identifying one failure boundary. The system adds under 50 ms overhead per tool call.

These results demonstrate that runtime agent governance can be expressed as a portable cryptographic evidence problem.

***Index Terms*—AI governance, cryptographic attestation, runtime integrity, Model Context Protocol, evidence bundles, compliance automation**

I. Introduction

Autonomous AI agents choose which tools to invoke, which data to access, and how to sequence operations. An agent operating under the Model Context Protocol (MCP) [1], an open standard for connecting AI models to external tools, can read files, execute commands, and send messages through JSON-RPC 2.0 interfaces. The governance infrastructure for recording these decisions has not kept pace with the autonomy of the agents making them.

Application logs stored in mutable databases can be altered, deleted, or fabricated after the fact. A log entry claiming a tool call was denied provides no cryptographic proof the denial occurred. Prior approaches address adjacent problems: Open Policy Agent [11] evaluates admission-time decisions without signed records; Sigstore [9] and SLSA [10] prove provenance without addressing runtime behavior; SCITT [13] defines signed supply chain claims without runtime authorization records; RATS [12] provides platform attestation but not software-level decision evidence. The missing layer is not policy evaluation or provenance, but verifiable evidence of runtime authorization decisions.

Regulatory frameworks are converging on requirements that demand better answers. The EU AI Act [6] requires post-deployment monitoring for high-risk systems. The NIST AI Risk Management Framework [4] calls for traceable decision-making. The NIST Zero Trust Architecture [7] defines policy enforcement points but not governance evidence formats. These frameworks establish requirements without specifying how to produce evidence that is cryptographically bound, tamper-resistant, and independently verifiable.

This paper makes the following claim: *runtime governance for autonomous AI agents can be expressed as offline-verifiable cryptographic evidence rather than mutable operational logs.* We demonstrate this through Attested Governance Artifacts (AGA), a receipt system combining four properties that, to our knowledge, no deployed system provides simultaneously: (1) sealed governance artifacts binding policy to agent identity; (2) a mandatory enforcement boundary holding all signing keys while the agent holds none; (3) structural metadata chains enabling integrity verification without content disclosure; and (4) signed receipts aggregated into Merkle-rooted evidence bundles for portable verification.

The contributions are: (1) a cryptographically specified governance receipt format with formally defined signing and chaining rules; (2) an offline-verifiable five-step verification procedure; (3) an evaluation of five adversarial scenarios with explicit failure boundaries; and (4) an implementation study across four independent codebases validated against 37 shared test vectors.

A. Scope and Non-Goals

This paper describes a governance evidence layer, not a complete security system. The receipt system is not a provenance framework (Sigstore and SLSA address that layer), not a platform attestation mechanism (RATS addresses that), not a policy language (OPA addresses that), not a replacement for hardware key protection, and not an end-to-end guarantee against a compromised portal. The system does not guarantee completeness of observation beyond the enforcement boundary: actions performed outside the enforcement boundary are out of scope. Section VII-D evaluates the portal-compromise scenario explicitly.

TABLE I
KEY TERMS USED IN THIS PAPER.

Term	Definition
Governance receipt	Signed JSON record of a single tool-call authorization decision.
Sealed artifact	Cryptographically signed policy object encoding the agent's known-good state and enforcement parameters.
Portal	Enforcement boundary process holding all signing keys; mediates agent interactions with external tools.
Gateway	Deployed proxy implementation of the portal, operating at the MCP protocol boundary.
Continuity chain	Append-only receipt sequence linked by hash references, providing ordering and tamper evidence.
Evidence bundle	Portable package of receipts, Merkle proofs, and public key, sufficient for offline verification.
Chain hash	SHA-256 of a receipt's complete canonical form including signature; links consecutive receipts.
Merkle proof	Sibling-path proof that a receipt was included in a checkpoint commitment.

B. Design Constraints

The architecture was shaped by four constraints. *Offline verifiability*: evidence must be verifiable on an air-gapped system using only the bundle and the signer's public key. *Minimal trust*: the system must not require trusting the bundle producer, the network, or any verification service. *Standard primitives*: Ed25519 [2], SHA-256, and RFC 8785 JCS [3] only. *Gateway deployability*: the system operates as a proxy without modifying the AI agent or the upstream tool server.

C. Terminology

Table I defines terms used throughout.

II. Threat Model and Trust Boundary

The system assumes an adversary with full access to the application database and network environment who does not possess the portal's Ed25519 signing key.

Trust boundary. The agent is untrusted. The portal is trusted only to the extent that its signing key has not been compromised. The bundle verifier is independent of the producing system. The upstream tool server is outside the governance trust domain entirely. Receipts establish evidence of what the portal decided, not what the upstream server did.

The adversary's goals span five categories: *retrospective tampering* (modifying records after decisions), *synthetic evidence* (fabricating authorization records), *silent omission* (deleting denial records), *policy rollback* (substituting an older, more permissive policy), and *privilege accumulation* (chaining individually authorized calls to produce an unauthorized composite effect).

A compromised portal can sign false receipts; this is the system's principal limitation (Section VII-D). Mitigations include hardware security modules, trusted execution environments, and multi-party signing. These are complementary and layerable.

Each adversarial goal maps to a verification property: retrospective tampering and insertion target chain integrity (Section IV), omission targets sequence continuity and Merkle inclusion (Section VI), policy rollback targets policy binding via `policy_reference`, and collusion targets the trust boundary itself. This mapping is evaluated explicitly in Section VII-D.

III. System Architecture

AGA operates in three phases: **Seal**, **Enforce**, and **Prove**.

A. Seal

Before execution, a governance artifact is sealed by computing a reference hash over the agent's known-good state. The sealed hash combines the binary digest, metadata digest, policy reference, and a random salt using 4-byte big-endian length-prefixed encoding:

$$\begin{aligned} \mathit{sealed} = \text{SHA-256}(&\text{len32}(\mathit{bytes\_hash}) \| \mathit{bytes\_hash} \\ &\| \text{len32}(\mathit{meta\_hash}) \| \mathit{meta\_hash} \\ &\| \text{len32}(\mathit{policy\_ref}) \| \mathit{policy\_ref} \\ &\| \text{len32}(\mathit{salt}) \| \mathit{salt}) \end{aligned} \tag{1}$$

The function $\text{len32}(x)$ encodes the byte length of x as a 4-byte big-endian integer, preventing field boundary ambiguity. An Ed25519 signature over the sealed hash binds the artifact to its issuing authority. The artifact is created by a policy author at attestation time and is immutable thereafter.

B. Enforce

The agent executes within the portal, a mandatory enforcement boundary. In NIST SP 800-207 [7] terms, the portal operates as a Policy Enforcement Point. The portal holds all signing keys; the agent holds none.

The portal measures runtime state across multiple dimensions: executable image digest, container image digest, configuration manifest, file system state, loaded modules, system prompt integrity, model identity, and tool-call frequency. The available measurements depend on the deployment mode: a host-level portal (e.g., a Kubernetes operator) can measure binaries and modules directly, while a protocol-level gateway measures tool calls, system prompts, and model identity at the MCP boundary. Measurements run at cadences from sub-second (safety-critical) to minute-interval (lower-risk).

When measured state diverges from the sealed reference, the portal executes a predetermined action: termination, quarantine, network isolation, or safe-state transition. For contexts where termination is unacceptable, safe-state transitions execute controlled degradation specified in the sealed artifact. Every action generates a signed receipt.

C. Prove

Evidence bundles package the sealed artifact, signed receipts, Merkle inclusion proofs, and the public key. Verification requires only SHA-256 and Ed25519 with no network access.

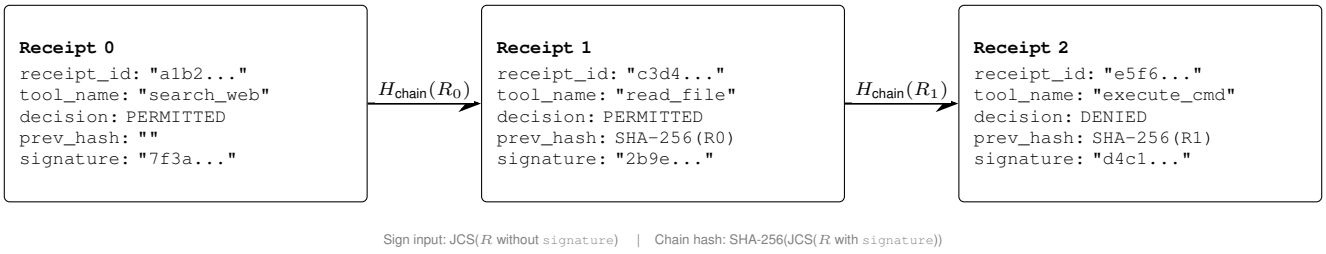

Fig. 1. Three receipts linked by SHA-256 chain hashes. Each receipt's `previous_receipt_hash` contains the hash of the preceding receipt's complete canonical form including signature. The signing input excludes the signature; the chain hash includes it.

IV. Receipt Format and Integrity Structures

A. Structural Metadata Hash

The leaf hash for each continuity chain event is computed over seven structural metadata fields. The event payload is excluded:

$$\begin{aligned} \mathit{leaf} = \text{SHA-256}(&\text{len32}(\mathit{SchemaVer}) \| \mathit{SchemaVer} \\ &\| \text{len32}(\mathit{ProtoVer}) \| \mathit{ProtoVer} \\ &\| \text{len32}(\mathit{EventType}) \| \mathit{EventType} \\ &\| \text{len32}(\mathit{EventId}) \| \mathit{EventId} \\ &\| \text{len32}(\mathit{SeqNum}) \| \mathit{SeqNum} \\ &\| \text{len32}(\mathit{Timestamp}) \| \mathit{Timestamp} \\ &\| \text{len32}(\mathit{PrevLeaf}) \| \mathit{PrevLeaf}) \end{aligned} \tag{2}$$

Payload exclusion enables an auditor to verify ordering, completeness, and tamper evidence without seeing event contents. Payload integrity is independently protected through event signatures and a content-addressable payload hash in each event's metadata.

B. Receipt Format

Each receipt contains 15 required fields with no optional security-critical fields. The schema is versioned via `receipt_version` and is forward-extensible; verifiers reject unknown algorithm identifiers but may ignore unknown non-critical fields. The fields are: `receipt_id`, `receipt_version`, `algorithm` (`"Ed25519-SHA256-JCS"`; verifiers MUST reject unknown values), `timestamp`, `request_id` (JSON-RPC identifier preserved exactly), `method`, `tool_name`, `decision` (`PERMITTED` or `DENIED`), `reason`, `policy_reference` (SHA-256 of canonical policy), `arguments_hash`, `previous_receipt_hash`, `gateway_id`, `signature`, and `public_key`.

The `arguments_hash` distinguishes three cases: absent arguments produce an empty string, an empty object produces the hash of canonical `{}`, and populated arguments produce the hash of their canonical form. Implementations must not conflate absent and empty. All fields are required; missing required fields cause verification failure.

C. Chain Construction

The signing input is the receipt's RFC 8785 canonical form with `signature` removed. The chain hash is SHA-256 of the complete canonical receipt including the signature. This prevents signature substitution: replacing a signature with a different valid one breaks the chain because subsequent receipts commit to the original via the chain hash. Verifiers use constant-time comparison when checking computed roots against the stated root to prevent timing side channels.

D. Three Integrity Layers

The system uses three hash structures, each serving a distinct verification purpose:

1) **Chain hash** (receipt-level): tamper-evident ordering. Modifying any receipt invalidates all subsequent hashes. Requires sequential processing of the full chain.
2) **Structural metadata hash** (event-level): integrity verification without payload disclosure. An auditor verifies chain structure without seeing tool-call arguments or sensitive content.
3) **Merkle root** (bundle-level): efficient subset verification. An auditor verifying one receipt confirms its inclusion in a checkpoint without processing the entire chain.

A chain alone requires processing every receipt. A Merkle tree alone does not enforce ordering. Structural hashing alone does not bind receipts to payloads. The three layers are complementary: ordering (chain), inclusion (Merkle), and privacy-preserving structure (metadata hash) are verified independently.

V. Gateway Deployment

The receipt system deploys as a proxy between MCP clients and upstream servers. The proxy evaluates `tools/call` messages against sealed policy, signs a receipt, and forwards or denies. All authorization decisions are made synchronously prior to tool invocation, ensuring that every permitted or denied action is captured as a receipt before execution proceeds. Non-tool-call messages pass through without receipts.

Three policy modes are supported: allowlist, denylist, and audit-only. Per-tool constraints enforce on path-segment boundaries. The gateway is fail-closed: unextractable tool names produce denial receipts.

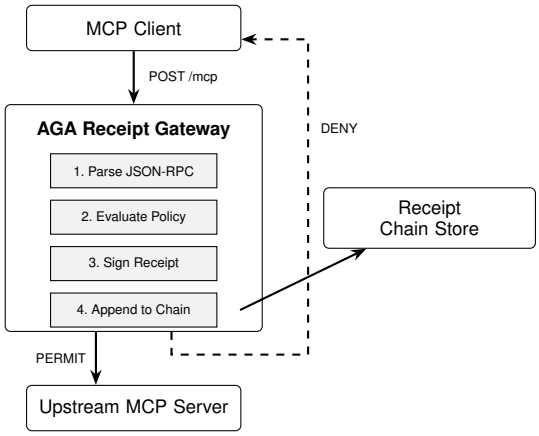

Fig. 2. Gateway proxy architecture. The proxy evaluates tool calls against sealed policy, signs a receipt, and forwards or denies. All receipts are persisted regardless of outcome.

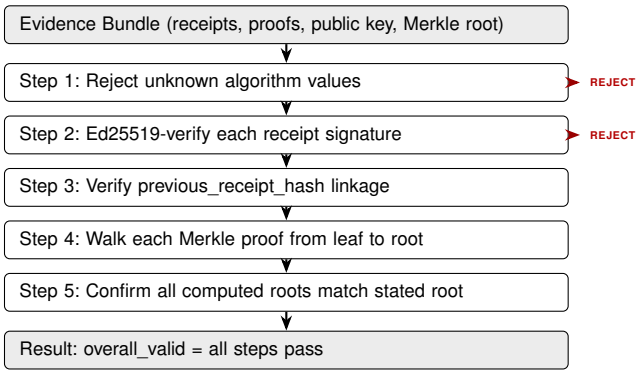

Fig. 3. Five-step verification. All steps execute offline. Unknown algorithm values cause rejection at Step 1 without cryptographic operations. Results report each step individually.

The agent's system prompt and model identity are sealed via SHA-256 digests. The gateway complements token-based authorization: an OAuth 2.0 token grants connectivity; the sealed artifact constrains what the agent does with it.

VI. Evidence Bundles and Verification

A bundle contains the schema version, algorithm identifier, public key, policy reference, receipt array, Merkle root, and one proof per receipt.

Definition (Valid Bundle). An evidence bundle is valid if and only if all five verification steps succeed for every included receipt. Verification is deterministic and stateless: the result depends only on the bundle contents and the public key.

Step 1 rejects unknown algorithm identifiers. Step 2 removes `signature`, canonicalizes per RFC 8785, and verifies Ed25519. Step 3 confirms chain linkage. Step 4 walks Merkle proofs from leaf to root. Step 5 confirms all roots match the bundle-level root. Any failure in Steps 1–5 results in immediate rejection. Verification produces a binary result with no partial validity states.

TABLE II
Cross-implementation bundle verification.

Source \ Verifier	Python	Go	TypeScript
Python	PASS	PASS	PASS
TypeScript	PASS	PASS	PASS
Simulated env.	PASS	PASS	PASS

A. Security Properties and Non-Goals

The verification procedure establishes four properties: *tamper evidence* (modifying any receipt invalidates subsequent chain hashes and its Merkle proof), *ordering integrity* (the chain enforces a total order), *offline auditability* (all steps except optional anchor validation require no network), and *payload privacy* (structural metadata hashes exclude payloads).

The system does not guarantee that the sealed policy is correct or complete, that the upstream server complied with the portal's decision, or that the signing boundary has not been compromised. These are non-goals evaluated in Section VII-D. The receipt system establishes evidence of governance decisions, not correctness of those decisions.

VII. Evaluation

A. Implementations

Four independent implementations validate the receipt format as an interoperable protocol rather than an implementation-specific artifact. The Go implementation (31,308 lines, 25 packages, 77.6% coverage) includes a CLI, Kubernetes webhook, and MCP proxy. TypeScript deploys as a Cloudflare Worker with one dependency. Python is published on PyPI as `aga-governance`. A fourth was tested in a simulated multi-domain autonomous systems environment, producing 33 receipts across 10 assets in 4 domains. Scenarios addressed industrial control with sub-second measurement cadences, autonomous navigation with safe-state enforcement, and multi-agent coordination with delegated authority constraints.

B. Cross-Language Verification

The 37 shared test vectors cover canonicalization edge cases (negative zero, Unicode, empty keys), algorithm identity, chain continuity, and cross-language equivalence, ensuring byte-identical behavior across all four codebases.

C. Performance

Primitive benchmarks confirm cryptographic overhead is negligible: sealing completes in 891 ns, leaf hashing in 1,874 ns. System composition cost is modest: a full lifecycle completes in 0.19 s. In a Cloudflare Worker deployment, total per-call overhead is under 50 ms, small relative to typical MCP tool latency. Concurrent testing with 50 requests confirmed correct decisions with zero data races.

The overhead is layered and can be understood in terms of individual feature costs. Signing alone (receipt generation without chaining) accounts for roughly 8 ms per call. Adding

TABLE III
ADVERSARIAL RESULTS. EACH ROW MAPS A THREAT TO THE TARGETED PROPERTY, DETECTION MECHANISM, VERIFICATION STEP, AND OUTCOME.

Threat	Property	Method	Step	Result
Replay	Ordering	Chain hash mismatch	3	Detected
Insertion	Integrity	Adjacent hashes fail	3	Detected
Omission	Coverage	Seq. gap, Merkle	3, 4	Detected
Rollback	Policy binding	policy_ref hash	N/A	Detected
Collusion	Trust boundary	Valid signatures	N/A	**Not detected**

chain append with persistent storage raises the total to approximately 33 ms. Adding Merkle proof computation for bundling adds marginal cost at export time rather than per-call time. Based on these measurements, persistent storage is the dominant observed contributor to per-call latency rather than cryptographic operations.

D. Adversarial Scenarios

To evaluate security properties rather than only engineering characteristics, we tested five scenarios against the threat model. The scenarios target the exact verification properties claimed by the system: ordering, completeness, policy binding, and trust-boundary integrity.

In *replay*, the attacker copies a valid receipt and appends it to the chain head. The copied receipt's `previous_receipt_hash` points to its original predecessor rather than the current chain head, producing an immediate mismatch at Step 3. In *insertion*, the attacker fabricates a receipt and places it between two legitimate ones. Both the fabricated receipt's previous hash and the following receipt's previous hash fail chain verification, since the following receipt still commits to the original predecessor. In *omission*, the attacker deletes a denial receipt to conceal an unauthorized tool-call attempt. The gap produces a sequence number discontinuity detectable at Step 3 and a Merkle proof failure at Step 4, since the deleted receipt's leaf is absent from the tree. In *rollback*, the `policy_reference` hash computed from the substituted policy does not match the hash recorded in receipts generated under the current policy.

Portal collusion is the scenario the system does not detect. A compromised portal signs `PERMITTED` receipts for actions that were denied or never evaluated. These pass all verification steps because they carry valid signatures from a trusted key. This defines the system's explicit trust-boundary failure condition. No cryptographic verification procedure can detect this condition without introducing an additional trust anchor.

Mitigations are complementary: HSM key storage, TEE-isolated portals, and multi-party signing.

Across five scenarios, the system detected four classes of chain, policy, and omission attacks and failed only in the portal-compromise case, which lies outside the trust boundary by design. All attacks targeting integrity, ordering, and completeness are detected by deterministic verification steps. The only undetected class requires adversary control of the signing boundary itself.

TABLE IV
FUNCTIONAL PROPERTY COMPARISON FOR RUNTIME AI AGENT GOVERNANCE. ✓ = SUPPORTED, ○ = PARTIAL.

	Logs	*OPA*	*Sigstore*	*SCITT*	*RATS*	***AGA***
Signed decisions				✓	✓	✓
Runtime enforce.		✓				✓
Tamper evidence	○	○	✓	✓	○	✓
Ordering						✓
Offline verify			✓	✓	○	✓
Agent tool gov.		○				✓

VIII. RELATED WORK

Table IV compares properties for runtime AI agent governance. The comparison is functional: each cell marks whether the system natively provides the property in the context of runtime agent tool-call governance, not whether it could be extended to do so.

Sigstore [9] addresses provenance. SLSA [10] addresses build integrity. AGA addresses the subsequent question of post-deployment behavior. SCITT [13] applies signed statements to supply chain claims; the receipt model applies a similar approach to runtime decisions. RATS [12] complements at the platform layer.

OPA [11] evaluates admission-time policy but leaves runtime evidence unspecified; its decision logs lack native cryptographic chaining or signature binding. SPIFFE/SPIRE provides workload identity; AGA answers a different question: what did this workload decide to do? Certificate Transparency [14] applies Merkle logs to TLS certificates; the receipt chain applies a similar structure at the application layer.

Prior approaches have not, to our knowledge, combined cryptographic binding, continuous measurement, autonomous enforcement, and independently verifiable evidence for runtime agent governance. The distinction is categorical: prior systems record or enforce decisions, whereas AGA produces independently verifiable evidence of those decisions. The comparison focuses on native capabilities rather than potential extensions.

IX. LIMITATIONS

Portal trust is the principal limitation (Section VII-D). The system provides evidence of decisions, not correctness of decisions; it does not prevent misuse of correctly authorized capabilities. The system does not attempt to solve correctness of policy, only verifiability of enforcement. The architecture is algorithm-agile: receipt metadata specifies the signature algorithm, enabling post-quantum migration

without architectural changes. Key rotation requires starting a new chain; a registry extension is planned. Domain-separated hashing would prevent cross-domain collisions. Multi-agent delegation where scope diminishes through delegation is defined but not yet implemented. Behavioral drift detection beyond binary measurement remains open.

X. Conclusion

We have presented a receipt system that produces cryptographic evidence of autonomous AI agent tool-call decisions. Evaluation against five adversarial scenarios detected four attack classes and identified one explicit failure boundary. Four implementations validated against 37 test vectors demonstrate interoperability, and performance confirms sub-50ms overhead per governed call.

The central result is that runtime governance for autonomous agents can be reduced to a deterministic, offline-verifiable cryptographic evidence problem. This shifts the question from trusting operational systems to verifying signed artifacts. The receipt system demonstrates a concrete, deployable instantiation of this model.

References

[1] Anthropic, "Model Context Protocol Specification," 2024. [Online]. Available: https://modelcontextprotocol.io/specification

[2] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," RFC 8032, IETF, Jan. 2017.

[3] A. Rundgren, B. Jordan, and S. Erdtman, "JSON Canonicalization Scheme (JCS)," RFC 8785, IETF, Jun. 2020.

[4] NIST, "Artificial Intelligence Risk Management Framework (AI RMF 1.0)," NIST AI 100-1, Jan. 2023.

[5] M. Souppaya, K. Scarfone, and D. Dodson, "Secure Software Development Framework (SSDF), Version 1.1," NIST SP 800-218, Feb. 2022.

[6] European Parliament, "Regulation (EU) 2024/1689 Laying Down Harmonised Rules on Artificial Intelligence (AI Act)," Jun. 2024.

[7] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture," NIST SP 800-207, Aug. 2020.

[8] CISA, "Secure by Design," 2024. [Online]. Available: https://www.cisa.gov/securebydesign

[9] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software Signing for Everybody," in *Proc. ACM CCS*, 2022, pp. 2353–2367.

[10] Google, "SLSA: Supply-chain Levels for Software Artifacts, v1.0," 2023. [Online]. Available: https://slsa.dev/spec/v1.0/

[11] OPA Contributors, "Open Policy Agent," 2021. [Online]. Available: https://www.openpolicyagent.org

[12] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote ATtestation procedureS (RATS) Architecture," RFC 9334, IETF, Jan. 2023.

[13] H. Birkholz *et al.*, "An Architecture for Trustworthy and Transparent Digital Supply Chains," Internet-Draft, IETF, 2024.

[14] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," RFC 6962, IETF, Jun. 2013.