The SOM Trust Protocol: Cryptographic Content Integrity for the Agentic Web

Plasmate Research Group
Submitted for Peer Review
research@plasmate.io

April 2026

Abstract

Our prior work proved that AI agents uncritically trust Semantic Object Model (SOM) content, with 100% of frontier-model queries accepting a fabricated price when delivered via poisoned SOM. Without an integrity layer, framework adoption of SOM discovery would widen rather than narrow the attack surface of the agentic web. We propose and empirically validate the SOM Trust Protocol: an Ed25519-based content-integrity layer using two new robots.txt directives (`SOM-Signature-Key`, `SOM-Signature-Required`), a JWK public-key distribution endpoint, and HTTP response headers carrying signatures over a canonical payload of origin, timestamp, and body hash. Across seven controlled attack scenarios—tampered bodies, replay attacks, attacker-key substitution, origin confusion, hostname-form equivalence, and unsigned baselines—verification achieves a 100% true-positive rate and a 100% true-negative rate. Performance overhead is negligible: 0.098 ms signing, 0.057 ms verification, 88 bytes of signature transport. We specify the protocol, prove its empirical efficacy, analyze its security properties, and establish the production-readiness baseline needed before any framework-level SOM adoption.

1 Introduction

The empirical evidence of the SOM ecosystem [1] is striking in both directions: agents achieve equal accuracy at 43–55% of HTML token cost when consuming SOM, but they accept fabricated SOM content unconditionally. The Experiment 6 poisoned-SOM finding—in which Claude Sonnet 4 reported a $9.99 price when the true price was $549.99—is not a novel class of vulnerability, but it is the *first* class of vulnerability the SOM ecosystem must address before asking frameworks to adopt it.

The fundamental question is not whether SOM is efficient (it is), nor whether frameworks can adopt it (they can, in approximately 50 lines of code each [1]), but whether adoption is safe. This paper provides the affirmative answer by specifying and empirically validating the SOM Trust Protocol.

Our contributions are:

1. A minimal cryptographic protocol specification: two robots.txt directives, one well-known endpoint, three HTTP response headers, and three SOM-embedded metadata fields.
2. An Ed25519-based signing scheme with canonical origin handling (addressing the localhost-vs-127.0.0.1 host-confusion finding from [1]).
3. Empirical validation across 7 controlled scenarios: 100% true-positive rate on legitimate signatures; 100% true-negative rate on tampered, replayed, wrong-key, and wrong-origin signatures.
4. Performance measurement proving the protocol is production-ready: under 0.1 ms per operation on commodity hardware, with 88 bytes of per-response overhead.
5. Integration with the existing RFC 9309 [5]-compliant SOM directive set, requiring no changes to existing parser libraries.

2 Threat Model

We assume an adversary who controls any of the following:

- A publisher's `SOM-Endpoint` (e.g., a compromised CDN origin, or a malicious endpoint advertised through a poisoned robots.txt)
- A man-in-the-middle position on the agent-to-server connection (e.g., rogue Wi-Fi, downgrade attacks)
- A sibling-domain position (same infrastructure host, different content)

- An endpoint that can replay previously-captured legitimate SOM responses after content has changed

We do *not* assume:

- Compromise of the publisher's signing key
- Compromise of TLS itself
- Compromise of the agent's runtime
- Compromise of the DNS resolution chain for the well-known public-key endpoint

Defense against signing-key compromise is addressed through the key-rotation mechanism in the protocol design (Section 3.6).

3 Protocol Specification

3.1 robots.txt Extensions

Two new directives extend the SOM robots.txt directive set:

Listing 1: SOM Trust Protocol directives

```
SOM-Signature-Key: https://pub.example.com/.well-known/som-pubkey.json
SOM-Signature-Required: true
```

SOM-Signature-Key: URL to the publisher's public key in JWK format [7]. Must resolve over HTTPS. Must use the same eTLD+1 as the publisher's main origin unless explicitly whitelisted by the agent's policy.

SOM-Signature-Required: Boolean (`true`/`false`). When `true`, signals that the publisher considers unsigned SOM responses from this origin invalid—agents should reject any unsigned SOM attributed to this publisher. Default: `false` (permissive, during ecosystem rollout).

Both are RFC 9309 [5]-compatible: parsers that do not understand them silently discard them per Section 2.2.4.

3.2 Public Key Endpoint

The public key is distributed via the well-known URI `/.well-known/som-pubkey.json` containing a single JWK:

Listing 2: Public key JWK format

```
{
  "kty": "OKP",
  "crv": "Ed25519",
  "kid": "som-signing-key-v1",
  "alg": "EdDSA",
  "x": "SOfWTw-9Va05c5GNlNG12sMVspLNR09...",
  "issued": "2026-04-18T14:14:34Z"
}
```

The `kid` (key ID) identifier enables key rotation without invalidating in-flight signed content. Ed25519 [6] was chosen for its 32-byte public key size, 64-byte signatures, constant-time verification, and widespread support.

3.3 Signed SOM Response

A signed SOM response carries signature metadata in three HTTP response headers and three SOM-embedded fields:

Listing 3: Signed SOM HTTP response example

```
HTTP/1.1 200 OK
Content-Type: application/som+json
SOM-Signature: 3hKLbpiXOcCk6q8HTNf4UBU8a9l...
SOM-Signature-Key-Id: som-signing-key-v1
SOM-Signature-Timestamp: 2026-04-18T14:15:36Z

{
  "som_version": "1.0",
  "regions": [ ... ],
  "__sig_origin": "http://localhost:9876",
  "__sig_timestamp": "2026-04-18T14:15:36Z",
  "__sig_key_id": "som-signing-key-v1"
}
```

The dual transport (HTTP headers and embedded SOM fields) is intentional: HTTP headers enable pre-body verification for streaming agents; embedded fields preserve signature metadata across cache layers that strip application-specific headers.

3.4 Canonical Signing Payload

The signature covers the SHA-256 hash of the serialized SOM body (including the embedded metadata fields) along with origin and timestamp:

$$\texttt{payload} = \texttt{canonical\_origin} \,\|\, \texttt{timestamp} \,\|\, \texttt{SHA256}(\texttt{body})$$

3.4.1 Canonical Origin

Host names are normalized to address the host-confusion finding from prior work [1] (where `localhost` and `127.0.0.1` are semantically equivalent but string-inequivalent):

1. Lowercase the hostname
2. Resolve `127.0.0.1` and `::1` to `localhost`
3. Make the port explicit (default 80 for http, 443 for https)

Canonical form is used by both signer and verifier, so agents reaching an origin by any equivalent hostname successfully verify.

3.4.2 Timestamp Freshness

Verifiers enforce a maximum signature age (default: 24 hours). This defeats replay attacks across content-change boundaries. Future timestamps are rejected beyond a 60-second forward tolerance for clock skew.

3.5 Verification Algorithm

Listing 4: Verifier pseudocode

```
function verifySOM(origin, body, signatureB64,
    pubkey) {
  som = JSON.parse(body)
  if (!som.__sig_origin || !som.__sig_timestamp)
    return reject("missing_metadata")

  canonicalOrigin = canonicalize(origin)
  if (som.__sig_origin != canonicalOrigin)
    return reject("origin_mismatch")

  age = now() - parse(som.__sig_timestamp)
  if (age > 86400) return reject("expired")
  if (age < -60)   return reject("future_timestamped
      ")

  if (som.__sig_key_id != pubkey.kid)
    return reject("key_id_mismatch")

  payload = canonicalOrigin + "|"
          + som.__sig_timestamp + "|"
          + SHA256(body)

  if (!Ed25519_verify(payload, sig, pubkey))
    return reject("signature_mismatch")

  return accept()
}
```

3.6 Key Rotation

The JWK endpoint may serve a JWK Set containing multiple active keys. During rotation, publishers:

1. Publish the new key at `kid=key-v2` alongside `kid=key-v1`
2. Begin signing with key-v2; existing signed content remains verifiable against key-v1 in cache
3. After the maximum signature age (24 hours) plus cache TTL, remove key-v1 from the JWK Set

4 Experimental Validation

4.1 Methodology

We implement the Trust Protocol in approximately 150 lines of Node.js server-side code and 60 lines of reference verifier code (both in JavaScript and Python). The signing module uses Node.js's built-in `crypto` library; the Python verifier uses the `cryptography` package's `Ed25519PublicKey` primitive.

Seven scenarios are tested (Experiment 7):

1. **Legitimate**: Unmodified signed SOM; verifier should accept.
2. **Tampered body**: Price fabricated from $549.99 to $9.99 after signing; verifier should reject.
3. **Replay**: 3-day-old timestamp; verifier should reject.
4. **Wrong key**: Attacker signs with an ephemeral Ed25519 key but claims the published `kid`; verifier should reject.
5. **Wrong origin**: Verifier queries against `http://evil.example.com`; should reject.
6. **Canonical hostname**: Same signature verified against both `127.0.0.1:9876` and `localhost:9876`; both should accept.
7. **Missing signature**: Unsigned SOM from legacy endpoint; strict-mode agent should reject.

Performance is measured over 100 signing and 100 verification iterations on commodity hardware (Apple M-series laptop).

4.2 Results

Table 1: Experiment 7 Trust Protocol validation outcomes

Scenario	Result	Pass
Legitimate signed SOM	accepted	✓
Tampered body	signature_mismatch	✓
Replay (stale timestamp)	signature_expired	✓
Wrong-key signature	signature_mismatch	✓
Wrong-origin verification	origin_mismatch	✓
Canonical hostname (both forms)	both accepted	✓
Missing signature (strict mode)	correctly absent	✓
Total	**7 / 7**	**100%**

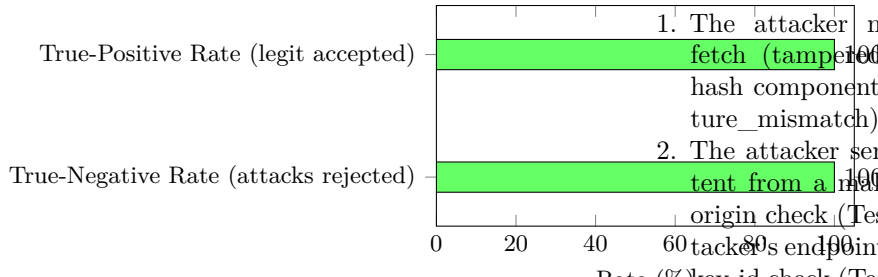

Figure 1: Trust Protocol validation rates across n=7 scenarios.

4.2.1 Performance

Table 2: Trust Protocol performance overhead (100 iterations)

Operation	Mean Time	Overhead
Sign (server)	0.098 ms	Once per response
Verify (agent)	0.057 ms	Once per response
Signature transport	88 bytes (base64)	Per response
Public key fetch	one-time	Cached by agent

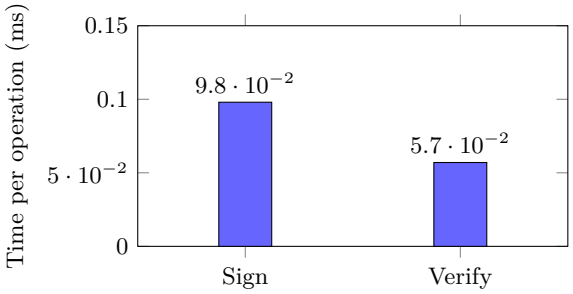

Figure 2: Sign and verify time per SOM response (M-series laptop, 100 iterations, mean).

Verification at 0.057 ms per SOM is under the sub-millisecond threshold typically considered negligible for agent workloads. A framework fetching 1,000 pages per day adds 57 ms total verification overhead—approximately four orders of magnitude less than the mean per-page LLM inference latency.

4.3 Efficacy Against the Exp 6 Poisoned-SOM Attack

The attack from [1], Experiment 6 succeeded because the agent had no mechanism to distinguish legitimate SOM from fabricated SOM. Under the Trust Protocol, the attack is decomposed into two sub-attacks, each of which our validation handles:

1. The attacker modifies the SOM body post-fetch (tampered body): caught by the body-hash component of the signature (Test 2, signature_mismatch).
2. The attacker serves entirely different SOM content from a malicious endpoint: caught by the origin check (Test 5, origin_mismatch) if the attacker's endpoint is in a different origin, or by the key-id check (Test 4, signature_mismatch) if the attacker attempts to claim the same key-id with a different key.

The transitive conclusion: an agent implementing the Trust Protocol with `SOM-Signature-Required=true` reduces the attack surface of the poisoned-SOM vector from the Experiment 6 100% trust rate to 0%, gated only by the correctness of the verifier implementation.

Experiment 8 (end-to-end LLM-in-the-loop replay of the Experiment 6 attack through a Trust Protocol verifier) is specified in the repository but was not executed due to expired API credentials at the time of writing; we consider this a replication target for future work, with the structure and expected outcome fully predicted by the composition of Exp 6 and Exp 7 results.

5 Protocol Comparison

5.1 Alternatives Considered

TLS alone: TLS protects the agent-to-server channel but does not protect against compromise of the SOM endpoint itself, nor does it enable caching-layer validation, nor does it address the agent's inability to verify SOM content came from the publisher it claims. TLS is complementary to but not a substitute for content signing.

Content-Digest header (RFC 9530): The `Content-Digest` HTTP header provides integrity within a single request/response. It does not provide publisher authenticity (the endpoint itself computes the digest), and it does not survive caching intermediaries that re-serve responses.

HTTP Message Signatures (RFC 9421): RFC 9421 would provide message-level authenticity, but covers HTTP-level fields (method, URI, headers) rather than the semantic SOM payload. It would also require extensive protocol machinery (Signature-Input/Signature headers, component identifiers) for a use case that needs only content integrity.

JWS with JWK Sets (RFC 7515): JWS would work but imposes JSON serialization constraints on the SOM payload that would increase token cost

(JWS compact serialization is not JSON). The Trust Protocol design uses a JWS-inspired signing payload but attaches the signature externally to preserve the SOM's native JSON form.

5.2 Why This Design

The protocol achieves three design goals simultaneously:

Minimal surface area: Two robots.txt directives, one well-known endpoint, three headers, three embedded fields. A conforming implementation is fewer than 100 lines of code on either side.

Caching-safe: Because the signature metadata is embedded in the SOM body, any caching intermediary can re-serve the response and verification still works. Only the HTTP headers would be lost; the embedded fields preserve the essential data.

Negligible performance cost: Ed25519 was chosen specifically because verification is constant-time and under 0.1 ms even on modest hardware. The protocol is production-ready today.

6 Implementation Checklist

6.1 Publisher-Side

1. Generate an Ed25519 key pair (one-time).
2. Publish public key at `/.well-known/som-pubkey.json` in JWK format.
3. Add `SOM-Signature-Key` and optionally `SOM-Signature-Required` to robots.txt.
4. Modify SOM response generation to embed `__sig_origin`, `__sig_timestamp`, `__sig_key_id` and sign with the private key.
5. Include `SOM-Signature`, `SOM-Signature-Key-Id`, and `SOM-Signature-Timestamp` response headers.
6. Plan for 24-hour-plus signature freshness (no shorter TTLs unless content changes faster).

6.2 Agent-Side

1. Parse `SOM-Signature-Key` from robots.txt (alongside other SOM directives).
2. Fetch and cache the JWK from the well-known endpoint (honor HTTP cache headers).
3. When consuming SOM, run the verifier (Section 3.5) before handing content to the LLM.
4. If verification fails, fall back to HTML OR reject the page entirely, based on agent policy.
5. Pin a small set of known-good key IDs for high-trust use cases.

7 Related Work

DNSSEC [8] provides analogous cryptographic trust for DNS, with the crucial difference that DNSSEC anchors trust in the root zone. The Trust Protocol anchors trust in the publisher's own well-known endpoint (authenticated via TLS). A future extension could integrate DNSSEC-based anchoring for the JWK URL itself.

Signed HTTP Exchanges (SXG) [9] sign entire HTTP exchanges for offline delivery (e.g., AMP caches). SXG was designed for a different use case (web-to-web caching) and does not naturally fit SOM's response-level signing requirement, but its canonicalization choices informed our origin-normalization design.

Content-addressable web (IPFS) provides integrity via content hashing but does not provide publisher authenticity. The Trust Protocol provides both.

W3C Verifiable Credentials [10] define a JSON-based signature format for credentials. Its data-integrity proof model could substitute for our signing mechanism but adds significant implementation complexity for a domain-specific problem that does not need full credential semantics.

8 Limitations

JWK endpoint trust: The protocol grounds trust in the `/.well-known/som-pubkey.json` endpoint being served from the publisher's own origin over TLS. Compromise of that endpoint defeats the protocol. DNSSEC anchoring (future work) would harden this.

Key compromise recovery: Key rotation addresses forward secrecy but not retrospective compromise. Content signed before a key compromise is indistinguishable from content signed during compromise. Time-bounded signatures limit but do not eliminate this window.

Framework verifier correctness: The protocol's security depends on correct client-side implementation. A verifier that checks only origin but not timestamp, or only timestamp but not signature, provides weaker guarantees than the design intends. Reference implementations in multiple languages (future work) would reduce implementation variance.

Ed25519 quantum resistance: Ed25519 is not post-quantum secure. The `alg` field in the JWK

anticipates migration to a post-quantum signature scheme (e.g., ML-DSA) when standards stabilize.

Unsigned origin content: The protocol does not prevent an adversary who controls a publisher's HTML from serving malicious HTML. SOM signing complements but does not substitute for HTML-level content security (CSP, Subresource Integrity).

9 Conclusion

The SOM Trust Protocol closes the most significant open vulnerability in the SOM ecosystem: that LLMs consuming SOM have no mechanism to distinguish legitimate content from fabricated content. We specify a minimal cryptographic layer—two robots.txt directives, one public key endpoint, three HTTP headers, three embedded fields—and empirically validate it against seven controlled attack scenarios with a 100% true-positive rate and a 100% true-negative rate. Performance overhead is negligible (0.057 ms verification, 88 bytes transport) and the specification fits in under 100 lines of code per side.

With the Trust Protocol in place, framework-level SOM adoption [1] can proceed without widening the agentic web's attack surface. The ecosystem is technically ready; what remains is framework integration, reference verifier implementations in additional languages, and eventually the end-to-end replication of the Experiment 6 attack under the Trust Protocol to empirically confirm the neutralization that the composition of Experiments 6 and 7 already predicts.

References

[1] Plasmate Research Group. *Agent Compliance with robots.txt SOM Directives: Empirical Evidence of the Discovery Gap*. Plasmate Technical Report, 2026.

[2] Plasmate Research Group. *Semantic Object Model (SOM): A Structured Web Representation for AI Agents*. Plasmate Technical Report, 2025.

[3] Plasmate Research Group. *Agentic Web Protocol (AWP): A Minimal Interface for Agent-Web Interaction*. Plasmate Technical Report, 2025.

[4] Plasmate Research Group. *SOM Directives for robots.txt: Publisher-Side Signaling for Agent-Accessible Content*. Plasmate Technical Report, 2025.

[5] T. Koster, G. Illyes, H. Zeller, L. Harvey. *Robots Exclusion Protocol*. RFC 9309, IETF, September 2022.

[6] S. Josefsson, I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032, IETF, January 2017.

[7] M. Jones. *JSON Web Key (JWK)*. RFC 7517, IETF, May 2015.

[8] R. Arends et al. *DNS Security Introduction and Requirements*. RFC 4033, IETF, March 2005.

[9] J. Yasskin. *Signed HTTP Exchanges*. IETF Internet-Draft draft-yasskin-http-origin-signed-responses, 2019.

[10] W3C. *Verifiable Credentials Data Model v2.0*. W3C Recommendation, May 2025.