# Contents

# Agent Trust Protocol (ATP): A Cryptographic Standard for AI Agent Identity, Scope, and Action Attestation

**Authors:** Guy Sheetrit
**Affiliations:** OTT Cybersecurity LLC / Lyrie.ai
**Status:** Public Draft v1.0 – May 2026
**IETF Intent:** Standards Track RFC (SEC Area)
**Repository:** https://github.com/OTT-Cybersecurity-LLC/lyrie-ai/tree/main/packages/atp
**License:** MIT (reference implementation)

---

## Abstract

AI agents deployed in production environments in 2025-2026 lack any standardized mechanism for cryptographic identity, authorized scope declaration, or attributable action logging. This gap enables identity spoofing, unbounded privilege escalation, action repudiation, and runtime state tampering – attack classes for which no existing standard (TLS, JWT, x.509, OAuth 2.0) provides coverage.

The Agent Trust Protocol (ATP) defines five cryptographic primitives to close this gap: (1) the **Agent Identity Certificate (AIC)**, a signed credential binding a unique agent instance to its model, system prompt, and authorized scope; (2) the **Scope Declaration Language (SDL)**, a formal authorization boundary with composition rules that prevent escalation across delegation chains; (3) the **Action Receipt**, a tamper-evident record attributing a specific tool invocation to a specific agent instance; (4) the **Trust Chain**, a verifiable ancestry linking root operator to leaf sub-agent with monotonically narrowing scope; and (5) **Breach Attestation**, a periodic signed state snapshot enabling runtime tamper detection. ATP uses Ed25519 exclusively for all signatures and canonical JSON (RFC 8785 JCS subset) for all serialization. This document specifies the protocol, its wire format, compliance levels, and security analysis. A reference implementation is available as `@lyrie/atp` (TypeScript, MIT, Node 20+). The authors intend to submit this specification to the IETF for Standards Track consideration.

---

## 1. Introduction

### 1.1 The Agent Security Gap

Between 2024 and 2026, AI agents transitioned from controlled research settings to production infrastructure. They run scheduled pipelines, execute shell commands, call external APIs,

manage files, and spawn sub-agents to decompose complex tasks. This transition happened faster than the security infrastructure supporting it.

The incident record illustrates the gap concretely. CVE-2026-30615 documented a remote code execution vulnerability in a widely deployed Model Context Protocol (MCP) server, where an attacker-controlled tool description injected shell commands into an agent's execution context. Post-incident analysis revealed that no component in the stack – the agent runtime, the orchestration layer, or the tool API – had any mechanism to verify that the agent invoking the tool was the agent it claimed to be, or that the invocation was within the agent's authorized scope. The attack exploited not a memory corruption bug or an authentication bypass in the classical sense, but rather the complete absence of authenticated identity at the agent layer.

The pattern recurs across incident classes:

- **Unbounded sub-agent privilege escalation**: A parent agent spawns a sub-agent and passes it credentials or tool access the sub-agent was never explicitly authorized to hold. The sub-agent, operating with no scope constraint, calls APIs the parent itself was not authorized to call directly.
- **Prompt injection hijacks**: An adversarial document in the agent's context window overwrites the agent's effective instructions, causing it to take actions attributable to the legitimate agent identity.
- **Tool poisoning**: A maliciously described tool in an agent's tool list causes the agent to exfiltrate data or execute unintended operations, with no audit trail linking the action to the specific tool invocation.

These are not alignment failures or model quality deficiencies. They are identity and authorization failures – the same class of problem that TLS solved for encrypted channel establishment, that x.509 solved for machine identity, that OAuth 2.0 solved for API authorization delegation. Each of those standards emerged because a class of infrastructure scaled faster than its security model.

The critical observation is that **none of the existing standards covers autonomous AI agents**:

- **TLS** (RFC 8446) authenticates the channel between two endpoints. It says nothing about what software is running at either end, what that software is authorized to do, or whether its actions should be trusted.
- **x.509** (RFC 5280) provides cryptographic identity for machines and services. It has no notion of a system prompt, an authorized tool list, or a delegation chain between computational agents.
- **JWT / JWS** (RFC 7519, RFC 7515) provides a signed token format typically used to convey user identity claims to APIs. It has no agent-specific semantics: no scope composition rules, no action receipts, no state attestation.
- **OAuth 2.0** (RFC 6749) governs how a user delegates API access to a client application. It was designed for human-in-the-loop consent flows; it has no model for a fully autonomous agent that issues its own sub-agents, each of which may further delegate.
- **SPIFFE/SPIRE** provides workload identity for containerized services. It answers "which service is calling which service" – not "which agent instance, running which

model, under which system prompt, with which authorized tool list, is taking which action."

The result is a structural gap: at the agent layer, there is no cryptographic identity, no authenticated scope, and no attributable action log. ATP exists to fill that gap.

### 1.2 ATP's Contribution

ATP specifies a minimal, composable set of cryptographic primitives that, taken together, provide:

1. **Verifiable agent identity** – any verifier can confirm that a specific agent instance was issued by a specific operator, is running a specific model under a specific system prompt, and holds a specific set of authorizations.
2. **Authenticated scope** – an agent's authorization boundary is declared at issuance time, signed by the issuing operator, and enforced at verification time with formal composition rules.
3. **Attributable action records** – every tool invocation can produce a signed receipt that irrefutably links the action to the issuing agent certificate, creating an auditable, tamper-evident log.
4. **Verifiable delegation chains** – when an agent spawns a sub-agent, the chain of custody from root operator to leaf agent is cryptographically verifiable and formally constrained to prevent scope widening.
5. **Runtime tamper detection** – periodic signed state snapshots allow operators and verifiers to detect whether an agent's context has been modified between attestation checkpoints.

ATP does **not** cover:

- **Model alignment**: ATP cannot verify that a model behaved correctly relative to its training objectives.
- **Output quality**: ATP makes no claims about the accuracy, safety, or usefulness of an agent's outputs.
- **Data privacy**: ATP specifies no encryption scheme for agent state or action payloads.
- **Behavioral detection**: ATP produces attestation artifacts that behavioral detection systems (such as the Agent Threat Rules corpus) can consume, but ATP itself specifies no behavioral rules.

The analogy to TLS is useful precisely once: e-commerce could not scale until every browser and server shared a common, interoperable mechanism for encrypted, authenticated channels. The agentic economy – where agents autonomously manage money, code, infrastructure, and communications – cannot scale responsibly until every agent runtime, orchestration layer, and tool API shares a common mechanism for identity, scope, and attribution. That is what ATP provides.

---

## 2. Threat Model

### 2.1 Principals

ATP recognizes five principals:

- **Operator**: The entity (human or organization) that issues Agent Identity Certificates and defines the root scope. The operator holds the root private key from which all agent trust derives.
- **Agent Instance**: A specific running instance of an AI model, identified by a UUID, bound to a system prompt hash and a scope declaration.
- **Tool/API**: An external service or capability that an agent may invoke. A tool may optionally counter-sign Action Receipts to provide mutual non-repudiation.
- **Sub-Agent**: An agent spawned by a parent agent. A sub-agent is itself an Agent Instance, but its AIC must reference its parent's CertId and its scope must be a subset of the parent's scope.
- **Verifier**: Any component (tool, orchestration layer, audit system) that inspects ATP artifacts and makes trust decisions based on them.

### 2.2 Attack Classes Addressed

**Identity Spoofing**: An adversarial process claims to be a legitimate agent instance in order to gain trust from a tool or orchestration layer. Without ATP, nothing prevents an arbitrary HTTP client from claiming any agent identity. With ATP, every agent identity is bound to a certificate signed by the operator's private key; a verifier that checks `verifyAIC()` will reject any unsigned or incorrectly signed claim.

**Scope Creep**: An agent – whether through prompt injection, misconfiguration, or adversarial sub-agent issuance – invokes tools or accesses resources outside its declared authorization boundary. ATP's SDL defines the boundary at issuance time. Scope composition rules ensure that no child agent in a delegation chain can hold permissions not granted to its parent. A verifier that enforces SDL will reject out-of-scope tool calls before execution.

**Action Repudiation**: An agent takes an action (deletes a file, transfers funds, exfiltrates data) and there is no reliable record attributing the action to a specific agent instance. ATP's Action Receipts are signed by the agent's private key over a canonical representation of the action and its parameters, binding action to identity irrefutably.

**State Tampering**: An adversary modifies an agent's memory, context window, or tool call history mid-session – through prompt injection, a compromised storage backend, or a man-in-the-middle attack on the agent's state – without the modification being detectable. ATP's Breach Attestation produces a hash chain over the agent's state at each checkpoint. A divergence between consecutive attestations signals tampering.

**Delegation Escalation**: A parent agent issues a sub-agent with broader permissions than the parent itself holds, either intentionally (misconfiguration) or through adversarial manipulation. ATP's Trust Chain verification enforces that each child's scope is a strict subset of its parent's scope, checked at verification time over the entire chain.

### 2.3 Out of Scope

ATP does not address:

- Whether the underlying model actually executes the certified system prompt faithfully (this is an attestation-of-execution problem, not an attestation-of-identity problem).
- The PKI design for distributing operator public keys (this specification defers key distribution to implementers; operator public keys may be distributed via HTTPS well-known endpoints, out-of-band registration, or existing PKI).
- Behavioral anomaly detection beyond state hash divergence.
- Encryption of ATP artifacts in transit (ATP assumes a confidential transport layer, e.g., TLS, for sensitive payloads).

---

## 3. Protocol Primitives

### 3.1 Agent Identity Certificate (AIC)

The Agent Identity Certificate is the foundational primitive. It is a signed JSON document that binds a unique agent instance identifier to its operator, model, system prompt, authorized scope, and public key.

#### 3.1.1 Structure

```
interface AgentIdentityCertificate {
  agentId: string;           // UUID v4 -- unique per agent instance
  modelId: string;           // Model identifier, e.g., "anthropic/claude-sonnet-4-6"
  systemPromptHash: string;  // SHA-256 of the system prompt, hex-encoded
  scope: ScopeDeclaration;   // SDL document (see Section3.2)
  operatorId: string;        // Operator identifier (URI or UUID)
  issuedAt: string;          // ISO 8601 UTC, e.g., "2026-05-12T08:00:00Z"
  expiresAt: string;         // ISO 8601 UTC
  publicKey: string;         // Ed25519 public key, base64-standard, 44 chars
  signature: string;         // Ed25519 signature over canonical JSON of above fields
  certId: string;            // SHA-256 of the canonical signed bytes, hex-encoded
  parentCertId?: string;     // Present if this is a sub-agent; references parent's certId
}
```

**3.1.2 Issuance** The issuing operator constructs the certificate with all fields except `signature` and `certId`, serializes it to canonical JSON (RFC 8785 JCS subset: keys lexicographically sorted, no insignificant whitespace, no trailing commas), signs the canonical bytes with the operator's Ed25519 private key, appends the base64-encoded signature, then computes `certId` as the SHA-256 of the canonical signed bytes (the complete canonical JSON including the `signature` field).

The `systemPromptHash` field binds the certificate to a specific system prompt. If an agent's runtime loads a different system prompt than the one hashed in the AIC, a conforming

verifier will reject the certificate. This binding is what prevents prompt injection from silently changing an agent's operating context without detection.

**3.1.3 Verification** `verifyAIC(cert: AgentIdentityCertificate, operatorPublicKey: string): boolean`

A conforming verifier MUST: 1. Check that `expiresAt` is after the current time. 2. Reconstruct the canonical JSON of the certificate excluding the `certId` field. 3. Verify the Ed25519 `signature` over those canonical bytes using `operatorPublicKey`. 4. Recompute `certId` and check it matches the stored value. 5. If `parentCertId` is present, retrieve the parent AIC and verify the Trust Chain (Section3.4).

**3.1.4 Certificate Lifecycle** An AIC follows the lifecycle: **issued -> active -> expired | revoked**. Expiry is enforced by `expiresAt`. Revocation is handled via the RevocationList mechanism (Section4.2). Short-lived certificates (minutes to hours) are recommended for high-risk agents; longer validity (days) is acceptable for low-privilege, read-only agents.

---

### 3.2 Scope Declaration Language (SDL)

The Scope Declaration Language provides a formal specification of what an agent instance is authorized to do. SDL documents are embedded in AICs and are therefore signed by the issuing operator.

#### 3.2.1 Structure

```
interface ScopeDeclaration {
  allowedTools: string[];         // Tool identifiers the agent may call
  deniedTools: string[];          // Explicit denials (take precedence over allowedTools)
  allowedDomains: string[];       // Domains the agent may make HTTP requests to
  maxSubAgentDepth: number;       // Maximum delegation chain depth (0 = cannot spawn sub-age
  requireApprovalFor: string[];   // Tool identifiers that require human approval before invo
  temporalScope: {
    notBefore: string;            // ISO 8601 UTC
    notAfter: string;             // ISO 8601 UTC
  };
  dataScope: {
    readPaths: string[];          // File system paths the agent may read
    writePaths: string[];         // File system paths the agent may write
    maxPayloadBytes: number;      // Maximum size of any single tool payload
  };
}
```

**3.2.2 Composition Rules** SDL composition is the mechanism that prevents delegation escalation. The invariant is: **a child agent's scope MUST be a subset of its parent agent's scope in every dimension.**

Formally, for a parent scope `P` and child scope `C`: - `C.allowedTools subset of P.allowedTools` - `C.deniedTools superset of P.deniedTools` (child inherits all parent denials, may add more) - `C.allowedDomains subset of P.allowedDomains` - `C.maxSubAgentDepth < P.maxSubAgentDepth` - `C.temporalScope` is contained within `P.temporalScope` - `C.dataScope.readPaths subset of P.dataScope.readPaths` - `C.dataScope.writePaths subset of P.dataScope.writePaths` - `C.dataScope.maxPayloadBytes <= P.dataScope.maxPayloadBytes`

A verifier that checks scope composition MUST reject any child AIC that violates these constraints. Scope composition is checked at Trust Chain verification time (Section3.4), not only at issuance time, ensuring that even if an incorrectly issued certificate reaches a verifier, the violation is caught.

#### 3.2.3 Example Declarations

**Research Agent** (read-only, no sub-agents):

```
{
  "allowedTools": ["web_search", "web_fetch", "memory_store", "memory_recall"],
  "deniedTools": ["exec", "file_write", "http_request"],
  "allowedDomains": ["*"],
  "maxSubAgentDepth": 0,
  "requireApprovalFor": [],
  "temporalScope": {
    "notBefore": "2026-05-12T00:00:00Z",
    "notAfter": "2026-05-12T23:59:59Z"
  },
  "dataScope": {
    "readPaths": [],
    "writePaths": [],
    "maxPayloadBytes": 65536
  }
}
```

**Executor Agent** (local filesystem + exec, no network):

```
{
  "allowedTools": ["exec", "file_read", "file_write", "file_delete"],
  "deniedTools": ["web_search", "web_fetch", "http_request"],
  "allowedDomains": [],
  "maxSubAgentDepth": 1,
  "requireApprovalFor": ["file_delete", "exec"],
  "temporalScope": {
    "notBefore": "2026-05-12T08:00:00Z",
    "notAfter": "2026-05-12T20:00:00Z"
  },
  "dataScope": {
    "readPaths": ["/home/user/project"],
    "writePaths": ["/home/user/project/output"],
```

```
    "maxPayloadBytes": 1048576
  }
}
```

---

### 3.3 Action Receipt

An Action Receipt is a signed record that a specific agent instance, identified by its AIC, took a specific action at a specific time. Action Receipts are the atomic unit of the ATP audit trail.

#### 3.3.1 Structure

```
interface ActionReceipt {
  receiptId: string;           // UUID v4 -- unique per receipt, for replay resistance
  agentCertId: string;         // certId of the issuing AIC
  action: {
    tool: string;              // Tool identifier as declared in SDL
    params: Record<string, unknown>; // Tool parameters (sanitized of secrets)
    timestamp: string;         // ISO 8601 UTC
  };
  result: {
    success: boolean;
    summary: string;           // Human-readable summary (not the full output)
    timestamp: string;         // ISO 8601 UTC
  };
  agentSignature: string;      // Ed25519 signature over canonical JSON of above fields
  receiverSignature?: string;  // Optional counter-signature by tool/API
  receiverPublicKey?: string;  // Required if receiverSignature is present
}
```

**3.3.2 Signing** The agent signs the canonical JSON of the receipt (all fields except `agentSignature`, `receiverSignature`, and `receiverPublicKey`) with its own Ed25519 private key – the key whose public counterpart appears in the agent's AIC. The signature binds the action irrefutably to the agent instance.

The optional `receiverSignature` is a counter-signature by the tool or API that processed the action, using its own key pair. When present, it provides **mutual non-repudiation**: the agent cannot deny issuing the action, and the tool cannot deny processing it.

**3.3.3 Verification** `verifyReceipt(receipt: ActionReceipt, agentPublicKey: string): boolean`

A conforming verifier MUST: 1. Reconstruct the canonical JSON of the receipt (excluding signatures). 2. Verify `agentSignature` over those bytes using `agentPublicKey` (retrieved from the agent's AIC). 3. If `receiverSignature` is present, verify it over the same canonical bytes using `receiverPublicKey`. 4. Check that `receiptId` has not been seen before (replay resistance; SHOULD maintain a seen-ID set with TTL).

**3.3.4 Audit Trail** Receipts SHOULD be persisted to an append-only log. The combination of UUID receipt IDs, agent cert bindings, and Ed25519 signatures ensures that the log is both attributable (each entry is linked to a specific agent identity) and tamper-evident (any modification to a receipt will invalidate its signature).

---

### 3.4 Trust Chain

The Trust Chain is the mechanism that makes delegation verifiable. It is not a separate data structure but a property of a chain of AICs, each referencing its parent via `parentCertId`.

**3.4.1 Chain Construction** A trust chain from root operator to leaf sub-agent is a sequence of AICs `[AIC_0, AIC_1, ..., AIC_n]` where: - `AIC_0` has no `parentCertId` (root, issued directly by operator). - For each `i > 0`, `AIC_i.parentCertId == AIC_{i-1}.certId`. - Each `AIC_i.issuedAt` is within the validity window of `AIC_{i-1}`.

**3.4.2 Verification Invariants** `verifyTrustChain(chain: AgentIdentityCertificate[], operatorPublicKey: string): boolean`

A conforming verifier MUST enforce all of the following:

1. **Signature validity**: Every AIC in the chain has a valid Ed25519 signature (verified using `operatorPublicKey` for the root, and the parent's public key for each subsequent link).
2. **CertId binding**: `AIC_i.certId` matches the SHA-256 of the canonical signed bytes of `AIC_i`.
3. **ParentCertId linkage**: `AIC_i.parentCertId` matches `AIC_{i-1}.certId`.
4. **Temporal containment**: `AIC_i.issuedAt >= AIC_{i-1}.issuedAt` and `AIC_i.expiresAt <= AIC_{i-1}.expiresAt`.
5. **Scope monotonic narrowing**: For each adjacent pair, the child's SDL is a subset of the parent's SDL per the composition rules in Section3.2.2.
6. **Depth limit**: The chain length does not exceed the `maxSubAgentDepth` declared in the root AIC.

Violation of any invariant MUST cause the entire chain to be rejected. Partial trust ("trust the first three links but not the fourth") is not permitted.

**3.4.3 Attack Blocked** The invariants collectively prevent the "spawn a super-agent" attack: a parent agent that attempts to issue a child with broader permissions will produce a child AIC that fails the scope narrowing check at step 5. The parent agent cannot pass the scope narrowing check even if it generates a syntactically valid AIC, because it cannot forge the operator's signature on a fraudulently widened scope.

---

### 3.5 Breach Attestation

Breach Attestation addresses a class of attack that the preceding primitives do not cover: runtime state modification. An agent that has been hijacked via prompt injection or a compromised state backend may continue to present a valid AIC (its identity has not changed) while operating under adversarially modified instructions. Breach Attestation provides a mechanism to detect this.

#### 3.5.1 Structure

```typescript
interface BreachAttestation {
  attestationId: string;       // UUID v4
  agentCertId: string;         // certId of the attesting agent's AIC
  timestamp: string;           // ISO 8601 UTC
  stateHash: string;           // SHA-256 of canonical state representation (see Section3.5.
  previousHash: string;        // stateHash of the previous attestation (or "genesis" for fi
  agentSignature: string;      // Ed25519 signature over canonical JSON of above fields
  attestorId?: string;         // Optional: third-party attestor identifier
  attestorSignature?: string;  // Optional: counter-signature by third-party attestor
  attestorPublicKey?: string;  // Required if attestorSignature is present
}
```

**3.5.2 State Hash Computation** The `stateHash` is computed as:

```
stateHash = SHA-256(
  canonicalJSON({
    systemPromptHash: SHA-256(currentSystemPrompt),
    memoryHash: SHA-256(canonicalJSON(currentMemoryState)),
    toolCallHistoryHash: SHA-256(canonicalJSON(toolCallHistory))
  })
)
```

Fields are combined in canonical JSON order (lexicographic key sort). The exact definition of `currentMemoryState` and `toolCallHistory` is implementation-specific, but implementations MUST include all persistent agent state that could be adversarially modified.

**3.5.3 Hash Chain** Each attestation includes the `stateHash` of the immediately preceding attestation as `previousHash`. This forms a hash chain analogous to a blockchain: each link commits to all prior history. An attacker who modifies past state cannot recompute a valid chain without invalidating all subsequent signatures.

The first attestation in a session uses the string `"genesis"` as `previousHash`.

**3.5.4 Divergence Detection** A verifier that maintains attestation history can detect tampering by checking: 1. The current attestation's `previousHash` matches the previous attestation's `stateHash`. 2. The `systemPromptHash` within the state has not changed between attestations (unless an authorized update occurred). 3. The `agentSignature` is valid, confirming the attestation was produced by the same agent key.

A break in the hash chain, or an unexpected change to `systemPromptHash`, is a tamper signal that SHOULD trigger incident response procedures.

---

## 4. ATP v2 Extensions (May 2026)

The following extensions were added in ATP v2, based on production deployment experience and security review feedback.

### 4.1 Delegation Chains (Section6.2)

Full AIC issuance is a heavyweight operation requiring operator private key access. In many production deployments, agents need to delegate scoped authority to sub-agents without engaging the root operator's key management infrastructure for each delegation event. The Delegation Certificate addresses this.

#### 4.1.1 Structure

```
interface DelegationCertificate {
  id: string;                  // UUID v4
  parentAgentId: string;       // agentId of the delegating agent
  childAgentId: string;        // agentId of the delegate
  delegatedScope: ScopeDeclaration; // Must be subset of parent's scope
  maxDepth: number;            // Remaining delegation depth; MUST be < parent's maxSubAgent
  issuedAt: string;            // ISO 8601 UTC
  expiresAt: string;           // ISO 8601 UTC; MUST be within parent AIC validity window
  parentSignature: string;     // Ed25519 signature by parent agent's private key
}
```

**4.1.2 Chain Verification** `verifyDelegationChain(certs: DelegationCertificate[], rootAIC: AgentIdentityCertificate): boolean`

Verification traverses the delegation chain and enforces at each hop: - **Structural**: `cert_i.parentAgentId == cert_{i-1}.childAgentId`. - **Temporal**: `cert_i.issuedAt >= cert_{i-1}.issuedAt`, `cert_i.expiresAt <= cert_{i-1}.expiresAt`. - **Scope non-widening**: `cert_i.delegatedScope` is a subset of `cert_{i-1}.delegatedScope` per SDL composition rules. - **Depth decrement**: `cert_i.maxDepth < cert_{i-1}.maxDepth`. - **Signature**: `cert_i.parentSignature` is a valid Ed25519 signature over the canonical JSON of `cert_i` (excluding the signature field) using the parent agent's public key (retrieved from the AIC chain).

The `maxDepth` field prevents unbounded re-delegation: if `maxDepth` reaches 0, no further delegation certificates may be issued in that chain.

---

### 4.2 Certificate Revocation (Section7.1)

AICs and Delegation Certificates may need to be invalidated before their natural expiry – due to key compromise, detected scope violation, or operational decommissioning.

#### 4.2.1 RevocationList Structure

```
interface RevocationList {
  issuer: string;              // Operator identifier matching AICs issued by this operator
  issuedAt: string;            // ISO 8601 UTC; verifiers SHOULD check for freshness
  entries: Array<{
    certId: string;            // certId of the revoked certificate
    revokedAt: string;         // ISO 8601 UTC
    reason: RevocationReason;
    revokedBy: string;         // operatorId or agentId that issued the revocation
  }>;
  signature: string;           // Ed25519 signature over canonical JSON of above fields
}

type RevocationReason =
  | "key_compromise"
  | "agent_compromised"
  | "scope_violation"
  | "expired"
  | "superseded"
  | "unspecified";
```

**4.2.2 Verification** `verifyRevocationList(list: RevocationList, operatorPublicKey: string): boolean`

MUST be called before trusting any CRL entries. Verifiers MUST check the operator's signature on the list before accepting its contents. An attacker who can present an unsigned or incorrectly signed CRL cannot selectively un-revoke certificates.

**4.2.3 Distribution** This specification does not mandate a specific CRL distribution mechanism. Implementers MAY distribute CRLs via HTTPS well-known endpoints (`/.well-known/atp-crl`), incorporate them into attestation checkpoints, or use push-based notification. CRL freshness SHOULD be validated; verifiers SHOULD reject lists older than a configurable staleness threshold (default: 1 hour for high-security deployments).

---

### 4.3 Multi-Party Authorization (Section8)

Some agent actions carry enough risk that a single agent's signature is insufficient authorization. The Multi-Party Authorization mechanism implements M-of-N threshold approval.

#### 4.3.1 Structure

```
interface MultiSigRequest {
  id: string;                    // UUID v4
  payload: Record<string, unknown>; // The action or decision requiring authorization
  requiredSigners: number;       // M: minimum valid signatures required
  signers: string[];             // N: list of authorized signer agentIds
  signatures: Array<{
    agentId: string;
    signature: string;           // Ed25519 signature over canonical JSON of {id, payload}
    publicKey: string;           // Signer's public key
    timestamp: string;           // ISO 8601 UTC
  }>;
}
```

#### 4.3.2 Authorization Check `isAuthorized(request: MultiSigRequest): boolean`

Counts valid signatures from distinct authorized signers. For each entry in `signatures`: 1. Verify `agentId` is in `signers`. 2. Verify the Ed25519 `signature` over `canonicalJSON({id: request.id, payload: request.payload})` using `publicKey`. 3. Deduplicate by `agentId` (multiple signatures from the same agent count as one).

Return `true` if the count of valid, distinct, authorized signatures >= `requiredSigners`.

This mechanism is appropriate for actions such as: deploying infrastructure changes, executing financial transfers above a threshold, deleting production data, or granting new operator permissions.

---

## 5. Wire Format and Cryptography

### 5.1 Serialization

All ATP data types are JSON-serializable. The following constraints apply:

- **No `Date` objects**: All timestamps are ISO 8601 strings with UTC timezone (`Z` suffix). This ensures consistent, cross-language serialization.
- **No `Buffer` or binary fields**: All binary data (keys, signatures, hashes) is encoded as base64 or hex strings.
- **No `undefined` values**: Any field that is not present is omitted entirely rather than set to `null` or `undefined`. This prevents JSON serializer divergence where some implementations serialize `undefined` as `null`.

### 5.2 Canonical JSON

ATP requires canonical JSON for all signature operations, based on the RFC 8785 JCS (JSON Canonicalization Scheme) subset. The canonicalization algorithm is:

1. **Key ordering**: All object keys are sorted lexicographically by Unicode code point value, recursively.
2. **No whitespace**: No insignificant whitespace (spaces, newlines, tabs) is emitted.
3. **No trailing commas**: Standard JSON – no trailing commas after the last array or object element.
4. **Unicode normalization**: Strings are left as-is; no additional Unicode normalization is applied.
5. **Number representation**: Numbers are serialized using their shortest decimal representation (per ECMAScript JSON.stringify semantics).

Canonical JSON prevents field-reordering attacks: an attacker who reorders JSON fields in a signed document and presents it to a verifier that does not canonicalize before verification would receive a valid verification. Canonical JSON ensures all verifiers compute signatures over the same byte sequence regardless of field ordering in the received document.

A canonical JSON implementation MUST produce byte-for-byte identical output for semantically equivalent inputs across all compliant implementations.

### 5.3 Cryptography

ATP uses **Ed25519** exclusively for all digital signatures (RFC 8032). This choice is deliberate:

- **No parameter choices**: Unlike ECDSA (which requires a curve parameter) or RSA (which requires a key size), Ed25519 has no user-configurable parameters. This eliminates a class of implementation errors where weak parameters are chosen.
- **Small keys**: Ed25519 public keys are 32 bytes (44 base64 characters); signatures are 64 bytes (88 base64 characters). This minimizes serialized certificate size.
- **Performance**: Ed25519 sign and verify operations are fast on modern hardware, enabling signature verification in tight loops (e.g., verifying the full trust chain on every tool invocation).
- **Immunity to DH small-subgroup attacks**: Ed25519 uses Curve25519, which has cofactor 8. The standard library implementations (including Node.js built-ins) handle cofactor clearing correctly. Implementations MUST NOT use raw low-level curve operations that bypass cofactor handling.
- **Security level**: Ed25519 provides approximately 128-bit security, matching AES-128 and SHA-256. There are no known practical attacks.

**Key encoding**: Ed25519 public keys are 32-byte raw payloads, base64-standard encoded (RFC 4648 Section4, using `+` and `/`, with `=` padding). URL-safe base64 is NOT used. PEM encoding is NOT used. This choice maximizes cross-language interoperability; every major language's standard library can decode base64-standard.

**Signature encoding**: Ed25519 signatures are 64-byte outputs, base64-standard encoded (88 characters including padding).

**Hash encoding**: SHA-256 hashes (used for `certId`, `systemPromptHash`, `stateHash`) are hex-encoded lowercase strings (64 characters). Hex is chosen for hashes because it is human-readable in logs and unambiguous (no padding edge cases).

**No external cryptographic dependencies**: The reference implementation uses only Node.js built-in `crypto` module (available since Node 15+). Implementers in other languages SHOULD use their platform's standard cryptographic library rather than third-party dependencies.

---

## 6. Compliance Levels

ATP defines three compliance levels to enable gradual adoption. The goal is that deploying even the minimum level provides meaningful security improvement over the current state (which is no cryptographic agent identity at all).

### ATP-Basic

**Primitives required**: Agent Identity Certificate (AIC) + Action Receipts.

**What this provides**: Every agent instance has a cryptographically verifiable identity. Every tool invocation is attributable to a specific agent. An audit log built on Action Receipts is tamper-evident.

**What this does not provide**: No formal scope enforcement, no delegation chain verification, no runtime tamper detection.

**Recommended for**: Initial adoption, low-risk read-only agents, environments where scope is enforced by other means (e.g., existing RBAC systems).

### ATP-Standard

**Primitives required**: ATP-Basic + SDL enforcement + Trust Chain validation.

**What this provides**: Agents cannot invoke out-of-scope tools (verifiers enforce SDL). Delegation chains are verifiable; sub-agents cannot escalate beyond parent scope. The full delegation lineage from operator to leaf agent is auditable.

**What this does not provide**: No runtime tamper detection between attestation checkpoints.

**Recommended for**: Production agents with meaningful tool access; multi-agent systems; any agent that can spawn sub-agents.

### ATP-Full

**Primitives required**: ATP-Standard + periodic Breach Attestation.

**What this provides**: Runtime state tampering is detectable. The complete chain of attestations forms a forensic record of agent state over time.

**What this does not provide**: Real-time tamper response (attestation is periodic, not continuous; a tamper event between checkpoints may not be immediately detected).

**Recommended for**: High-privilege agents (code execution, financial operations, infrastructure management); agents in adversarial environments (processing untrusted user inputs, browsing arbitrary websites); regulated environments requiring forensic audit capability.

## 7. Security Analysis

### 7.1 Signature Scheme Security

Ed25519 provides approximately 128-bit security against all known classical attacks. The Bernstein et al. (2011) design addresses the primary implementation pitfalls of prior ECDSA deployments: nonce reuse (Ed25519 uses deterministic nonce derivation, making nonce-reuse attacks impossible), side-channel vulnerabilities (constant-time implementations are standard), and weak curve parameters (Curve25519 was chosen for rigid security criteria, not for NIST compliance).

There are no known practical attacks against correctly implemented Ed25519. Quantum adversaries with a large-scale fault-tolerant quantum computer could run Shor's algorithm against Ed25519; ATP does not address post-quantum threats in this version. A future ATP revision may add a hybrid signature scheme (e.g., Ed25519 + ML-DSA per NIST FIPS 204) for post-quantum resilience.

### 7.2 Canonical JSON Resistance

Field-reordering attacks against JSON-signed systems are well-documented. An attacker who can present the same JSON payload with reordered keys to a non-canonicalizing verifier may cause the verifier to verify a signature over different bytes than were originally signed. ATP's mandatory canonical JSON eliminates this surface.

The use of lexicographic key ordering (per RFC 8785) ensures that any two semantically equivalent JSON objects produce byte-for-byte identical canonical representations across all compliant implementations.

### 7.3 CertId Binding

`certId` is computed as SHA-256 of the full canonical signed bytes (including the `signature` field). This binding serves a critical function: a verifier can retrieve an AIC by `certId` from any source, recompute the hash, and confirm it matches – without needing to trust the transport or storage layer. It also prevents certificate substitution: an attacker who presents a different AIC with the same `agentId` but different fields cannot claim the same `certId`.

### 7.4 Scope Non-Widening

The SDL composition invariant (Section3.2.2) is enforced at verification time, not only at issuance time. This means that even if an operator issues a malformed child AIC (e.g., due to a bug in the issuance code), a conforming verifier will catch the scope widening when it processes the Trust Chain. Defense-in-depth: the scope constraint is checked by the issuer (policy) AND by the verifier (enforcement).

### 7.5 Replay Resistance

Action Receipts include a UUID v4 `receiptId`. UUID v4 provides 122 bits of randomness; collision probability across 10^12 receipts is approximately 10^{-13} – negligible for practical deployments. Verifiers SHOULD maintain a set of seen `receiptId` values with a TTL matching the receipt validity window. The `timestamp` field in the action further constrains replay: a verifier MAY reject receipts with timestamps outside a configurable freshness window.

### 7.6 Key Compromise

Key compromise is handled via the RevocationList mechanism (Section4.2). An operator who detects private key compromise issues a new RevocationList entry with reason `key_compromise` and provisions new key material for affected agents. The time between key compromise and revocation list distribution is the primary residual risk; implementers SHOULD minimize this window through automated key rotation policies and real-time CRL distribution.

### 7.7 Known Limitations

**Attestation of execution, not of compliance**: ATP's `systemPromptHash` binding establishes that a specific system prompt was declared at AIC issuance time. ATP cannot verify that the underlying model actually executed that prompt faithfully – i.e., that the model's outputs were determined by the specified system prompt rather than by adversarial inputs overriding it. This is a fundamental limitation of the attestation model: ATP attests to declared configuration, not to runtime model behavior. Behavioral monitoring layers (such as ATR-based detection) are complementary and address the behavioral compliance gap that ATP does not.

**PKI for operator public keys**: ATP specifies cryptographic operations over operator public keys but defers the design of the PKI that distributes and authenticates those public keys to implementers. Without a trusted mechanism for distributing operator public keys, an attacker who can substitute a fraudulent operator public key can forge any ATP artifact. Implementers MUST treat operator public key distribution as a first-class security concern; recommended approaches include HTTPS well-known endpoints with certificate pinning, integration with existing PKI, or out-of-band key registration.

**Periodic, not continuous, attestation**: Breach Attestation produces snapshots at configurable intervals. A state tampering event that occurs and is remediated between two attestation checkpoints may not be detected. The attestation interval SHOULD be calibrated to the risk profile of the agent; high-risk agents in adversarial environments SHOULD attest on every tool call.

---

## 8. Reference Implementation

The ATP reference implementation is available as `@lyrie/atp` on npm.

**Package**: `@lyrie/atp`
**License**: MIT

**Runtime**: Node.js 20+
**Dependencies**: None (Ed25519 via Node.js built-in `crypto` module)
**Tests**: 138 tests, 0 failures
**Repository**: https://github.com/OTT-Cybersecurity-LLC/lyrie-ai/tree/main/packages/atp

### 8.1 Installation

```
npm install @lyrie/atp
```

### 8.2 Full Lifecycle Example

The following example demonstrates the complete ATP lifecycle: key pair generation, AIC issuance, action signing, and verification.

```
import {
  generateKeyPair,
  issueAIC,
  verifyAIC,
  signReceipt,
  verifyReceipt,
  type AgentIdentityCertificate,
  type ActionReceipt,
  type ScopeDeclaration,
} from '@lyrie/atp';
import { createHash } from 'crypto';

//  Step 1: Generate key pairs

// Operator key pair (held securely by the deploying organization)
const operatorKeys = generateKeyPair();

// Agent instance key pair (generated fresh per agent instance)
const agentKeys = generateKeyPair();

//  Step 2: Define the agent's authorized scope

const scope: ScopeDeclaration = {
  allowedTools: ['web_search', 'web_fetch', 'memory_store'],
  deniedTools: ['exec', 'file_write', 'http_request'],
  allowedDomains: ['*.wikipedia.org', '*.arxiv.org'],
  maxSubAgentDepth: 0,
  requireApprovalFor: [],
  temporalScope: {
    notBefore: new Date().toISOString(),
    notAfter: new Date(Date.now() + 3600 * 1000).toISOString(), // 1 hour
  },
  dataScope: {
```

```
    readPaths: [],
    writePaths: [],
    maxPayloadBytes: 65536,
  },
};

// Step 3: Issue the Agent Identity Certificate

const systemPrompt = 'You are a research assistant. You may search the web and store memorie
const systemPromptHash = createHash('sha256').update(systemPrompt).digest('hex');

const aic: AgentIdentityCertificate = issueAIC({
  modelId: 'anthropic/claude-sonnet-4-6',
  systemPromptHash,
  scope,
  operatorId: 'urn:operator:ott-cybersecurity:prod',
  validitySeconds: 3600,
  agentPublicKey: agentKeys.publicKey,
  operatorPrivateKey: operatorKeys.privateKey,
});

console.log('Issued AIC:', aic.certId);

// Step 4: Verify the AIC (as any verifier would)

const aicValid = verifyAIC(aic, operatorKeys.publicKey);
console.log('AIC valid:', aicValid); // true

// Step 5: Agent takes an action and signs a receipt

const receipt: ActionReceipt = signReceipt({
  agentCertId: aic.certId,
  action: {
    tool: 'web_search',
    params: { query: 'Ed25519 cryptography overview' },
    timestamp: new Date().toISOString(),
  },
  result: {
    success: true,
    summary: 'Retrieved 10 search results for Ed25519 cryptography.',
    timestamp: new Date().toISOString(),
  },
  agentPrivateKey: agentKeys.privateKey,
});
```

```
console.log('Receipt ID:', receipt.receiptId);

// Step 6: Verify the receipt

const receiptValid = verifyReceipt(receipt, agentKeys.publicKey);
console.log('Receipt valid:', receiptValid); // true

// Step 7: Verify scope compliance before executing the tool

import { checkScope } from '@lyrie/atp';

const scopeCheck = checkScope(aic.scope, 'web_search', { query: 'test' });
if (!scopeCheck.allowed) {
  throw new Error(`Scope violation: ${scopeCheck.reason}`);
}
// Only reaches here if tool is in allowedTools and not in deniedTools

// Step 8: Produce a Breach Attestation snapshot

import { issueAttestation, verifyAttestation } from '@lyrie/atp';

const memoryState = { key: 'session_topic', value: 'Ed25519 research' };
const toolCallHistory = [{ tool: 'web_search', timestamp: receipt.action.timestamp }];

const attestation = issueAttestation({
  agentCertId: aic.certId,
  systemPromptHash,
  memoryState,
  toolCallHistory,
  previousHash: 'genesis',
  agentPrivateKey: agentKeys.privateKey,
});

const attestationValid = verifyAttestation(attestation, agentKeys.publicKey);
console.log('Attestation valid:', attestationValid); // true
```

This example runs without network access, without mocking, and without external dependencies on any system with Node.js 20+.

---

## 9. Related Work

### SPIFFE/SPIRE

SPIFFE (Secure Production Identity Framework for Everyone) and its reference implementation SPIRE provide workload identity for services in containerized infrastructure. SPIFFE

solves the problem of "which service is this?" by issuing X.509 SVIDs (SPIFFE Verifiable Identity Documents) bound to workload processes. SPIFFE does not address AI agent-specific concerns: it has no notion of a system prompt, an authorized tool list, or a delegation chain between computational agents. It also does not produce per-action attributable receipts. ATP is complementary to SPIFFE: an agent runtime could use SPIFFE for service-layer identity while using ATP for agent-layer identity.

### W3C Decentralized Identifiers (DIDs)

W3C DIDs provide a framework for decentralized, self-sovereign identity. DID Documents describe verification methods and service endpoints for a DID subject. DIDs are designed for broad identity use cases and are intentionally general-purpose. They provide no agent-specific primitives: no scope declaration, no action receipts, no state attestation, no delegation chain with monotonic scope narrowing. ATP could be expressed as a DID method, but the generality of DIDs would add significant complexity without adding security value for the agent-specific use case ATP addresses.

### OpenID Connect

OpenID Connect (OIDC) builds an identity layer on top of OAuth 2.0, enabling clients to verify end-user identity and obtain basic profile claims. OIDC is designed for human-in-the-loop authentication flows. It has no model for fully autonomous agents issuing their own sub-agents, no action receipts, no state attestation, and no scope composition rules that prevent delegation escalation. OIDC tokens (JWTs) can express custom claims, but OIDC defines no standard claim vocabulary for agent identity. ATP borrows the signed token concept from JWT but adds agent-specific semantics throughout.

### JWT / JWS (RFC 7519, RFC 7515)

JSON Web Tokens and JSON Web Signatures provide a general-purpose signed token format widely used for API authorization. ATP's AIC and Action Receipt formats are conceptually similar to JWTs: signed JSON payloads with standard fields. The critical difference is semantics: ATP defines a specific, non-extensible set of fields with precise security meaning for agent identity, whereas JWT is a generic container whose security properties depend entirely on how it is used. ATP implementations MAY serialize AICs and receipts in JWT-compatible formats for interoperability with existing JWT infrastructure, provided they maintain all ATP-required fields and canonical JSON serialization for signature verification.

### Microsoft Agent Governance Toolkit (AGT)

Microsoft's Agent Governance Toolkit provides a behavioral governance layer for AI agents, focusing on policy enforcement, content filtering, and behavioral monitoring. AGT operates at the behavioral layer – what the agent does and whether its outputs conform to policy. ATP operates at the identity and authorization layer – who the agent is, what it is authorized to do, and whether its actions are attributable. These are complementary layers: AGT can consume ATP attestation artifacts as inputs to its policy engine, and ATP can use AGT-reported violations as triggers for certificate revocation.

### Agent Threat Rules (ATR)

The Agent Threat Rules corpus, developed by Panguard AI, is a behavioral detection framework defining rule conditions for detecting adversarial agent behavior. ATR rules can reference ATP attestation artifacts as conditions: for example, a rule that fires when an agent takes an action outside its declared SDL scope, or when Breach Attestation reveals a system prompt hash change. ATP and ATR are complementary layers – ATP provides the attestation infrastructure that ATR rules can consume. They are not competing approaches; a complete agent security stack benefits from both.

---

## 10. IETF Submission

The authors intend to submit ATP to the IETF for Standards Track RFC consideration.

**Target working group**: IETF SEC Area. The most natural fit is a new BoF (Birds of a Feather) session to assess community interest in an AI Agent Security working group. Depending on WG charter alignment, dispatch to an existing group (e.g., OAUTH, JOSE, or a newly chartered agent security WG) is also possible.

**Current status**: Reference implementation (`@lyrie/atp`) is complete and publicly available. This public draft is circulating for community review.

**Draft timeline**: The authors plan to submit an Internet-Draft to the IETF datatracker within 90 days of this public draft's release, targeting the IETF 121 or 122 meeting cycle.

**Co-authorship**: The authors welcome co-authors from the implementer community, particularly from teams with production ATP deployments who can contribute empirical deployment data and implementation notes. Contact: `atp-draft@lyrie.ai`.

**What we need from the community**: - Independent implementations in languages other than TypeScript (Python, Go, Rust, Java) to validate the specification's language-agnostic claims. - Deployment case studies documenting where ATP primitives were insufficient and where they provided clear value. - Formal security analysis (ProVerif, Tamarin, or equivalent) of the Trust Chain and Breach Attestation protocols. - Review of the canonical JSON subset definition for edge cases.

---

## 11. Acknowledgments

The threat model in Section2 was refined through discussions with the security research community following the MCP RCE incidents of early 2026. The authors are grateful to the reviewers who pointed out the attestation-of-execution limitation described in Section7.7 – honest acknowledgment of what a protocol does not do is as important as specifying what it does.

The `@lyrie/atp` implementation benefited from the Node.js cryptography team's work on making Ed25519 a first-class built-in, eliminating the external dependency problem that plagued early JWT implementations.

## References


[**RFC4122**] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, https://www.rfc-editor.org/info/rfc4122.

[**RFC4648**] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, https://www.rfc-editor.org/info/rfc4648.

[**RFC5280**] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, https://www.rfc-editor.org/info/rfc5280.

[**RFC6749**] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, https://www.rfc-editor.org/info/rfc6749.

[**RFC7515**] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, https://www.rfc-editor.org/info/rfc7515.

[**RFC7517**] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, https://www.rfc-editor.org/info/rfc7517.

[**RFC7519**] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, https://www.rfc-editor.org/info/rfc7519.

[**RFC8032**] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, https://www.rfc-editor.org/info/rfc8032.

[**RFC8446**] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, https://www.rfc-editor.org/info/rfc8446.

[**RFC8785**] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, https://www.rfc-editor.org/info/rfc8785.

[**ED25519**] Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., and B.-Y. Yang, "High-speed high-security signatures", Journal of Cryptographic Engineering, 2(2):77-89, 2012. DOI: 10.1007/s13389-012-0027-1.

[**SPIFFE**] Secure Production Identity Framework for Everyone (SPIFFE), CNCF Project, https://spiffe.io/, 2024.

[**DID**] Sporny, M., Longley, D., Sabadello, M., Reed, D., Steele, O., and C. Allen, "Decentralized Identifiers (DIDs) v1.0", W3C Recommendation, July 2022, https://www.w3.org/TR/did-core/.

[**OIDC**] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", OpenID Foundation, November 2014, https://openid.net/specs/openid-

connect-core-1_0.html.

[**JCS**] Rundgren, A., "Explaining JCS – The JSON Canonicalization Scheme", 2020, https://cyberphone.github.io/doc/security/browser-json-canonicalization.pdf.

[**NISTFIPS204**] National Institute of Standards and Technology, "Module-Lattice-Based Digital Signature Standard (ML-DSA)", FIPS 204, August 2024, https://doi.org/10.6028/NIST.FIPS.204.

---

*Agent Trust Protocol (ATP) – Public Draft v1.0*

*Feedback: atp-draft@lyrie.ai | Issues: https://github.com/OTT-Cybersecurity-LLC/lyrie-ai/issues*