Independent Submission
Internet-Draft
Intended status: Informational
Expires: 11 November 2026

D. Bates
SVT Robotics

10 May 2026

# ATP Core Test Vectors
## draft-bates-atp-test-vectors-00

Abstract

This document publishes golden test vectors for the Agent Transaction Protocol (ATP) [ATP-CORE]. The vectors cover canonicalization (RFC 8785 JCS), nodeId computation, and Ed25519 signature generation and verification. ATP Core conformance per [ATP-CORE] Section 13.7 requires that conformant implementations pass all vectors in this document for canonicalization, nodeId computation, and Ed25519 signature verification.

Table of Contents



## 1. Introduction

[ATP-CORE] specifies a deterministic identity and signature model for agent-action nodes. Two independent implementations presented with the same node content are required to compute byte-identical canonical forms, byte-identical nodeId values, and to verify each other's Ed25519 signatures.

This document publishes a starter set of test vectors so that interoperability claims can be substantiated mechanically rather than by inspection. Each vector is computed and recorded in this document; an implementation passes a vector if and only if it produces the exact byte sequence and hex value recorded for that vector.

The vectors in this -00 release cover the minimum required for ATP Core conformance: canonicalization fixtures, nodeId computation across representative node shapes, and one Ed25519 sign/verify vector with deterministic key material. Future revisions will add fixtures for validation modes, the Section 13.6 validation result format, profile-field canonicalization (strict and permissive), parent-state handling, and adversarial cases (large numbers, surrogate pairs, deeply nested structures).

## 2. Conventions

The key words MUST, MUST NOT, REQUIRED, SHOULD, SHOULD NOT, RECOMMENDED, NOT RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals.

All canonical byte sequences in this document are presented as UTF-8 text. All hash and signature outputs are presented in lowercase hexadecimal unless explicitly stated otherwise.

## 3. Canonicalization Vectors (RFC 8785 JCS)

These vectors verify that an implementation’s canonicalizer matches RFC 8785 JSON Canonicalization Scheme [RFC8785]. They are intentionally minimal so that any conformant JCS implementation produces identical output.

### 3.1. Vector C1 -- empty object

Input:

```
{}
```

Canonical bytes (UTF-8):

```
{}
```

Length: 2 bytes.

### 3.2. Vector C2 -- object with key reordering

Input:

```
{"b": 1, "a": 2}
```

Canonical bytes (UTF-8):

```
{"a":2,"b":1}
```

Length: 13 bytes.

### 3.3. Vector C3 -- null-valued field omission per ATP Section 9

Input:

```
{"a": 1, "b": null}
```

Canonical bytes (UTF-8) per ATP Section 9 (null fields omitted):

```
{"a":1}
```

Length: 7 bytes.

*Note:* Strict RFC 8785 JCS would preserve "b":null. ATP Section 9 modifies this rule and requires null-valued fields to be omitted. Profiles MAY override the omission rule per [ATP-CORE] Section 9.

### 3.4. Vector C4 -- array order preservation

Input:

```
{"items": [3, 1, 2]}
```

Canonical bytes (UTF-8):

```
{"items":[3,1,2]}
```

Array order MUST be preserved as declared.

### 3.5. Vector C5 -- nested object key sorting

Input:

```
{"outer": {"z": 1, "a": 2}, "alpha": 3}
```

Canonical bytes (UTF-8):

```
{"alpha":3,"outer":{"a":2,"z":1}}
```

Both top-level and nested keys MUST be sorted.

## 4. nodeId Computation Vectors

These vectors verify the full nodeId formula:

nodeId = SHA256(canonical(node_without_signature))

Each vector presents a node, the exact canonical bytes produced by ATP Section 9 + RFC 8785 JCS, and the resulting SHA-256 hash in lowercase hex. The signature field is intentionally absent from the input -- implementations MUST exclude it from canonicalization per [ATP-CORE] Section 10.1.

### 4.1. Vector V1 -- minimum-viable request node

Input node (no actor, no profile, no outputHash, empty parents -- represents an autonomous root request):

```
{
  "timestamp": "2026-01-01T00:00:00Z",
  "scope": "test-scope-001",
  "issuer":  {"issuerId": "test-issuer", "keyId": "test-key-1"},
  "agent":   {"agentId": "test-agent",   "version": "1.0.0"},
  "action":  {
    "type": "atp:request",
    "inputHash": "sha256:0000000000000000000000000000000000000000000000000000000000000000
000"
  },
  "parents": []
}
```

Canonical bytes (UTF-8):

```
{"action":{"inputHash":"sha256:0000000000000000000000000000000000000000000000000000000000000000
000000","type":"atp:request"},"agent":{"agentId":"test-agent","version":"1.0.0"},"issuer"
:{"issuerId":"test-issuer","keyId":"test-key-1"},"parents":[],"scope":"test-scope-001","t
imestamp":"2026-01-01T00:00:00Z"}
```

Expected nodeId:

```
77d803c2d67e6cbe893172e5676e52b8f1bb80910bcbe1ca4c9aa5273f46ce70
```

### 4.2. Vector V2 -- completion node with actor and parent reference

Input node (completion of V1, actor present, references V1 as parent):

```
   {
     "timestamp": "2026-01-01T00:00:01Z",
     "scope": "test-scope-001",
     "issuer":  {"issuerId": "test-issuer", "keyId": "test-key-1"},
     "agent":   {"agentId": "test-agent",   "version": "1.0.0"},
     "actor":   {"actorId": "psn:0000-test-actor", "authContext": "test:none"},
     "action":  {
       "type": "atp:completion",
       "inputHash":  "sha256:0000000000000000000000000000000000000000000000000000000000000000
0000",
       "outputHash": "sha256:1111111111111111111111111111111111111111111111111111111111111111
1111"
     },
     "parents": ["77d803c2d67e6cbe893172e5676e52b8f1bb80910bcbe1ca4c9aa5273f46ce70"]
   }
```

Canonical bytes (UTF-8):

```
{"action":{"inputHash":"sha256:0000000000000000000000000000000000000000000000000000000000000000
000000","outputHash":"sha256:1111111111111111111111111111111111111111111111111111111111111111
1111","type":"atp:completion"},"actor":{"actorId":"psn:0000-test-actor","authContext":"te
st:none"},"agent":{"agentId":"test-agent","version":"1.0.0"},"issuer":{"issuerId":"test-i
ssuer","keyId":"test-key-1"},"parents":["77d803c2d67e6cbe893172e5676e52b8f1bb80910bcbe1ca
4c9aa5273f46ce70"],"scope":"test-scope-001","timestamp":"2026-01-01T00:00:01Z"}
```

Expected nodeId:

```
881b552dd7d4a8598abe44ceab49257bb63b5e6420eeaf949ac2657b5495ae5e
```

### 4.3. Vector V3 -- same content as V1 with mis-ordered keys

This vector verifies that canonicalization produces identical output regardless of input key order. Implementations MUST produce the same canonical bytes and the same nodeId for V3 as for V1.

Input node (logically equivalent to V1, all keys in reverse-ish order):

```
   {
     "scope": "test-scope-001",
     "agent":   {"version": "1.0.0", "agentId": "test-agent"},
     "action":  {
       "inputHash": "sha256:0000000000000000000000000000000000000000000000000000000000000000
000",
       "type": "atp:request"
     },
     "issuer":  {"keyId": "test-key-1", "issuerId": "test-issuer"},
     "timestamp": "2026-01-01T00:00:00Z",
     "parents": []
   }
```

Canonical bytes (UTF-8):

```
{"action":{"inputHash":"sha256:0000000000000000000000000000000000000000000000000000000000000000
000000","type":"atp:request"},"agent":{"agentId":"test-agent","version":"1.0.0"},"issuer"
:{"issuerId":"test-issuer","keyId":"test-key-1"},"parents":[],"scope":"test-scope-001","t
imestamp":"2026-01-01T00:00:00Z"}
```

Expected nodeId:

77d803c2d67e6cbe893172e5676e52b8f1bb80910bcbe1ca4c9aa5273f46ce70

(Identical to V1; this is the conformance signal.)

### 4.4. Vector V4 -- fan-in decision with two parents

Input node (decision synthesis with parents=[V1, V2], signed by a different issuer):


```
{
  "timestamp": "2026-01-01T00:00:02Z",
  "scope": "test-scope-001",
  "issuer":  {"issuerId": "test-issuer-2", "keyId": "test-key-2"},
  "agent":   {"agentId": "synth-agent", "version": "1.0.0"},
  "action":  {
    "type": "atp:decision",
    "inputHash":  "sha256:2222222222222222222222222222222222222222222222222222222222222222",
    "outputHash": "sha256:3333333333333333333333333333333333333333333333333333333333333333"
  },
  "parents": [
    "77d803c2d67e6cbe893172e5676e52b8f1bb80910bcbe1ca4c9aa5273f46ce70",
    "881b552dd7d4a8598abe44ceab49257bb63b5e6420eeaf949ac2657b5495ae5e"
  ]
}
```


Canonical bytes (UTF-8):


```
{"action":{"inputHash":"sha256:2222222222222222222222222222222222222222222222222222222222222222","outputHash":"sha256:3333333333333333333333333333333333333333333333333333333333333333","type":"atp:decision"},"agent":{"agentId":"synth-agent","version":"1.0.0"},"issuer":{"issuerId":"test-issuer-2","keyId":"test-key-2"},"parents":["77d803c2d67e6cbe893172e5676e52b8f1bb80910bcbe1ca4c9aa5273f46ce70","881b552dd7d4a8598abe44ceab49257bb63b5e6420eeaf949ac2657b5495ae5e"],"scope":"test-scope-001","timestamp":"2026-01-01T00:00:02Z"}
```


Expected nodeId:

25abc84ddbd4ca932502e83e92050f00b1ecb70b4e3cf071d5823b3d3d23de4c

Parent array order is canonicalization-significant per [ATP-CORE] Section 8: V1 before V2 produces a different nodeId than V2 before V1 would. Implementations MUST preserve declared parent order.

### 4.5. Vector V5 -- node with profile field

This vector verifies that the profile field participates in canonicalization and nodeId computation per [ATP-CORE] Section 10.1 and Section 20.4.

Input node:

```
   {
     "timestamp": "2026-01-01T00:00:03Z",
     "scope": "test-scope-001",
     "issuer":  {"issuerId": "test-issuer", "keyId": "test-key-1"},
     "agent":   {"agentId": "test-agent",   "version": "1.0.0"},
     "action":  {
       "type": "atp:request",
       "inputHash": "sha256:4444444444444444444444444444444444444444444444444444444444444444
444"
     },
     "parents": [],
     "profile": "urn:ietf:params:atp:profile:test:1.0"
   }
```

Canonical bytes (UTF-8):

```
{"action":{"inputHash":"sha256:4444444444444444444444444444444444444444444444444444444444444444
444444","type":"atp:request"},"agent":{"agentId":"test-agent","version":"1.0.0"},"issuer"
:{"issuerId":"test-issuer","keyId":"test-key-1"},"parents":[],"profile":"urn:ietf:params:
atp:profile:test:1.0","scope":"test-scope-001","timestamp":"2026-01-01T00:00:03Z"}
```

Expected nodeId:

2356e89a5e787e9312287dfa4b3440d823b7fac59e401f060d42757e8f452803

If an implementation produces a different nodeId for V5, the most likely cause is that the profile field has been excluded from canonicalization. ATP Section 10.1 requires it to be included whenever present.

## 5. Ed25519 Signature Vector

This section publishes one Ed25519 sign/verify vector using deterministic key material. ATP Section 10.3 requires that the signature be computed over the nodeId, which is a 32-byte SHA-256 output.

### 5.1. Test Key Material

The vectors in this document use a fixed Ed25519 keypair derived from a deterministic 32-byte seed. Implementations MUST use the same seed to reproduce these vectors. *This key material is for test use only and MUST NOT be used in production deployments.*

| Field | Value |
|---|---|
| Seed (32 bytes, hex) | aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa |
| Public key (raw, 32 bytes, hex) | e734ea6c2b6257de72355e472aa05a4c487e6b463c029ed306df2f01b5636b58 |
| Public key (raw, 32 bytes, base64) | 5zTqbCtiV95yNV5HKqBaTEh+a0Y8Ap7TBt8vAbVja1g= |

Table 1

The seed is the Ed25519 private-key seed per [RFC8032]. Implementations using PKCS#8 or other encodings MUST derive the equivalent raw seed before applying the test.

## 5.2. Vector S1 -- sign V1’s nodeId

Sign the V1 nodeId (Section 4.1) as the message. ATP Section 10.3 specifies that the signature input is the nodeId value treated as 32 bytes -- that is, bytes.fromhex(nodeId), not the hex string itself.

| Field | Value |
|---|---|
| Message (hex, 32 bytes) | 77d803c2d67e6cbe893172e5676e52b8f1bb80910bcbe1ca4c9aa5273f46ce70 |
| Signature (hex, 64 bytes) | 3f4d9fb756aba9bca11cfac15d65d82441dbf6f69adc9ba527b506c3379855500a2ef1a4e471323f2e8c8d190868e4f5ef303bef1e3e57e1988b1b46d83d5509 |
| Signature (base64) | P02ft1arqbyhHPrBXWXYJEHb9vaa3JulJ7UGwzeYVVAKLvGk5HEyPy6MjRkIaOT17zA77x4+V+GYixtG2D1VCQ== |

Table 2

A conformant implementation MUST:

1. Reproduce the public key bytes from the seed.

2. Reproduce the signature bytes from the seed and message.

3. Verify the signature against the public key and message and return success.

4. Reject the signature if any byte of the message, signature, or public key is altered.

Ed25519 is deterministic per [RFC8032] Section 5.1.6: signing the same message under the same key MUST produce the same signature bytes. Implementations that produce different signature bytes for this vector are not interoperable with [ATP-CORE].

## 6. Implementation Notes

### 6.1. Canonicalization Edge Cases Not Covered in -00

This release deliberately uses simple JSON values (strings, integers, arrays of strings, plain objects) that all conformant RFC 8785 implementations handle identically. The following edge cases are not covered in -00 and will be added in subsequent revisions:

* Floating-point and exponent-form numbers (RFC 8785's ECMAScript number serialization rules).

* Unicode surrogate-pair handling and codepoints in the supplementary planes.

* Strings containing escapes for control characters not in the basic ASCII range.

* Very large nested structures (depth > 16, breadth > 256 keys).

* Empty arrays and empty strings as values.

Implementations passing the -00 vectors satisfy the floor for ATP Core conformance per [ATP-CORE] Section 13.7. Implementations claiming broader RFC 8785 conformance SHOULD additionally pass the test data published by [JCS-TESTDATA].

### 6.2. Canonicalization Library Choices

Implementations MAY use any RFC 8785 conformant canonicalization library. The reference implementation by the RFC 8785 author is available in multiple languages at [JCS-TESTDATA] and provides additional test data covering the edge cases listed above.

ATP Core does not endorse a specific implementation. The deterministic-output guarantee of [ATP-CORE] Section 9 makes the choice immaterial provided the chosen library is RFC 8785 conformant.

### 6.3. Reproducing the Vectors

The vectors in this document are reproducible from the inputs shown. A conformant implementation given the same input MUST produce the canonical bytes and nodeId exactly as recorded. The Python script that generated these vectors is available alongside the published draft for cross-checking.

## 7. Security Considerations

The test key material in Section 5.1 is deterministic and public. It MUST NOT be used to sign production ATP nodes. Doing so would expose those nodes to forgery by anyone who has read this document.

These vectors are integrity tests, not security tests. Passing them confirms that an implementation produces correct deterministic output for the inputs shown. It does not, by itself, establish that the implementation is free of cryptographic implementation bugs that would only manifest on different inputs (timing side channels in Ed25519 signing, malleability in canonicalizer parsing, hash-truncation bugs in nodeId computation, and similar). Implementations SHOULD additionally adopt the security practices recommended by the underlying primitive specifications [RFC8032] [RFC8785].

## 8. IANA Considerations

This document has no IANA actions.

## 9. Normative References

[ATP-CORE] D. Bates, "Agent Transaction Protocol (ATP)", Work in Progress, Internet-Draft, draft-bates-atp-00, 2026, <https://datatracker.ietf.org/doc/draft-bates-atp/>.

[RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

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

[RFC8174] B. Leiba, "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", RFC 8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

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

## 10. Informative References

[JCS-TESTDATA] A. Rundgren et al., "JSON Canonicalization Scheme -- multi-language reference implementations and test data", 2025, <https://github.com/cyberphone/json-canonicalization>.

## Author’s Address

David A. Bates
SVT Robotics
Email: david.bates@svtrobotics.com
URI: https://www.svtrobotics.com/