Network Working Group
Internet-Draft
Intended status: Informational
Expires: 29 October 2026

J. Fassbender
Umarise

27 April 2026

External Temporal Anchoring for Transparency Services
draft-fassbender-scitt-time-anchor-01

Abstract

This document defines a mechanism for external temporal anchoring of digital artifacts by committing cryptographic hashes to an independent, publicly verifiable ledger -- specifically, the Bitcoin blockchain via the OpenTimestamps protocol. The resulting proof is independently verifiable by any party with access to ledger state, without reliance on a trusted third party. The SCITT Architecture [RFC9943] is used as the primary integration example, but the anchoring primitive is applicable to any system requiring externally verifiable temporal proof. No changes to the SCITT architecture are required.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 29 October 2026.

Copyright Notice

Table of Contents

1. Introduction

Cryptographic time-stamping -- proving that a datum existed at or before a given point in time -- is a foundational primitive for audit, compliance, and non-repudiation on the Internet.

RFC 3161 [RFC3161] defines a widely deployed protocol in which a trusted Time Stamping Authority (TSA) signs a timestamp token binding a hash to a point in time. The security of an RFC 3161 timestamp depends on the TSA's private key, its operational continuity, and the validity of its certificate chain. If the TSA ceases operations, its certificate expires without renewal, or its key is compromised, previously issued timestamps may become unverifiable or disputed.

This document defines a complementary mechanism in which temporal proof derives from inclusion in a public, append-only ledger maintained by computational consensus -- specifically, the Bitcoin blockchain. The resulting proof is independently verifiable by any party with access to ledger state, without reliance on any single authority or certificate chain.

The distinction is structural:

* RFC 3161: a trusted authority attests to the time of a hash commitment. Verification requires trust in that authority.
* This document: temporal proof is a consequence of ledger inclusion. Verification requires only access to public ledger state.

These approaches are not mutually exclusive. A system MAY use both RFC 3161 timestamps and ledger-based anchoring to provide complementary assurance under different trust assumptions.

The SCITT Architecture [RFC9943] is used as the primary integration example throughout this document. SCITT defines a framework for Transparency Services that record signed claims about digital artifacts. A Transparency Service receives Signed Statements, appends them to a verifiable log, and returns cryptographic Receipts proving inclusion.

SCITT is deliberately ledger-agnostic and does not mandate a specific time source. Time is derived from log position -- an internal clock controlled by the Transparency Service operator. This creates an architectural gap: the system that manages the evidence also manages the timeline. The operator can:

* Delay recording without detection
* Backdate entries (within operational constraints)
* Present different log views to different auditors (equivocation)

Furthermore, the Verifier does not need to trust any Calendar Server or anchoring intermediary -- the trust root is Bitcoin consensus itself. This property, termed "verification independence" in this document (Section 3.2), is the primary architectural distinction from existing time-stamping mechanisms. SCITT mitigates equivocation through consistency proofs, but these proofs are relative to the log itself. There is no external reference point.

This document defines an OPTIONAL profile that closes this gap by anchoring operations to an external, publicly verifiable ledger. The anchoring mechanism is generic and applicable beyond SCITT to any system requiring externally verifiable temporal proof.

1.1. Motivating Example

An AI research laboratory produces model weights and safety evaluations that must be auditable by regulators and the public. The lab registers each artifact with a SCITT Transparency Service and receives a Receipt. However, regulators ask: "How do we know the lab did not register these weights after the safety evaluation was already public -- backdating the claim?"

With External Temporal Anchoring, the Transparency Service submits the SHA-256 hash of the Signed Statement to an anchoring service. The anchoring service returns an Anchor Proof -- a portable, self-contained cryptographic proof that the hash was committed to the Bitcoin blockchain. The proof is independently verifiable by any party with access to Bitcoin block headers, without contacting the anchoring service or the Transparency Service.

Within the next Bitcoin confirmation -- typically 10 to 60 minutes -- the regulator obtains a temporal guarantee: these model weights existed no later than block height H. The anchoring service provides proof of existence and time, not claims about authorship, quality, or regulatory compliance.

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "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, as shown here.

1.3. Terminology

* *Temporal Anchor*: A cryptographic commitment of a hash value to a public, append-only ledger, proving that the hashed content existed at or before the ledger’s recorded time.

* *Anchor Proof*: A portable, self-contained proof object (e.g., an OpenTimestamps .ots file) that is independently verifiable without contacting the anchoring service.

* *Anchor Ledger*: A public, append-only data structure where no single controlling authority -- including the proof issuer -- can rewrite historical state or timestamps. Bitcoin is the reference Anchor Ledger in this document.

2. Anchoring Model

2.1. Overview

External temporal anchoring adds an independent time reference to SCITT operations without modifying the SCITT architecture. A Transparency Service that implements this profile MUST anchor operations to an Anchor Ledger at one or both of the following levels:

1. *Statement Anchoring* (per-statement)
2. *Log Root Anchoring* (periodic)

The following diagram illustrates the actors and message flow in a federated anchoring deployment:

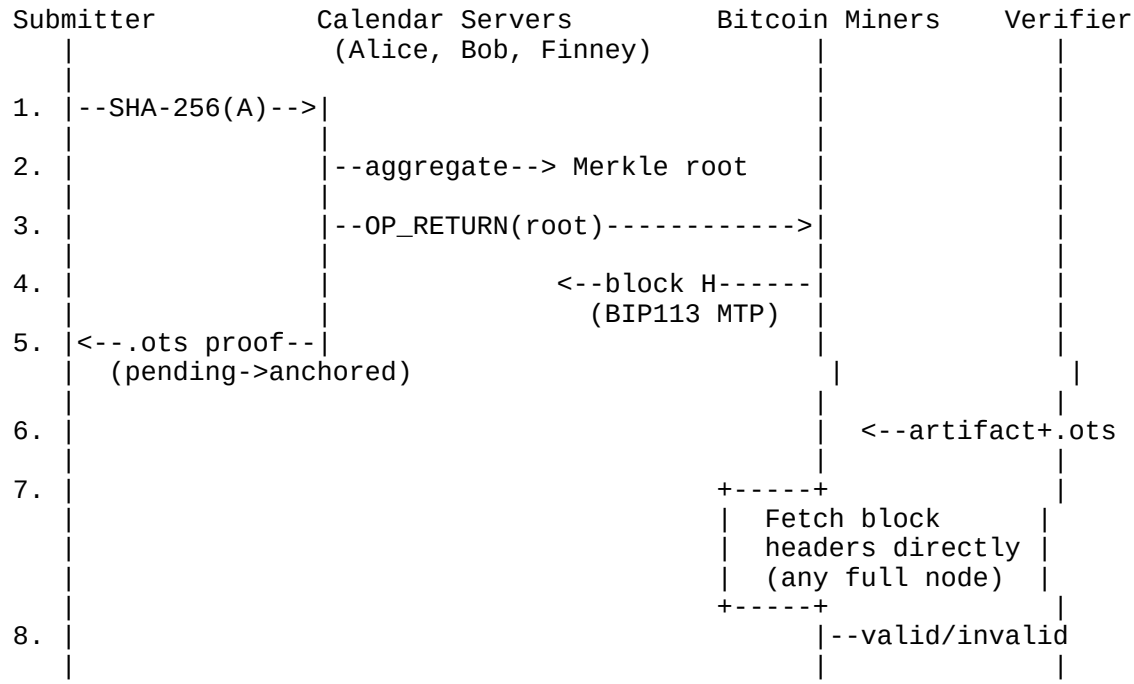


Figure 1: Federated Anchoring Message Flow

Key trust property: the Verifier (step 7) retrieves block headers directly from the Bitcoin network. The Verifier does NOT need to trust any Calendar Server -- if a Calendar Server equivocated, the .ots proof simply fails verification against the blockchain. The trust root is Bitcoin’s Proof-of-Work consensus, not any intermediary.

2.2. Statement Anchoring

When a Transparency Service receives a Signed Statement, it MAY compute a Temporal Anchor for that statement:

```
anchor_input = SHA-256(SignedStatement)
anchor_proof = Anchor(anchor_input, AnchorLedger)
```

The resulting Anchor Proof is stored alongside the SCITT Receipt. Together, they provide:

* *Receipt*: proof of inclusion in the Transparency Service log
* *Anchor Proof*: proof of existence at or before time T on the Anchor Ledger

2.3. Log Root Anchoring

A Transparency Service SHOULD periodically anchor the root of its verifiable data structure:

```
log_root = MerkleRoot(TransparencyLog)
anchor_proof = Anchor(log_root, AnchorLedger)
```

This creates an external checkpoint. If the operator later presents a different log state, the anchored root provides a publicly verifiable commitment against which inconsistencies can be detected.

2.4. Anchor Proof Format

An Anchor Proof MUST be:

1. *Self-contained*: Verifiable without contacting the anchoring service or the Transparency Service
2. *Portable*: A standalone file that travels with the Receipt
3. *Deterministic*: Given the same input bytes and ledger state, verification MUST produce the same result

The verification function is:

```
V(B, P, L) -> { valid | invalid | unverifiable }
```

Where: - *B* = the bytes being verified - *P* = the Anchor Proof - *L* = the Anchor Ledger

This function is defined in Section 4 of the Anchoring Specification [ANCHORING].

2.4.1. Abstract Field Requirements

A conformant Anchor Proof MUST encode the following abstract fields, regardless of serialization format:

#	Field	Required	Description
F1	artifact_hash	MUST	SHA-256 digest of the anchored byte sequence
F2	hash_algorithm	MUST	Algorithm identifier (MUST be "sha-256")
F3	merkle_path	MUST	Ordered sequence of operations linking F1 to the ledger commitment
F4	ledger_id	MUST	Identifier of the Anchor Ledger (e.g., "bitcoin-mainnet")
F5	block_height	MUST	Block number in which the anchor commitment appears
F6	block_hash	SHOULD	Hash of the block header for cross-verification
F7	tx_id	SHOULD	Transaction identifier containing the commitment
F8	block_time	MUST	Timestamp from the block header (see Section 2.6)
F9	anchor_status	MUST	One of: "submitted", "pending", "anchored", "failed"
F10	calendar_url	MAY	URL of the calendar server used for submission

Table 1: Anchor Proof Abstract Fields

Fields F1-F5, F8, and F9 are REQUIRED for a proof with anchor_status "anchored". Fields F6-F7 are RECOMMENDED. Field F10 is informational.

A proof with anchor_status "pending" MUST contain at minimum F1, F2, F9, and a partial merkle_path (F3) sufficient for later upgrade.

2.4.2. OpenTimestamps Wire Format Mapping

This profile uses the OpenTimestamps binary format as the reference serialization. The mapping from abstract fields to OTS encoding is as follows:

#	Abstract Field	OTS Encoding
F1	artifact_hash	Initial hash input to the proof chain
F2	hash_algorithm	Implicit: SHA-256 (OTS magic byte 0x08)
F3	merkle_path	Sequence of append (0xf0), prepend (0xf1), and hash (0x08) operations
F4	ledger_id	Attestation tag: Bitcoin (0x0588960d73d71901)
F5	block_height	Derived: verifier resolves from Bitcoin block headers
F6	block_hash	Derived: verifier resolves from Bitcoin block headers
F7	tx_id	Derived: verifier resolves from Bitcoin block data
F8	block_time	Derived: from block header timestamp field
F9	anchor_status	Implicit: presence of attestation tag = "anchored"; absence = "pending"
F10	calendar_url	Encoded as pending attestation URL in incomplete proofs

Table 2: OTS Wire Format Mapping

Note: Fields F5-F8 are "derived" in OTS because the binary proof encodes the Merkle path to the Bitcoin transaction, not the block metadata directly. The verifier extracts these values by replaying the proof operations against the Bitcoin blockchain. This is a design strength: the proof is compact and self-contained, while the ledger provides the authoritative metadata.

An implementation MAY use an alternative serialization format provided it encodes all REQUIRED abstract fields from Table 1. A future specification MAY define a CBOR-based encoding (see Section 4.4).

2.5. Anchor Ledger Requirements

An Anchor Ledger used with this profile MUST satisfy the properties defined in [ANCHORING] Section 7 (Ledger Qualification):

1. *Append-only*: Historical entries cannot be modified or deleted
2. *Public*: Any party can read and verify entries without permission
3. *No single controlling authority*: No single entity -- including the proof issuer -- can rewrite historical state or timestamps
4. *Independently verifiable*: Verification does not require trust in any specific service or operator

Bitcoin satisfies all four requirements and is the reference Anchor Ledger for this profile.

2.6. Temporal Precision

This section formally defines the temporal claim established by a verified Anchor Proof, the assumptions under which the claim holds, and the bounds on temporal uncertainty.

2.6.1. Temporal Claim

Given artifact A and Anchor Proof P verified against Bitcoin block B at height H:

A existed at or before T_B

where T_B is the Median Time Past (MTP) of block B as defined in [BIP113]. T_B is a ledger-derived temporal reference; it is not a wall-clock creation time and MUST NOT be interpreted as such.

The claim is exact with respect to T_B: the uncertainty is in T_B itself relative to wall-clock time, not in the claim relative to T_B. The phrase "at or before" is definitive -- no tighter bound is claimed or implied.

2.6.2. Assumptions

The temporal claim holds under the following assumptions:

ASSUMPTION 1 (Hash Collision Resistance): No second-preimage or collision has been found for the hash algorithm identified in P. For SHA-256 [RFC6234], this assumption is supported by the current state of cryptanalytic research.

ASSUMPTION 2 (Ledger Immutability): Block B at height H has not been replaced by a competing chain. This assumption strengthens with each subsequent confirmation. After six confirmations (~60 minutes), reorganization is considered computationally infeasible under current network conditions.

ASSUMPTION 3 (Timestamp Validity): Bitcoin miners comply with the consensus rule that a block’s timestamp MUST be strictly greater than the MTP of the previous 11 blocks and MUST be less than the network-adjusted time plus two hours [BIP113].

2.6.3. Temporal Bound

The temporal resolution of the claim is bounded by:

* Bitcoin’s block interval (~10 minutes average)
* MTP consensus rules (up to 2 hours of variance from wall-clock time)

These bounds are inherent to the Anchor Ledger and cannot be reduced by the anchoring service or the verifier. For use cases requiring sub-minute precision, an additional time source (e.g., RFC 3161) MAY be combined with anchoring to provide a complementary, finer-grained timestamp.

2.6.4. Non-Claims

An Anchor Proof explicitly does NOT establish:

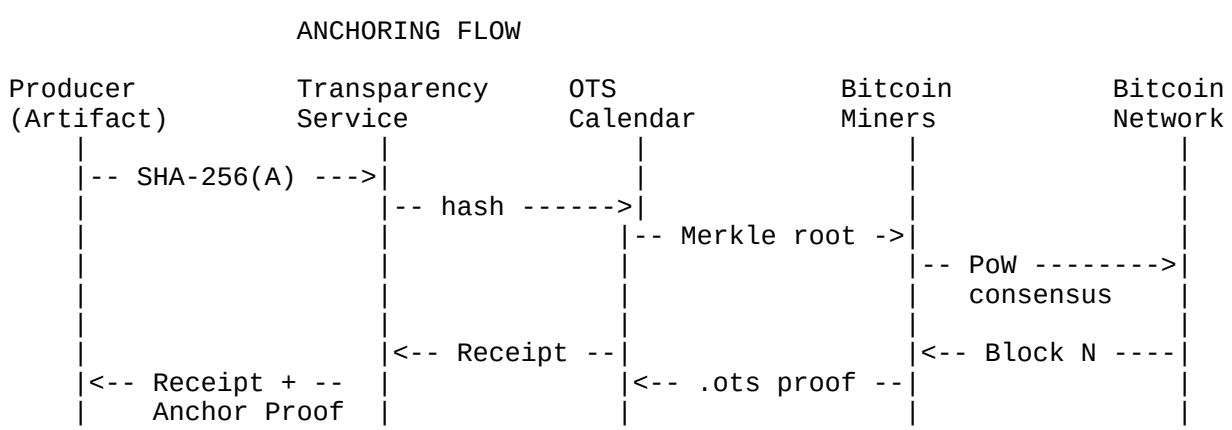
* That A was created at T_B (only: existed at or before T_B)
* That A was created by any specific party
* That A did not exist before T_B
* That T_B corresponds to wall-clock time (T_B is consensus-accurate, not wall-clock-accurate)

These exclusions are consistent with [ANCHORING] Section 8 (Semantic Exclusions) and Section 15 (Non-Retroactivity).

2.7. Architectural Overview

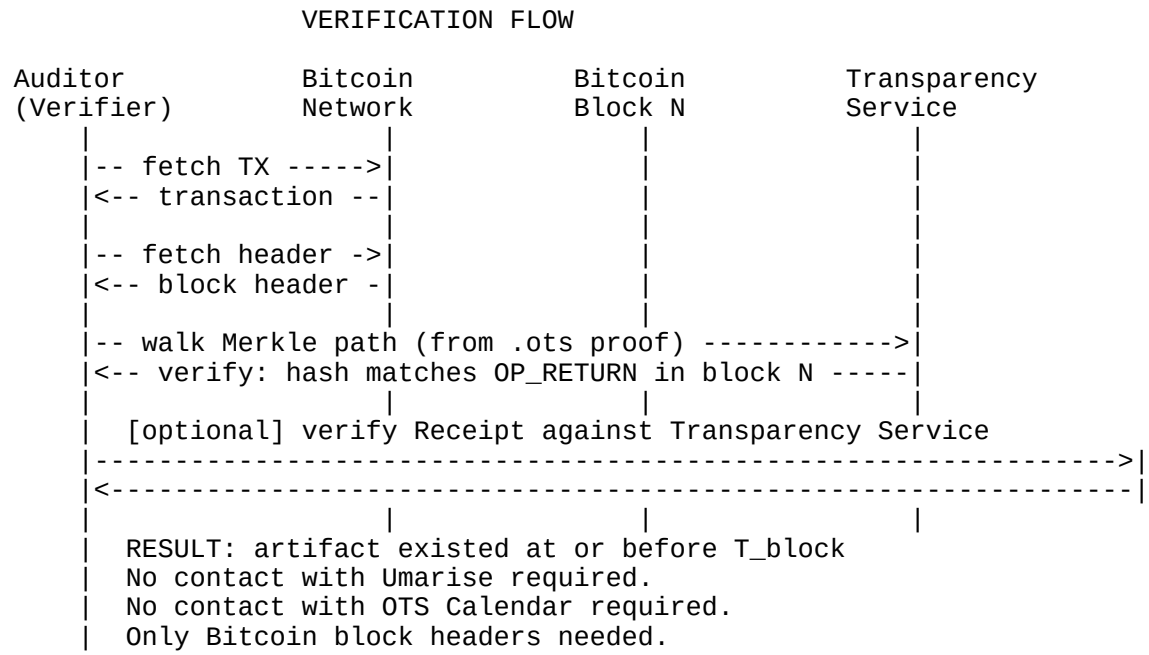
This section provides a complete actor-level view of the anchoring and verification flows. The role of Bitcoin miners is shown explicitly, since miners -- not any intermediary -- are the trust anchor that makes the temporal claim binding.

2.7.1. Anchoring Flow



Note: Miners include the Merkle root in a block. The block is accepted by the network via proof-of-work consensus. This is the moment the temporal anchor becomes binding and independently verifiable. Until inclusion in a confirmed block, the Anchor Proof carries status "pending" (see F9 in Section 2.4.1) and MUST NOT be relied upon as a temporal claim.

2.7.2. Verification Flow



The Verifier's only required trust dependency is the Bitcoin block header chain, which can be obtained from any full node or independent block explorer. Contact with the Transparency Service is OPTIONAL and only relevant if the Verifier wishes to additionally confirm SCITT log inclusion. This separation is what allows the Anchor Proof to satisfy the self-containment requirement defined in Section 2.4.

3. Verification Procedure

This section defines the normative verification algorithm for Anchor Proofs produced under this profile. An implementation that claims conformance to this profile MUST implement the procedure specified in Section 3.1.

3.1. Verification Algorithm (VERIFY-ANCHOR)

The verification function V(B, P, L) defined in Section 2.4 is instantiated as follows:

Algorithm: VERIFY-ANCHOR(artifact_bytes, proof_bundle)

Input:

```
  artifact_bytes    -- the original artifact byte sequence
  proof_bundle      -- contains: ots_proof (.ots file),
                       claimed_hash, origin_id,
```

```
                          bitcoin_block_height (optional)

   Output:
     { valid, invalid, unverifiable }

   Steps:

   1.  RECOMPUTE HASH
       computed_hash <- SHA-256(artifact_bytes)
       The verifier MUST recompute the hash from the original
       bytes. The verifier MUST NOT rely on any claimed hash
       value without independent computation.

   2.  COMPARE HASH
       IF HEX(computed_hash) != STRIP-PREFIX(
           proof_bundle.claimed_hash):
         RETURN invalid
         // The artifact does not match the anchored hash.

   3.  PARSE OTS PROOF
       parsed <- OTS-DESERIALIZE(proof_bundle.ots_proof)
       The verifier MUST parse the binary .ots proof according
       to the OpenTimestamps wire format (Section 2.4.2).
       IF parsed IS NULL OR parsed.hash != computed_hash:
         RETURN invalid

   4.  CHECK PROOF STATUS
       IF parsed contains only a calendar commitment (no
       Bitcoin attestation tag 0x0588960d73d71901):
         RETURN unverifiable
         // The proof has not been upgraded to a Bitcoin
         // anchor. It depends on the Calendar Server.

   5.  VERIFY BITCOIN MERKLE PATH
       The verifier MUST replay the sequence of append (0xf0),
       prepend (0xf1), and hash (0x08) operations encoded in
       the proof to derive the expected OP_RETURN value.
       tx_id <- OTS-EXTRACT-TX(parsed)
       expected_op_return <- OTS-WALK-MERKLE-PATH(
           parsed, computed_hash)

   6.  VERIFY AGAINST BITCOIN
       The verifier MUST retrieve the Bitcoin transaction from
       any full node, block explorer, or local block header
       cache. The verifier MUST NOT depend on any single
       service for this lookup.
       bitcoin_tx <- FETCH-TX(tx_id)              [NAKAMOTO]
       IF bitcoin_tx IS NULL:
```

```
        RETURN unverifiable              // ledger unavailable

      IF bitcoin_tx.OP_RETURN != expected_op_return:
        RETURN invalid

  7.  EXTRACT TEMPORAL BOUND
      block_header  <- FETCH-BLOCK-HEADER(
          bitcoin_tx.block_hash)
      prev_headers  <- FETCH-PREV-HEADERS(block_header, 11)
      timestamp_T_B <- MEDIAN(prev_headers.timestamp)  [BIP113]
      // T_B is the Median Time Past (MTP) of block B.
      // MTP is always strictly less than the actual
      // inclusion time, so "existed at or before T_B"
      // is a conservative, forward-drift-free claim.
      // See Section 2.6.1.

  8.  RETURN valid
      // The proof demonstrates: these exact bytes existed
      // at or before T_B (the MTP of block B). No
      // authorship, ownership, or identity claim is made.
      // See Section 2.6 (Temporal Precision) for the
      // semantics of T_B.
```

Figure 2: VERIFY-ANCHOR Algorithm

3.2. Verification Independence

A conformant verifier MUST be able to complete the VERIFY-ANCHOR procedure using ONLY:

1. The artifact bytes (possessed by the verifier)
2. The .ots proof file (portable, self-contained)
3. Access to Bitcoin block headers (any full node, any block explorer, or a local header cache)

A conformant verifier MUST NOT require:

* Contact with any anchoring service or Transparency Service
* An API key, account, or authentication credential
* Trust in any certificate authority
* The Calendar Server that issued the pending commitment

This satisfies the Independence Requirement defined in [ANCHORING] Section 9: verification is possible even if the anchoring service ceases to exist.

3.3. Error Conditions

The VERIFY-ANCHOR algorithm produces three possible outputs. Implementations MUST handle each as follows:

Output	Meaning
valid	The artifact bytes match the anchored hash, the Merkle path is correct, and the Bitcoin transaction confirms the commitment. The artifact existed at or before T_B (the MTP of the containing block), as defined in Section 2.6.1.
invalid	The artifact does not match the anchored hash (step 2), the proof fails to parse (step 3), or the Merkle path does not match the Bitcoin OP_RETURN (step 6). The verifier SHOULD treat this as a verification failure.
unverifiable	The proof is pending (step 4) or the Bitcoin ledger is unavailable (step 6). The verifier SHOULD retry after a delay. A pending proof MAY become verifiable after Bitcoin confirmation (~2-4 hours).

Table 3: VERIFY-ANCHOR Output Semantics

4. Integration with SCITT Architecture

4.1. No Protocol Changes Required

This profile is additive. It does not modify:

* The Signed Statement format
* The Receipt format
* The Transparency Service API
* The consistency or inclusion proof mechanisms

A Transparency Service operator MAY adopt this profile unilaterally. Auditors MAY verify Anchor Proofs independently of the SCITT verification flow.

4.2. Metadata Extension

A Transparency Service that implements this profile SHOULD include anchor metadata in its service parameters:

```
{
  "anchor_ledger": "bitcoin",
  "anchor_method": "opentimestamps",
  "anchor_level": "log_root",
  "anchor_interval": "batch",
  "anchor_spec": "https://anchoring-spec.org/v1.0/"
}
```

4.3. Receipt Association

An Anchor Proof MAY be associated with a Receipt by including the Anchor Proof hash in the Receipt's metadata, or by co-locating the .ots file with the Receipt in a proof bundle.

4.4. Relationship to RFC 9921 (COSE Timestamp Headers)

RFC 9921 defines two COSE header parameters -- 3161-ctt and 3161-ttc -- for embedding RFC 3161 Time-Stamp Tokens directly inside COSE_Sign1 structures [RFC9052]. This establishes a precedent: COSE already supports binding external temporal proofs to signed objects without modifying the object's payload.

This profile extends the same principle to a different trust model:

RFC 9921	This Profile
Trust root: CA	Trust root: Consensus (PoW)
Proof: TST token	Proof: .ots file
Precision: ms	Precision: block (~10 min)
Binding: COSE hdr	Binding: Anchor Proof or hdr
Verifier trusts: TSA + CA chain	Verifier trusts: Bitcoin blockchain
Offline verify: With CA certs	Offline verify: With block headers

The two mechanisms are complementary. A Transparency Service MAY implement both -- producing a dual-anchored Anchor Proof that combines CA-rooted precision (RFC 3161 via RFC 9921) with consensus-rooted independence (OpenTimestamps via this profile).

A COSE header parameter for OpenTimestamps proofs is out of scope for this document but could be defined in a future specification, following the pattern established by RFC 9921. Such a specification could define a CBOR-based serialization of the abstract fields in Section 2.4.1 (Table 1), using COSE_Sign1 as the envelope, analogous to how RFC 9921 embeds RFC 3161 TST tokens. The abstract field table in this document is designed to facilitate such mapping without requiring changes to the anchoring model itself.

5. Formal Security Argument

This section provides a formal argument supporting the central claim of the External Temporal Anchoring mechanism.

5.1. Claim

Theorem (Existence-at-or-before-T): If VERIFY-ANCHOR(A, P) = valid (Section 3.1), then the byte sequence A existed at or before time T, where T is the timestamp of the Bitcoin block containing the anchor commitment.

5.2. Assumptions

The proof relies on the following assumptions, each of which is a well-established property of the underlying primitives:

* *A1 (Collision Resistance)*: SHA-256 is collision-resistant. That is, no computationally bounded adversary can find distinct inputs x != y such that SHA-256(x) = SHA-256(y) [FIPS180-4].

* *A2 (Ledger Immutability)*: The Bitcoin blockchain is append-only and infeasible to rewrite for any confirmed block. Specifically, a transaction included in block B with k >= 6 confirmations cannot be removed or altered without controlling a majority of the network’s hash power [NAKAMOTO].

* *A3 (Temporal Ordering)*: Bitcoin block timestamps are constrained by consensus rules. A block’s timestamp must be greater than the median of the previous 11 blocks [BIP113]. The maximum forward drift permitted by nodes is 2 hours. Therefore, for a block with timestamp T_b, the block was accepted by the network within the interval [T_b - tolerance, T_b + 2h], and any transaction in that block was submitted before the block was mined.

* *A4 (OTS Correctness)*: The OpenTimestamps proof format provides a deterministic chain of hash operations from an input hash H to a value embedded in a Bitcoin transaction's OP_RETURN output [OTS]. This chain is publicly verifiable. The normative algorithms for construction and verification are specified in Appendix D of this document; see Section 5.2.1.

* *A5 (Operator Independence)*: The anchoring service and any intermediary may fail, be compromised, or cease operations without affecting the validity of previously issued proofs. The temporal claim is grounded in Bitcoin consensus, not in the continued operation or trustworthiness of the anchoring service operator. This assumption is supported by the verification independence requirement in Section 3.2: a conformant verifier requires only the artifact bytes, the .ots proof, and access to Bitcoin block headers.

5.3. Note on Assumption A4

OpenTimestamps [OTS] is a deployed open-source protocol with multiple independent implementations and a public project site [OTS-SITE]. It does not, however, have a formal IETF or ISO specification. This profile therefore treats OpenTimestamps as a *reference implementation*, not as a normative dependency.

The security argument of Section 5.3 does not rely on the correctness of any particular OTS software library. Assumption A4 reduces to the correctness of two well-understood primitives that are fully specified in Appendix D:

1. *SHA-256 Merkle tree construction* (Appendix D.1, D.2): Binary hash trees as described in [RFC6962], Section 2.1.

2. *Bitcoin transaction parsing and block header verification* (Section 3.1, steps 5-7): OP_RETURN output identification and block confirmation depth checking per [BIP141].

Both primitives are independently verifiable against the Bitcoin blockchain without reference to any OTS software. Appendix D provides self-contained pseudocode sufficient to implement verification from first principles. If the OTS reference implementation were to become unavailable, the algorithms in Appendix D remain sufficient to verify any proof produced under this profile.

5.4. Proof

Let A be an artifact (byte sequence), and let P be a proof Anchor Proof for which VERIFY-ANCHOR(A, P) returns valid.

Step 1: By steps 1-2 of the verification algorithm (Section 3.1), SHA-256(A) = H, where H is the hash committed in the Anchor Proof. By assumption A1, A is the unique preimage of H with overwhelming probability (2^-128 security level for second preimage).

Step 2: By steps 3 and 5, the .ots proof contains a deterministic sequence of hash operations linking H to a value V embedded in a Bitcoin transaction TX. By assumption A4, this chain is correct and verifiable: V = f(H) where f is the composition of the Merkle path operations.

Step 3: By step 6, the Bitcoin transaction TX exists in block B and TX.OP_RETURN = V. By assumption A2, TX was included in B at the time B was mined, and this inclusion cannot be retroactively altered.

Step 4: By step 7, block B has Median Time Past T_B, computed per [BIP113] as the median of the timestamps of the 11 blocks preceding B. By assumption A3, Bitcoin consensus requires that the actual time at which B was accepted by the network is strictly greater than T_B (since T_B is the median of prior blocks, it is always behind actual inclusion time). Therefore, transaction TX -- which commits to V, which commits to H, which commits to A -- existed before block B was mined.

Step 5: The hash commitment is causal: to produce H, the artifact A must have existed before H was computed. To include H in the OTS Merkle tree that produces V, H must have existed before TX was broadcast. To include TX in block B, TX must have existed before B was mined.

Therefore: A existed -> H was computed -> TX was broadcast -> B was mined at an actual time strictly greater than T_B. The artifact A existed at or before T_B (the Median Time Past of block B at height H), as defined in Section 2.6.1. No 2-hour tolerance caveat applies: T_B is a conservative lower bound on the actual inclusion time by construction. [end]

5.5. Strength and Limitations

The above argument is a *computational security* argument, not an information-theoretic proof. Its strength is bounded by:

1. The collision resistance of SHA-256 (currently 128-bit security level)
2. The cost of a sustained 51% attack on Bitcoin, which depends on then-current network conditions and is external to this specification
3. The 2-hour temporal tolerance inherent to Bitcoin block timestamps

The argument does NOT prove: - When exactly A was created (only an upper bound on existence) - Who created A (no identity binding) - That A has not been modified since anchoring (only that these specific bytes existed)

These limitations are inherent to the mechanism and are documented in [ANCHORING] Section 8 (Semantic Exclusions).

5.6. Anchor Proof Integrity

The Anchor Proof does not replace the SCITT Receipt. It provides an independent temporal claim. If the Anchor Proof is lost or corrupted, the Receipt remains valid within the SCITT framework. The temporal independence guarantee is degraded but the claim integrity is unaffected.

5.7. Hash Algorithm Agility

This profile specifies SHA-256 as the hash function for anchor inputs. If SHA-256 is deprecated, the Anchor Ledger requirement (Section 2.5) remains valid with a successor hash function. The anchoring mechanism is hash-agile by design.

5.8. Ledger Availability

Bitcoin’s availability characteristics exceed those of any single-operator Transparency Service. However, Anchor Proof verification requires access to the Bitcoin blockchain (or a trusted copy). Offline verification is possible with a local blockchain copy or cached block headers.

5.9. Equivocation Detection

Log Root Anchoring (Section 2.3) enables equivocation detection: if a Transparency Service presents different log states to different parties, the anchored root provides a public commitment that can be compared. This is a strictly stronger guarantee than SCITT provides without external anchoring.

6. Security Considerations

This section follows the guidelines in [RFC3552] for describing threats and mitigations relevant to the External Temporal Anchoring mechanism defined in this document.

The attacker model assumes the following capabilities:

* The attacker can observe all network traffic between the submitter, the anchoring service, Calendar Servers, and the Anchor Ledger.
* The attacker can operate one or more Calendar Servers.
* The attacker can submit arbitrary hashes to the anchoring service.
* The attacker cannot find collisions or second pre-images for SHA-256 (Assumption A1, Section 5.2).
* The attacker cannot rewrite confirmed Bitcoin blocks with k >= 6 confirmations (Assumption A2, Section 5.2).
* The attacker cannot control a majority of Bitcoin’s network hash power.

The following subsections enumerate specific threats under this model.

6.1. Threat: Hash Collision (Forged Commitment)

An attacker could construct a second artifact B such that SHA-256(B) = SHA-256(A), thereby claiming that the anchor for artifact A also proves the existence of artifact B.

This is remediated by the collision resistance of SHA-256 [FIPS180-4]. No known practical collision attack exists against SHA-256 as of the date of this document. The Construction Algorithm (Appendix D.1) is hash-algorithm- agile: if SHA-256 is weakened, the hash_algo field permits migration to a successor algorithm without protocol changes.

6.2. Threat: Anchor Ledger Rewrite (51% Attack)

An attacker with majority hash power on the Anchor Ledger could rewrite the block containing the commitment, thereby invalidating or altering the temporal proof.

This is remediated by the economic cost of sustaining a majority attack on Bitcoin, which is the only Anchor Ledger currently qualified under the Anchoring Specification [ANCHORING] Section 7 (Ledger Qualification). The economic and operational cost of producing a competing chain with greater accumulated proof-of-work depends on then-current Bitcoin network conditions and is external to

this specification. The Verification Algorithm (Section 3.1) requires a minimum confirmation depth before accepting an anchor as valid.

6.3. Threat: Calendar Server Equivocation

An attacker operating a compromised OpenTimestamps calendar server could return divergent intermediate commitments to different clients, or withhold a valid commitment entirely.

This is remediated by the OTS protocol design: multiple independent Calendar Servers provide redundant commitment paths. The final anchor is a Bitcoin transaction, not a calendar assertion. A verifier does not need to trust any Calendar Server -- verification uses only the Bitcoin blockchain (Section 2.1, step 7). A Calendar Server that equivocates produces proofs that fail verification.

6.4. Threat: Transparency Service Equivocation

A malicious Transparency Service operator could present different log states to different relying parties while anchoring only one version.

This is remediated by Log Root Anchoring (Section 2.3). Because the anchored Merkle root is committed to the public ledger, any party holding a Receipt can independently compute the expected root and compare it against the anchored value. Divergent log states are detectable by any two parties that compare their anchored roots.

6.5. Threat: Temporal Claim Inflation

An attacker could claim that the anchor proves existence at a time earlier than the actual anchoring. For example, asserting that T equals the block timestamp minus an arbitrary margin.

This is remediated by the formal time semantics defined in Section 5.1. The anchor provides an existence-at-or-before-T guarantee where T is the consensus-confirmed block inclusion time. The protocol does not claim to prove existence at the exact moment of hash creation -- only that the hash existed no later than T. Verifiers MUST NOT interpret T as the creation time of the artifact.

6.6. Threat: Anchor Proof Tampering

An attacker could modify the certificate.json or .ots proof file within an Anchor Proof after generation, for example by altering the captured_at timestamp or substituting a different hash value.

This is remediated by the cryptographic binding between the components. The .ots proof commits to a specific hash value; any modification to certificate.json that changes the hash breaks the OTS verification chain. The Verification Algorithm (Section 3.1) re-derives the hash from the original artifact bytes and verifies it against the .ots proof independently -- it does not trust the metadata in certificate.json.

6.7. Threat: Denial of Anchoring Service

An attacker could prevent the Transparency Service from submitting commitments to the Anchor Ledger, for example via a denial-of-service attack on the Calendar Servers or the Transparency Service’s network connectivity.

This is remediated by the asynchronous design of the protocol (Section 2.1). Commitments can be retried. The Transparency Service retains the pending .ots proof and resubmits when connectivity is restored. During the outage, existing anchored proofs remain independently verifiable. New Signed Statements are still recorded by the Transparency Service; only their external temporal anchoring is delayed.

6.8. Threat: Long-Term Hash Algorithm Compromise

Over decades, SHA-256 may become vulnerable to collision or pre-image attacks due to advances in computing (including quantum computing).

This is remediated by the algorithm-agility provision in the protocol. The hash_algo field in the anchor record (Section 2.4.1, field F2) permits migration to a successor hash algorithm. Existing proofs anchored with SHA-256 retain their validity for the period during which SHA-256 was considered secure. The Anchoring Specification [ANCHORING] Section 15 defines the temporal semantics that bound this validity window.

6.9. Trust Boundary: Hash Intake

The anchoring service proves that a given hash existed at or before a certain time. It does not, and cannot, prove the truth, accuracy, or origin of the data that produced that hash.

An attacker could submit the hash of a fabricated or falsified artifact before anchoring. The resulting proof would be cryptographically valid and temporally bound, yet the underlying data would be false.

This is not remediated by the protocol. It is an explicit trust boundary: the anchoring service guarantees temporal existence of a hash commitment, not the integrity or authenticity of the pre-image data. Consumers of anchor proofs MUST apply independent verification of the artifact content, authorship, and provenance outside the scope of this specification.

6.10. Positive Property: No Long-Term Key Dependency

Unlike certificate-based timestamping mechanisms (e.g., RFC 3161), Anchor Proofs do not depend on any signing key, certificate chain, or key management infrastructure. The proof’s validity derives from the mathematical properties of hash functions and the computational consensus of the Anchor Ledger.

This eliminates three threat categories that apply to key-based timestamping:

* *Key compromise*: There is no private key that, if exposed, would allow an attacker to forge timestamps.
* *Certificate expiration*: There is no certificate whose expiry would invalidate existing proofs.
* *Authority revocation*: There is no trusted authority whose revocation would render proofs unverifiable.

An Anchor Proof verified today remains verifiable indefinitely, provided SHA-256 retains its collision resistance (Section 6.8) and the Anchor Ledger remains accessible (Section 6.7).

6.11. Threat: Premature Anchored State (False Finality)

An attacker -- or a faulty implementation -- could mark a proof as "anchored" before the Bitcoin transaction has reached a durable confirmation depth. A relying party that accepts an "anchored" status asserted by the anchoring service could be misled into relying on a temporal proof that is subsequently invalidated by a chain reorganization.

This is remediated by two independent controls. First, the Verification Algorithm (Section 3.1) requires the verifier to independently retrieve the Bitcoin transaction and confirm block inclusion -- the verifier does not rely on any status field asserted by the anchoring service. Second, Assumption A2 (Section 5.2) specifies that a minimum of six confirmations (~60 minutes) must be reached before the immutability assumption is considered computationally sound. Implementations MUST NOT promote a proof from "pending" to "anchored" (field F9, Section 2.4.1) until the transaction has reached the minimum confirmation depth required by their deployment policy.

6.12. Threat: Batch Integrity Compromise (Merkle Mixing)

In the batch anchoring mode (Appendix D.3), an implementation error in the Merkle tree construction could incorrectly associate hashes from different submitters within the same batch. A defective batch could produce a valid-appearing Anchor Proof that binds a hash to an incorrect Merkle position, undermining the evidence integrity of all origins in the batch.

This is remediated by the Batch Anchoring algorithm (Appendix D.3), which specifies deterministic leaf ordering (lexicographic sort before concatenation) and requires individual origin records per hash in addition to the shared Merkle root anchor. A verifier can independently recompute the Merkle path from a leaf hash to the root and confirm correct positioning. Post-batch verification sampling SHOULD be performed by implementations to detect systematic construction errors before they are exposed to relying parties.

7. Privacy Considerations

The External Temporal Anchoring mechanism operates on cryptographic hashes only. No artifact content, personal data, or identity information is transmitted to or stored on the Anchor Ledger.

However, the following privacy-relevant properties apply:

* The hash of an artifact is a unique identifier. An observer who independently possesses the artifact can confirm whether it was anchored by computing the hash and searching for a matching commitment.

* Bitcoin transactions are publicly visible and permanent. An anchored hash cannot be removed from the ledger.

* The temporal ordering of anchored hashes is public. An observer can determine that hash A was anchored before hash B.

Implementations that anchor hashes of privacy-sensitive artifacts SHOULD inform users that the hash becomes a permanent, publicly queryable identifier on the anchor ledger.

In multi-tenant deployments, anchored hashes from different tenants appear on the same public ledger within the same Bitcoin blocks. An observer with access to hashes from multiple tenants can determine temporal ordering relationships across tenants -- including whether two artifacts were anchored in the same batch -- even when no artifact content is disclosed. Implementations that anchor on behalf of multiple tenants SHOULD be aware that the public ledger creates a permanent, cross-tenant ordering record. The 2-hour maximum forward drift in Bitcoin block timestamps (Section 2.6.3, Assumption A3) defines the worst-case window within which ordering observations are unreliable; within a single confirmed block (~10 minutes), ordering between co-batched hashes is not preserved by the protocol.

8. IANA Considerations

This document has no IANA actions.

9. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[FIPS180-4] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015, <https://doi.org/10.6028/NIST.FIPS.180-4>.

[ANCHORING] Fassbender, J., "Anchoring Specification (IEC), Version 1.0", DOI 10.5281/zenodo.19537321, February 2026, <https://doi.org/10.5281/zenodo.19537321>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

10. Informative References

[RFC3161] Adams, C., Cain, P., Pinkas, D., and R. Zuccherato, "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)", RFC 3161, DOI 10.17487/RFC3161, August 2001, <https://www.rfc-editor.org/info/rfc3161>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", RFC 9052, DOI 10.17487/RFC9052, August 2022, <https://www.rfc-editor.org/info/rfc9052>.

[RFC9921] Pinkas, D., Jones, M., and K. Yasuda, "CBOR Object Signing and Encryption (COSE) Header Parameter for Carrying and Conveying a Timestamp Token", RFC 9921, DOI 10.17487/RFC9921, February 2025, <https://www.rfc-editor.org/info/rfc9921>.

[RFC9162] Laurie, B., Messeri, E., and R. Stradling, "Certificate Transparency Version 2.0", RFC 9162, DOI 10.17487/RFC9162, December 2021, <https://www.rfc-editor.org/info/rfc9162>.

[RFC9943] Birkholz, H., Delignat-Lavaud, A., Fournet, C., Deshpande, Y., and S. Lasker, "An Architecture for Trustworthy and Transparent Digital Supply Chains", RFC 9943, DOI 10.17487/RFC9943, March 2026, <https://www.rfc-editor.org/info/rfc9943>.

[OTS] Todd, P., "OpenTimestamps: Scalable, Trust-Minimized, Distributed Timestamping with Bitcoin (Reference Implementation, Release v0.4.3)", November 2022, <https://github.com/opentimestamps/python-opentimestamps/tree/python-opentimestamps-v0.4.3>.

[OTS-SITE] Todd, P., "OpenTimestamps (Project Site)", 2016, <https://opentimestamps.org>.

[OTS-DESIGN] Todd, P., "OpenTimestamps Announcement", September 2016, <https://petertodd.org/2016/opentimestamps-announcement>.

[NAKAMOTO] Nakamoto, S., "Bitcoin: A Peer-to-Peer Electronic Cash System", DOI 10.2139/ssrn.3440802, October 2008, <https://doi.org/10.2139/ssrn.3440802>.

[BIP113] Kerin, T. and M. Friedenbach, "Median time-past as endpoint for lock-time calculations (BIP 113, Status: Deployed)", August 2015, <https://github.com/bitcoin/bips/blob/24e96e870fffaa257b465ce1f0370c14aac588e8/bip-0113.mediawiki>.

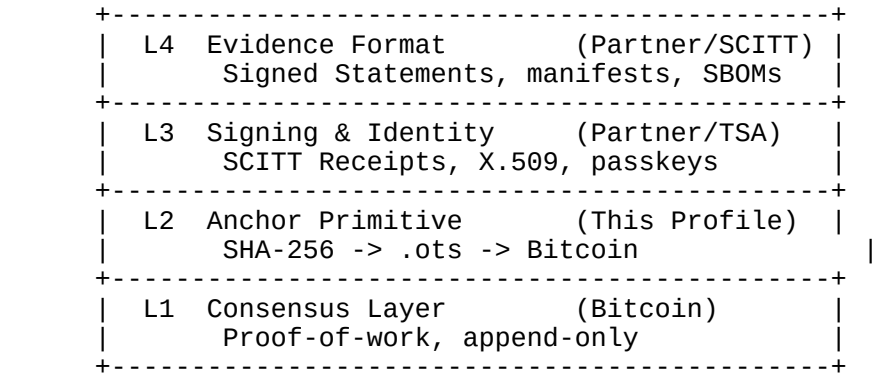
[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[BIP141] Lombrozo, E., Lau, J., and P. Wuille, "Segregated Witness (Consensus layer) (BIP 141, Status: Deployed)", December 2015, <https://github.com/bitcoin/bips/blob/1f0b563738199ca60d32b4ba779797fc97d040fe/bip-0141.mediawiki>.

[MERKLE] Merkle, R., "A Digital Signature Based on a Conventional Encryption Function", Advances in Cryptology - CRYPTO '87, Lecture Notes in Computer Science, vol 293, Springer, DOI 10.1007/3-540-48184-2_32, 1988, <https://doi.org/10.1007/3-540-48184-2_32>.

Appendix A. Relationship to Four-Layer Evidence Stack



SCITT operates at L3-L4. This profile defines the L2 integration. L1 is the Bitcoin consensus layer. The layers are independent: each can be verified without the others.

Appendix B. Example: Anchored SCITT Flow

1. Producer creates artifact A
2. Producer signs Signed Statement S about A
3. Transparency Service receives S
4. Transparency Service appends S to log -> Receipt R
5. Transparency Service computes SHA-256(S) -> H
6. Transparency Service submits H to anchoring service
7. Anchoring service returns Anchor Proof P (.ots)
8. Transparency Service stores (R, P) together

Verification (by any auditor):

```
  a. Verify R against Transparency Service log  -> SCITT valid
  b. Verify P against Bitcoin                   -> Temporal valid
  c. Compare: H in P == SHA-256(S)              -> Binding valid
```

Note: If the Transparency Service is no longer available, step (a) cannot be performed -- steps (b) and (c) remain independently valid. The temporal and binding proofs do not depend on the continued operation of the log.

Appendix C. Implementation Status

As of March 2026, the following implementations exist:

* *Umarise Core API*: Production anchoring service implementing the Anchoring Specification, with Merkle-tree batching and OpenTimestamps anchoring. RFC 3161 TSA integration is planned. 253,000+ attestations anchored.

* *verify-anchoring.org*: 100% client-side verification tool. Zero API contact, zero tracking. Public domain.

* *@umarise/cli*: Node.js CLI for automated anchoring. Published on npm.

* *anchor-action*: GitHub Actions integration for CI/CD pipeline anchoring. Published on GitHub Marketplace.

Appendix D. Acknowledgements

The authors thank Eliot Lear (Independent Submissions Editor) for his detailed review of the initial submission and his concrete guidance on strengthening the formal security argument, algorithmic specification, and Security Considerations structure.

The authors thank Nicole Bates (Microsoft, SCITT Working Group Chair) for her review of the SCITT integration approach and her assessment that the mechanism requires no changes to existing SCITT protocols.

The OpenTimestamps protocol was designed by Peter Todd, whose reference implementation [OTS] underpins the anchoring mechanism described in this document.

The Merkle tree construction in Appendix D.3 follows the conventions established in RFC 6962 (Certificate Transparency).

Author's Address:

Jonna Fassbender
Umarise
The Netherlands

Email: j.fassbender@umarise.com
URI: https://umarise.com

Appendix E. OTS Anchoring Protocol -- Construction and Verification

Sections D.1 through D.3 are informative examples of a compliant construction. Sections D.4 and D.5 have been promoted to the normative main body (Section 3) and are retained here as cross-references only. This appendix supports Assumption A4 (Section 5.2). It defines the construction algorithms for OpenTimestamps-based anchoring as used in this document. The algorithms are presented in pseudocode and are self-contained: they can be implemented independently of any OpenTimestamps software library. They correspond to the reference implementation described in Appendix C, but do not depend on it.

For background on the OpenTimestamps protocol design, see [OTS-SITE] and [OTS-DESIGN].

E.1. D.1. Construction Algorithm (Anchor)

```
Algorithm: CONSTRUCT-ANCHOR(artifact_bytes)

Input:
  artifact_bytes    -- arbitrary byte sequence (the artifact)

Output:
  origin_record     -- {origin_id, hash, captured_at, proof_status}
  pending_proof     -- serialised OTS pending proof (.ots file)

Steps:

1.  HASH COMPUTATION
    hash_value  <- SHA-256(artifact_bytes)              // [FIPS180-4]
    hash_string <- "sha256:" || HEX(hash_value)        // canonical form

2.  TIMESTAMP CAPTURE
    captured_at <- NOW()                                // UTC ISO 8601

3.  ORIGIN REGISTRATION
    origin_id   <- UUID-v4()                            // unique identifier
    short_token <- RANDOM-ALPHANUMERIC(8)               // human-readable ref
    INSERT origin_attestations {
      origin_id, hash: hash_string, hash_algo: "sha256",
      captured_at, short_token
    }

4.  OTS CALENDAR SUBMISSION
    // Submit hash to >=1 OpenTimestamps Calendar Server(s) [OTS]
    FOR EACH calendar IN configured_calendars:
      pending_commitment <- HTTP-POST(calendar.url, hash_value)
      STORE pending_commitment

5.  PENDING PROOF ASSEMBLY
    // The .ots file at this stage contains calendar commitment(s)
    // but NOT a Bitcoin anchor. It is NOT independently verifiable.
    pending_proof <- OTS-SERIALIZE(hash_value, pending_commitments)
    INSERT core_ots_proofs {
      origin_id, ots_proof: pending_proof, status: "pending"
    }

6.  RETURN {origin_id, hash_string, captured_at, proof_status: "pending"}
```

E.2. D.2. Upgrade Algorithm (Bitcoin Confirmation)

```
   Algorithm: UPGRADE-PENDING-PROOFS()

   // Runs periodically (e.g. every 15 minutes) as a background worker.
   // Converts calendar-level commitments to Bitcoin-level proofs.

   Input:
     none (reads from core_ots_proofs WHERE status = "pending")

   Steps:

   1.  pending_proofs <- SELECT * FROM core_ots_proofs
                         WHERE status = "pending"
                         AND created_at < NOW() - INTERVAL '2 hours'

   2.  FOR EACH proof IN pending_proofs:

       2a. CONTACT CALENDAR
           upgraded_proof <- OTS-UPGRADE(proof.ots_proof)
           // OTS-UPGRADE contacts the Calendar Server that issued
           // the pending commitment and requests the Bitcoin
           // Merkle path if available.

       2b. IF upgraded_proof IS NULL:
           // Bitcoin transaction not yet confirmed, or calendar
           // not yet merged into a block. Retry next cycle.
           CONTINUE

       2c. EXTRACT BITCOIN BINDING
           block_height <- OTS-EXTRACT-BLOCK-HEIGHT(upgraded_proof)
           block_time   <- OTS-EXTRACT-BLOCK-TIME(upgraded_proof)

       2d. VERIFY LOCALLY
           // Verify the Merkle path from hash -> Bitcoin OP_RETURN
           valid <- OTS-VERIFY(upgraded_proof, proof.origin_hash)
           IF NOT valid:
             LOG-ERROR("Upgrade verification failed", proof.origin_id)
             CONTINUE

       2e. PERSIST UPGRADED PROOF
           UPDATE core_ots_proofs SET
             ots_proof           = upgraded_proof,
             status              = "anchored",
             bitcoin_block_height = block_height,
             anchored_at          = block_time,
             upgraded_at          = NOW()
           WHERE origin_id = proof.origin_id
```

E.3. D.3. Batch Anchoring with Merkle Trees

```
Algorithm: BATCH-ANCHOR(hash_list)

// For high-volume partners (>1,000 hashes per request).
// Anchors a single Merkle root instead of N individual hashes.

Input:
  hash_list       -- ordered list of SHA-256 hashes [h_0 ... h_{n-1}]

Output:
  batch_record    -- {batch_id, merkle_root, merkle_origin_id, origins[]}

Steps:

1.  VALIDATE
    ASSERT 1 <= |hash_list| <= 1000
    FOR EACH h IN hash_list:
      ASSERT h matches /^(sha256:)?[0-9a-f]{64}$/

2.  COMPUTE MERKLE ROOT                                  // [RFC9162] Section 2.1
    // Canonical leaf ordering: strip "sha256:" prefix, sort pairs
    // lexicographically before concatenation.
    level <- [STRIP-PREFIX(h) FOR h IN hash_list]
    WHILE |level| > 1:
      next_level <- []
      FOR i <- 0 TO |level|-1 STEP 2:
        IF i+1 < |level|:
          pair <- SORT([level[i], level[i+1]])
          next_level.APPEND(SHA-256(pair[0] || pair[1]))
        ELSE:
          next_level.APPEND(level[i])           // odd element: promote
      level <- next_level
    merkle_root <- "sha256:" || level[0]

3.  CREATE INDIVIDUAL ORIGINS
    FOR EACH h IN hash_list:
      CALL CONSTRUCT-ANCHOR-RECORD(h)           // DB insert only, no OTS

4.  ANCHOR MERKLE ROOT
    // Only the root is submitted to OTS calendar(s).
    // This reduces N OTS operations to 1.
    CALL CONSTRUCT-ANCHOR(merkle_root)

5.  LINK BATCH
    INSERT batch_submissions {
      batch_id, merkle_root, merkle_origin_id,
      hashes: hash_list, origin_ids: [per-hash origin_ids]
```

```
    }

6.  RETURN batch_record
```

E.4. D.4. Verification Algorithm

The normative verification algorithm has been promoted to Section 3.1 of this document. This appendix section is retained as a cross-reference for continuity.

See Section 3.1 (VERIFY-ANCHOR) for the full eight-step verification procedure with MUST/SHOULD requirements.

E.5. D.5. Verification Independence

The normative verification independence requirements have been promoted to Section 3.2 of this document.

See Section 3.2 for the definitive statement of what a conformant verifier requires and what it MUST NOT depend on.

Author’s Address

Jonna Fassbender
Umarise
Email: j.fassbender@umarise.com