SCITT
Internet-Draft
Intended status: Informational
Expires: 29 November 2026

J. Fassbender
Umarise

28 May 2026

Bitcoin-Anchored Temporal Proof for Transparency Services
draft-fassbender-scitt-time-anchor-02

Abstract

This document defines a mechanism for temporal anchoring of digital artifacts by committing cryptographic hashes to the Bitcoin blockchain via the OpenTimestamps protocol [OTS]. The resulting proof is independently verifiable by any party with access to Bitcoin block headers, without reliance on a trusted third party. The SCITT Architecture [RFC9943] is used as the primary integration example. 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 November 2026.

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 the Bitcoin blockchain [NAKAMOTO] via the OpenTimestamps protocol [OTS].

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?"

Under this profile, 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.

* *External Temporal Anchor*: A Temporal Anchor that is computed and stored outside the system whose artifacts are being anchored. In the SCITT context, "external" means external to the Transparency Service's verifiable log: the anchor is committed to a separate ledger (the Anchor Ledger, see below) controlled by neither the Transparency Service operator nor any single authority.

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

* *Proof Bundle*: The structured input to the VERIFY-ANCHOR algorithm (Section 3.1). A Proof Bundle MUST contain the following fields:

* *ots_proof* (REQUIRED): The binary OpenTimestamps proof file (.ots), as specified in Section 2.4.2.

* *claimed_hash* (REQUIRED): The SHA-256 hash of the artifact, encoded as a hex string with optional "sha256:" prefix (e.g., "sha256:781bb71a..."). The verifier MUST recompute this value independently and MUST NOT rely on the claimed_hash field as authoritative.

* *origin_id* (OPTIONAL): An implementation-defined identifier for the anchoring record. Not used in verification; included for traceability.

* *bitcoin_block_height* (OPTIONAL): A cached block height from a prior verification. Not used in verification; included for informational purposes only.

The normative CDDL definition of the Proof Bundle structure is given in Section 2.4.3.

* *Anchor Ledger*: In this document, the Anchor Ledger is the Bitcoin blockchain [NAKAMOTO]. The properties that make Bitcoin suitable for this role -- public verifiability, append-only history, absence of a single controlling authority, and independent verifiability -- are documented in Section 2.5.

* *OpenTimestamps (OTS)*: A protocol and reference implementation for committing hashes to the Bitcoin blockchain via aggregating Calendar Servers, producing a portable proof file (the .ots file) that is independently verifiable against Bitcoin block headers. See [OTS], [OTS-SITE], and [OTS-DESIGN].

* *Attestation*: The term has two distinct uses in this document. (1) In the OpenTimestamps wire format [OTS], an attestation is a typed terminal element of an .ots proof that binds the proof to an external commitment. This document references two such types: the Bitcoin attestation (tag 0x0588960d73d71901), which binds the proof to a confirmed Bitcoin block, and the pending attestation, which references a Calendar Server URL pending Bitcoin confirmation. (2) In implementation-level contexts (Appendix C, pseudocode field origin_attestations), the word denotes an anchored origin record stored by the anchoring service. Neither use implies legal or evidentiary attestation in any jurisdictional sense.

* *Calendar Server*: A server in the OpenTimestamps protocol that aggregates submitted hashes into a Merkle tree and periodically commits the Merkle root to the Bitcoin blockchain. A Calendar Server is a convenience for batching; it is not a trust authority. Verification of a completed Anchor Proof does not require contacting any Calendar Server (see Section 3.2).

* *Bitcoin Miner*: A network participant that constructs and proposes new Bitcoin blocks via proof-of-work [NAKAMOTO]. In the context of this document, miners are relevant only as the consensus mechanism by which OpenTimestamps Calendar Server commitments become embedded in immutable block history. This document does not depend on the identity, intent, or cooperation of any specific miner.

* *Transparency Service*: As defined in [RFC9943], a service that records signed claims about digital artifacts in a verifiable log and returns cryptographic Receipts proving inclusion.

* *Signed Statement*: As defined in [RFC9943] Section 4, a signed claim about a digital artifact submitted by a Producer to a Transparency Service for inclusion in the verifiable log.

* *Receipt*: As defined in [RFC9943] Section 5, a cryptographic structure returned by a Transparency Service that proves inclusion of a Signed Statement in the log.

* *Producer*: A party that creates a digital artifact and submits a Signed Statement about it to a Transparency Service. Also referred to as the Submitter when the emphasis is on the act of submission rather than artifact creation. See [RFC9943].

* *Verifier*: A party that evaluates an Anchor Proof (and optionally a Receipt) to determine whether an artifact's temporal claim is valid, invalid, or unverifiable. The Verifier is independent from the Transparency Service operator and the anchoring service.

* *Auditor*: A specialized Verifier acting on behalf of regulators, courts, or compliance functions, with the additional goal of detecting equivocation or backdating by the Transparency Service operator. An Auditor uses the same verification procedure (Section 3.1) as any other Verifier.

* *Median Time Past (MTP)*: As specified in [BIP113], the median timestamp of the eleven Bitcoin blocks immediately preceding a given block. MTP is constrained by Bitcoin consensus rules to be strictly less than the actual block inclusion time, and is therefore a conservative lower bound on the moment a block was accepted by the network.

* *OP_RETURN*: A Bitcoin transaction output type that permits embedding a small payload (up to 80 bytes) in a provably unspendable output. OpenTimestamps Calendar Servers use OP_RETURN to commit Merkle roots to the Bitcoin blockchain. See [BIP141] for the wire format and [OTS] for the specific use in this profile.

* *Merkle path*: An ordered sequence of sibling hash values and concatenation operations that, when applied to a leaf hash, deterministically reproduces the root of a Merkle tree. The OpenTimestamps .ots proof encodes a Merkle path from the artifact hash to the OP_RETURN value of a Bitcoin transaction. See [RFC6962] Section 2.1.1 for the general construction.

* *Block header*: As specified in [NAKAMOTO], the fixed-size 80-byte structure at the start of every Bitcoin block, containing the block's version, previous block hash, Merkle root of transactions, timestamp, difficulty target, and nonce. Block headers are sufficient to verify Bitcoin proof-of-work without downloading transaction data.

* *Full node*: A Bitcoin network participant that independently validates the entire blockchain (all blocks and transactions) against consensus rules. A full node requires no trust in third parties for the data it has validated. See [NAKAMOTO].

* *Block explorer*: A public web service that exposes Bitcoin blockchain data (blocks, transactions, headers) via HTTP. Block explorers are convenient sources of block headers but are not trust authorities; a Verifier MAY use any block explorer or its own full node interchangeably.

* *Confirmation depth*: The number of blocks that have been appended to the Bitcoin blockchain after a given block, including the block itself. A transaction with k confirmations is considered increasingly resistant to chain reorganization as k grows. This profile requires k >= 6 before promoting an Anchor Proof from pending to anchored (see Section 6.11).

* *Block-height binding*: The normative chronological binding established by an Anchor Proof under this profile. The binding identifies the Bitcoin block at height H that contains the OP_RETURN commitment of (the Merkle root containing) the artifact's hash. Block-height binding is exact: a proof either binds an artifact to block H or it does not. The binding is consensus-verified and is not subject to clock drift, miner-set timestamp manipulation, or the 贼2 hour block.time envelope discussed in Section 2.6.6.

* *Reference Wall-Clock Projection*: The wall-clock value block.time(H), where H is the block height of the block-height binding (above). The Reference Wall-Clock Projection is the human-readable temporal value exposed by deployed OpenTimestamps verifiers and SHOULD be presented alongside H for human consumption. The Reference Wall-Clock Projection is informative and is NOT the normative temporal value of the proof; the normative value is the block-height binding (see Section 2.6.1).

1.4. Notation

This document uses pseudocode in Sections 2.2, 2.3, 3.1, and Appendix D. The notation follows these conventions:

Function and algorithm names are written in UPPER-CASE with hyphens as word separators (e.g., VERIFY-ANCHOR, OTS-DESERIALIZE). The hyphen is part of the identifier and MUST NOT be parsed as subtraction. This convention is used to distinguish algorithm names from arithmetic expressions.

Variables and abstract objects are written in snake_case (e.g., artifact_bytes, anchor_proof). Abstract objects passed as parameters refer to terms defined in Section 1.3 (Terminology); when an abstract object name appears in pseudocode, it denotes an instance of the term defined there.

The following abstract objects appear as pseudocode parameters and refer to defined terms:

* AnchorLedger -- the Anchor Ledger as defined in Section 1.3.

* AnchorProof -- the Anchor Proof as defined in Section 1.3.

* SignedStatement -- a Signed Statement as defined in Section 1.3, serialized to a canonical byte sequence prior to hashing. The exact serialization is specified by [RFC9943] and is out of scope for this profile.

* TransparencyLog -- the verifiable log of a Transparency Service as defined in [RFC9943] Section 4.

The following pseudocode functions are used. Each is either a standard cryptographic primitive, a Bitcoin protocol operation, an OpenTimestamps operation, or an implementation utility. All are defined or referenced as follows:

Top-level anchoring abstraction:

* Anchor(input, AnchorLedger) -- takes a byte sequence input and an Anchor Ledger reference, and returns an Anchor Proof binding input to a commitment on that ledger. This is an abstract operation; its concrete instantiation for the Bitcoin Anchor Ledger via OpenTimestamps is specified in Appendix D.1 (CONSTRUCT-ANCHOR).

Cryptographic primitives:

* SHA-256(x) -- the SHA-256 hash function applied to byte sequence x, as specified in [FIPS180-4].

* MerkleRoot(L) -- the root of a Merkle tree constructed over the ordered list L, as specified in [RFC6962] Section 2.1.

Encoding utilities:

* HEX(b) -- the lowercase hexadecimal string representation of byte sequence b, with no separators or prefixes (e.g., HEX(0xAB12) = "ab12").

* STRIP-PREFIX(s) -- given a string s of the form "sha256:" || h, returns the substring h. If no prefix is present, returns s unchanged.

* UUID-v4() -- generates a random UUID per [RFC4122] Section 4.4.

* RANDOM-ALPHANUMERIC(n) -- returns a random string of n characters drawn from the alphabet [0-9a-zA-Z]. Used for human-readable references only; not security-critical.

* NOW() -- returns the current wall-clock time, as determined by the local system, in UTC ISO 8601 format (e.g., "2026-04-29T12:26:30Z"). NOW() is used for recording when an artifact was submitted to the anchoring service; it is NOT a security-critical time source and MUST NOT be confused with the consensus-derived temporal claim established by an Anchor Proof (see Section 2.6).

* HTTP-POST(url, body) -- performs an HTTP POST request per [RFC9110]. Used for submitting hashes to Calendar Servers (Appendix D.1, step 4). Not security-critical: Calendar Server compromise is addressed in Section 6.3.

Bitcoin protocol operations:

* FETCH-TX(tx_id) -- retrieves a Bitcoin transaction by its identifier from any Bitcoin full node, block explorer, or cached source. Returns the transaction or NULL if not found. Defined operationally per [NAKAMOTO]; not specific to any single client implementation.

* FETCH-BLOCK-HEADER(block_hash) -- retrieves a Bitcoin block header by its hash. Same source independence as FETCH-TX.

* FETCH-PREV-HEADERS(header, n) -- retrieves the n block headers immediately preceding the given header, walking the prev_block field. Used to compute Median Time Past per [BIP113].

* MEDIAN(values) -- the statistical median of an ordered list of integer values.

OpenTimestamps operations:

The following functions are operations on OpenTimestamps proof structures, as documented in [OTS] and [OTS-DESIGN]. The normative semantics required by this document are specified in Appendix D; conformant implementations need only implement the algorithms there and need not depend on any particular OTS software library.

* OTS-SERIALIZE(hash, commitments) -- produces a binary .ots proof structure containing the input hash and the given commitments. See Appendix D.1.

* OTS-DESERIALIZE(ots_bytes) -- parses a binary .ots proof into an internal structure. Returns NULL if parsing fails.

* OTS-UPGRADE(pending_proof) -- contacts a Calendar Server and replaces a calendar-level commitment with a Bitcoin Merkle path, when available. See Appendix D.2.

* OTS-VERIFY(proof, hash) -- returns true if the proof's internal Merkle path correctly resolves the given input hash to the embedded ledger commitment. See Section 3.1.

* OTS-EXTRACT-TX(parsed) -- extracts the Bitcoin transaction identifier referenced by the OTS attestation in parsed.

* OTS-EXTRACT-BLOCK-HEIGHT(proof) -- extracts the Bitcoin block height referenced by the proof.

* OTS-EXTRACT-BLOCK-TIME(proof) -- extracts the Bitcoin block timestamp referenced by the proof.

* OTS-WALK-MERKLE-PATH(parsed, hash) -- replays the sequence of append/prepend/hash operations encoded in parsed, starting from hash, to derive the expected ledger commitment value. See Section 3.1, step 5.

Storage and control flow:

The pseudocode uses INSERT, SELECT, UPDATE, STORE, and LOG-ERROR as implementation-detail placeholders for persistent storage operations. These are not normative: an implementation MAY use any storage mechanism. Standard control flow keywords (IF, WHILE, FOR EACH, RETURN, ASSERT, CONTINUE, APPEND, CALL) follow the conventions of pseudocode as commonly used in the IETF (e.g., [RFC9162]). The SORT(L) operation returns the input list L in ascending lexicographic order; it is used in Appendix D.3 to ensure deterministic Merkle leaf ordering.

The arrow ← denotes assignment. The symbols || denote concatenation. The operand types determine whether this is byte or string concatenation. Where the distinction is interoperability-critical, the surrounding pseudocode states the operand type explicitly (see, for example, the Merkle construction in Appendix D.3, step 2c, which performs hex-string concatenation). The function notation f(x, y) denotes function application; brackets [a, b, c] denote ordered lists.

Subscript notation: In running text and security arguments, subscripts are occasionally used to disambiguate values that share a common name (for example, distinguishing the timestamp value of block b from a generic value T). Subscripts are notational only; they do not denote function application or array indexing unless explicitly stated.

Single-letter symbols in running text and the security argument: This document uses the following single-letter symbols, with meaning fixed by local context. Each occurrence specifies its referent the first time it appears in a given section.

* A -- an artifact (byte sequence) under verification. Used in the formal security argument (Section 5).

* B -- a Bitcoin block. Used in the formal security argument (Section 5) and threat model (Section 6).

* H -- a Bitcoin block height (a non-negative integer identifying a position on the canonical chain). Used in Sections 1.3, 2.6.1, 5.3, and throughout for the block-height binding.

* h -- a hash value (a 256-bit byte string). Used in the formal security argument (Section 5.3) for the hash committed in an Anchor Proof. The lower-case form distinguishes the hash value from the block height H.

* V -- the verification function defined in Section 2.4, invoked as V(ArtifactBytes, AnchorProof, AnchorLedger).

* k -- a Bitcoin confirmation depth (a non-negative integer counting blocks appended on top of the block containing the commitment).

These symbols are used with the meanings above only. The distinction between H (block height) and h (hash value) is maintained throughout the document; where confusion could arise the surrounding text states the referent explicitly.

Top-level procedures. The procedures VERIFY-ANCHOR, CONSTRUCT-ANCHOR, UPGRADE-PENDING-PROOFS, and BATCH-ANCHOR are defined in Section 3.1 and Appendix D (D.1 through D.3). They use the functions and conventions defined above.

2. Anchoring Model

2.1. Overview

This profile 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:

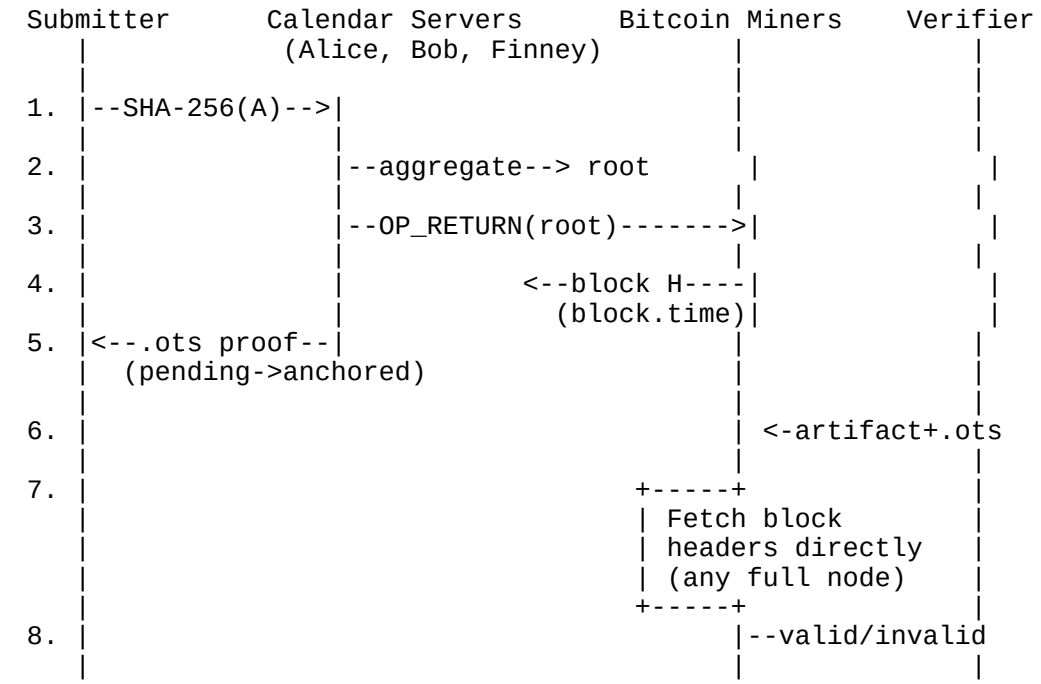


Figure 1: Federated Anchoring Message Flow

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 that the bytes of SignedStatement were committed to the Bitcoin blockchain, where the commitment is included in the block at height H. This block-height binding is the normative temporal claim of this profile; its precise definition appears in Section 2.6.1. The Reference Wall-Clock Projection block.time(H) (Section 1.3) SHOULD be presented alongside H for human readability and is informative, not normative.

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 anchored to a specific Bitcoin block height H. If the operator later presents a different log state, the anchored root provides a publicly verifiable commitment against which inconsistencies can be detected. The block-height binding (Section 1.3) of the log root anchor is the normative temporal reference for this checkpoint.

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 V is defined as follows:

```
V(ArtifactBytes, AnchorProof, AnchorLedger)
    → { valid | invalid | unverifiable }
```

Where:

* *ArtifactBytes* -- the byte sequence being verified

* *AnchorProof* -- the Anchor Proof (Section 1.3)

* *AnchorLedger* -- the Anchor Ledger (Section 1.3)

The short-form notation V(B, P, L) from the Anchoring Specification [ANCHORING] Section 4 is equivalent: this profile uses the descriptive parameter names recommended in the companion-note of [ANCHORING] Section 4 for clarity in running text.

2.4.1. Abstract Field Requirements

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

#	Field	Reqd	CDDL Type	Description
F1	artifact_hash	MUST	bstr .size 32	SHA-256 digest of anchored byte sequence
F2	hash_algorithm	MUST	"sha256"	Algorithm identifier
F3	merkle_path	MUST	[* operation]	Ordered operations linking F1 to the ledger commitment
F4	ledger_id	MUST	tstr	Anchor Ledger identifier (e.g., "bitcoin-mainnet")
F5	block_height	MUST	uint	Block number containing the commitment

F6	block_hash	SHLD	bstr .size 32	Block header hash for cross-verification
F7	tx_id	SHLD	bstr .size 32	Transaction identifier containing the commitment
F8	block_time	MUST	uint	Block header nTime (see Section 2.6)
F9	anchor_status	MUST	status-tstr	"submitted" / "pending" / "anchored" / "failed"
F10	calendar_url	MAY	tstr / nil	URL of calendar server used for submission

Table 1: Anchor Proof Abstract Fields

The "Reqd" column uses MUST, SHLD (SHOULD), and MAY per [RFC2119]. The "CDDL Type" column lists the concrete CDDL [RFC8610] type for each field; status-tstr is a tstr restricted to the four enumerated values shown in the Description column. The complete normative CDDL schema is published as a machine-readable file (Section 2.4.3).

The literal "sha256" is used for the F2 hash_algorithm field, aligned with the OpenTimestamps internal naming convention used throughout this profile. The same algorithm is registered as sha-256 in the IANA Named Information Hash Algorithm Registry [RFC6920]; both names refer to the function specified in [FIPS180-4]. Implementations MUST use the literal "sha256" in the F2 field for compatibility with the canonical hash-prefix notation "sha256:" || HEX(h) employed in the algorithms of Appendix D.

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).

2.4.3. CDDL Schema for OpenTimestamps Reference Implementation

The following CDDL schema [RFC8610] defines the concrete encoding of the abstract fields in Table 1 for the OpenTimestamps reference implementation. An implementation MAY use an alternative encoding provided all REQUIRED fields from Table 1 are present.

```
; CDDL schema for OpenTimestamps Anchor Proof bundle
; Corresponds to abstract fields F1-F10 in Table 1

AnchorProofBundle = {
  artifact_hash:    bytes .size 32,          ; F1: SHA-256 digest
  hash_algorithm:   HashAlgorithm .default "sha256",
                                             ; F2: algorithm id
  merkle_path:      [+ MerkleOp],            ; F3: path to ledger
  ledger_id:        "bitcoin-mainnet",       ; F4: anchor ledger id
  ? block_height:   uint,                    ; F5: block number
                                             ;     (block-height
                                             ;     binding,
                                             ;     Section 2.6.1;
                                             ;     REQUIRED when
                                             ;     anchor_status =
                                             ;     "anchored")
  ? block_hash:     bytes .size 32,          ; F6: block header hash
  ? tx_id:          bytes .size 32,          ; F7: transaction id
  ? block_time:     uint,                    ; F8: Unix timestamp
                                             ;     (block header
                                             ;     nTime; Reference
                                             ;     Wall-Clock
                                             ;     Projection,
                                             ;     Section 1.3;
                                             ;     REQUIRED when
                                             ;     anchor_status =
                                             ;     "anchored")
  anchor_status:    AnchorStatus,            ; F9: proof lifecycle
  ? calendar_url:   uri,                     ; F10: submission calendar
}

AnchorStatus = "pending" / "anchored" / "failed"

HashAlgorithm = "sha256"                     ; SHA-256 [FIPS180-4]

MerkleOp  = PrependOp / AppendOp / HashOp
PrependOp = { 1 => bytes }                   ; OTS prepend (0xf1)
AppendOp  = { 2 => bytes }                   ; OTS append  (0xf0)
HashOp    = { 3 => HashAlgorithm }           ; OTS hash op (0x08)
```

Fields F5-F8 are derived by the verifier from the Bitcoin blockchain during execution of the VERIFY-ANCHOR algorithm (Section 3.1, steps 6-7) and need not be present in the serialized proof bundle. Their presence in the CDDL schema reflects the logical fields of a fully-verified proof, not mandatory wire format fields.

The normative semantic distinction between F5 (block_height, the block-height binding) and F8 (block_time, the Reference Wall-Clock Projection) is defined in Section 2.6.1. The CDDL schema records both values; the temporal claim of the proof is established by F5, with F8 presented for human readability.

The following CDDL schema defines the Proof Bundle structure passed to VERIFY-ANCHOR (Section 3.1). This is the normative definition of the proof_bundle input type referenced in Section 1.3.

```
; CDDL definition of Proof Bundle (input to VERIFY-ANCHOR)

ProofBundle = {
  ots_proof:              bytes,   ; REQUIRED: binary .ots file
  claimed_hash:           tstr,    ; REQUIRED: "sha256:<hex>"
  ? origin_id:            tstr,    ; OPTIONAL: traceability id
  ? bitcoin_block_height: uint,    ; OPTIONAL: cached block height
}
```

An implementation constructing a ProofBundle MUST populate ots_proof and claimed_hash. The verifier MUST NOT treat claimed_hash as authoritative; it MUST recompute the hash independently from artifact_bytes (Section 3.1, step 1).

The CDDL schema published at https://anchoring-spec.org/v1.2/cddl/anchor-proof-bundle.cddl is semantically equivalent to the schema above and serves as the machine-readable normative form. Implementations MUST treat that file as the authoritative concrete syntax in case of any typesetting discrepancy with this document.

2.5. Anchor Ledger Requirements

This profile uses the Bitcoin blockchain [NAKAMOTO] as the Anchor Ledger. This section defines the qualification properties that any system used as an Anchor Ledger under this profile MUST satisfy, and establishes that the Bitcoin blockchain satisfies all of them.

An implementation MUST NOT treat a system as an Anchor Ledger under this profile unless it satisfies all of the following properties:

1. *Append-Only Property*: The ledger MUST be append-only. Once data is recorded, it cannot be modified or removed without detection. Any attempt to alter historical entries MUST be computationally or economically infeasible.

1. *Public Accessibility*: The ledger MUST be publicly accessible for verification. Any party MUST be able to independently retrieve and inspect the relevant data required to validate an Anchor Proof.

1. *Decentralization / No Single Controlling Authority*: The ledger MUST NOT be controlled by a single trusted party -- including the proof issuer. Its integrity MUST derive from distributed consensus, cryptographic verification, or equivalent mechanisms that prevent unilateral manipulation of historical state or timestamps.

1. *Independent Verifiability*: The ledger MUST allow independent verification of inclusion proofs without reliance on the original anchoring service. A verifier MUST be able to validate that a given commitment exists in the ledger using only publicly available data and standard verification procedures.

The Bitcoin blockchain satisfies all four properties:

* *Append-only*: Bitcoin’s proof-of-work consensus and the cumulative work of the longest chain make alteration of historical blocks economically infeasible. After six confirmations (~60 minutes), reorganization is considered computationally infeasible under current network conditions (see Section 2.6.2, Assumption A2).

* *Public Accessibility*: Bitcoin block headers and transaction data are public. Any party can run a full node, query a public block explorer, or use the Bitcoin peer-to-peer network to retrieve the data required to verify an Anchor Proof.

* *Decentralization*: Bitcoin has no central operator. The network consensus is established by globally distributed miners and full nodes operating under publicly specified consensus rules. No single entity -- including the proof issuer or the anchoring service -- can rewrite historical blocks or timestamps.

* *Independent Verifiability*: Verification of an Anchor Proof requires only the artifact bytes, the OpenTimestamps .ots file, and Bitcoin block headers. No contact with the anchoring service or the OpenTimestamps Calendar Server is required (see Section 3.2).

This profile is deliberately scoped to the Bitcoin blockchain. Implementations MAY in principle use any other ledger that demonstrably satisfies the four properties above; however, this document does not specify the verification semantics for any ledger

other than Bitcoin. A future profile MAY extend this work to additional qualifying ledgers; such a profile would specify the ledger-specific verification algorithm, attestation format, and confirmation-depth requirements analogous to those defined for Bitcoin in Section 3.1 and Section 6.11.

The qualification properties above take precedence over [ANCHORING] Section 7 in case of conflict for the purposes of conformance to 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 that successfully verifies against the Bitcoin blockchain (i.e., VERIFY-ANCHOR(ArtifactBytes, ProofBundle) = valid), the normative temporal claim established by P is:

A was committed to the Bitcoin blockchain in the block at height H.

This claim is referred to as the *block-height binding* (Section 1.3). The binding is exact: a verified proof either binds the artifact to block height H or it does not. The binding is consensus-verified by the same proof-of-work process that secures the Bitcoin blockchain itself.

The block-height binding implies an existence claim: artifact A existed before block H was added to the canonical chain, since the hash of A was necessarily computed before it could be committed to a transaction included in H.

Reference Wall-Clock Projection. The wall-clock value block.time(H), also known as the nTime field of the block header at height H, SHOULD be presented to human verifiers as the Reference Wall-Clock Projection of the binding (Section 1.3). This value is the temporal reading returned by deployed OpenTimestamps verifiers. It is informative and is NOT the normative temporal value. Discussion of block.time and its drift envelope appears in Section 2.6.6.

Optional tighter bound: Median Time Past. Verifiers MAY additionally compute the Median Time Past (MTP) of block H per [BIP113]. MTP is a strict consensus-derived lower bound: the actual

time at which block H was accepted by the network is provably greater than MTP. Verifiers seeking a strict-lower-bound interpretation of "existed at or before [wall-clock time]" MAY use MTP as that bound. Computing MTP requires retrieval of the eleven block headers immediately preceding H. The block-height binding above does not require MTP and is normative regardless of whether MTP is computed.

Backwards-compatibility note. Anchor Proofs created under earlier interpretations of this profile, where the temporal value T was inferred to be block.time(H) rather than the block-height binding at H, remain valid under this profile. The two interpretations are logically equivalent for verification purposes: any proof that successfully binds an artifact to block H also fixes block.time(H) as a derivative value, and any wall-clock interpretation must ultimately resolve to a specific block height to be verifiable.

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): The block at height H, identified by its block hash, 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 (Block-Inclusion Causality): A transaction included in the block at height H was necessarily broadcast to the Bitcoin network before block H was mined. By extension, the OP_RETURN payload of that transaction (and hence the artifact hash committed to it) existed before block H was mined. This assumption follows directly from Bitcoin protocol mechanics; it does not depend on the specific value of any timestamp field in the block header.

Note: the claim "the artifact existed before block H was mined" is independent of block.time(H). The Reference Wall-Clock Projection block.time(H) (Section 1.3) provides a human-readable approximation of when block H was mined but is not part of the normative temporal claim. See Sections 2.6.3 and 2.6.6.

2.6.3. Temporal Bound

The temporal resolution of the block-height binding is bounded by Bitcoin's block interval, which targets an average of approximately ten minutes per block under the network's difficulty adjustment mechanism. The actual interval between two consecutive blocks is variable; individual block intervals can range from seconds to several hours.

When the Reference Wall-Clock Projection block.time(H) is presented alongside the block-height binding, an additional uncertainty applies. Bitcoin block timestamps are miner-set within consensus-allowed bounds and may diverge from the actual block-mining wall-clock time by up to approximately two hours in either direction (see Section 2.6.6 for full discussion). This uncertainty applies to the projection only; the block-height binding itself is exact.

For use cases requiring wall-clock precision finer than the block interval (e.g., sub-minute timestamps), an additional time source such as RFC 3161 [RFC3161] MAY be combined with this profile to provide a complementary, finer-grained timestamp. The two mechanisms operate under different trust models (see Section 4.4) and can be presented together.

2.6.4. Non-Claims

An Anchor Proof explicitly does NOT establish:

* That A was created at the time block H was mined (only: A existed before block H was mined)

* That A was created by any specific party

* That A did not exist before block H was mined

* That the Reference Wall-Clock Projection block.time(H) corresponds to true wall-clock time (it is a miner-set value within consensus bounds; see Section 2.6.6)

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

2.6.5. Pending Proofs and Temporal Definition

Anchor Proofs produced by this profile have two states with respect to the Bitcoin blockchain (see also Section 2.4.1 field F9, anchor_status):

* *pending*: the artifact hash has been submitted to one or more OpenTimestamps Calendar Servers, but no Bitcoin block containing the resulting commitment has yet been observed with sufficient confirmation depth.

* *anchored*: the proof has been upgraded (Appendix D.2) and binds to a specific block at height H with at least six confirmations.

The block-height binding (Section 2.6.1) becomes defined only when the proof reaches the anchored state. Pending proofs carry no normative temporal claim under this profile: the hash is in transit toward a block, but no block has yet been identified.

A Verifier that encounters a pending proof MUST return unverifiable (see Section 3.1, step 4) rather than attempting to construct a temporal claim from the calendar commitment alone. Calendar commitments are not consensus- verified; relying on them for temporal claims would reintroduce trust in the Calendar Server, which Section 3.2 explicitly excludes.

This treatment is structurally cleaner than alternatives that would assign provisional temporal values to pending proofs: under this profile, a temporal claim either exists (because a block-height binding has been established) or does not (because no block has yet been identified). There is no intermediate state in which a partial temporal claim is asserted.

2.6.6. Relationship to Wall-Clock Time Stamping

Time Stamping Authorities (TSAs) operating under [RFC3161] deliver a wall-clock timestamp as their normative value. That timestamp is derived from the TSA’s local clock and signed by the TSA’s private key. Verification depends on trust in the TSA, its certificate chain, and the integrity of its operational clock.

This profile takes a structurally different approach. The normative value delivered by a verified Anchor Proof is the block-height binding (Section 2.6.1): the artifact’s commitment is included in the Bitcoin block at height H. The reasons for this divergence from TSA convention are:

* A TSA’s timestamp is the assertion of a single trusted authority based on that authority’s local clock. The block-height binding is a consensus-verified fact: the block at height H contains the commitment because the Bitcoin network agreed, via proof-of-work, to include it there. The trust root differs (single authority vs. computational consensus), and this profile names the consensus-verified value as normative.

* Bitcoin block timestamps (block.time or nTime) are miner-set within consensus-allowed bounds. The consensus rules permit a block's nTime to be in the range approximately [MTP + 1, network_time + 2h], where network_time is the median of the Bitcoin node's peers' reported times [BIP113]. As a consequence:

* block.time can move backward relative to its parent block. Documented examples include block 156114 (~2 hours before its parent) and block 790402 (2 minutes before its parent). Such non-monotonicity is consensus- valid.

* A miner can set block.time up to approximately two hours ahead of the actual mining moment, which would cause an anchored artifact to appear to have existed earlier than it did.

These properties make block.time unsuitable as a normative wall-clock value. The block-height binding is not affected by these properties: a block exists at a specific height, or it does not, and that fact is determined by consensus.

* Deployed OpenTimestamps verifiers, including the upstream reference implementation [OTS], surface block.time(H) as the temporal value associated with a verified proof. This profile preserves that behaviour by defining block.time(H) as the Reference Wall-Clock Projection (Section 1.3) and recommending its presentation alongside H for human readability.

The result is a two-layer model:

* *Normative layer (cryptographic):* the block-height binding. Exact, consensus-verified, not subject to clock drift or miner manipulation.

* *Informative layer (human-readable):* the Reference Wall-Clock Projection block.time(H). Convenient for display, subject to the bounded uncertainty described above.

This separation -- of what is proved from what is displayed -- allows this profile to claim only what Bitcoin consensus guarantees, while still enabling implementations to present a familiar wall-clock value to human consumers of the proof.

A profile that combined this mechanism with an [RFC3161] TSA timestamp can present both a consensus-verified block-height binding and a CA-verified wall-clock timestamp. The two are complementary; neither subsumes the other. See Section 4.4 for the COSE-level binding pattern.

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

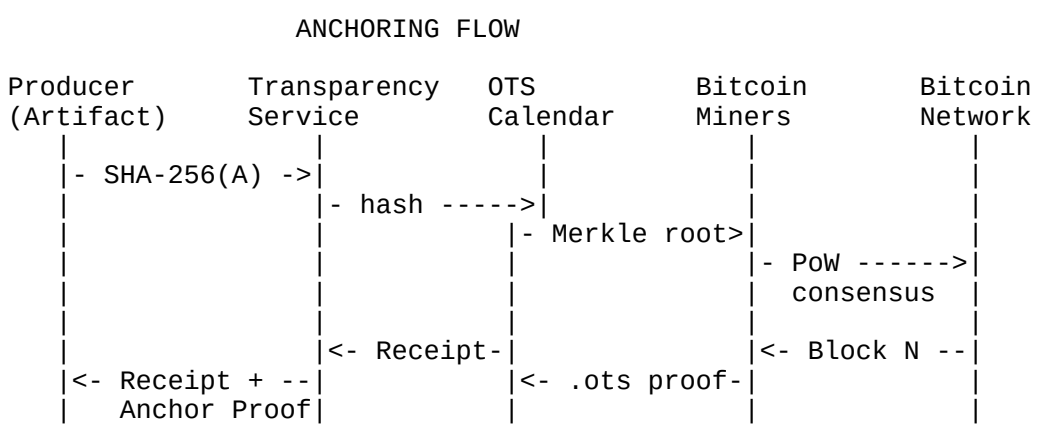


Figure 2: Actor-Level Anchoring Flow

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

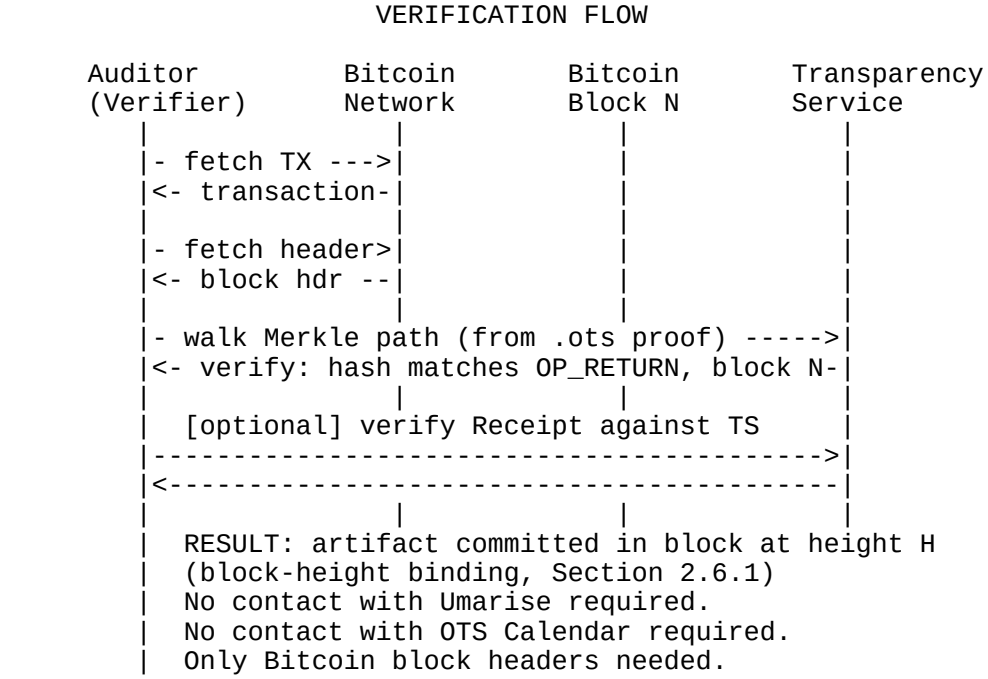


Figure 3: Actor-Level Verification Flow

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 (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 BLOCK-HEIGHT BINDING AND REFERENCE WALL-CLOCK
      PROJECTION
      block_header   ← FETCH-BLOCK-HEADER(
          bitcoin_tx.block_hash)
      block_height_H ← OTS-EXTRACT-BLOCK-HEIGHT(parsed)
      // The block_height_H is the normative temporal binding
      // (Section 2.6.1). It is exact and consensus-verified.

      block_time_RWCP ← block_header.nTime
      // block_time_RWCP is the Reference Wall-Clock Projection
      // (Section 1.3). It is informative, not normative, and
      // is subject to the bounded uncertainty described in
      // Section 2.6.6. Verifiers SHOULD present block_time_RWCP
      // alongside block_height_H for human readability.

      // OPTIONAL: a verifier MAY additionally compute the
      // Median Time Past per [BIP113] as a strict-lower-bound
      // wall-clock interpretation of the binding. This step
      // is OPTIONAL and is not required for conformance.
      //   prev_headers ← FETCH-PREV-HEADERS(block_header, 11)
      //   mtp_optional ← MEDIAN(prev_headers.nTime)

  8.  RETURN valid
      // The proof demonstrates: these exact bytes were
      // committed to the Bitcoin blockchain in the block at
      // height block_height_H, and therefore existed before
      // that block was mined. The Reference Wall-Clock
      // Projection block_time_RWCP SHOULD be presented for human
      // readability. No authorship, ownership, or identity
      // claim is made. See Section 2.6 (Temporal Precision)
      // for full semantics.
```

Figure 4: 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 block-height binding (Section 2.6.1) is established: the artifact was committed in the block at height H, and therefore existed before that block was mined.
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 this profile.

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 $x \neq y$ 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 (Block-Inclusion Causality)*: A transaction included in a Bitcoin block was necessarily broadcast to the network before that block was mined. By the standard Bitcoin protocol, a miner cannot include a transaction in a block unless the transaction was already in the miner's mempool or otherwise available at block-construction time. Therefore, the OP_RETURN payload of any transaction in the block at height H -- and hence the artifact hash committed to that payload -- existed before block H was mined. This assumption follows from Bitcoin protocol mechanics and does not depend on the specific value of any timestamp field in the block header. (For verifiers seeking a conservative wall-clock interpretation, the Median Time Past per [BIP113] provides a strict lower bound on block.time(H); see Section 2.6.1.)

* *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.

1. *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 an Anchor Proof for which VERIFY-ANCHOR(ArtifactBytes, ProofBundle) = 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 the block at height H and TX.OP_RETURN = V. By Assumption A2, the inclusion of TX in the block at height H cannot be retroactively altered.

Step 4: By Assumption A3 (Block-Inclusion Causality), TX was necessarily broadcast to the Bitcoin network before the block at height H was mined. Therefore the OP_RETURN payload V -- which commits to h, which commits to A -- existed before the block at height H was mined.

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

Therefore: A existed → h was computed → TX was broadcast → block at height H was mined. The artifact A existed before the block at height H was added to the canonical chain (the block-height binding of Section 2.6.1).

This argument does not depend on the block's nTime field, on the Median Time Past, or on any wall-clock interpretation of the binding. The Reference Wall-Clock Projection block.time(H) (Section 1.3) is informative only and is not required for the security argument above.

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 granularity of Bitcoin's block-mining process: the block-height binding identifies a block but does not resolve when within the block's mining interval the binding became fixed

The argument does NOT prove:

* When exactly A was created (only that A existed before the block at height H was mined)

* Who created A (no identity binding)

* That A has not been modified since anchoring (only that these specific bytes existed)

* Any wall-clock interpretation of the binding (the Reference Wall-Clock Projection is informative only; see Section 2.6.6)

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 this profile.

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

A claimant could attempt to present an Anchor Proof with a temporal interpretation broader than what the proof establishes. Two variants are possible under this profile:

Variant 1: claiming an earlier creation time. A claimant could assert that the proof establishes the artifact's creation at a specific moment, rather than its existence before the block at height H was mined.

This is remediated by the explicit semantics defined in Section 2.6.1: the normative claim is the block-height binding ("A was committed to the Bitcoin blockchain in the block at height H"), which implies "A existed before block H was mined." The proof does not establish a creation time. Section 2.6.4 (Non-Claims) makes this exclusion explicit. Verifiers MUST NOT interpret the binding as a creation-time assertion.

Variant 2: misrepresenting the Reference Wall-Clock Projection as normative. A claimant could present block.time(H) (the Reference Wall-Clock Projection, Section 1.3) as if it were the normative temporal value of the proof, ignoring that block.time is miner-set within consensus bounds and may diverge from true wall-clock time by up to approximately two hours (Section 2.6.6).

This is remediated by the explicit two-layer model in Section 2.6.6: the block-height binding is normative and exact; the Reference Wall-Clock Projection is informative. Implementations that present the projection to humans SHOULD clearly indicate that the value is a block-derived projection, not a verified wall-clock timestamp. Verifiers that require strict-lower-bound wall-clock semantics MAY compute the Median Time Past per [BIP113] as documented in Section 2.6.1 ("Optional tighter bound").

The block-height binding itself is not subject to inflation: a block exists at a specific height or it does not, and that fact is determined by Bitcoin consensus.

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 block-height binding that is subsequently invalidated by a chain reorganization.

This is remediated by three 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. Third, Section 2.6.5 (Pending Proofs and Temporal Definition) establishes that a pending proof carries no normative temporal claim: a Verifier encountering a pending proof MUST return unverifiable (Section 3.1, step 4) rather than treating the calendar commitment as a provisional binding.

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

This profile 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. References

9.1. Normative References

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

[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>.

[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>.

[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>.

[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>.

9.2. Informative References

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

[BIP141] Lombrozo, E., Lau, J., and P. Wuille, "Segregated Witness (Consensus layer)", BIP 141, 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, LNCS vol 293, Springer, DOI 10.1007/3-540-48184-2_32, 1988, <https://doi.org/10.1007/3-540-48184-2_32>.

[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>.

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

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

[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>.

[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>.

[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>.

[RFC6920] Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keranen, A., and P. Hallam-Baker, "Naming Things with Hashes", RFC 6920, DOI 10.17487/RFC6920, April 2013, <https://www.rfc-editor.org/info/rfc6920>.

[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>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[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>.

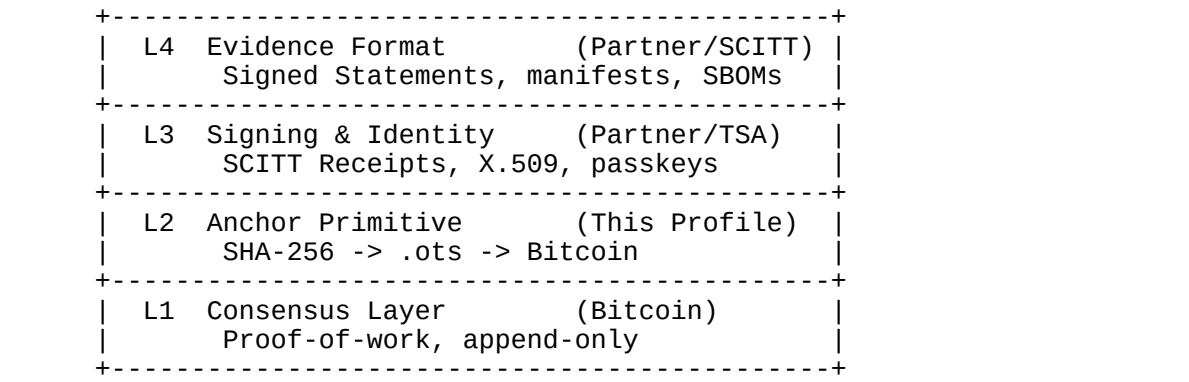
[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[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>.

[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>.

[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>.

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. 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].

D.1. Construction Algorithm (Anchor)

The construction algorithm CONSTRUCT-ANCHOR is defined as follows. Given an artifact byte sequence, it produces an origin record and a pending OpenTimestamps proof; subsequent upgrade to an anchored proof is specified in Appendix D.2.

```
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"}
```

Figure 5: CONSTRUCT-ANCHOR Algorithm

D.2. Upgrade Algorithm (Bitcoin Confirmation)

The upgrade algorithm UPGRADE-PENDING-PROOFS is defined as follows. It runs as a background worker, periodically converting calendar-level commitments produced by D.1 into Bitcoin-level proofs once the relevant Bitcoin transaction has been confirmed.

```
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
```

Figure 6: UPGRADE-PENDING-PROOFS Algorithm

D.3. Batch Anchoring with Merkle Trees

The batch anchoring algorithm BATCH-ANCHOR is defined as follows. It is a high-throughput variant of CONSTRUCT-ANCHOR (Appendix D.1) that anchors a single Merkle root for up to 1000 hashes in one Bitcoin commitment, instead of one commitment per hash. Each individual hash retains its own origin record; only the Merkle root is submitted to the OpenTimestamps Calendar Server(s).

```
Algorithm: BATCH-ANCHOR(hash_list)

// Anchors up to 1000 hashes via a single Merkle root.
// Each hash retains its own origin record (step 3); only
// the Merkle root is submitted to the OTS calendar(s) in
// step 4. This reduces N OTS submissions to 1.

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] § 2.1
    // Canonical leaf ordering: strip "sha256:" prefix, sort pairs
    // lexicographically before concatenation. After STRIP-PREFIX,
    // level elements are hex-encoded strings; concatenation in
    // step 2c below is hex-string concatenation, and SHA-256 is
    // applied to the resulting ASCII byte sequence. A second
    // implementer reading || as raw-byte concatenation of decoded
    // hash bytes would compute a different root and lose
    // interoperability with this profile.
    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]])
          // pair[0] and pair[1] are hex strings; || concatenates
          // them as ASCII; SHA-256 hashes the resulting bytes.
          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 ORIGIN RECORDS
    // Per-hash bookkeeping only; no OTS calendar submission
    // is performed at this step (the Merkle root is anchored
    // once in step 4 below).
    FOR EACH h IN hash_list:
      origin_id   ← UUID-v4()                        [RFC4122]
      short_token ← RANDOM-ALPHANUMERIC(8)
      captured_at ← NOW()                            // UTC ISO 8601
      INSERT origin_attestations {
        origin_id, hash: h, hash_algo: "sha256",
        captured_at, short_token
      }

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
```

Figure 7: BATCH-ANCHOR Algorithm

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.

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.

Appendix E. Test Vectors

This appendix provides concrete test vectors derived from a production anchoring run. Each vector can be independently verified using the VERIFY-ANCHOR algorithm (Section 3.1) and the verification tool at https://verify-anchoring.org.

The vectors are drawn from an AI model training pipeline (a Hugging Face Transformers fine-tuning run) in which each pipeline event was anchored to Bitcoin via the Umarise anchoring service. The pipeline anchored four events: on_train_begin, on_evaluate, on_train_end, and a post-run manifest. All four resulting Anchor Proofs are Bitcoin-confirmed.

For each vector the following values are provided:

* The artifact description and SHA-256 hash.

* The OpenTimestamps proof file name and size.

* The *block-height binding* -- the Bitcoin block at which the commitment is consensus-verified. This is the normative temporal value of the proof under this profile (Section 2.6.1).

* The *Reference Wall-Clock Projection* block.time(H) -- the nTime field of the Bitcoin block header, presented for human readability. This is informative only (Section 1.3, Section 2.6.6).

A verifier MAY additionally compute the Median Time Past per [BIP113] as a strict-lower-bound wall-clock interpretation of the binding (Section 2.6.1, "Optional tighter bound"). MTP is not required for conformance and is not reported in the vectors below.

E.1. Test Vector 1: on_train_begin

This vector represents the on_train_begin event from a Hugging Face training pipeline. The vector binds the serialised pipeline state at training start to a specific Bitcoin block.

```
   Artifact description:  Hugging Face training pipeline --
                          on_train_begin event (serialised
                          pipeline state at training start)
   SHA-256(artifact):     781bb71a88c82d1f009178d3e2a48fba
                          5023f52f510553ce74bf8d64db9985dd
   Anchor Proof file:     1777911147926_on_train_begin.ots
   Proof file size:       2086 bytes
   Anchor Ledger:         Bitcoin mainnet
   Block-height binding:  Bitcoin block at height 947124
                          (normative; Section 2.6.1)
   RWCP block.time(H):    2026-04-29T08:29:20Z
                          (Reference Wall-Clock Projection;
                          informative; Section 1.3)
   Expected VERIFY-ANCHOR result: valid

   Expected step-by-step execution of VERIFY-ANCHOR:

   Step 1: SHA-256(artifact) = 781bb71a...9985dd  [matches proof]
   Step 2: STRIP-PREFIX match:                    PASS
   Step 3: OTS-DESERIALIZE(.ots):                 valid OTS structure
   Step 4: Bitcoin attestation tag present:       0x0588960d73d71901
   Step 5: OTS-WALK-MERKLE-PATH produces
           OP_RETURN value matching Bitcoin
           transaction in block 947124:           PASS
   Step 6: FETCH-TX confirms block inclusion:     PASS
   Step 7: Block-height binding established:
             block_height_H = 947124
           Reference Wall-Clock Projection:
             block_time_RWCP = 2026-04-29T08:29:20Z
   Step 8: RETURN valid
```

Interpretation: The bytes of the on_train_begin event were committed to the Bitcoin block at height 947124. By Block-Inclusion Causality (Section 2.6.2, Assumption A3), the artifact existed before block 947124 was mined. The Reference Wall-Clock Projection 2026-04-29T08:29:20Z is presented as a human-readable approximation; it is not the normative temporal value of the proof.

E.2. Test Vector 2: on_evaluate

This vector represents the on_evaluate event captured during the training run. Together with E.1 and E.3, it provides a sequence of block-height bindings spanning the pipeline.

```
   Artifact description:  Hugging Face training pipeline --
                          on_evaluate event (serialised
                          evaluation state during training)
   SHA-256(artifact):     67b86bc99ad01edf7351610f57291d28
                          84c20309bf3355fbc4020b6426eaa6be
   Anchor Proof file:     on_evaluate.ots
   Proof file size:       2877 bytes
   Anchor Ledger:         Bitcoin mainnet
   Block-height binding:  Bitcoin block at height 947150
                          (normative; Section 2.6.1)
   RWCP block.time(H):    2026-04-29T12:26:30Z
                          (Reference Wall-Clock Projection;
                          informative; Section 1.3)
   Expected VERIFY-ANCHOR result: valid
```

Interpretation: The bytes of the on_evaluate event were committed to the Bitcoin block at height 947150. The artifact existed before that block was mined.

E.3. Test Vector 3: on_train_end

```
   Artifact description:  Hugging Face training pipeline --
                          on_train_end event (serialised
                          pipeline state at training end)
   SHA-256(artifact):     644f84d34a4dd97a6ce3e8dfd9c73adb
                          7ba2d97bd98d92adbeb38d2e3780e70d
   Anchor Proof file:     1777911147927_on_train_end.ots
   Proof file size:       2877 bytes
   Anchor Ledger:         Bitcoin mainnet
   Block-height binding:  Bitcoin block at height 947150
                          (normative; Section 2.6.1)
   RWCP block.time(H):    2026-04-29T12:26:30Z
                          (Reference Wall-Clock Projection;
                          informative; Section 1.3)
   Expected VERIFY-ANCHOR result: valid
```

Interpretation: The bytes of the on_train_end event were committed to the Bitcoin block at height 947150. The artifact existed before that block was mined.

E.4. Test Vector 4: post_run_manifest

```
Artifact description:  Hugging Face training pipeline --
                       post-run manifest (serialised record
                       of pipeline outputs and hashes)
SHA-256(artifact):     367a1cf0f5e5c80dceea6564c440b7a4
                       99e273e5673be8833524bfc867ec7583
Anchor Proof file:     1777911147927_post_run_manifest.ots
Proof file size:       2877 bytes
Anchor Ledger:         Bitcoin mainnet
Block-height binding:  Bitcoin block at height 947150
                       (normative; Section 2.6.1)
RWCP block.time(H):    2026-04-29T12:26:30Z
                       (Reference Wall-Clock Projection;
                       informative; Section 1.3)
Expected VERIFY-ANCHOR result: valid
```

Interpretation: The bytes of the post-run manifest were committed to the Bitcoin block at height 947150. The artifact existed before that block was mined.

Note on E.2, E.3, E.4: these three artifacts share the same block-height binding (block 947150). This is consistent with batched anchoring (Appendix D.3): multiple artifact hashes can be committed via a single Merkle root in one Bitcoin transaction. Each artifact retains its own .ots proof; the proofs differ in their Merkle path while sharing the same OP_RETURN commitment.

E.5. Negative Test Vector

A conformant implementation MUST return invalid if the artifact bytes do not match the anchored hash. Substituting any byte in the artifact while presenting the original .ots proof MUST cause VERIFY-ANCHOR to return invalid at step 2 (hash mismatch).

```
Modified artifact:    [any single byte change to the
                       original artifact]
Original proof:       any .ots proof from E.1-E.4
Expected result:      invalid (step 2: hash mismatch)
```

This vector demonstrates the binding between artifact bytes and Anchor Proof: a proof valid for one byte sequence is necessarily invalid for any other byte sequence.

E.6. Independent Verification

All test vectors can be independently verified without contacting the Umarise anchoring service:

1. Obtain the .ots proof file from the IETF document repository or the Umarise reference implementation repository.

2. Navigate to https://verify-anchoring.org.

3. Select "Hash + OTS" tab.

4. Enter the SHA-256 hash and upload the .ots file.

5. Click "Verify against Bitcoin."

The verifier contacts Bitcoin block explorers directly and performs no server-side computation on behalf of Umarise. This demonstrates the verification independence property defined in Section 3.2.

The verifier UI presents the result with the block height as the primary (normative) value and block.time(H) as the Reference Wall-Clock Projection, consistent with the two-layer model defined in Section 2.6.6.

All four positive vectors (E.1-E.4) have been independently verified using https://verify-anchoring.org and return "VALID, LEDGER-CONFIRMED" with the block heights and block.time(H) values shown above.

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
Netherlands

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