Independent Submission
Internet-Draft
Intended status: Experimental
Expires: 11 August 2026

T. Kamimura
VeritasChain Co., Ltd.

7 February 2026

Content Provenance Profile (CPP) Core
draft-vso-cpp-core-02

Abstract

The Content Provenance Profile (CPP) is an open specification for cryptographically verifiable media capture provenance. This document defines the core data model, hashing conventions, Merkle tree construction rules, RFC 3161 Time-Stamp Authority (TSA) anchoring protocol, and offline verification procedures for CPP.

CPP enables capture devices to produce tamper-evident provenance records that bind media content to external timestamps via trusted third parties. Unlike self-attestation models, CPP requires independent timestamp verification through RFC 3161 TSA services, providing externally verifiable proof of when media was captured.

This revision (-01) incorporates implementation experience from multi-platform deployments, adding self-attested signer identity, hardware-backed key requirements, chain context for partial submission detection, depth analysis extensions for screen detection, and a Pre-Publish Verification Extension for social media sharing workflows. It also defines interoperability mappings with the C2PA specification.

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 11 August 2026.

Table of Contents

1. Introduction

1.1. Problem Statement

Digital media authenticity faces several fundamental challenges:

* _Self-Attestation Weakness_: Systems where creators sign their own claims provide no independent verification. A verifier must trust that the creator's claimed timestamp is accurate.

* _Metadata Stripping_: Social media platforms and messaging applications routinely strip embedded metadata, breaking provenance chains that depend on file-level embedding.

* _Omission Attacks_: Systems that prove individual media authenticity cannot detect when unfavorable evidence has been selectively deleted from a collection.

* _Terminology Confusion_: Terms like "verified" mislead users into believing content truthfulness has been established, when only provenance has been recorded.

* _Analog Hole Attacks_: Photographing screens displaying manipulated content can bypass digital provenance systems that only track file-level operations.

1.2. Design Goals

CPP addresses these challenges through the following design principles:

1. _External Timestamp Verification_: Timestamps MUST be anchored to independent RFC 3161 Time-Stamp Authorities, enabling third-party verification without trusting the capture device.

2. _Omission Detection_: The Completeness Invariant mechanism enables detection of deleted events within a collection.

3. _Offline Verification_: All data necessary for verification is included in the Evidence Pack, enabling verification without network access.

4. _Provenance != Truth_: CPP proves when and by what device media was captured. It does NOT prove content truthfulness or scene authenticity.

5. _Hardware-Backed Security_: Private keys SHOULD be stored in hardware security modules where available (Secure Enclave, StrongBox, TPM).

1.3. Scope

This document specifies:

* The CPP event data model

* Hash computation and canonicalization rules

* Merkle tree construction and proof verification

* RFC 3161 TSA anchoring requirements

* Verification procedures for implementers

* Self-attested signer identity (SignerInfo)

* Chain context for partial submission detection

* Depth analysis extension for screen detection (OPTIONAL)

* Pre-Publish Verification extension (OPTIONAL)

This document does NOT specify:

* Application user interface requirements

* Network protocols for proof distribution

* Key management or certificate policies

* Content authenticity claims

1.4. Changes from draft-vso-cpp-core-01

This revision incorporates the following changes:

BREAKING CHANGE — Merkle Tree Domain Separation: This document mandates domain-separated Merkle hashing (Section 4.6.1). LeafHash MUST be computed as SHA256(0x00 || EventHash_bytes) and internal nodes as SHA256(0x01 || Left || Right). Earlier CPP specification documents (v1.0 through v1.4) used non-domain-separated hashing (LeafHash = SHA256(EventHash_bytes), Node = SHA256(Left || Right)). Those constructions are now deprecated. Implementations using the legacy (non-prefixed) construction MUST migrate to the domain-separated construction defined in this document. Test vectors in Appendix B reflect the domain-separated construction and differ from pre-domain-separation outputs.

BREAKING CHANGE — AnchorDigest Verification: Verifiers MUST now perform explicit format and digest matching checks on AnchorDigest, MerkleRoot, and TSA messageImprint fields as specified in Section 7.5. These checks were RECOMMENDED in -01 but are now REQUIRED.

* Added Section 7.5 with mandatory format checks, TSA binding verification, and PROHIBITED implementation patterns

* Added Normative Authority and Legacy Migration guidance to Section 1.6

* Strengthened Domain Separation (Section 4.6.1) with MUST/MUST NOT/DEPRECATED language

* Added cross-reference from TSA Verification (Section 7.4) to AnchorDigest verification requirements

* Clarified test vector applicability for legacy migration (Appendix B)

* Corrected author contact email address

1.5. Changes from draft-vso-cpp-core-00

The -01 revision incorporated the following changes:

* Added SignerInfo for self-attested identity (Section 4.2)

* Added DeviceInfo structure for cross-platform support (Section 4.3)
* Added CaptureContext for environmental metadata (Section 4.4)
* Added Chain Context for partial submission detection (Section 4.8)
* Added Depth Analysis Extension for screen detection (Section 8)
* Added Pre-Publish Verification Extension (Section 9)
* Added C2PA interoperability mappings (Section 10)
* Clarified hardware-backed key storage requirements
* Added EXPORT and additional event types
* Expanded implementation experience with Android and cross-platform validation

1.6. Relationship to Other Specifications

CPP defines its own Merkle tree construction that is NOT compatible with Certificate Transparency [RFC6962]. While inspired by similar principles, CPP uses different domain separation prefixes and padding rules optimized for media provenance use cases. Implementations MUST NOT assume RFC 6962 compatibility.

CPP is complementary to the C2PA specification [C2PA]. C2PA tracks edit history of content; CPP proves capture provenance with deletion detection. See Section 10 for interoperability mappings.

Normative Authority: Where the cryptographic algorithms defined in this document (domain-separated Merkle hashing, AnchorDigest computation, verification procedures) conflict with earlier CPP specification documents (v1.0 through v1.5), this document takes precedence. The CPP specification series published by VSO serves as the design-level reference; this Internet-Draft is the normative interoperability specification for implementations seeking cross-platform compatibility.

Legacy Migration: Implementations using non-domain-separated Merkle hashing (LeafHash = SHA256(EventHash_bytes) without the 0x00 prefix) MUST migrate to the domain-separated construction defined in Section 4.6.1. During a transition period, implementations MAY accept both legacy and domain-separated proofs for verification but MUST generate only domain-separated proofs. Implementations SHOULD log a deprecation warning when encountering legacy proofs.

2. Conventions and Definitions

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.

Additionally, this document uses the following terms:

Event A discrete, signed record representing a provenance action (capture, export, deletion).

EventHash A SHA-256 hash of the canonicalized event data, formatted as sha256:<64_hex_chars>.

LeafHash SHA-256(0x00 || EventHash_bytes), used as input to the Merkle tree. The 0x00 prefix provides domain separation from internal nodes.

MerkleRoot The root hash of a binary Merkle tree containing one or more LeafHashes.

AnchorDigest The 32-byte value submitted to the TSA. Represented as a 64-character lowercase hexadecimal string without prefix. MUST equal the MerkleRoot value (after stripping the sha256: prefix).

TreeSize The count of original leaves in the Merkle tree before any padding. An unsigned integer. MUST be >= 1.

PaddedSize The count of leaves after padding to a power of 2. Computed as the smallest power of 2 greater than or equal to TreeSize.

Evidence Pack A self-contained data structure containing all information necessary for offline verification.

Tombstone An event that records the legitimate deletion of a previous event.

Provenance Available The recommended terminology for UI display, indicating capture provenance is recorded without implying content truth verification.

Genesis PrevHash The constant value used as PrevHash for the first event in a chain: sha256:0000000000000000000000000000000000000000000000000000000000000000 (64 zeros).

SignerInfo An OPTIONAL self-attested identity claim included in events. Not independently verified; provides tamper-evident name association.

Chain Context Metadata embedded in proofs describing the event's position within its chain, enabling detection of partial evidence submission.

DeviceClass A classification of the capture device: SMARTPHONE, TABLET, EMBEDDED, PHYSICAL_CAMERA, DRONE, or INDUSTRIAL.

3. Threat Model

3.1. Addressed Threats

Threat	Mitigation
Timestamp forgery	RFC 3161 TSA provides independent timestamp
Evidence tampering	EventHash binds content; Merkle proof binds to anchor
Selective deletion	Completeness Invariant detects missing events
TSA token swapping	messageImprint must match AnchorDigest
Partial evidence submission	Chain Context reveals event position and deletion counts
Screen photography (analog hole)	Depth Analysis Extension detects flat surfaces (OPTIONAL)

Table 1

3.2. Explicitly Not Addressed

* Content truthfulness (scene staging, deepfakes)

* Device compromise before capture

* Key extraction from secure hardware

* TSA collusion with adversary

* Identity verification (SignerInfo is self-attested only)

4. Data Model

4.1. Events

An Event is the fundamental unit of provenance in CPP. Events are signed records that capture discrete provenance actions.

4.1.1. Event Types

Type	Description
INGEST	Media captured from device sensor
SEAL	Collection sealed with Completeness Invariant
EXPORT	Proof shared externally
TOMBSTONE	Legitimate deletion record

Table 2

4.1.2. Event Structure

The following fields are REQUIRED for all events:

Field	Type	Required	Description
EventID	string	REQUIRED	Unique identifier (UUID per [RFC9562])
ChainID	string	REQUIRED	Identifier linking events in a sequence
PrevHash	string	REQUIRED	Hash of previous event in chain
Timestamp	string	REQUIRED	ISO 8601 timestamp with millisecond precision
EventType	string	REQUIRED	One of: INGEST, SEAL, EXPORT, TOMBSTONE
HashAlgo	string	REQUIRED	Always "SHA256"
SignAlgo	string	REQUIRED	"ES256" or "Ed25519"
EventHash	string	REQUIRED	SHA-256 hash of canonicalized event
Signature	string	REQUIRED	Raw Base64-encoded signature (no prefix)
SignerInfo	object	OPTIONAL	Self-attested identity claim
DeviceInfo	object	OPTIONAL	Device metadata
CaptureContext	object	OPTIONAL	Environmental capture metadata

Table 3

4.1.3. Encoding Requirements

All Base64-encoded fields (Signature, TSA.Token, public_key) MUST conform to:

* [RFC4648] Section 4 (standard base64 alphabet, NOT base64url)

* No whitespace characters (no line breaks, spaces, or tabs)

* Padding characters (=) MUST be included when required

PROHIBITED:

* base64:MEUCIQDx... - Prefixes not allowed

* data:application/octet-stream;base64,... - Data URIs not allowed

* Base64url alphabet (- and _ instead of + and /)

* Line wrapping or embedded whitespace

4.1.4. INGEST Event

INGEST events MUST include:

Field	Type	Required	Description
Asset.AssetHash	string	REQUIRED	SHA-256 hash of media bytes
Asset.AssetType	string	REQUIRED	IMAGE or VIDEO
Asset.MimeType	string	REQUIRED	MIME type of asset
Asset.AssetID	string	OPTIONAL	Unique asset identifier
Asset.AssetName	string	OPTIONAL	Original filename
Asset.AssetSize	integer	OPTIONAL	File size in bytes

Table 4

4.1.5. SEAL Event

SEAL events finalize a collection and commit the Completeness Invariant. A SEAL event MUST include:

Field	Type	Required	Description
CollectionID	string	REQUIRED	Identifier for the sealed collection
EventCount	integer	REQUIRED	Number of events in collection (excluding SEAL)
CompletenessInvariant	object	REQUIRED	Completeness verification data
MerkleRoot	string	REQUIRED	Root hash of events in collection

Table 5

4.1.5.1. CompletenessInvariant Object

Field	Type	Required	Description
ExpectedCount	integer	REQUIRED	Number of events that MUST be present
HashSum	string	REQUIRED	XOR of all EventHash values (sha256: format)
FirstTimestamp	string	REQUIRED	ISO 8601 timestamp of first event
LastTimestamp	string	REQUIRED	ISO 8601 timestamp of last event

Table 6

The HashSum is computed as:

HashSum = EventHash[0] XOR EventHash[1] XOR ... XOR EventHash[n-1]

Where XOR operates on the 32-byte binary values of each EventHash.

4.1.5.2. SEAL Anchoring Pattern

The recommended anchoring pattern for SEAL events:

1. Compute the Merkle tree over all INGEST events in the collection
2. Create the SEAL event with MerkleRoot referencing this tree
3. Include the SEAL event's EventHash as an additional leaf in a NEW Merkle tree
4. Anchor this new tree (containing the SEAL event) to a TSA

This pattern ensures the Completeness Invariant itself is bound to an external timestamp. The SEAL event's EventHash covers all CI fields, so the TSA anchor proves the CI existed at GenTime.

Alternative Pattern (NOT RECOMMENDED): Including the SEAL event in the same Merkle tree it references creates a circular dependency and is prohibited.

4.1.6. TOMBSTONE Event

TOMBSTONE events MUST additionally include:

Field	Type	Required	Description
DeletedEventId	string	REQUIRED	EventID being invalidated
Reason	string	REQUIRED	Deletion reason code
DeletedAt	string	REQUIRED	ISO 8601 deletion timestamp

Table 7

4.2. SignerInfo

SignerInfo provides self-attested identity claims. It is OPTIONAL and MUST NOT be interpreted as verified identity.

Field	Type	Required	Description
Name	string	REQUIRED	Self-attested display name
Identifier	string	OPTIONAL	Self-attested identifier (email, URL)
AttestedAt	string	REQUIRED	ISO 8601 timestamp of attestation

Table 8

4.2.1. Verification Semantics

When a third party verifies a CPP proof with SignerInfo:

What CPP Proves	What CPP Does NOT Prove
Someone claimed this name at capture time	The name is real or legal
The claim is tamper-evident (signed)	Identity verification occurred
The claim existed before TSA timestamp	The person is who they claim

Table 9

Implementations displaying SignerInfo MUST use terminology such as "Self-Attested Name" and MUST NOT use "Verified Identity" or similar phrases that imply independent verification.

4.3. DeviceInfo

DeviceInfo provides metadata about the capture device. It is OPTIONAL but RECOMMENDED for INGEST events.

Field	Type	Required	Description
Manufacturer	string	OPTIONAL	Device manufacturer
Model	string	OPTIONAL	Device model identifier
DeviceClass	string	OPTIONAL	One of: SMARTPHONE, TABLET, EMBEDDED, PHYSICAL_CAMERA, DRONE, INDUSTRIAL
OSName	string	OPTIONAL	Operating system name
OSVersion	string	OPTIONAL	Operating system version
AppVersion	string	OPTIONAL	Capture application version

Table 10

4.4. CaptureContext

CaptureContext provides environmental metadata at capture time. It is OPTIONAL for INGEST events.

Field	Type	Required	Description
SensorData	object	OPTIONAL	Sensor readings (GPS, accelerometer)
HumanAttestation	object	OPTIONAL	Biometric verification record (boolean result only)
DepthAnalysis	object	OPTIONAL	Screen detection results (see Section 8)

Table 11

4.5. Hash Chain

Events form a hash chain through the PrevHash field:

```
Event 1: PrevHash = sha256:0000...0000 (genesis - 64 zeros)
Event 2: PrevHash = EventHash(Event 1)
Event 3: PrevHash = EventHash(Event 2)
```

Verification of chain integrity:

1. For each event after the first, the verifier MUST compute EventHash of the previous event.

2. The computed hash MUST match the current event's PrevHash field.

3. If any mismatch is detected, verification MUST fail with status CHAIN_INTEGRITY_VIOLATION.

4.6. Merkle Tree Structure

CPP defines its own binary Merkle tree construction optimized for media provenance. This construction uses domain separation prefixes to prevent attacks where leaf values could be confused with internal node values.

Important: CPP Merkle trees are NOT compatible with RFC 6962 (Certificate Transparency). Implementations MUST use the exact algorithms specified in this section.

4.6.1. Domain Separation

CPP MUST use single-byte prefixes to separate leaf and internal node domains. This construction prevents second preimage attacks where an attacker substitutes a leaf value for an internal node or vice versa.

Implementations MUST apply these prefixes. The legacy (non-prefixed) construction where LeafHash = SHA256(EventHash_bytes) and Node = SHA256(Left || Right) is DEPRECATED and MUST NOT be used for generating new proofs.

Domain	Prefix Byte	Description
Leaf	0x00	Applied to EventHash bytes
Internal	0x01	Applied to concatenated child hashes

Table 12

4.6.2. Leaf Nodes

```
LeafHash = SHA256(0x00 || EventHash_bytes)
```

Where:

* 0x00 is a single byte with value zero
* EventHash_bytes is the 32-byte binary representation of EventHash (after stripping the sha256: prefix)
* || denotes byte concatenation

4.6.3. Internal Nodes

```
InternalHash = SHA256(0x01 || Left_bytes || Right_bytes)
```

Where:

* 0x01 is a single byte with value one
* Left_bytes is the 32-byte hash of the left child
* Right_bytes is the 32-byte hash of the right child
* || denotes byte concatenation

4.6.4. Tree Construction

Step 1: Compute Leaf Hashes

For each event, compute LeafHash = SHA256(0x00 || EventHash_bytes).

Step 2: Determine Padding

PaddedSize is the smallest power of 2 >= TreeSize:

```
function computePaddedSize(treeSize):
    if treeSize == 0:
        return 0  // Invalid - TreeSize MUST be >= 1
    paddedSize = 1
    while paddedSize < treeSize:
        paddedSize = paddedSize * 2
    return paddedSize
```

Step 3: Pad Leaf Array

If TreeSize < PaddedSize, duplicate the last leaf hash until the array length equals PaddedSize.

Step 4: Build Tree

```
function buildTree(paddedLeaves):
    levels = [paddedLeaves]
    current = paddedLeaves

    while current.length > 1:
        nextLevel = []
        for i in range(0, current.length, 2):
            left = current[i]
            right = current[i + 1]
            parent = SHA256(0x01 || left || right)
            nextLevel.append(parent)
        levels.append(nextLevel)
        current = nextLevel

    return levels  // levels[0] = leaves, levels[-1] = [root]
```

4.6.5. Merkle Proof Structure

Field	Type	Description
TreeSize	integer	Original leaf count (before padding), unsigned, MUST be >= 1
LeafHashMethod	string	MUST be exactly SHA256(0x00||EventHash) (18 ASCII characters)
LeafHash	string	Computed LeafHash for this event with sha256: prefix
LeafIndex	integer	0-based position in tree, range [0, TreeSize-1]
Proof	array	Sibling hashes from bottom to top, each with sha256: prefix
Root	string	MerkleRoot with sha256: prefix

Table 13

4.7. Anchor Structure

Field	Type	Description
AnchorID	string	Unique anchor identifier
AnchorType	string	MUST be "RFC3161"
AnchorDigest	string	MerkleRoot without prefix, 64 lowercase hex chars
AnchorDigestAlgorithm	string	MUST be "sha-256"
Merkle	object	Merkle proof structure
TSA	object	TSA response data

Table 14

The TSA object MUST include:

Field	Type	Description
Token	string	Complete DER-encoded TimeStampToken, Base64
MessageImprint	object	Extracted messageImprint from TSTInfo
GenTime	string	Extracted GenTime from TSTInfo, ISO 8601
Service	string	TSA service URL (informational)

Table 15

The MessageImprint object MUST include:

Field	Type	Description
HashAlgorithm	string	MUST be "sha-256"
HashedMessage	string	64 lowercase hex chars, MUST equal AnchorDigest

Table 16

4.8. Chain Context

Chain Context is OPTIONAL metadata embedded in Evidence Packs to enable partial submission detection. It describes the event's position within its chain without requiring the full chain for initial assessment.

Field	Type	Description
ChainID	string	Unique chain identifier
TotalEvents	integer	Total events in chain (including Tombstones)
ActiveEvents	integer	Non-invalidated events
TombstoneCount	integer	Number of deleted events
EventPosition	integer	Position of this event (1-indexed)
CompletenessInvariant	object	XOR-based completeness data for the chain
GeneratedAt	string	ISO 8601 timestamp of context generation

Table 17

Chain Context is informational when embedded in a single proof. Full completeness verification requires the complete chain (Forensic Export). Verifiers SHOULD use Chain Context to alert users when:

* TombstoneCount > 0 (events have been deleted)

* ActiveEvents < TotalEvents (some events invalidated)

* Only a subset of the chain is presented

5. Canonicalization and Hashing

5.1. JSON Canonicalization

Events MUST be canonicalized using [RFC8785] (JSON Canonicalization Scheme) before hashing.

The following fields MUST be excluded from canonicalization:

* EventHash

* Signature

All other fields, including SignerInfo if present, MUST be included. Field names in the canonical event object use PascalCase (e.g., EventID, ChainID, PrevHash).

5.2. EventHash Computation

```
function computeEventHash(event):
    eventCopy = copy(event)
    delete eventCopy.EventHash
    delete eventCopy.Signature
    canonical = JCS_canonicalize(eventCopy)  // RFC 8785
    hashBytes = SHA256(canonical)
    return "sha256:" + lowercase_hex(hashBytes)
```

The resulting EventHash is a 71-character string: the prefix "sha256:" followed by 64 lowercase hexadecimal characters.

Note: SignerInfo, DeviceInfo, and CaptureContext are included in the EventHash computation when present. This ensures these fields are tamper-evident and bound to the TSA timestamp.

5.3. LeafHash Computation

```
function computeLeafHash(eventHash):
    hexStr = eventHash.substring(7)  // Remove "sha256:" prefix
    eventHashBytes = hexDecode(hexStr)  // 32 bytes
    prefixedData = [0x00] + eventHashBytes  // 33 bytes
    leafHashBytes = SHA256(prefixedData)
    return "sha256:" + lowercase_hex(leafHashBytes)
```

5.4. Internal Node Hash Computation

```
function computeInternalHash(left, right):
    leftBytes = hexDecode(left.substring(7))
    rightBytes = hexDecode(right.substring(7))
    prefixedData = [0x01] + leftBytes + rightBytes  // 65 bytes
    hashBytes = SHA256(prefixedData)
    return "sha256:" + lowercase_hex(hashBytes)
```

5.5. AnchorDigest Computation

AnchorDigest is the MerkleRoot value WITHOUT the sha256: prefix, represented as 64 lowercase hexadecimal characters.

```
function computeAnchorDigest(merkleRoot):
    return lowercase(merkleRoot.substring(7))
```

PROHIBITED:

* SHA256(merkleRoot) - This would create double hashing
* SHA256(stringEncode(merkleRoot)) - This would hash the string representation
* Any transformation other than prefix removal
* Mixed case output - MUST be lowercase

6. Anchoring Protocol

6.1. TSA Request Construction

The messageImprint in TimeStampReq [RFC3161] MUST contain:

* hashAlgorithm: SHA-256 (OID 2.16.840.1.101.3.4.2.1)
* hashedMessage: AnchorDigest as 32-byte OCTET STRING

```
TimeStampReq ::= SEQUENCE {
   version         INTEGER { v1(1) },
   messageImprint  MessageImprint,
   reqPolicy       OBJECT IDENTIFIER OPTIONAL,
   nonce           INTEGER OPTIONAL,
   certReq         BOOLEAN DEFAULT FALSE,
   extensions      [0] IMPLICIT Extensions OPTIONAL
}

MessageImprint ::= SEQUENCE {
   hashAlgorithm   AlgorithmIdentifier,  -- SHA-256
   hashedMessage   OCTET STRING          -- AnchorDigest (32 bytes)
}
```

6.1.1. certReq Recommendation

Producers SHOULD set certReq to TRUE to request the TSA's signing certificate be included in the response. This enables:

* Offline verification without fetching certificates separately

* Long-term verification even if TSA infrastructure changes

* Self-contained Evidence Packs

If certReq is FALSE and the TSA certificate is not included in the response, verifiers MUST attempt to obtain the certificate through other means (e.g., AIA extension, local cache) or return VALID_WARNING.

6.2. TSA Response Processing

Upon receiving TimeStampResp, the producer:

1. MUST verify the response status is granted (0) or grantedWithMods (1)

2. MUST extract the TimeStampToken from the response

3. MUST store the complete DER-encoded TimeStampToken

4. MUST extract and store the messageImprint from TSTInfo

5. MUST extract and store GenTime from TSTInfo

6.3. Single-Leaf Tree Rules

When TreeSize equals 1, the following invariants MUST hold:

* LeafIndex MUST equal 0
* Proof MUST be an empty array
* Root MUST equal LeafHash
* LeafHash MUST equal SHA256(0x00 || EventHash_bytes)

If any of these conditions fail, verification MUST return INVALID.

6.4. Multi-Leaf Tree Rules

For TreeSize greater than 1:

* LeafIndex MUST be in range [0, TreeSize-1]
* PaddedSize = smallest power of 2 >= TreeSize
* Proof length MUST NOT exceed log2(PaddedSize)
* Sibling hashes are ordered from bottom (leaf level) to top (root level)
* Index parity determines pairing order: even=left, odd=right
* All internal nodes use SHA256(0x01 || left || right)

7. Verification Procedures

7.1. Verification Result Codes

Code	Meaning
VALID	All checks passed, including TSA signature verification
VALID_WARNING	Cryptographic checks passed, but TSA certificate chain could not be fully validated
INVALID	Cryptographic verification failed

CHAIN_INTEGRITY_VIOLATION	Hash chain is broken
COMPLETENESS_VIOLATION	Completeness Invariant mismatch

Table 18

7.2. Event Verification

```
function verifyEvent(event, publicKey):
    // Step 1: Recompute EventHash
    computedHash = computeEventHash(event)
    if computedHash != event.EventHash:
        return INVALID("EventHash mismatch")

    // Step 2: Verify signature
    hashBytes = hexDecode(event.EventHash.substring(7))
    sigBytes = base64Decode(event.Signature)
    if not verifySignature(publicKey, hashBytes, sigBytes):
        return INVALID("Signature verification failed")

    return VALID
```

7.3. Merkle Proof Verification

```
function verifyMerkleProof(eventHash, leafIndex, proof,
                           expectedRoot, treeSize):
    // Step 1: Validate inputs
    if treeSize < 1:
        return INVALID("TreeSize must be >= 1")
    if leafIndex < 0 or leafIndex >= treeSize:
        return INVALID("LeafIndex out of range")

    paddedSize = computePaddedSize(treeSize)
    maxProofLength = log2(paddedSize)
    if proof.length > maxProofLength:
        return INVALID("Proof too long")

    // Step 2: Compute leaf hash with domain separation
    currentHash = computeLeafHash(eventHash)

    // Step 3: Handle single-leaf case
    if treeSize == 1:
        if leafIndex != 0:
            return INVALID("LeafIndex must be 0 for single-leaf")
        if proof.length != 0:
            return INVALID("Proof must be empty for single-leaf")
        if lowercase(currentHash) != lowercase(expectedRoot):
            return INVALID("Root != LeafHash for single-leaf")
        return VALID

    // Step 4: Traverse proof from bottom to top
    index = leafIndex
    for siblingHash in proof:
        if index % 2 == 0:
            currentHash = computeInternalHash(currentHash, siblingHash)
        else:
            currentHash = computeInternalHash(siblingHash, currentHash)
        index = floor(index / 2)

    // Step 5: Compare with expected root
    if lowercase(currentHash) != lowercase(expectedRoot):
        return INVALID("Computed root != expected root")

    return VALID
```

7.4. TSA Verification

TSA verification ensures the timestamp token was legitimately issued by a Time-Stamp Authority and binds the correct digest. Implementations MUST also perform the format and binding checks specified in Section 7.5 prior to or as part of this procedure.

```
   function verifyTSAAnchor(eventHash, anchor):
       // Step 1: Verify Merkle structure
       merkle = anchor.Merkle
       result = verifyMerkleProof(
           eventHash, merkle.LeafIndex, merkle.Proof,
           merkle.Root, merkle.TreeSize)
       if result != VALID:
           return result

       // Step 2: Verify LeafHashMethod
       if merkle.LeafHashMethod != "SHA256(0x00||EventHash)":
           return INVALID("Unsupported LeafHashMethod")

       // Step 3: Verify AnchorDigest == MerkleRoot
       expectedDigest = lowercase(merkle.Root.substring(7))
       if lowercase(anchor.AnchorDigest) != expectedDigest:
           return INVALID("AnchorDigest != MerkleRoot")

       // Step 4: Parse TSA Token (RFC 5652 ContentInfo)
       tsaToken = base64Decode(anchor.TSA.Token)
       contentInfo = parseContentInfo(tsaToken)
       signedData = parseSignedData(contentInfo.content)
       tstInfo = parseTSTInfo(signedData.encapContentInfo.eContent)

       // Step 5: Verify hash algorithm is SHA-256
       if tstInfo.messageImprint.hashAlgorithm != SHA256_OID:
           return INVALID("Unsupported TSA hash algorithm")

       // Step 6: Verify messageImprint == AnchorDigest
       tstImprint = lowercase_hex(tstInfo.messageImprint.hashedMessage)
       if tstImprint != lowercase(anchor.AnchorDigest):
           return INVALID("TSA messageImprint != AnchorDigest")

       // Step 7: Verify CMS signature over TSTInfo
       signerInfo = signedData.signerInfos[0]
       signatureValid = verifyCMSSignature(
           signedData.encapContentInfo.eContent,
           signerInfo.signature,
           signerInfo.signatureAlgorithm,
           extractSignerCert(signedData.certificates, signerInfo.sid))
       if not signatureValid:
           return INVALID("TSA signature verification failed")

       // Step 8: Verify certificate chain (SHOULD)
       certValid = verifyCertificateChain(
           signedData.certificates, signerInfo.sid, trustAnchors)

       if certValid:
```

```
            return VALID(genTime = tstInfo.genTime)
        else:
            return VALID_WARNING(genTime = tstInfo.genTime,
                warning = "TSA certificate chain could not be verified")
```

7.4.1. CMS Signature Verification Requirements

Per [RFC5652], verifiers MUST:

1. Parse the TimeStampToken as a ContentInfo structure
2. Extract the SignedData from the content field
3. Locate the SignerInfo corresponding to the TSA
4. Verify the signature over the encapsulated TSTInfo
5. Verify the signer’s certificate was valid at signing time

Verifiers SHOULD:

1. Build and validate the certificate chain to a trust anchor
2. Verify the TSA certificate contains the id-kp-timeStamping extended key usage
3. Check for certificate revocation

7.5. AnchorDigest Verification Requirements

This section consolidates the mandatory checks for AnchorDigest, MerkleRoot, and TSA messageImprint consistency. These checks are REQUIRED and failure of any single check MUST result in INVALID.

Format Requirements:

1. MerkleRoot MUST be a 71-character string consisting of the prefix sha256: followed by exactly 64 lowercase hexadecimal characters (regex: ^sha256:[0-9a-f]{64}$). Verifiers MUST reject MerkleRoot values containing uppercase characters, missing prefix, or incorrect length.
2. AnchorDigest MUST be exactly 64 lowercase hexadecimal characters (regex: ^[0-9a-f]{64}$) representing the 32-byte binary digest. No prefix, no whitespace, no uppercase.

3. AnchorDigest MUST exactly equal MerkleRoot with the sha256: prefix stripped. No rehashing, no encoding transformation — purely prefix removal.

TSA Binding Requirements:

1. Verifiers MUST extract messageImprint from TSTInfo within the TimeStampToken.

2. Verifiers MUST confirm that messageImprint.hashAlgorithm equals the SHA-256 OID (2.16.840.1.101.3.4.2.1). If the OID differs, verification MUST return INVALID.

3. Verifiers MUST confirm that the lowercase hexadecimal encoding of messageImprint.hashedMessage (32 bytes) is identical to AnchorDigest. This is a byte-for-byte comparison after hex encoding; case-insensitive comparison is NOT sufficient because AnchorDigest is already required to be lowercase.

4. If an implementation transmits AnchorDigest to the TSA as a UTF-8 string rather than a 32-byte binary OCTET STRING, the TSA will timestamp a 64-byte value (the hex encoding) rather than the correct 32-byte digest. This is a known implementation error. Verifiers MUST detect this by comparing the messageImprint length: if hashedMessage is not exactly 32 bytes, verification MUST return INVALID.

PROHIBITED Patterns (known implementation errors):

* Submitting the hex string of AnchorDigest (64 ASCII bytes) instead of the decoded binary (32 bytes) to the TSA — results in messageImprint mismatch

* Computing SHA256(MerkleRoot) as the TSA input — results in double-hashing

* Computing SHA256(AnchorDigest_as_string) — results in hashing the hex encoding rather than the binary value

* Using the MerkleRoot string (with sha256: prefix) directly as the TSA hashedMessage — results in a 71-byte input instead of 32 bytes

```
   function verifyAnchorDigestBinding(anchor):
       // Format checks
       if not matches(anchor.Merkle.Root, /^sha256:[0-9a-f]{64}$/):
           return INVALID("MerkleRoot format violation")

       if not matches(anchor.AnchorDigest, /^[0-9a-f]{64}$/):
           return INVALID("AnchorDigest format violation")

       // Prefix-strip equivalence
       expectedDigest = anchor.Merkle.Root.substring(7)
       if anchor.AnchorDigest != expectedDigest:
           return INVALID("AnchorDigest != MerkleRoot hex part")

       // TSA binding
       tstInfo = extractTSTInfo(anchor.TSA.Token)
       if tstInfo.messageImprint.hashAlgorithm != SHA256_OID:
           return INVALID("TSA hash algorithm is not SHA-256")

       imprintBytes = tstInfo.messageImprint.hashedMessage
       if length(imprintBytes) != 32:
           return INVALID("TSA hashedMessage is not 32 bytes"
                          + " (string-as-binary error?)")

       imprintHex = lowercase_hex(imprintBytes)
       if imprintHex != anchor.AnchorDigest:
           return INVALID("TSA messageImprint != AnchorDigest")

       return VALID
```

7.6. Chain Integrity Verification

```
   GENESIS_PREV_HASH = "sha256:0000000000000000000000000000000000" +
                       "0000000000000000000000000000000000"

   function verifyChainIntegrity(events):
       if events.length == 0:
           return VALID

       if events[0].PrevHash != GENESIS_PREV_HASH:
           return CHAIN_INTEGRITY_VIOLATION("Invalid genesis PrevHash")

       for i in range(1, events.length):
           expectedPrevHash = events[i-1].EventHash
           if events[i].PrevHash != expectedPrevHash:
               return CHAIN_INTEGRITY_VIOLATION(
                   "Break at event " + i)

       return VALID
```

7.7. Completeness Invariant Verification

```
   function verifyCompleteness(events, sealEvent):
       ci = sealEvent.CompletenessInvariant

       // Step 1: Verify count
       if events.length != ci.ExpectedCount:
           return COMPLETENESS_VIOLATION("Count mismatch")

       // Step 2: Compute XOR hash sum
       computed = bytes(32)
       for event in events:
           eventHashBytes = hexDecode(event.EventHash.substring(7))
           computed = XOR(computed, eventHashBytes)

       // Step 3: Compare with sealed value
       expectedHashSum = hexDecode(ci.HashSum.substring(7))
       if computed != expectedHashSum:
           return COMPLETENESS_VIOLATION("Hash sum mismatch")

       // Step 4: Verify timestamp bounds
       for event in events:
           if event.Timestamp < ci.FirstTimestamp:
               return COMPLETENESS_VIOLATION("Before collection start")
           if event.Timestamp > ci.LastTimestamp:
               return COMPLETENESS_VIOLATION("After collection end")

       return VALID
```

7.7.1. Attack Detection

Attack	Detection
Delete event	Hash sum mismatch and/or count mismatch
Add fake event	Count mismatch and/or hash sum mismatch
Reorder events	Chain integrity violation (PrevHash mismatch)
Modify event	EventHash mismatch in chain

Table 19

8. Depth Analysis Extension

The Depth Analysis Extension is OPTIONAL and provides screen detection capabilities to mitigate analog hole attacks (photographing screens displaying manipulated content).

8.1. DepthAnalysis Structure

When present in CaptureContext, DepthAnalysis MUST include:

Field	Type	Required	Description
SensorType	string	REQUIRED	Type of depth sensor used
FlatnessScore	number	REQUIRED	0.0 (natural scene) to 1.0 (flat screen)
DepthVariance	number	REQUIRED	Statistical variance of depth values
ScreenDetected	boolean	REQUIRED	Whether a screen was detected
Confidence	number	REQUIRED	Detection confidence 0.0 to 1.0
AnalysisVersion	string	REQUIRED	Version of analysis algorithm

Table 20

8.2. Sensor Types

Value	Description
LIDAR	LiDAR time-of-flight sensor
STRUCTURED_LIGHT	Structured light projection
STEREO	Stereo camera pair
TOF	Non-LiDAR time-of-flight
RADAR	Millimeter-wave radar

ULTRASONIC	Ultrasonic depth sensing
MONOCULAR_ESTIMATED	AI-estimated depth from single camera
MULTI_CAMERA	Multi-camera triangulation
ACTIVE_IR	Active infrared projection
HYBRID	Multiple sensor fusion
UNKNOWN	Sensor type not determined
NONE	No depth sensor available

Table 21

8.3. Depth Analysis Verification

Depth analysis data is included in the EventHash computation, making it tamper-evident. Verifiers MAY use DepthAnalysis to assess capture environment but MUST NOT treat it as definitive proof of scene authenticity. Depth sensors can be spoofed by sophisticated adversaries.

When ScreenDetected is true, implementations SHOULD display a warning to users but MUST NOT automatically reject the proof.

9. Pre-Publish Verification Extension

The Pre-Publish Verification Extension is OPTIONAL and enables verification of CPP provenance at the moment of social media sharing. It allows users to indicate their content has traceable origin without blocking the sharing flow or making truth claims.

9.1. Design Principles

* _Silent Failure_: If verification fails or times out, sharing MUST proceed without any user-visible error or delay.

* _No Truth Claims_: The indicator communicates provenance availability, not content truthfulness.

* _Performance Budget_: Verification MUST complete within 200ms or short-circuit to silent passthrough.

9.2. VerificationResult

Status	Behavior
PROVENANCE_AVAILABLE	Show indicator, attach metadata
PROVENANCE_PARTIAL	Silent passthrough (no indicator)
PROVENANCE_UNAVAILABLE	Silent passthrough
VERIFICATION_TIMEOUT	Silent passthrough
VERIFICATION_ERROR	Silent passthrough

Table 22

Only PROVENANCE_AVAILABLE results in visible indication. All other statuses MUST result in silent passthrough where the original content is shared without modification or delay.

9.3. Prohibited Terminology

Implementations MUST NOT use the following terms in user-facing displays related to CPP provenance:

* "Verified" or "Verified Content"
* "Authentic" or "Authenticated"
* "True" or "Truthful"
* "Certified"
* "Guaranteed"
* "Real"
* "Trustworthy"

Recommended terminology: "Provenance Available", "Capture Provenance Recorded", "Origin Traceable".

10. C2PA Interoperability

Implementations MAY support both CPP and C2PA [C2PA] manifests. This section defines field mappings for dual-standard implementations.

10.1. Field Mapping

CPP Field	C2PA Equivalent	Notes
DeviceInfo.Manufacturer	claim_generator	Partial mapping
DeviceInfo.Model	claim_generator	Partial mapping
Timestamp	dc:created	ISO 8601 format
SensorData.GPS	Exif:GPS*	Standard EXIF mapping
Anchor.TSA	c2pa.time_stamp	RFC 3161 compatible
DepthAnalysis	No equivalent	CPP-specific extension
HumanAttestation	No equivalent	CPP-specific extension
CompletenessInvariant	No equivalent	CPP-specific extension

Table 23

10.2. Dual-Standard Output

Implementations generating both CPP and C2PA manifests MUST ensure shared fields (Timestamp, GPS, DeviceInfo) are consistent across both manifests. CPP-specific fields (DepthAnalysis, HumanAttestation, CompletenessInvariant) have no C2PA equivalent and are carried only in the CPP manifest.

11. Privacy Considerations

11.1. Location Data

Location collection SHOULD be disabled by default. When enabled, implementations SHOULD:

* Clearly indicate when location is being recorded

* Allow users to delete location from individual events

* Consider privacy implications of location precision

* Support approximate location modes that reduce GPS precision

11.2. Biometric Data

Implementations MUST NOT store raw biometric data (fingerprints, face images). Human presence verification, if implemented, SHOULD:

* Process biometrics locally on-device

* Store only verification results (boolean flags)

* Never transmit biometric data to external services

11.3. Tombstone Privacy

When events are deleted via TOMBSTONE:

* Original event content is removed

* TOMBSTONE preserves chain integrity

* Reason codes allow selective disclosure

11.4. Shareable vs Forensic Proofs

Level	Includes	Use Case
Shareable	Timestamp, device info, asset hash	Social sharing
Forensic	All metadata including location, chain context	Legal proceedings

Table 24

12. Security Considerations

12.1. Hash Algorithm Agility

This specification mandates SHA-256 for all hash computations. Future versions MAY define additional algorithms via the HashAlgo field. Verifiers MUST reject unknown hash algorithms.

Post-quantum consideration: ML-DSA-65 (NIST Module-Lattice Digital Signature Algorithm) is RESERVED for future adoption. XOR-based accumulators used in the Completeness Invariant, being purely symmetric operations, face fewer quantum vulnerabilities than public-key constructions.

12.2. Signature Algorithm Requirements

Implementations MUST support ES256 (ECDSA with P-256 and SHA-256, per [FIPS186-5]) for mobile device compatibility. Ed25519 MAY be supported for non-mobile implementations.

Private keys SHOULD be stored in hardware security modules where available:

* iOS: Secure Enclave (kSecAttrTokenIDSecureEnclave)

* Android: StrongBox or TEE via Android Keystore

* Desktop/Server: TPM 2.0 or HSM

Hardware-backed keys provide non-exportability guarantees that strengthen the binding between device and signature.

12.3. TSA Trust

Security of timestamp proofs depends on TSA trustworthiness. Implementations:

* MUST verify the CMS signature in the TimeStampToken per [RFC5652]

* SHOULD validate the certificate chain to a configured trust anchor

* SHOULD use TSAs with published certificate policies

* MAY support multiple TSA services for redundancy

12.4. Merkle Tree Security

The 0x00/0x01 prefix bytes ensure leaf hashes cannot equal internal node hashes for any input. This prevents second preimage attacks on the tree structure. This construction differs from Certificate Transparency [RFC6962] which uses a similar but incompatible scheme.

12.5. Clock Accuracy

Device timestamps (Timestamp field) are self-attested and may be inaccurate. The authoritative timestamp is GenTime from the TSA response. Implementations SHOULD warn users when device time differs significantly from TSA GenTime (e.g., more than 5 minutes).

12.6. Deletion Detection Limitations

The Completeness Invariant detects deletions within a sealed collection. It does NOT detect:

* Events never created (adversary captured but never recorded)

* Events in other collections

* Deletions before sealing

XOR is commutative and self-inverse. An attacker who can forge TWO events with EventHashes that XOR to zero can delete both without detection. The CI is designed to work WITH the Merkle tree anchor, not replace it.

12.7. Depth Analysis Security

Depth sensors can be spoofed by sophisticated adversaries using 3D displays or structured light interference. Depth analysis provides an additional signal but is not a definitive screen detection mechanism. Implementations MUST NOT represent depth analysis as conclusive proof of scene authenticity.

12.8. Canonicalization Attacks

JSON canonicalization per [RFC8785] prevents ordering and whitespace attacks. Implementations MUST ensure field names exactly match the specification (PascalCase for events) and Unicode normalization is handled consistently.

13. IANA Considerations

This document has no IANA actions.

14. Implementation Experience

14.1. VeraSnap iOS (Non-Normative)

VeraSnap is a consumer iOS application implementing CPP, available in 175 countries with 10-language localization. It demonstrates:

* Secure Enclave key storage with ES256 signatures

* RFC 3161 TSA integration with multiple providers (failover)

* Merkle tree construction per this specification

* Offline proof verification

* LiDAR-based depth analysis for screen detection

* Cross-platform proof export and QR code sharing

14.2. VeraSnap Android (Non-Normative)

A Kotlin-based Android implementation validates cross-platform interoperability:

* Android Keystore (StrongBox/TEE) key storage with ES256

* RFC 8785 JSON canonicalization producing identical hashes to iOS

* Proof JSON verification across platforms

* QR code generation and scanning interoperability

Cross-platform testing confirmed that proofs generated on iOS verify correctly on Android and vice versa, validating the canonicalization and hashing specifications.

15. References

15.1. Normative References

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

[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>.

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

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/info/rfc5652>.

[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>.

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

15.2. Informative References

[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>.

[RFC9562] Davis, K., Peabody, B., and P. Leach, "Universally Unique IDentifiers (UUIDs)", RFC 9562, DOI 10.17487/RFC9562, May 2024, <https://www.rfc-editor.org/info/rfc9562>.

[C2PA] Coalition for Content Provenance and Authenticity, "C2PA Specification", 2024, <https://c2pa.org/specifications/>.

[FIPS186-5] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS 186-5, February 2023, <https://csrc.nist.gov/publications/detail/fips/186/5/final>.

Appendix A. JSON Examples

A.1. Canonical Event with SignerInfo (Normative)

The canonical event structure uses PascalCase field names. This is the structure that MUST be used for EventHash computation.

```
{
  "EventID": "550e8400-e29b-41d4-a716-446655440001",
  "ChainID": "urn:uuid:550e8400-e29b-41d4-a716-446655440000",
  "PrevHash": "sha256:0000000000000000000000000000000000000000000000000000000000000000",
  "Timestamp": "2026-01-27T10:30:00.000Z",
  "EventType": "INGEST",
  "HashAlgo": "SHA256",
  "SignAlgo": "ES256",
  "Asset": {
    "AssetID": "asset-001",
    "AssetType": "IMAGE",
    "AssetHash": "sha256:e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
",
    "AssetName": "IMG_0001.HEIC",
    "MimeType": "image/heic"
  },
  "SignerInfo": {
    "Name": "John Doe",
    "Identifier": null,
    "AttestedAt": "2026-01-27T10:29:55.000Z"
  },
  "DeviceInfo": {
    "Manufacturer": "Apple",
    "Model": "iPhone 15 Pro",
    "DeviceClass": "SMARTPHONE",
    "OSName": "iOS",
    "OSVersion": "17.3",
    "AppVersion": "1.5.34"
  },
  "EventHash": "sha256:7d865e959b2466918c9863afca942d0fb89d7c9ac0c99bafc3749504ded97730",
  "Signature": "MEUCIQDKsRwMv..."
}
```


A.2. Anchor Structure with MessageImprint (Normative)

```
{
  "Anchor": {
    "AnchorID": "anchor-001",
    "AnchorType": "RFC3161",
    "AnchorDigest": "719f871f1018a17ebe199d4f0db27e3a4929f8ab3e46f5c0d30054f4b331e929",
    "AnchorDigestAlgorithm": "sha-256",
    "Merkle": {
      "TreeSize": 1,
      "LeafHashMethod": "SHA256(0x00||EventHash)",
      "LeafHash": "sha256:719f871f1018a17ebe199d4f0db27e3a4929f8ab3e46f5c0d30054f4b331e92
9",
      "LeafIndex": 0,
      "Proof": [],
      "Root": "sha256:719f871f1018a17ebe199d4f0db27e3a4929f8ab3e46f5c0d30054f4b331e929"
    },
    "TSA": {
      "Token": "MIIEzAYJKoZIhvcNAQcCoIIEvTCCBLkCAQMx...",
      "MessageImprint": {
        "HashAlgorithm": "sha-256",
        "HashedMessage": "719f871f1018a17ebe199d4f0db27e3a4929f8ab3e46f5c0d30054f4b331e92
9"
      },
      "GenTime": "2026-01-27T10:31:00.000Z",
      "Service": "https://freetsa.org/tsr"
    }
  }
}
```

A.3. Chain Context Example (Non-Normative)

```
   {
     "ChainContext": {
       "ChainID": "urn:uuid:550e8400-e29b-41d4-a716-446655440000",
       "TotalEvents": 100,
       "ActiveEvents": 53,
       "TombstoneCount": 47,
       "EventPosition": 15,
       "CompletenessInvariant": {
         "ExpectedCount": 100,
         "HashSum": "sha256:a3f2c8d1e5b9...",
         "FirstTimestamp": "2026-01-27T10:30:00.000Z",
         "LastTimestamp": "2026-01-27T17:45:00.000Z"
       },
       "GeneratedAt": "2026-01-27T18:00:00.000Z"
     }
   }
```

A.4. Depth Analysis Example (Non-Normative)

```
   {
     "CaptureContext": {
       "DepthAnalysis": {
         "SensorType": "LIDAR",
         "FlatnessScore": 0.12,
         "DepthVariance": 0.847,
         "ScreenDetected": false,
         "Confidence": 0.95,
         "AnalysisVersion": "1.4.0"
       }
     }
   }
```

Appendix B. Test Vectors

All test vectors in this section use the domain-separated hash construction defined in this specification. These vectors are identical to those in draft-vso-cpp-core-00 and -01, which also used domain separation. Note: implementations migrating from legacy (non-domain-separated) CPP hashing will produce different LeafHash and MerkleRoot values for the same EventHash inputs. The domain-separated outputs below are the only correct values for compliant implementations.

B.1. Test Vector 1: Single-Leaf Tree

Input:

```
EventHash = "sha256:7d865e959b2466918c9863afca942d0fb89d7c9ac0c99bafc3749504ded97730"
```

Computation:

```
LeafHash = SHA256(0x00 || EventHash_bytes)
         = sha256:719f871f1018a17ebe199d4f0db27e3a4929f8ab3e46f5c0d30054f4b331e929

For TreeSize=1:
  Root = LeafHash
  LeafIndex = 0
  Proof = []
  AnchorDigest = 719f871f1018a17ebe199d4f0db27e3a4929f8ab3e46f5c0d30054f4b331e929

Result: VALID
```

B.2. Test Vector 2: Two-Leaf Tree

```
EventHash[0] = "sha256:aaaa...aaaa"  (32 bytes of 0xaa)
EventHash[1] = "sha256:bbbb...bbbb"  (32 bytes of 0xbb)

L0 = SHA256(0x00 || 0xaa...aa)
   = sha256:e0bb82791bae3c50bd9c20fa4ccdcb8064a56e5c12bc69b07e6712ac9b4429e6
L1 = SHA256(0x00 || 0xbb...bb)
   = sha256:4f16119d36ccd0da91102f57692d73934fd0ad2494280df88449accedbbfb7ea

Root = SHA256(0x01 || L0 || L1)
     = sha256:03938e2c8f758e6cae443d499b41c899c373eb0c0198bae61796a069f2b05904

For index 0: Proof = [L1]
For index 1: Proof = [L0]

Result: VALID
```

B.3. Test Vector 3: TSA messageImprint Verification

```
AnchorDigest = "719f871f1018a17ebe199d4f0db27e3a4929f8ab3e46f5c0d30054f4b331e929"

TSTInfo.messageImprint.hashAlgorithm = SHA-256
TSTInfo.messageImprint.hashedMessage = 0x719f871f...1e929

lowercase_hex(hashedMessage) == AnchorDigest ? YES

Result: VALID
```

Acknowledgements

The authors thank:

* The VeritasChain Standards Organization (VSO) for developing the CPP specification series (v1.0 through v1.5)

* The implementers of VeraSnap (iOS and Android) for cross-platform validation feedback that improved this specification

* Early reviewers who identified the domain separation and Completeness Invariant modeling issues

* Contributors to the depth analysis and pre-publish verification extensions

Author’s Address

Tokachi Kamimura
VeritasChain Co., Ltd.
Email: kamimura@veritaschain.org