PLANTS Working Group
Internet-Draft
Intended status: Standards Track
Expires: 26 July 2026

D. Benjamin
Google LLC
D. O'Brien

B. E. Westerbaan
L. Valenta
Cloudflare
F. Valsorda
Geomys

22 January 2026

Merkle Tree Certificates
draft-davidben-tls-merkle-tree-certs-10

Abstract

This document describes Merkle Tree certificates, a new form of X.509 certificates which integrate public logging of the certificate, in the style of Certificate Transparency. The integrated design reduces logging overhead in the face of both shorter-lived certificates and large post-quantum signature algorithms, while still achieving comparable security properties to traditional X.509 and Certificate Transparency. Merkle Tree certificates additionally admit an optional signatureless optimization, which decreases the message size by avoiding signatures altogether, at the cost of only applying to up-to-date relying parties and older certificates.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://davidben.github.io/merkle-tree-certs/draft-davidben-tls-merkle-tree-certs.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-davidben-tls-merkle-tree-certs/.

Discussion of this document takes place on the PLANTS Working Group mailing list (mailto:plants@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/plants/. Subscribe at https://www.ietf.org/mailman/listinfo/plants/.

Source for this draft and an issue tracker can be found at https://github.com/davidben/merkle-tree-certs.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Authors' Note: This is an early draft of a proposal with many parts. We expect most details will change as the proposal evolves. This document has a concrete specification of these details, but this is only intended as a starting point, and to help convey the overall idea. The name of the draft says "tls" to keep continuity with earlier iterations of this work, but the protocol itself is not TLS-specific.

In Public Key Infrastructures (PKIs) that use Certificate Transparency (CT) [RFC6962] for a public logging requirement, an authenticating party must present Signed Certificate Timestamps (SCTs) alongside certificates. CT policies often require two or more SCTs per certificate [APPLE-CT] [CHROME-CT], each of which carries a signature. These signatures are in addition to those in the certificate chain itself.

Current signature schemes can use as few as 32 bytes per key and 64 bytes per signature [RFC8032], but post-quantum replacements are much larger. For example, ML-DSA-44 [FIPS204] uses 1,312 bytes per public key and 2,420 bytes per signature. ML-DSA-65 uses 1,952 bytes per public key and 3,309 bytes per signature. Even with a directly-trusted intermediate (Section 7.5 of [I-D.ietf-tls-trust-anchor-ids]), two SCTs and a leaf certificate signature adds 7,260 bytes of authentication overhead with ML-DSA-44 and 9,927 bytes with ML-DSA-65.

This increased overhead additionally impacts CT logs themselves. Most of a log's costs scale with the total storage size of the log. Each log entry contains both a public key, and a signature from the CA. With larger public keys and signatures, the size of each log entry will grow.

Additionally, as PKIs transition to shorter-lived certificates [CABF-153] [CABF-SC081], the number of entries in the log will grow.

This document introduces Merkle Tree certificates, a new form of X.509 certificate that integrates logging with certificate issuance. Each CA maintains a log of everything it issues, signing views of the log to assert it has issued the contents. The CA signature is combined with cosignatures from other parties who verify correct operation and optionally mirror the log. These signatures, together with an inclusion proof for an individual entry, constitute a certificate.

This achieves the following:

* Log entries do not scale with public key and signature sizes. Entries replace public keys with hashes and do not contain signatures, while preserving non-repudiability (Section 12.4).

* To bound growth, long-expired entries can be pruned from logs and mirrors without interrupting existing clients. This allows log sizes to scale by retention policies, not the lifetime of the log, even as certificate lifetimes decrease.

* After a processing delay, authenticating parties can obtain a second "signatureless" certificate for the same log entry. This second certificate is an optional size optimization that avoids the need for any signatures, assuming an up-to-date client that has some predistributed log information.

Section 3 gives an overview of the system. Section 4 describes a Merkle Tree primitive used by this system. Section 5 describes the log structure. Finally, Section 6 and Section 7 describe how to construct and consume a Merkle Tree certificate.

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.

This document additionally uses the TLS presentation language defined in Section 3 of [RFC8446], as well as the notation defined in Section 2.1.1 of [RFC9162].

U+ followed by four hexadecimal characters denotes a Unicode codepoint, to be encoded in UTF-8 [RFC3629]. 0x followed by two hexadecimal characters denotes a byte value in the 0-255 range.

[start, end), where start <= end, denotes the half-open interval containing integers x such that start <= x < end.

Given a non-negative integer n,

* LSB(n) refers to the least-significant bit of n’s binary representation. Equivalently, it is the remainder when n is divided by 2.

* BIT_WIDTH(n) refers to the smallest number of bits needed to represent n. BIT_WIDTH(0) is zero.

* POPCOUNT(n) refers to the number of set bits in n’s binary representation.

* BIT_CEIL(n) refers to the smallest power of 2 that is greater or equal to n.

To _left-shift_ a non-negative integer n is to shift each bit in its binary representation to one upper position. Equivalently, it is n times 2. Given non-negative integers a and b, a << b refers to a left-shifted b times.

To _right-shift_ a non-negative integer n is to shift each bit in its binary representation to one lower position, discarding the least-significant bit. Equivalently, it is the floor of n divided by 2. Given non-negative integers a and b, a >> b refers to a right-shifted b times.

Given two non-negative integers a and b, a & b refers to the non-negative integer such that each bit position is set if the corresponding bit is set in both a and b, and unset otherwise. This is commonly referred to as the bitwise AND operator.

2.1. Terminology and Roles

This document discusses the following roles:

Authenticating party: The party that authenticates itself in the protocol. In TLS, this is the side sending the Certificate and CertificateVerify message.

Certification authority (CA): The service that issues certificates to the authenticating party, after performing some validation process on the certificate contents.

Relying party: The party to whom the authenticating party presents its identity. In TLS, this is the side receiving the Certificate and CertificateVerify message.

Monitor: Parties who watch logs for certificates of interest, analogous to the role in Section 8.2 of [RFC9162].

Issuance log: A log, maintained by the CA, of everything issued by that CA.

Cosigner: A service that signs views of an issuance log, to assert correct operation and other properties about the entries.

Additionally, there are several terms used throughout this document to describe this proposal. This section provides an overview. They will be further defined and discussed in detail throughout the document.

Checkpoint: A description of the complete state of the log at some time.

Entry: An individual element of the log, describing information which the CA has validated and certified.

Subtree: A smaller Merkle Tree over a portion of the log, defined by

an interior node of some snapshot of the log. Subtrees can be efficiently shown to be consistent with the whole log.

Inclusion proof: A sequence of hashes that efficiently proves some entry is contained in some checkpoint or subtree.

Consistency proof: A sequence of hashes that efficiently proves a checkpoint or subtree is contained within another checkpoint.

Cosignature: A signature from either the CA or other cosigner, over some checkpoint or subtree.

Landmark: One of an infrequent subset of tree sizes that can be used to predistribute trusted subtrees to relying parties for signatureless certificates.

Landmark subtree: A subtree determined by a landmark. Landmark subtrees are common points of reference between relying parties and signatureless certificates.

Full certificate: A certificate containing an inclusion proof to some subtree, and several cosignatures over that subtree.

Signatureless certificate: An optimized certificate containing an inclusion proof to a landmark subtree, and no signatures.

3. Overview

In Certificate Transparency, a CA first certifies information by signing it, then submits the resulting certificate (or precertificate) to logs for logging. Merkle Tree Certificates invert this process: the CA certifies information by logging it, then submits the log to cosigners to verify log operation. A certificate is assembled from the result and proves the information is in the CA’s log.

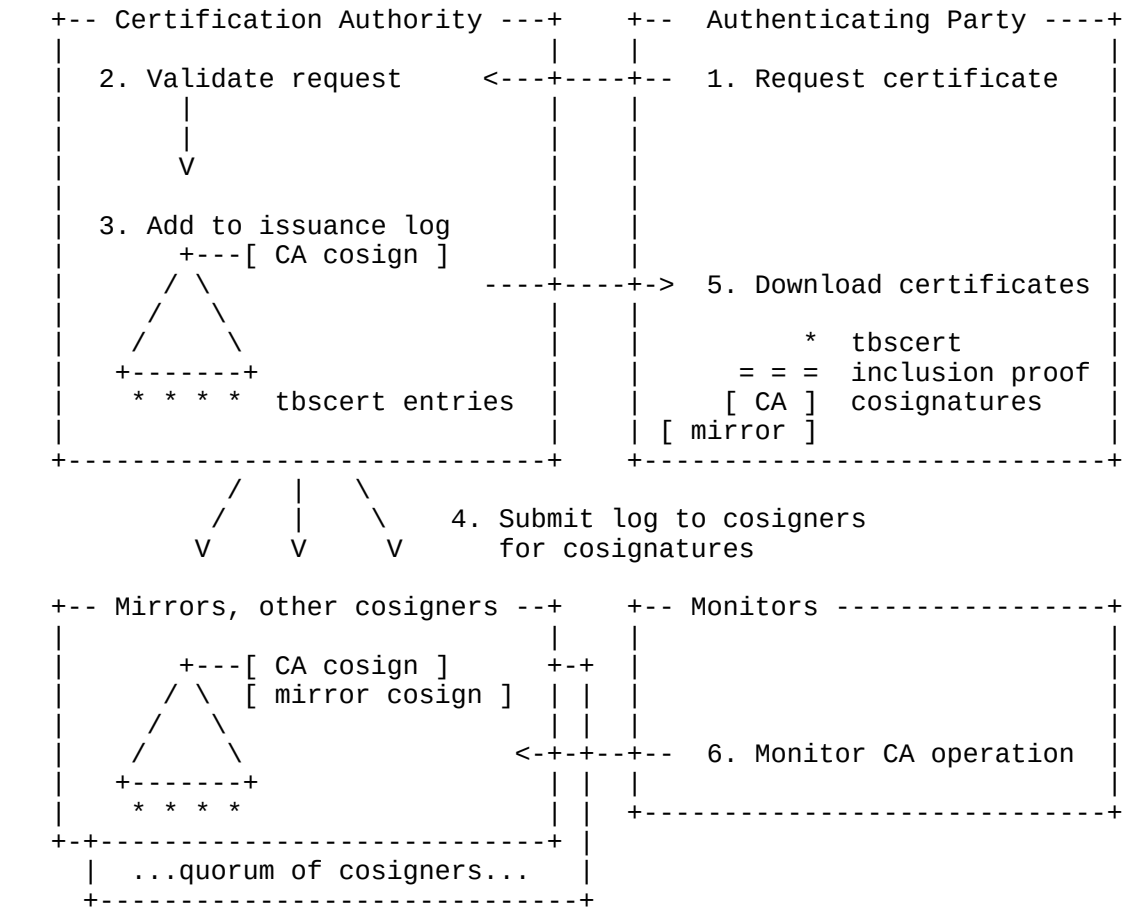


Figure 1: A diagram of the issuance architecture, detailed below

Merkle Tree Certificates are issued as follows. Figure 1 depicts this process.

1. The authenticating party requests a certificate, e.g. over ACME [RFC8555]

2. The CA validates each incoming issuance request, e.g. with ACME challenges. From there, the process differs.

3. The CA operates an append-only _issuance log_ (Section 5). Unlike a CT log, this issuance log only contains entries added by the CA:

1. The CA adds a TBSCertificateLogEntry (Section 5.3) to its log, describing the information it is certifying.

2. The CA signs a _checkpoint_, which describes the current state of the log. A signed checkpoint certifies that the CA issued _every_ entry in the Merkle Tree (Section 5.5).

3. The CA additionally signs _subtrees_ (Section 4) that together contain certificates added since the last checkpoint (Section 4.5). This is an optimization to reduce inclusion proof sizes. A signed subtree certifies that the CA has issued _every_ entry in the subtree.

4. The CA submits the new log state to _cosigners_. Cosigners validate the log is append-only and optionally provide additional services, such as mirroring its contents. They cosign the CA's checkpoints and subtrees.

5. The CA now has enough information to construct a certificate and give it to the authenticating party. A certificate contains:

   * The TBSCertificate being certified

   * An inclusion proof from the TBSCertificate to some subtree

   * Cosignatures from the CA and cosigners on the subtree

6. As in Certificate Transparency, monitors observe the issuance log to ensure the CA is operated correctly.

A certificate with cosignatures is known as a _full certificate_. Analogous to X.509 trust anchors and trusted CT logs, relying parties are configured with trusted cosigners (Section 7.3) that allow them to accept Merkle Tree certificates. The inclusion proof proves the TBSCertificate is part of some subtree, and cosignatures from trusted cosigners prove the subtree was certified by the CA and available to monitors. Where CT logs entire certificates, the issuance log's entries are smaller TBSCertificateLogEntry (Section 5.3) structures, which do not scale with public key or signature size.

This same issuance process also produces a _signatureless certificate_. This is an optional, optimized certificate that avoids all cosignatures, including the CA signature. Signatureless certificates are available after a short period of time and usable with up-to-date relying parties.

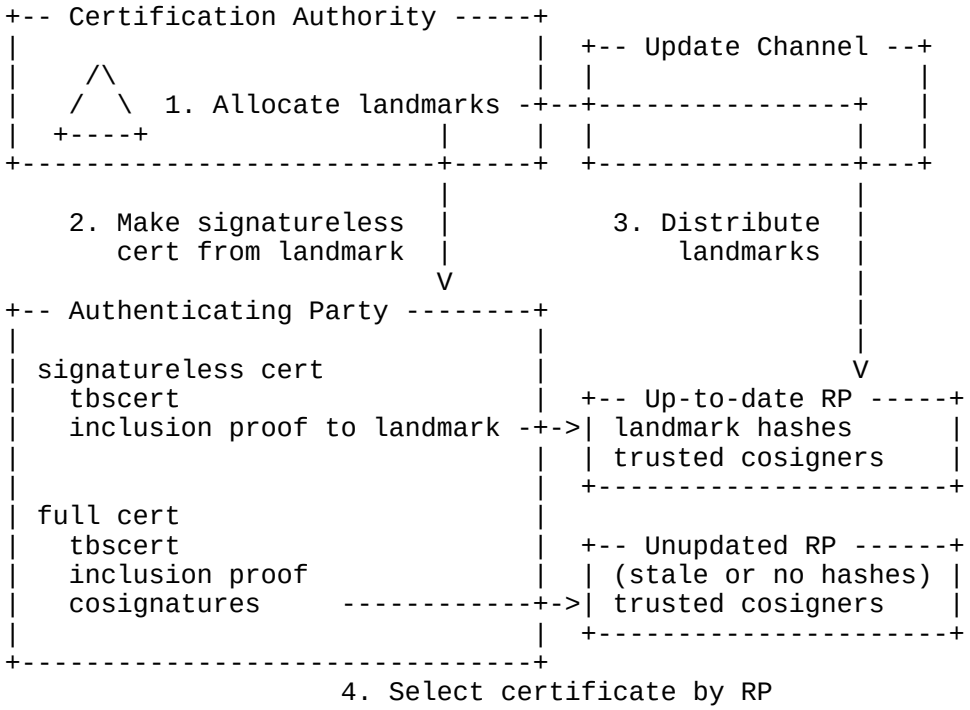


Figure 2: A diagram of signatureless certificate construction and usage, detailed below

Signatureless certificates are constructed and used as follows. Figure 2 depicts this process.

1. Periodically, the tree size of the CA's most recent checkpoint is designated as a _landmark_. This determines _landmark subtrees_, which are common points of reference between relying parties and signatureless certificates.

2. Once some landmark includes the TBSCertificate, the signatureless certificate is constructed with:

   * The TBSCertificate being certified

   * An inclusion proof from the TBSCertificate to a landmark subtree

3. In the background, landmark subtrees are predistributed to relying parties, with cosignatures checked against relying party requirements. This occurs periodically in the background, separate from the application protocol.

4. During the application protocol, such as TLS [RFC8446], if the relying party already supports the landmark subtree, the authenticating party can present the signatureless certificate. Otherwise, it presents a full certificate. The authenticating party may also select between several signatureless certificates, as described in Section 10.4.

4. Subtrees

This section extends the Merkle Tree definition in Section 2.1 of [RFC9162] by defining a _subtree_ of a Merkle Tree. A subtree is an interior node of a Merkle Tree, which can be efficiently shown consistent with the original Merkle Tree and any Merkle Tree with additional elements appended. This specification uses subtrees to reduce the size of inclusion proofs.

4.1. Definition of a Subtree

Given an ordered list of n inputs, D_n = {d[0], d[1], ..., d[n-1]}, Section 2.1.1 of [RFC9162] defines the Merkle Tree via the Merkle Tree Hash MTH(D_n).

A _subtree_ of this Merkle Tree is itself a Merkle Tree, defined by MTH(D[start:end]). start and end are integers such that:

* 0 <= start < end <= n

* start is a multiple of BIT_CEIL(end - start)

Note that, if start is zero, the second condition is always true.

In the context of a single Merkle Tree, the subtree defined by start and end is denoted by half-open interval [start, end). It contains the entries whose indices are in that half-open interval.

The _size_ of the subtree is end - start. If the subtree’s size is a power of two, it is said to be _full_, otherwise it is said to be _partial_.

If a subtree is full, then it is directly contained in the tree of hash operations in MTH(D_n) for n >= end.

If a subtree is partial, it is directly contained in MTH(D_n) only if n = end.

4.2. Example Subtrees

Figure 3 shows the subtrees [4, 8) and [8, 13):

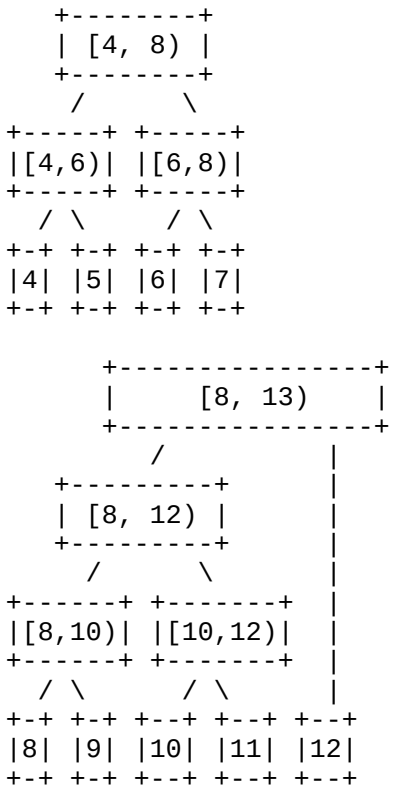


Figure 3: Two example subtrees, one full and one partial

Both subtrees are directly contained in a Merkle Tree of size 13, depicted in Figure 4. [4, 8) is contained because, although n (13) is not end (8), the subtree is full. [8, 13) is contained because n (13) is end (13).

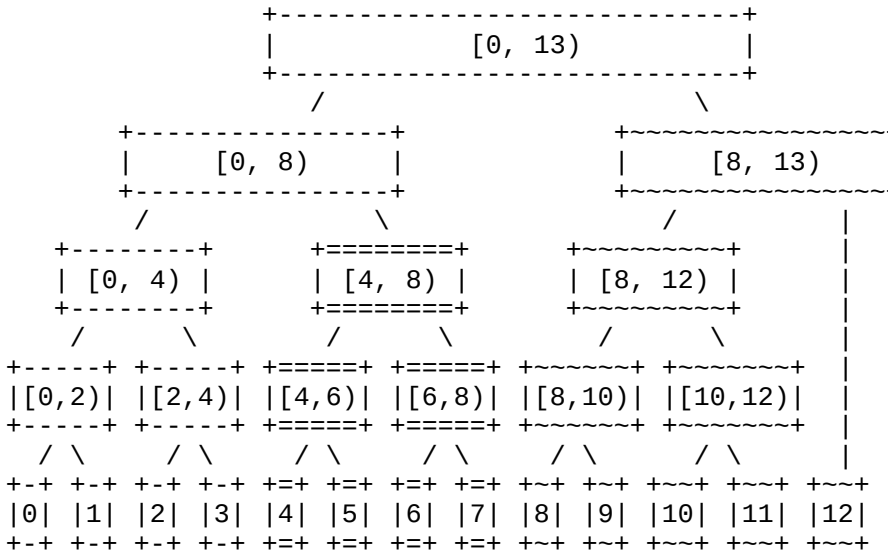


Figure 4: A Merkle Tree of size 13

In contrast, [8, 13) is not directly contained in a Merkle Tree of size 14, depicted in Figure 5. However, the subtree is still computed over consistent elements.

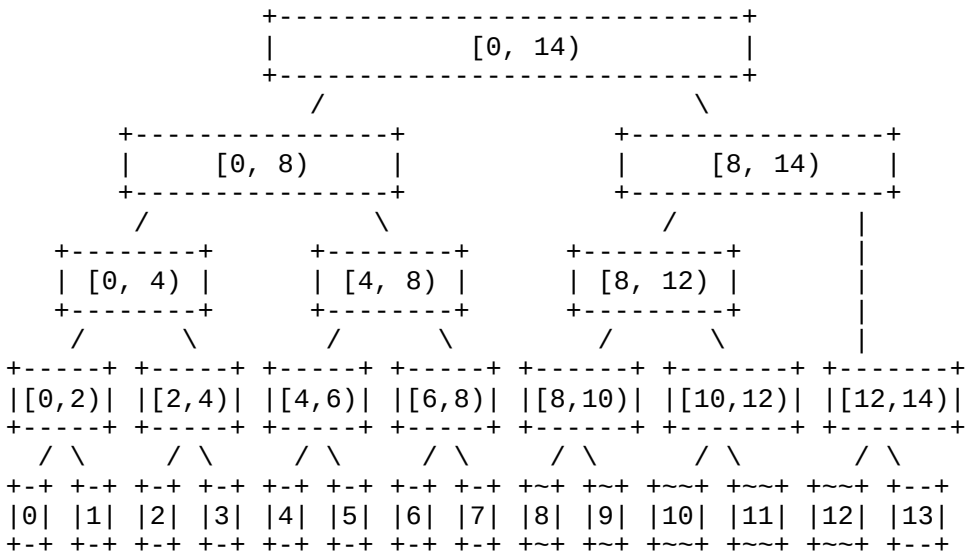


Figure 5: A Merkle Tree of size 14

4.3. Subtree Inclusion Proofs

Subtrees are Merkle Trees, so entries can be proven to be contained in the subtree. A subtree inclusion proof for entry index of the subtree [start, end) is a Merkle inclusion proof, as defined in Section 2.1.3.1 of [RFC9162], where m is index - start and the tree inputs are D[start:end].

Subtree inclusion proofs contain a sequence of nodes that are sufficient to reconstruct the subtree hash, MTH(D[start:end]), out of the hash for entry index, MTH({d[index]}), thus demonstrating that the subtree hash contains the entry’s hash.

4.3.1. Example Subtree Inclusion Proofs

The inclusion proof for entry 10 of subtree [8, 13) contains the hashes MTH({d[11]}), MTH(D[8:10]), and MTH({d[12]}), depicted in Figure 6. MTH({d[10]}) is not part of the proof because the verifier is assumed to already know its value.

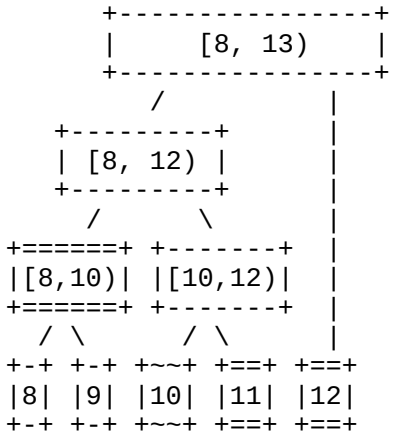


Figure 6: An example subtree inclusion proof

4.3.2. Evaluating a Subtree Inclusion Proof

Given a subtree inclusion proof, inclusion_proof, for entry index, with hash entry_hash, of a subtree [start, end), the subtree inclusion proof can be _evaluated_ to compute the expected subtree hash:

1. Check that [start, end) is a valid subtree (Section 4.1), and that start <= index < end. If either do not hold, fail proof evaluation.

2. Set fn to index - start and sn to end - start - 1.

3. Set r to entry_hash.

4. For each value p in the inclusion_proof array:

    1. If sn is 0, then stop the iteration and fail proof evaluation.

    2. If LSB(fn) is set, or if fn is equal to sn, then:

        1. Set r to HASH(0x01 || p || r).

        2. Until LSB(fn) is set, right-shift fn and sn equally.

        Otherwise:

        1. Set r to HASH(0x01 || r || p).

    3. Finally, right-shift both fn and sn one time.

5. If sn is not zero, fail proof evaluation.

6. Return r as the expected subtree hash.

This is the same as the procedure in Section 2.1.3.2 of [RFC9162], where leaf_index is index - start, tree_size is end - start, and r is returned instead of compared with root_hash.

Appendix B.2 explains this procedure in more detail.

4.3.3. Verifying a Subtree Inclusion Proof

Given a subtree inclusion proof, inclusion_proof, for entry index, with hash entry_hash, of a subtree [start, end) with hash subtree_hash, the subtree inclusion proof can be _verified_ to verify the described entry is contained in the subtree:

1. Let expected_subtree_hash be the result of evaluating the inclusion proof as described Section 4.3.2. If evaluation fails, fail the proof verification.

2. If subtree_hash is equal to expected_subtree_hash, the entry is contained in the subtree. Otherwise, fail the proof verification.

4.4. Subtree Consistency Proofs

A subtree [start, end) can be efficiently proven to be consistent with the full Merkle Tree. That is, given MTH(D[start:end]) and MTH(D_n), the proof demonstrates that the input D[start:end] to the subtree hash was equal to the corresponding elements of the input D_n to the Merkle Tree hash.

Subtree consistency proofs contain sufficient nodes to reconstruct both the subtree hash, MTH(D[start:end]), and the full tree hash, MTH(D_n), in such a way that every input to the subtree hash was also incorporated into the full tree hash.

4.4.1. Generating a Subtree Consistency Proof

The subtree consistency proof, SUBTREE_PROOF(start, end, D_n) is defined similarly to Section 2.1.4.1 of [RFC9162], in terms of a helper function that tracks whether the subtree hash is known:

```
SUBTREE_PROOF(start, end, D_n) =
    SUBTREE_SUBPROOF(start, end, D_n, true)
```

If start = 0 and end = n, the subtree is the root:

```
SUBTREE_SUBPROOF(0, n, D_n, true) = {}
SUBTREE_SUBPROOF(0, n, D_n, false) = {MTH(D_n)}
```

Otherwise, n > 1. Let k be the largest power of two smaller than n. The consistency proof is defined recursively as:

* If end <= k, the subtree is on the left of k. The proof proves consistency with the left child and includes the right child:

  ```
  SUBTREE_SUBPROOF(start, end, D_n, b) =
      SUBTREE_SUBPROOF(start, end, D[0:k], b) : MTH(D[k:n])
  ```

* If k <= start, the subtree is on the right of k. The proof proves consistency with the right child and includes the left child.

  ```
  SUBTREE_SUBPROOF(start, end, D_n, b) =
      SUBTREE_SUBPROOF(start - k, end - k, D[k:n], b) : MTH(D[0:k])
  ```

* Otherwise, start < k < end, which implies start = 0. The proof proves consistency with the right child and includes the left child.

  ```
  SUBTREE_SUBPROOF(0, end, D_n, b) =
      SUBTREE_SUBPROOF(0, end - k, D[k:n], false) : MTH(D[0:k])
  ```

When start is zero, this computes a Merkle consistency proof:

SUBTREE_PROOF(0, end, D_n) = PROOF(end, D_n)

When end = start + 1, this computes a Merkle inclusion proof:

SUBTREE_PROOF(start, start + 1, D_n) = PATH(start, D_n)

Appendix B.3 explains the structure of a subtree consistency proof in more detail.

4.4.2. Example Subtree Consistency Proofs

The subtree consistency proof for [4, 8) and a tree of size 14 contains MTH(D[0:4]) and MTH(D[8:14]), depicted in Figure 7. The verifier is assumed to know the subtree hash, so there is no need to include MTH(D[4:8]) itself in the consistency proof.

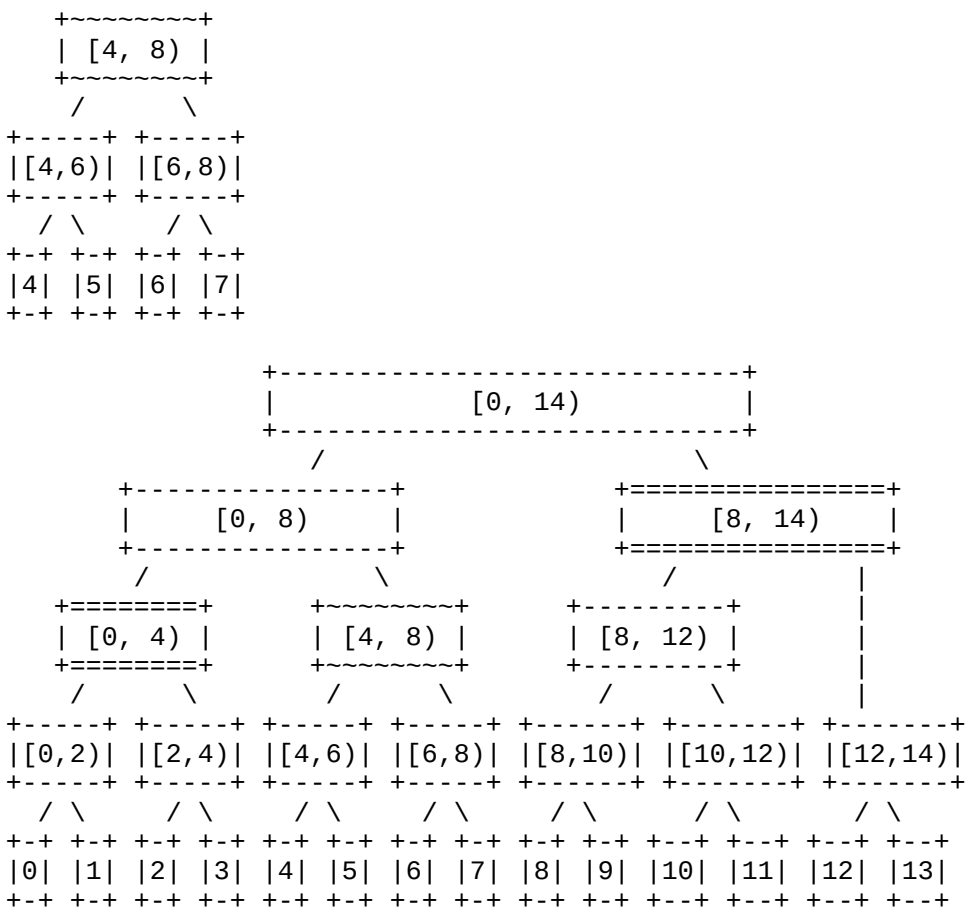
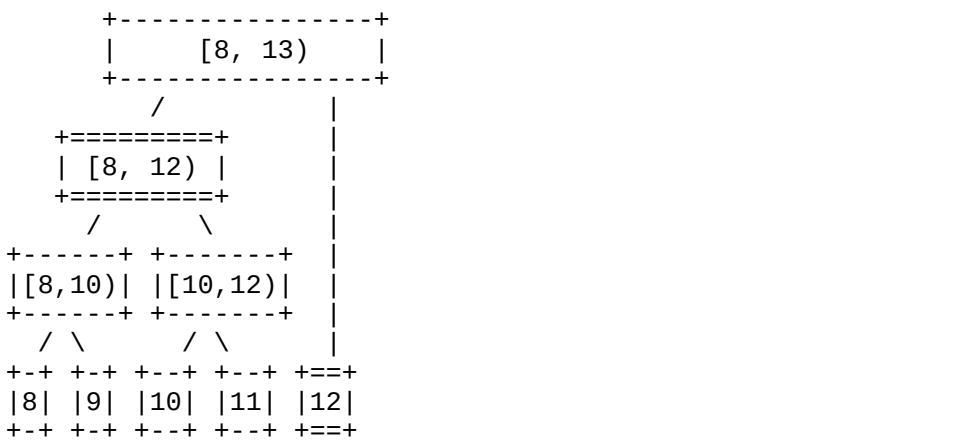


Figure 7: An example subtree consistency proof for a subtree that is directly contained in the full tree

The subtree consistency proof for [8, 13) and a tree of size 14 contains MTH({d[12]}), MTH({d[13]}), MTH(D[8:12]), and MTH(D[0:8]), depicted in Figure 8. [8, 13) is not directly contained in the tree, so the proof must include sufficient nodes to reconstruct both hashes.



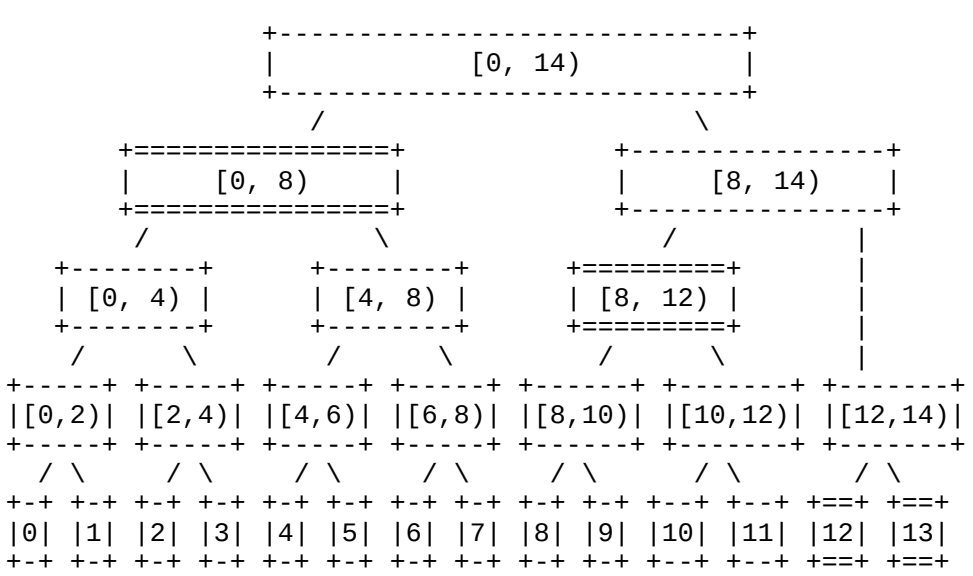


Figure 8: An example subtree consistency proof for a subtree that is not directly contained in the full tree

4.4.3. Verifying a Subtree Consistency Proof

The following procedure can be used to verify a subtree consistency proof.

Given a Merkle Tree over n elements, a subtree defined by [start, end), a consistency proof proof, a subtree hash node_hash, and a root hash root_hash:

1. Check that [start, end) is a valid subtree (Section 4.1), and that end <= n. If either do not hold, fail proof verification. These checks imply 0 <= start < end <= n.

2. Set fn to start, sn to end - 1, and tn to n - 1.

3. If sn is tn, then:

    1. Until fn is sn, right-shift fn, sn, and tn equally.

4. Otherwise:

    1. Until fn is sn or LSB(sn) is not set, right-shift fn, sn, and tn equally.

5. If fn is sn, set fr and sr to node_hash.

6. Otherwise:

    1. If proof is an empty array, stop and fail verification.

    2. Remove the first value of the proof array and set fr and sr to the removed value.

7. For each value c in the proof array:

    1. If tn is 0, then stop the iteration and fail the proof verification.

    2. If LSB(sn) is set, or if sn is equal to tn, then:

        1. If fn < sn, set fr to HASH(0x01 || c || fr).

        2. Set sr to HASH(0x01 || c || sr).

        3. Until LSB(sn) is set, right-shift fn, sn, and tn equally.

    3. Otherwise:

        1. Set sr to HASH(0x01 || sr || c).

    4. Right-shift fn, sn, and tn once more.

8. Compare tn to 0, fr to node_hash, and sr to root_hash. If any are not equal, fail the proof verification. If all are equal, accept the proof.

Appendix B.4 explains this procedure in more detail.

4.5. Arbitrary Intervals

Not all [start, end) intervals of a Merkle Tree are valid subtrees. This section describes how, for any start < end, to determine up to two subtrees that efficiently cover the interval. The subtrees are determined by the following procedure:

1. If end - start is one, return a single subtree, [start, end).

2. Otherwise, run the following to return a pair of subtrees:

    1. Let last be end - 1, the last index in [start, end).

    2. Let split be the bit index of the most significant bit where start and last differ. Bits are numbered from the least significant bit, starting at zero. split is the height at which start and last's paths in the tree diverge.

    3. Let mid be last with the least significant split bits set to zero. mid is the leftmost leaf node in the above divergence point's right branch.

    4. Within the least significant split bits of left, let b be the bit index of the most significant bit with value zero, if any:

        1. If there is such a bit, let left_split be b + 1.

        2. Otherwise, let left_split be zero.

        left_split is the height of the lowest common ancestor of the nodes in [start, mid).

    5. Let left_start be start with the least significant left_split bits set to zero. left_start is the above lowest common ancestor's leftmost leaf node.

    6. Return the subtrees [left_start, mid) and [mid, end).

When the procedure returns a single subtree, the subtree is [start, start+1). When it returns two subtrees, left and right, the subtrees satisfy the following properties:

* left.end = right.start. That is, the two subtrees cover adjacent intervals.

* left.start <= start and end = right.end. That is, the two subtrees together cover the entire target interval, possibly with some extra entries before start left, but not after end.

* left.end - left.start < 2 * (end - start) and right.end - right.start <= end - start. That is, the two subtrees efficiently cover the interval.

* left is full, while right may be partial.

The following Python code implements this procedure:

```
def find_subtrees(start, end):
    """ Returns a list of one or two subtrees that efficiently
    cover [start, end). """
    assert start < end
    if end - start == 1:
        return [(start, end),]
    last = end - 1
    # Find where start and last's tree paths diverge. The two
    # subtrees will be on either side of the split.
    split = (start ^ last).bit_length() - 1
    mask = (1 << split) - 1
    mid = last & ~mask
    # Maximize the left endpoint. This is just before start's
    # path leaves the right edge of its new subtree.
    left_split = (~start & mask).bit_length()
    left_start = start & ~((1 << left_split) - 1)
    return [(left_start, mid), (mid, end)]
```

Figure 9 shows the subtrees which cover [5, 13) in a Merkle Tree of 13 elements. The two subtrees selected are [4, 8) and [8, 13). Note that the subtrees cover a slightly larger interval than [5, 13).

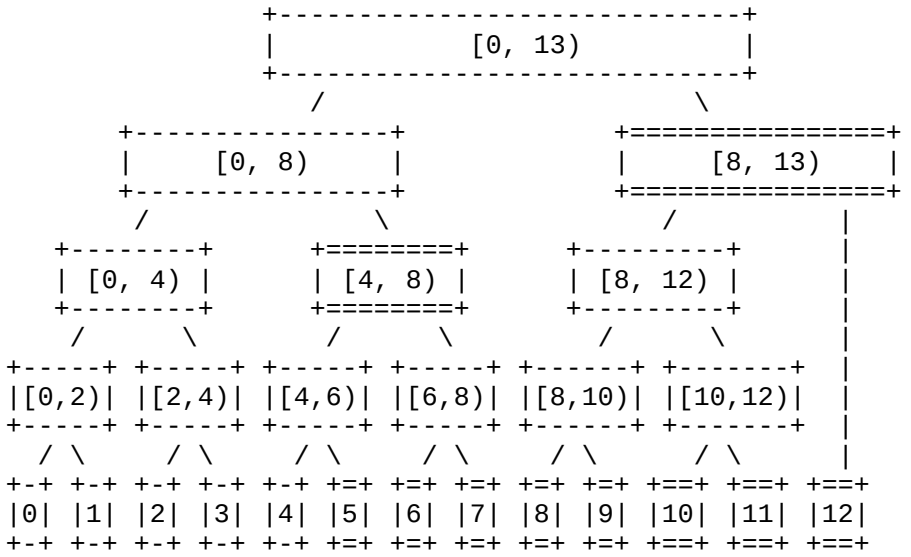


Figure 9: An example selection of subtrees to cover an interval

Two subtrees are needed because a single subtree may not be able to efficiently cover an interval. Figure 10 shows the smallest subtree that contains [7, 9) in a 9-element tree. The smallest single subtree that contains the interval is [0, 9) but this is the entire tree. Using two subtrees, the interval can be described by [7, 8) and [8, 9).

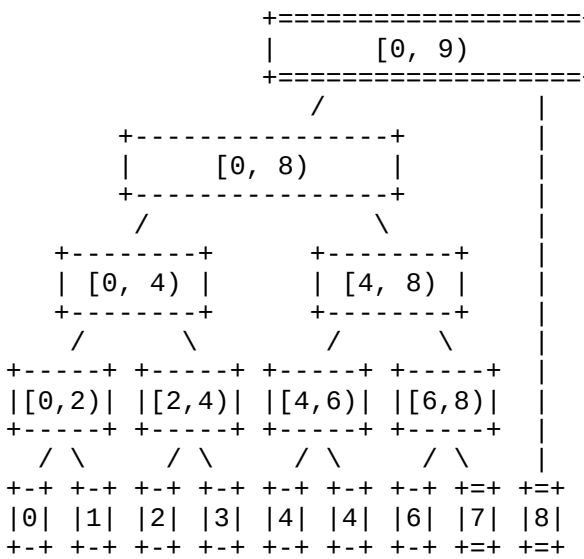


Figure 10: An example showing an inefficient choice of a single subtree

5. Issuance Logs

This section defines the structure of an _issuance log_.

An issuance log describes an append-only sequence of _entries_ (Section 5.3), identified consecutively by an index value, starting from zero. Each entry is an assertion that the CA has certified. The entries in the issuance log are represented as a Merkle Tree, described in Section 2.1 of [RFC9162].

Unlike [RFC6962] and [RFC9162], an issuance log does not have a public submission interface. The log only contains entries which the log operator, i.e. the CA, chose to add. As entries are added, the Merkle Tree is updated to be computed over the new sequence.

A snapshot of the log is known as a _checkpoint_. A checkpoint is identified by its _tree size_, that is the number of elements comitted to the log at the time. Its contents can be described by the Merkle Tree Hash (Section 2.1.1 of [RFC9162]) of entries zero through tree_size - 1.

Cosigners (Section 5.4) sign assertions about the state of the issuance log. A Merkle Tree CA operates a combination of an issuance log and one or more CA cosigners (Section 5.5) that authenticate the log state and certifies the contents. External cosigners may also be deployed to assert correct log operation or provide other services to relying parties (Section 7.3).

5.1. Log Parameters

An issuance log has the following parameters:

* A log ID, which uniquely identifies the log. See Section 5.2.
* A collision-resistant cryptographic hash function. SHA-256 [SHS] is RECOMMENDED.
* A minimum index, which is the index of the first log entry which is available. See Section 5.6.1. This value changes over the lifetime of the log.

Throughout this document, the hash algorithm in use is referred to as HASH, and the size of its output in bytes is referred to as HASH_SIZE.

5.2. Log IDs

Each issuance log is identified by a _log ID_, which is a trust anchor ID [I-D.ietf-tls-trust-anchor-ids].

An issuance log's log ID determines an X.509 distinguished name (Section 4.1.2.4 of [RFC5280]). The distinguished name has a single relative distinguished name, which has a single attribute. The attribute has type id-rdna-trustAnchorID, defined below:

```
id-rdna-trustAnchorID OBJECT IDENTIFIER ::= {
    iso(1) identified-organization(3) dod(6) internet(1) security(5)
    mechanisms(5) pkix(7) rdna(25) TBD}
```

The attribute's value is a RELATIVE-OID containing the trust anchor ID's ASN.1 representation. For example, the distinguished name for a log named 32473.1 would be represented in syntax of [RFC4514] as:

```
1.3.6.1.5.5.7.25.TBD=#0d0481fd5901
```

For initial experimentation, early implementations of this design will:

1. Use UTF8String to represent the attribute's value rather than RELATIVE-OID. The UTF8String contains trust anchor ID's ASCII representation, e.g. 324731.1.

2. Use the OID 1.3.6.1.4.1.44363.47.1 instead of id-rdna-trustAnchorID.

For example, the distinguished name for a log named 32473.1 would be represented in syntax of [RFC4514] as:

```
1.3.6.1.4.1.44363.47.1=#0c0733323437332e31
```

5.3. Log Entries

Each entry in the log is a MerkleTreeCertEntry, defined with the TLS presentation syntax below. A MerkleTreeCertEntry describes certificate information that the CA has validated and certified.

```
struct {} Empty;

enum {
    null_entry(0), tbs_cert_entry(1), (2^16-1)
} MerkleTreeCertEntryType;

struct {
    MerkleTreeCertEntryType type;
    select (type) {
       case null_entry: Empty;
       case tbs_cert_entry: opaque tbs_cert_entry_data[N];
       /* May be extended with future types. */
    }
} MerkleTreeCertEntry;
```

When type is tbs_cert_entry, N is the number of bytes needed to consume the rest of the input. A MerkleTreeCertEntry is expected to be decoded in contexts where the total length of the entry is known.

tbs_cert_entry_data contains the contents octets (i.e. excluding the initial identifier and length octets) of the DER [X.690] encoding of a TBSCertificateLogEntry, defined below. Equivalently, tbs_cert_entry_data contains the DER encodings of each field of the TBSCertificateLogEntry, concatenated. This construction allows a single-pass implementation in Section 7.2.

```
TBSCertificateLogEntry  ::=  SEQUENCE  {
      version             [0]  EXPLICIT Version DEFAULT v1,
      issuer                   Name,
      validity                 Validity,
      subject                  Name,
      subjectPublicKeyInfoHash OCTET STRING,
      issuerUniqueID      [1]  IMPLICIT UniqueIdentifier OPTIONAL,
      subjectUniqueID     [2]  IMPLICIT UniqueIdentifier OPTIONAL,
      extensions          [3]  EXPLICIT Extensions{{CertExtensions}} OPTIONAL }
```

The version, issuer, validity, subject, issuerUniqueID, subjectUniqueID, and extensions fields have the corresponding semantics as in Section 4.1.2 of [RFC5280], with the exception of subjectPublicKeyInfoHash. subjectPublicKeyInfoHash contains the hash of subject's public key as a SubjectPublicKeyInfo (Section 4.1.2.7 of [RFC5280]). The hash uses the log's hash function (Section 5.1) and is computed over the SubjectPublicKeyInfo's DER [X.690] encoding. The issuer field MUST be the issuance log's log ID as an X.509 distinguished name, as described in Section 5.2.

When type is null_entry, the entry does not represent any information. The entry at index zero of every issuance log MUST be of type null_entry. Other entries MUST NOT use null_entry. null_entry exists to avoid zero serial numbers in the certificate format (Section 6.1).

MerkleTreeCertEntry is an extensible structure. Future documents may define new values for MerkleTreeCertEntryType, with corresponding semantics. See Section 5.5 and Section 12.5 for additional discussion.

5.4. Cosigners

This section defines a log _cosigner_. A cosigner follows some append-only view of the log and signs subtrees (Section 4) consistent with that view. The signatures generated by a cosigner are known as _cosignatures_. All subtrees signed by a cosigner MUST be consistent with each other. The cosigner may be external to the log, in which case it might ensure consistency by checking consistency proofs. The cosigner may be operated together with the log, in which case it can trust its log state.

A cosignature MAY implicitly make additional statements about a subtree, determined by the cosigner's role. This document defines one concrete cosigner role, a CA cosigner (Section 5.5), to authenticate the log and certify entries. Other documents and specific deployments may define other cosigner roles, to perform different functions in a PKI. For example, [TLOG-WITNESS] defines a cosigner that only checks the log is append-only, and [TLOG-MIRROR] defines a cosigner that mirrors a log.

Each cosigner has a public key and a _cosigner ID_, which uniquely identifies the cosigner. The cosigner ID is a trust anchor ID [I-D.ietf-tls-trust-anchor-ids]. By identifying the cosigner, the cosigner ID specifies both the public key and the additional statements made by the cosigner's signatures. If a single operator performs multiple cosigner roles in an ecosystem, each role MUST use a distinct cosigner ID and SHOULD use a distinct key.

A single cosigner, with a single cosigner ID and public key, MAY generate cosignatures for multiple logs. In this case, signed subtrees only need to be consistent with others for the same log.

5.4.1. Signature Format

A cosigner computes a cosignature for a subtree in some log by signing a MTCSubtreeSignatureInput, defined below using the TLS presentation language (Section 3 of [RFC8446]):

```
opaque HashValue[HASH_SIZE];

/* From Section 4.1 of draft-ietf-tls-trust-anchor-ids */
opaque TrustAnchorID<1..2^8-1>;

struct {
    TrustAnchorID log_id;
    uint64 start;
    uint64 end;
    HashValue hash;
} MTCSubtree;

struct {
    uint8 label[16] = "mtc-subtree/v1\n\0";
    TrustAnchorID cosigner_id;
    MTCSubtree subtree;
} MTCSubtreeSignatureInput;
```

log_id MUST be the issuance log's ID (Section 5.2), in its binary representation (Section 3 of [I-D.ietf-tls-trust-anchor-ids]). start and end MUST define a valid subtree of the log, and hash MUST be the subtree's hash value in the cosigner's view of the log. The label is a fixed prefix for domain separation. Its value MUST be the string mtc-subtree/v1, followed by a newline (U+000A), followed by a zero byte (U+0000). cosigner_id MUST be the cosigner ID, in its binary representation.

The resulting signature is known as a _subtree signature_. When start is zero, the resulting signature describes the checkpoint with tree size end and is also known as a _checkpoint signature_.

For each supported log, a cosigner retains its checkpoint signature with the largest end. This is known as the cosigner's _current_ checkpoint. If the cosigner's current checkpoint has tree size tree_size, it MUST NOT generate a signature for a subtree [start, end) if start > 0 and end > tree_size. That is, a cosigner can only sign a non-checkpoint subtree if it is contained in its current checkpoint. In a correctly-operated cosigner, every signature made by the cosigner can be proven consistent with its current checkpoint with a subtree consistency proof (Section 4.4). As a consequence, a cosigner that signs a subtree is held responsible for all the entries in the tree of size matching the subtree end, even if the corresponding checkpoint is erroneously unavailable.

Before signing a subtree, the cosigner MUST ensure that hash is consistent with its log state. Different cosigner roles may obtain this assurance differently. For example, a cosigner may compute the hash from its saved log state (e.g. if it is the log operator or

maintains a copy of the log) or by verifying a subtree consistency proof (Section 4.4) from its current checkpoint. When a cosigner signs a subtree, it is held responsible _both_ for the subtree being consistent with its other signatures, _and_ for the cosigner-specific additional statements.

Cosigners SHOULD publish their current checkpoint, along with the checkpoint signature.

[[TODO: CT and tlog put timestamps in checkpoint signatures. Do we want them here? In CT and tlog, the timestamps are monotonically increasing as the log progresses, but we also sign subtrees. We can separate subtree and checkpoint signatures, with timestamps only in the latter, but it's unclear if there is any benefit to this.]]

5.4.2. Signature Algorithms

The cosigner's public key specifies both the key material and the signature algorithm to use with the key material. In order to change key or signature parameters, a cosigner operator MUST deploy a new cosigner, with a new cosigner ID. Signature algorithms MUST fully specify the algorithm parameters, such as hash functions used. This document defines the following signature algorithms:

* ECDSA with P-256 and SHA-256 [FIPS186-5]
* ECDSA with P-384 and SHA-384 [FIPS186-5]
* Ed25519 [RFC8032]
* ML-DSA-44 [FIPS204]
* ML-DSA-65 [FIPS204]
* ML-DSA-87 [FIPS204]

Other documents or deployments MAY define other signature schemes and formats. Log clients that accept cosignatures from some cosigner are assumed to be configured with all parameters necessary to verify that cosigner's signatures, including the signature algorithm and version of the signature format.

5.5. Certification Authority Cosigners

A _CA cosigner_ is a cosigner (Section 5.4) that certifies the contents of a log.

When a CA cosigner signs a subtree, it makes the additional statement that it has certified each entry in the subtree. For example, a domain-validating CA states that it has performed domain validation for each entry, at some time consistent with the entry's validity dates. CAs are held responsible for every entry in every subtree they sign. Proving an entry is included (Section 4.3) in a CA-signed subtree is sufficient to prove the CA certified it.

What it means to certify an entry depends on the entry type:

* To certify an entry of type null_entry is a no-op. A CA MAY freely certify null_entry without being held responsible for any validation.

* To certify an entry of type tbs_cert_entry is to certify the TBSCertificateLogEntry, as defined in Section 5.3.

Entries are extensible. Future documents MAY define type values and what it means to certify them. A CA MUST NOT sign a subtree if it contains an entry with type that it does not recognize. Doing so would certify that the CA has validated the information in some not-yet-defined entry format. Section 12.5 further discusses security implications of new formats.

A CA operator MAY operate multiple CA cosigners that all certify the same log in parallel. This may be useful when, e.g., rotating CA keys. In this case, each CA instance MUST have a distinct name. The CA operator's ACME server can return all CA cosignatures together in a single certificate, with the application protocol selecting the cosignatures to use. Section 8 describes how this is done in TLS [RFC8446].

If the CA operator additionally operates a traditional X.509 CA, that CA key MUST be distinct from any Merkle Tree CA cosigner keys.

5.6. Publishing Logs

[[NOTE: This section is written to avoid depending on a specific serving protocol. The current expectation is that a Web PKI deployment would derive from [TLOG-TILES], to match the direction of Certificate Transparency and pick up improvements made there.

For now, we avoid a normative reference on [TLOG-TILES] and also capture the fact that the certificate construction is independent of the choice of protocol. Similar to how the CT ecosystem is migrating to a tiled interface, were someone to improve on [TLOG-TILES], a PKI could migrate to that new protocol without impacting certificate verification.

That said, this is purely a starting point for describing the design. We expect the scope of this document, and other related documents to adapt as the work evolves across the IETF, C2SP, Certificate Transparency, and other communities.]]

Issuance logs are intended to be publicly accessible in some form, to allow monitors to detect misissued certificates.

The access method does not affect certificate interoperability, so this document does not prescribe a specific protocol. An individual issuance log MAY be published in any form, provided other parties in the PKI are able to consume it. Relying parties SHOULD define log serving requirements, including the allowed protocols and expected availability, as part of their policies on which CAs to support. See also Section 10.3.

For example, a log ecosystem could use [TLOG-TILES] to serve logs. [TLOG-TILES] improves on [RFC6962] and [RFC9162] by exposing the log as a collection of cacheable, immutable "tiles". This works well with a variety of common HTTP [RFC9110] serving architectures. It also allows log clients to request arbitrary tree nodes, so log clients can fetch the structures described in Section 4.

5.6.1. Log Pruning

Over time, an issuance log’s entries will expire and likely be replaced with certificate renewals. As this happens, the total size of the log grows, even if the unexpired subset remains fixed. To mitigate this, issuance logs MAY be _pruned_, as described in this section.

Pruning makes some prefix of the log unavailable, without changing the tree structure. It may be used to reduce the serving cost of long-lived logs, where any entries have long expired. Section 10.3 discusses policies on when pruning may be permitted. This section discusses how it is done and the impact on log structure.

An issuance log is pruned by updating its _minimum index_ parameter (Section 5.1). The minimum index is the index of the first log entry that the log publishes. (See Section 5.6.) It MUST be less than or equal to the tree size of the log’s current checkpoint, and also satisfy any availability policies set by relying parties who trust the CA.

An entry is said to be _available_ if its index is greater than or equal to the minimum index. A checkpoint is said to be available if its tree size is greater than the minimum index. A subtree [start, end) is said to be available if end is greater than the minimum index.

Log protocols MUST serve enough information to allow a log client to efficiently obtain the following:

* Signatures over the latest checkpoint by the CA's cosigners (Section 5.5)

* Any individual available log entry (Section 5.3)

* The hash value of any available checkpoint

* An inclusion proof (Section 2.1.3 of [RFC9162]) for any available entry to any containing checkpoint

* A consistency proof (Section 2.1.4 of [RFC9162]) between any two available checkpoints

* The hash value of any available subtree (Section 4)

* A subtree inclusion proof (Section 4.3) for any available entry in any containing subtree

* A subtree consistency proof (Section 4.4) between any available subtree to any containing checkpoint

Meeting these requirements requires a log to retain some information about pruned entries. Given a node [start, end) in the Merkle Tree, if end is less than or equal to the minimum index, the node's children MAY be discarded in favor of the node's hash.

Figure 11 shows an example pruned tree with 13 elements, where the minimum index is 7. It shows the original tree, followed by the pruned tree. The pruned tree depicts the nodes that MUST be available or computable. Note that entry 6 MAY be discarded, only the hash of entry 6 must be available.

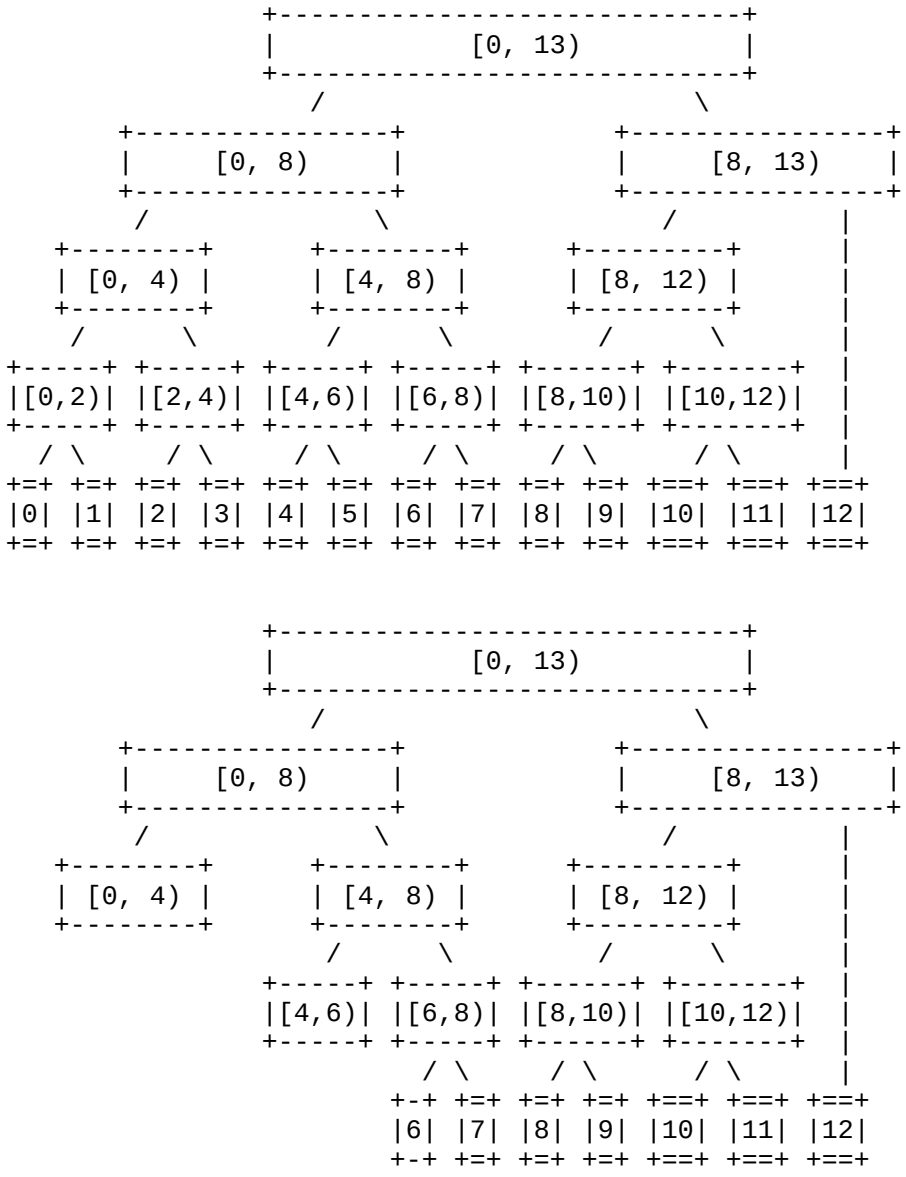


Figure 11: An example showing the minimum nodes that must be available after pruning

Logs MAY retain additional nodes, or expect log clients to compute required nodes from other nodes. For example, in Figure 11, the log's serving protocol MAY instead serve [0, 2) and [2, 4), with the log client computing [0, 4) from those values.

6. Certificates

This section defines how to construct Merkle Tree Certificates, which are X.509 Certificates [RFC5280] that assert the information in an issuance log entry. A Merkle Tree Certificate is constructed from the following:

* A TBSCertificateLogEntry (Section 5.3) contained in the issuance log (Section 5)
* A subject public key whose hash matches the TBSCertificateLogEntry
* A subtree (Section 4) that contains the log entry
* Zero or more signatures (Section 5.4) over the subtree, which together satisfy relying party requirements (Section 7.3)

For any given TBSCertificateLogEntry, there are multiple possible certificates that may prove the entry is certified by the CA and publicly logged, varying by choice of subtree and signatures. Section 6.1 defines how the certificate is constructed based on those choices. Section 6.2 and Section 6.3 define two profiles of Merkle Tree Certificates, full certificates and signatureless certificates, and how to select the subtree and signatures for them.

6.1. Certificate Format

The information is encoded in an X.509 Certificate [RFC5280] as follows:

The TBSCertificate’s version, issuer, validity, subject, issuerUniqueID, subjectUniqueID, and extensions MUST be equal to the corresponding fields of the TBSCertificateLogEntry. If any of issuerUniqueID, subjectUniqueID, or extensions is absent in the TBSCertificateLogEntry, the corresponding field MUST be absent in the TBSCertificate. Per Section 5.3, this means issuer MUST be the issuance log’s log ID as an X.509 distinguished name, as described in Section 5.2.

The TBSCertificate’s serialNumber MUST contain the zero-based index of the TBSCertificateLogEntry in the log. Section 4.1.2.2 of [RFC5280] forbids zero as a serial number, but Section 5.3 defines a null_entry type for use in entry zero, so the index will be positive. This encoding is intended to avoid implementation errors by having the serial numbers and indices off by one.

The TBSCertificate's subjectPublicKeyInfo contains the specified public key. Its hash MUST match the TBSCertificateLogEntry's subjectPublicKeyInfoHash.

The TBSCertificate's signature and the Certificate's signatureAlgorithm MUST contain an AlgorithmIdentifier whose algorithm is id-alg-mtcProof, defined below, and whose parameters is omitted.

```
id-alg-mtcProof OBJECT IDENTIFIER ::= {
    iso(1) identified-organization(3) dod(6) internet(1) security(5)
    mechanisms(5) pkix(7) algorithms(6) TBD}
```

For initial experimentation, early implementations of this design will use the OID 1.3.6.1.4.1.44363.47.0 instead of id-alg-mtcProof.

The signatureValue contains an MTCProof structure, defined below using the TLS presentation language (Section 3 of [RFC8446]):

```
opaque HashValue[HASH_SIZE];

struct {
    TrustAnchorID cosigner_id;
    opaque signature<0..2^16-1>;
} MTCSignature;

struct {
    uint64 start;
    uint64 end;
    HashValue inclusion_proof<0..2^16-1>;
    MTCSignature signatures<0..2^16-1>;
} MTCProof;
```

start and end MUST contain the corresponding parameters of the chosen subtree. inclusion_proof MUST contain a subtree inclusion proof (Section 4.3) for the log entry and the subtree. signatures contains the chosen subtree signatures. In each signature, cosigner_id contains the cosigner ID (Section 5.4) in its binary representation (Section 3 of [I-D.ietf-tls-trust-anchor-ids]), and signature contains the signature value as described in Section 5.4.1.

The MTCProof is encoded into the signatureValue with no additional ASN.1 wrapping. The most significant bit of the first octet of the signature value SHALL become the first bit of the bit string, and so on through the least significant bit of the last octet of the signature value, which SHALL become the last bit of the bit string.

6.2. Full Certificates

A _full certificate_ is a Merkle Tree certificate which contains sufficient signatures to allow a relying party to trust the choice of subtree, without any predistributed information beyond the cosigner(s) parameters. Full certificates can be issued without significant processing delay.

When issuing a certificate, the CA first adds the TBSCertificateLogEntry to its issuance log. It then schedules a job to construct a checkpoint and collect cosignatures. The job proceeds as follows:

1. The CA signs the checkpoint with its key(s) (Section 5.5).
2. Using the procedure in Section 4.5, the CA determines the two subtrees that cover the entries added between this checkpoint and the most recent checkpoint.
3. The CA signs each subtree with its key(s) (Section 5.4).
4. The CA requests sufficient checkpoint cosignatures (Section 5.4) from external cosigners to meet relying party requirements (Section 7.3).
5. The CA requests subtree cosignatures (Appendix C.2) from the cosigners above.
6. For each certificate in the interval, the CA constructs certificates (Section 6.1) using the covering subtree.

Steps 4 and 5 are analogous to requesting SCTs from CT logs in Certificate Transparency, except that a single run of this job collects signatures for many certificates at once. The CA MAY request signatures from a redundant set of cosigners and select the ones that complete first.

This document does not prescribe the specific cosigner roles, or a particular protocol for requesting cosignatures. Protocols for cosigners MAY vary depending on the needs for that cosigner. A consistency-only cosigner, such as [TLOG-WITNESS], might only require a checkpoint signature and consistency proof, while a mirroring cosigner, such as [TLOG-MIRROR] might require the full log contents.

A cosigner MAY expose a private interface for the CA, to reduce denial-of-service risk, or a cosigner MAY expose a public interface for other parties to request additional cosignatures. The latter may be useful if a relying party requires a cosigner that the CA does not

communicate with. In this case, an authenticating party MAY request cosignatures and add them to the certificate. However, it is RECOMMENDED that the CA collect cosignatures for the authenticating party. This simplifies deployment, as relying party policies change over time.

This document does not place any requirements on how frequently this job runs. More frequent runs results in lower issuance delay, but higher signing overhead. It is RECOMMENDED that CAs run at most one instance of this job at a time, starting the next instance after the previous one completes. A single run collects signatures for all entries since the most recent checkpoint, so there is little benefit to overlapping them. Less frequent runs may also aid relying parties that wish to directly audit signatures, as described in Section 5.2 of [AuditingRevisited], though this document does not define such a system.

6.3. Signatureless Certificates

A _signatureless certificate_ is a Merkle Tree certificate which contains no signatures and instead assumes the relying party had predistributed information about which subtrees were trusted. Signatureless certificates are an optional size optimization. They require a processing delay to construct, and only work in a sufficiently up-to-date relying party. Authenticating parties thus SHOULD deploy a corresponding full certificate alongside any signatureless certificate, and use some application-protocol-specific mechanism to select between the two. Section 8 discusses such a mechanism for TLS [RFC8446].

6.3.1. Landmarks

A signatureless certificate is constructed based on a _landmark sequence_, which is a sequence of _landmarks_. Landmarks are agreed-upon tree sizes across the ecosystem for optimizing certificates. Landmarks SHOULD be allocated by the CA, but they can also be allocated by some other coordinating party. It is possible, but NOT RECOMMENDED, for multiple landmark sequences to exist per CA. Landmarks are allocated to balance minimizing the delay in obtaining a signatureless certificate with minimizing the size of the relying party’s predistributed state.

A landmark sequence has the following fixed parameters:

* base_id: An OID arc for trust anchor IDs of individual landmarks

* max_landmarks: A positive integer, describing the maximum number of landmarks that may contain unexpired certificates at any time

* landmark_url: Some URL to fetch the current list of landmarks

Landmarks are numbered consecutively from zero. Each landmark has a trust anchor ID, determined by appending the landmark number to base_id. For example, the trust anchor ID for landmark 42 of a sequence with base_id of 32473.1 would be 32473.1.42.

Each landmark specifies a tree size. The first landmark, numbered zero, is always a tree size of zero. The sequence of tree sizes MUST be append-only and strictly monotonically increasing.

Landmarks determine _landmark subtrees_: for each landmark, other than number zero, let tree_size be the landmark's tree size and prev_tree_size be that of the previous landmark. As described in Section 4.5, select the one or two subtrees that cover [prev_tree_size, tree_size). Each of those subtrees is a landmark subtree. Landmark zero has no landmark subtrees.

The most recent max_landmarks landmarks are said to be _active_. Landmarks MUST be allocated such that, at any given time, only active landmarks contain unexpired certificates. The active landmark subtrees are those determined by the active landmarks. There are at most 2 * max_landmarks active landmark subtrees at any time. Every unexpired entry will be contained in one or more landmark subtree, or between the last landmark subtree and the latest checkpoint. Active landmark subtrees are predistributed to the relying party as trusted subtrees, as described in Section 7.4.

It is RECOMMENDED that landmarks be allocated following the procedure described in Section 6.3.2. If landmarks are allocated incorrectly (e.g. past landmarks change, or max_landmarks is inaccurate), there are no security consequences, but some older certificates may fail to validate.

Relying parties will locally retain up to 2 * max_landmarks hashes (Section 7.4) per CA, so max_landmarks should be set to balance the delay between landmarks and the amount of state the relying party must maintain. Using the recommended procedure above, a CA with a maximum certificate lifetime of 7 days, allocating a landmark every hour, will have a max_landmarks of 168. The client state is then 336 hashes, or 10,752 bytes with SHA-256.

landmark_url MUST serve a resource with Content-Type: text/plain; charset=utf-8 and the following lines. Each line MUST be terminated by a newline character (U+000A):

* Two space-separated non-negative decimal integers: <last_landmark> <num_active_landmarks>. This line MUST satisfy the following, otherwise it is invalid:

  - num_active_landmarks <= max_landmarks

  - num_active_landmarks <= last_landmark

* num_active_landmarks + 1 lines each containing a single non-negative decimal integer, containing a tree size. Numbered from zero to num_active_landmarks, line i contains the tree size for landmark last_landmark - i. The integers MUST be strictly monotonically decreasing and lower or equal to the log's latest tree size.

6.3.2. Allocating Landmarks

It is RECOMMENDED that landmarks be allocated using the following procedure:

1. Select some time_between_landmarks duration. Define a series of consecutive, non-overlapping time intervals, each of duration time_between_landmarks.

2. At most once per time interval, append the latest checkpoint tree size to the landmark sequence if it is greater than the last landmark's tree size.

To ensure that only active landmarks contain unexpired certificates, set max_landmarks to ceil(max_cert_lifetime / time_between_landmarks) + 1, where max_cert_lifetime is the CA's maximum certificate lifetime.

6.3.3. Constructing Signatureless Certificates

Given a TBSCertificateLogEntry in the issuance log and a landmark sequence, a signatureless certificate is constructed as follows:

1. Wait for the first landmark to be allocated that contains the entry.

2. Determine the landmark's subtrees and select the one that contains the entry.

3. Construct a certificate (Section 6.1) using the selected subtree and no signatures.

Before sending this certificate, the authenticating party SHOULD obtain some application-protocol-specific signal that implies the relying party has been configured with the corresponding landmark. (Section 7.4 defines how relying parties are configured.) The trust anchor ID of the landmark may be used as an efficient identifier in the application protocol. Section 8 discusses how to do this in TLS [RFC8446].

6.4. Size Estimates

The inclusion proofs in full and signatureless certificates scale logarithmically with the size of the subtree. These sizes can be estimated with the CA's issuance rate. The byte counts below assume the issuance log's hash function is SHA-256.

Some organizations have published statistics which can be used to estimate this rate for the Web PKI. As of June 9th, 2025:

* [LetsEncrypt] reported around 558,000,000 active certificates for a single CA

* [MerkleTown] reported around 2,100,000,000 unexpired certificates in CT logs, across all CAs

* [MerkleTown] reported an issuance rate of around 444,000 certificates per hour, across all CAs

The current issuance rate across the Web PKI may not necessarily be representative of the Web PKI after a transition to short-lived certificates. Assuming a certificate lifetime of 7 days, and that subscribers will update their certificates 75% of the way through their lifetime (see Section 10.4), every certificate will be reissued every 126 hours. This gives issuance rate estimates of around 4,400,000 certificates per hour and 17,000,000 certificates per hour, for the first two values above. Note the larger estimate is across all CAs, while subtrees would only span one CA.

Using the per-CA short lifetime estimate, if the CA mints a checkpoint every 2 seconds, full certificate subtrees will span around 2,500 certificates, leading to 12 hashes in the inclusion proof, or 384 bytes. Full certificates additionally must carry a sufficient set of signatures to meet relying party requirements.

If a new landmark is allocated every hour, signatureless certificate subtrees will span around 4,400,000 certificates, leading to 23 hashes in the inclusion proof, giving an inclusion proof size of 736 bytes, with no signatures. This is significantly smaller than a single ML-DSA-44 signature, 2,420 bytes, and almost ten times smaller than the three ML-DSA-44 signatures necessary to include post-quantum SCTs.

The proof sizes grow logarithmically, so 32 hashes, or 1024 bytes, is sufficient for subtrees of up to 2^32 (4,294,967,296) certificates.

7. Relying Parties

This section discusses how relying parties verify Merkle Tree Certificates.

7.1. Trust Anchors

In order to accept certificates from a Merkle Tree CA, a relying party MUST be configured with:

* The log ID (Section 5.2)
* A set of supported cosigners, as pairs of cosigner ID and public key
* A policy on which combinations of cosigners to accept in a certificate (Section 7.3)
* An optional list of trusted subtrees, with their hashes, that are known to be consistent with the relying party's cosigner requirements (Section 7.4)
* A list of revoked ranges of indices (Section 7.5)

[[TODO: Define some representation for this. In a trust anchor, there's a lot of room for flexibility in what the client stores. In principle, we could even encode some of this information in an X.509 intermediate certificate, if an application wishes to use this with a delegation model with intermediates, though the security story becomes more complex. Decide how/whether to do that.]]

7.2. Verifying Certificate Signatures

When verifying the signature on an X.509 certificate (Step (a)(1) of Section 6.1.3 of [RFC5280]) whose issuer is a Merkle Tree CA, the relying party performs the following procedure:

1. Check that the TBSCertificate's signature field is id-alg-mtcProof with omitted parameters. If either check fails, abort this process and fail verification.

2. Decode the signatureValue as an MTCProof, as described in Section 6.1.

3. Let index be the certificate's serial number. If index is contained in one of the relying party's revoked ranges (Section 7.5), abort this process and fail verification.

4. Construct a TBSCertificateLogEntry as follows:

    1. Copy the version, issuer, validity, subject, issuerUniqueID, subjectUniqueID, and extensions fields from the TBSCertificate.

    2. Set subjectPublicKeyInfoHash to the hash of the DER encoding of subjectPublicKeyInfo.

5. Construct a MerkleTreeCertEntry of type tbs_cert_entry with contents the TBSCertificateLogEntry. Let entry_hash be the hash of the entry, MTH({entry}) = HASH(0x00 || entry), as defined in Section 2.1.1 of [RFC9162].

6. Let expected_subtree_hash be the result of evaluating the MTCProof's inclusion_proof for entry index, with hash entry_hash, of the subtree described by the MTCProof's start and end, following the procedure in Section 4.3.2. If evaluation fails, abort this process and fail verification.

7. If [start, end) matches a trusted subtree (Section 7.4), check that expected_subtree_hash is equal to the trusted subtree's hash. Return success if it matches and failure if it does not.

8. Otherwise, check that the MTCProof's signatures contain a sufficient set of valid signatures from cosigners to satisfy the relying party's cosigner requirements (Section 7.3). Unrecognized cosigners MUST be ignored. Signatures are verified as described in Section 5.4.1. The hash field of the MTCSubtree is set to expected_subtree_hash.

This procedure only replaces the signature verification portion of X.509 path validation. The relying party MUST continue to perform other checks, such as checking expiry.

In this procedure, entry_hash can equivalently be computed in a single pass from the DER-encoded TBSCertificate, without storing the full TBSCertificateLogEntry or MerkleTreeCertEntry in memory:

1. Initialize a hash instance.

2. Write the big-endian, two-byte tbs_cert_entry value to the hash.

3. Write the TBSCertificate contents octets to the hash, up to the subjectPublicKeyInfo field.

4. Write the octet 0x04 to the hash. This is an OCTET STRING identifer.

5. Write the octet L to the hash, where L is the hash length. (This assumes L is at most 127.)

6. Write H to the hash, where H is the hash of the subjectPublicKeyInfo field.

7. Write the remainder of the TBSCertificate contents octets to the hash, starting just after the subjectPublicKeyInfo field.

8. Finalize the hash and set entry_hash to the result.

This is possible because the structure in Section 5.3 omits the TBSCertificateLogEntry's identifier and length octets.

7.3. Trusted Cosigners

A relying party's cosigner policy determines the sets of cosigners that must sign a view of the issuance log before it is trusted.

This document does not prescribe a particular policy, but gives general guidance. Relying parties MAY implement policies other than those described below, and MAY incorporate cosigners acting in roles not described in this document.

In picking trusted cosigners, the relying party SHOULD ensure the following security properties:

Authenticity: The relying party only accepts entries certified by the CA

Transparency: The relying party only accepts entries that are publicly accessible, so that monitors, particularly the subject of the certificate, can notice any unauthorized certificates

Relying parties SHOULD ensure authenticity by requiring a signature from the most recent CA cosigner key. If the CA is transitioning from an old to new key, the relying party SHOULD accept both until certificates that predate the new key expire. This is analogous to the signature in a traditional X.509 certificate.

While a CA signature is sufficient to prove a subtree came from the CA, this is not enough to ensure the certificate is visible to monitors. A misbehaving CA might not operate the log correctly, either presenting inconsistent versions of the log to relying parties and monitors, or refuse to publish some entries.

To mitigate this, relying parties SHOULD ensure transparency by requiring a quorum of signatures from additional cosigners. At minimum, these cosigners SHOULD enforce a consistent view of the log. For example, [TLOG-WITNESS] describes a lightweight "witness" cosigner role that checks this with consistency proofs. This is not sufficient to ensure durable logging. Section 7.5 discusses mitigations for this. Alternatively, a relying party MAY require cosigners that serve a copy of the log, in addition to enforcing a consistent view. For example, [TLOG-MIRROR] describes a "mirror" cosigner role.

Relying parties MAY accept the same set of additional cosigners across issuance logs.

Cosigner roles are extensible without changes to certificate verification itself. Future specifications and individual deployments MAY define other cosigner roles to incorporate into relying party policies.

Section 10.2 discusses additional deployment considerations in cosigner selection.

7.4. Trusted Subtrees

As an optional optimization, a relying party MAY incorporate a periodically updated, predistributed list of active landmark subtrees, determined as described in Section 6.3.1. The relying party configures these as trusted subtrees, allowing it to accept signatureless certificates (Section 6.3) constructed against those subtrees.

Before configuring the subtrees as trusted, the relying party MUST obtain assurance that each subtree is consistent with checkpoints observed by a sufficient set of cosigners (see Section 5.4) to meet its cosigner requirements. It is not necessary that the cosigners have generated signatures over the specific subtrees, only that they are consistent.

This criteria can be checked given:

* Some _reference checkpoint_ that contains the latest landmark
* For each cosigner, either:
  - A cosignature on the reference checkpoint
  - A cosigned checkpoint containing the referenced checkpoint and a valid Merkle consistency proof (Section 2.1.4 of [RFC9162]) between the two
* For each subtree, a valid subtree consistency proof (Section 4.4) between the subtree and the reference checkpoint

[[TODO: The subtree consistency proofs have many nodes in common. It is possible to define a single "bulk consistency proof" that verifies all the hashes at once, but it’s a lot more complex.]]

This document does not prescribe how relying parties obtain this information. A relying party MAY, for example, use an application-specific update service, such as the services described in [CHROMIUM] and [FIREFOX]. If the relying party considers the service sufficiently trusted (e.g. if the service provides the trust anchor list or certificate validation software), it MAY trust the update service to perform these checks.

The relying party SHOULD incorporate its trusted subtree configuration in application-protocol-specific certificate selection mechanisms, to allow an authenticating party to select a signatureless certificate. The trust anchor IDs of the landmarks may be used as efficient identifiers in the application protocol. Section 8 discusses how to do this in TLS [RFC8446].

7.5. Revocation by Index

For each supported Merkle Tree CA, the relying party maintains a list of revoked ranges of indices. This allows a relying party to efficiently revoke entries of an issuance log, even if the contents are not necessarily known. This may be used to mitigate the security consequences of misbehavior by a CA, or other parties in the ecosystem.

When a relying party is first configured to trust a CA, it SHOULD be configured to revoke all entries from zero up to but not including the first available unexpired certificate at the time. This revocation SHOULD be periodically updated as entries expire and logs are pruned (Section 5.6.1). In particular, when CAs prune entries, relying parties SHOULD be updated to revoke all newly unavailable entries. This gives assurance that, even if some unavailable entry had not yet expired, the relying party will not trust it. It also allows monitors to start monitoring a log without processing expired entries.

A misbehaving CA might correctly construct a globally consistent log, but refuse to make some entries or intermediate nodes available. Consistency proofs between checkpoints and subtrees would pass, but monitors cannot observe the entries themselves. Relying parties whose cosigner policies (Section 7.3) do not require durable logging (e.g. via [TLOG-MIRROR]) are particularly vulnerable to this. In this case, the indices of the missing entries will still be known, so relying parties can use this mechanism to revoke the unknown entries, possibly as an initial, targeted mitigation before a complete CA removal.

When a CA is found to be untrustworthy, relying parties SHOULD remove trust in that CA. To minimize the compatibility impact of this mitigation, index-based revocation can be used to only distrust entries after some index, while leaving existing entries accepted. This is analogous to the [SCTNotAfter] mechanism used in some PKIs.

8. Use in TLS

Most X.509 fields such as subjectPublicKeyInfo and X.509 extensions such as subjectAltName are unmodified in Merkle Tree certificates. They apply to TLS-based applications as in a traditional X.509 certificate. The primary new considerations for use in TLS are:

* Whether the authenticating party should send a certificate from one Merkle Tree CA, another Merkle Tree CA, or a traditional X.509 CA

* Whether the authenticating party should send a full or signatureless certificate

* What the relying party should communicate to the authenticating party to help it make this decision

Certificate selection in TLS, described in Section 4.4.2.2 and Section 4.4.2.3 of [RFC8446], incorporates both explicit relying-party-provided information in the ClientHello and CertificateRequest messages and implicit deployment-specific assumptions. This section describes a RECOMMENDED integration of Merkle Tree certificates into TLS trust anchor IDs ([I-D.ietf-tls-trust-anchor-ids]), but applications MAY use application-specific criteria in addition to, or instead of, this recommendation.

8.1. Extensions to Trust Anchor IDs

[[TODO: Move this into draft-ietf-tls-trust-anchor-ids once the PLANTS WG is further along. See https://github.com/tlswg/tls-trust-anchor-ids/issues/62]]

A TLS deployment may know that all relying parties that accept one trust anchor must additionally accept another trust anchor, or desire identifiers for groups of related trust anchors. For example, in this document, the relying party will recognize up to max_landmark consecutive landmarks, so the latest landmark can be used to represent the range.

Incorporating this knowledge into certificate selection can optimize the ClientHello or CertificateRequest extension. It is RECOMMENDED that this information be provisioned alongside the certificate, e.g. provided by the CA. This section extends the CertificatePropertyList structure (Section 6 of [I-D.ietf-tls-trust-anchor-ids]) with the additional_trust_anchor_ranges certificate property to do this:

```
enum {
    additional_trust_anchor_ranges(1), (2^16-1)
} CertificatePropertyType;

struct {
    TrustAnchorID base;
    uint64 min;
    uint64 max;
} TrustAnchorRange;

TrustAnchorRange TrustAnchorRangeList<1..2^16-1>;
```

A trust anchor range r is said to _contain_ a trust anchor ID id, if id, as a relative OID, is the concatenation of r.base and some integer component between min and max, inclusive.

The following procedure can be used to perform this check. It succeeds if r contains id and fails otherwise:

1. Check that r.base does not end in the middle of an OID component. That is, check that the most-significant bit of the last byte of r.base is unset. If it is set, fail the procedure.

2. Check that r.base is a prefix of id. If not, fail the procedure. Let rest be id with the r.base prefix removed.

3. Decode rest as a minimally-encoded, big-endian, base-128 OID component as follows:

    1. If rest is empty, fail the procedure.

    2. If the most-significant bit of the last byte of rest is set, fail the procedure.

    3. If the most-significant bit of any other byte of rest is unset, fail the procedure.

    4. If the first byte of rest is 0x80, fail the procedure.

    5. Set v to zero. Throughout this procedure, v will be less than 2^64.

    6. For each byte b of rest:

        1. If v is greater than or equal to 2^57, fail the procedure.

        2. Set v to (v << 7) + (b & 127).

4. Check if min <= v <= max. If this is not true, fail the procedure. Otherwise, the procedure succeeds.

Section 4.2 of [I-D.ietf-tls-trust-anchor-ids] is updated as follows. If the ClientHello or CertificateRequest contains a trust_anchors extension, the authenticating party SHOULD send a certification path such that one of the following is true:

* The certification path’s trust anchor ID appears in the relying party’s trust_anchors extension, or

* One of the certification path's additional trust anchor ranges contains some ID in the relying party's trust_anchors extension

Trust anchor ranges do not impact an authenticating party's list of available trust anchors in EncryptedExtensions (see Section 4.3 of [I-D.ietf-tls-trust-anchor-ids]) or the HTTPS/SVCB record (see Section 5 of [I-D.ietf-tls-trust-anchor-ids]). Those continue to reference the single trust anchor ID that corresponds to each certificate.

In applications that use additional trust anchor ranges, relying parties MAY send a single trust anchor ID to represent all certificates whose trust anchor ranges contain that trust anchor ID. This includes:

* Trust anchors that are sent in response to an EncryptedExtensions or HTTPS/SVCB message from the authenticating party

* Trust anchors that are sent in trust_anchors, independently of the authenticating party

8.2. Using Trust Anchor IDs

A full certificate will generally be accepted by relying parties that trust the issuing CA. To determine this, a full certificate has a trust anchor ID of the corresponding log ID (Section 5.2). The authenticating party can obtain this information either by parsing the certificate's issuer field or via out-of-band information as described in Section 3.2 of [I-D.ietf-tls-trust-anchor-ids]. Authenticating and relying parties SHOULD use the trust_anchors extension to determine whether the full certificate would be acceptable.

[[TODO: Ideally we would negotiate cosigners. https://github.com/tlswg/tls-trust-anchor-ids/issues/54 has a sketch of how one might do this, though other designs are possible. Negotiating cosigners allows the ecosystem to manage cosigners efficiently, without needing to collect every possible cosignature and send them all at once. This is wasteful, particularly with post-quantum algorithms.]]

A full certificate MAY also be sent without explicit relying party trust signals, however doing so means the authenticating party implicitly assumes the relying party trusts the issuing CA. This may be viable if, for example, the CA is relatively ubiquitous among supported relying parties.

A signatureless certificate, defined against landmark number L, has a trust anchor ID of base_id, concatenated with L, as described in Section 6.3.1, and SHOULD be provisioned with this value. Additionally, relying parties that trust later landmarks may also be assumed to trust landmark L, so a signatureless certificate SHOULD additionally provisioned with an additional trust anchor range whose base is base_id, min is L, and max is L + max_landmarks - 1.

A relying party that has been configured with trusted subtrees (Section 7.4) derived from a set of landmarks SHOULD configure the trust_anchors extension to advertise the highest supported landmark in the set. The selection procedures defined in [I-D.ietf-tls-trust-anchor-ids] and Section 8.1 will then correctly determine whether a signatureless certificate is compatible with the relying party.

When both a signatureless and full certificate are supported by a relying party, an authenticating party SHOULD preferentially use the signatureless certificate. A signatureless certificate asserts the same information as its full counterpart, but is expected to be smaller. An authenticating party SHOULD NOT send a signatureless certificate without a signal that the relying party trusts the corresponding landmark subtree. Even if the relying party is assumed to trust the issuing CA, the relying party may not have sufficiently up-to-date trusted subtrees.

9. ACME Extensions

This section describes how to issue Merkle Tree certificates using ACME [RFC8555].

When downloading the certificate (Section 7.4.2 of [RFC8555]), ACME clients supporting Merkle Tree certificates SHOULD send "application/pem-certificate-chain-with-properties" in their Accept header (Section 12.5.1 of [RFC9110]). ACME servers issuing Merkle Tree certificates SHOULD then respond with that content type and include trust anchor ID information as described in Section 6 of [I-D.ietf-tls-trust-anchor-ids]. Section 8 decribes the trust anchor ID assignments for full and signatureless certificates.

When processing an order for a Merkle Tree certificate, the ACME server moves the order to the "valid" state once the corresponding entry is sequenced in the issuance log. The order’s certificate URL then serves the full certificate, constructed as described in Section 6.2.

The full certificate response SHOULD additionally carry a alternate URL for the signatureless certificate, as described Section 7.4.2 of [RFC8555]. Before the signatureless certificate is available, the alternate URL SHOULD return a HTTP 503 (Service Unavailable) response, with a Retry-After header (Section 10.2.3 of [RFC9110]) estimating when the certificate will become available. Once the next landmark is allocated, the ACME server constructs a signatureless certificate, as described in Section 6.3 and serves it from the alternate URL.

ACME clients supporting Merkle Tree certificates SHOULD support fetching alternate chains. If an alternate chain returns an HTTP 503 with a Retry-After header, as described above, the client SHOULD retry the request at the specified time.

10. Deployment Considerations

10.1. Operational Costs

10.1.1. Certification Authority Costs

While Merkle Tree certificates expects CAs to operate logs, the costs of these logs are expected to be much lower than a CT log from [RFC6962] or [RFC9162]:

Section 5.6 does not constrain the API to the one defined in [RFC6962] or [RFC9162]. If the PKI uses a tile-based protocol, such as [TLOG-TILES], the issuance log benefits from the improved caching properties of such designs.

Unlike a CT log, an issuance log does not have public submission APIs. Log entries are only added by the CA directly. The costs are thus expected to scale with the CA’s own operations.

A CA only needs to produce a digital signature for every checkpoint, rather than for every certificate. The lower signature rate requirements could allow more secure and/or economical key storage choices.

Individual entries are kept small and do not scale with public key or signature sizes. This mitigates growth from post-quantum algorithms. Public keys in entries are replaced with fixed-sized hashes. There are no signatures in entries themselves, and only signatures on the very latest checkpoint are retained. Every new checkpoint completely subsumes the old checkpoint, so there is no need to retain older signatures. Likewise, a subtree is only signed if contained in another signed checkpoint.

Log pruning (Section 5.6.1) allows a long-lived log to serve only the more recent entries, scaling with the size of the retention window, rather than the log's total lifetime.

Mirrors of the log can also reduce CA bandwidth costs, because monitors can fetch data from mirrors instead of CAs directly. In PKIs that deploy mirrors as part of cosigner policies, relying parties could set few availability requirements on CAs, as described in Section 10.3.

10.1.2. Cosigner Costs

The costs of cosigners vary by cosigner role. A consistency-checking cosigner, such as [TLOG-WITNESS], requires very little state and can be run with low cost.

A mirroring cosigner, such as [TLOG-MIRROR], performs comparable roles as CT logs, but several of the cost-saving properties in Section 10.1.1 also apply: improved protocols, smaller entries, less frequent signatures, and log pruning. While a mirror does need to accommodate another party's (the CA's) growth rate, it grows only from new issuances from that one CA. If one CA's issuance rate exceeds the mirror's capacity, that does not impact the mirror's copies of other CAs. Mirrors also do not need to defend against a client uploading a large number of existing certificates all at once. Submissions are also naturally batched and serialized.

10.1.3. Monitor Costs

In a CT-based PKI, every log carries a potentially distinct subset of active certificates, so monitors must check the contents of every CT log. At the same time, certificates are commonly synchronized between CT logs. As a result, a monitor will typically download each certificate multiple times, once for every log. In Merkle Tree Certificates, each entry appears in exactly one log. A relying party might require a log to be covered by a quorum of mirrors, but each mirror is cryptographically verified to serve the same contents. Once a monitor has obtained some entry from one mirror, it does not need to download it from the others.

In addition to downloading each entry only once, the entries themselves are smaller, as discussed in Section 10.1.1.

10.2. Choosing Cosigners

In selecting trusted cosigners and cosigner requirements (Section 7.3), relying parties navigate a number of trade-offs:

A consistency-checking cosigner, such as [TLOG-WITNESS], is very cheap to run, but does not guarantee durable logging, while a mirroring cosigner is more expensive and may take longer to cosign structures. Requiring a mirror signature provides stronger guarantees to the relying party, which in turn can reduce the requirements on CAs (see Section 10.3), however it may cause certificate issuance to take longer. That said, mirrors are comparable to CT logs, if not cheaper (see Section 10.1), so they may be appropriate in PKIs where running CT logs is already viable.

Relying parties that require larger quorums of trusted cosigners can reduce the trust placed in any individual cosigner. However, these larger quorums result in larger, more expensive full certificates. The cost of this will depend on how frequently the signatureless optimization occurs in a given PKI. Conversely, relying parties that require smaller quorums have smaller full certificates, but place more trust in their cosigners.

Relying party policies also impact monitor operation. If a relying party accepts any one of three cosigners, monitors SHOULD check the checkpoints of all three. Otherwise, a malicious CA may send different split views to different cosigners. More generally, monitors SHOULD check the checkpoints in the union of all cosigners trusted by all supported relying parties. This is an efficient check because, if the CA is operating correctly, all cosigners will observe the same tree. Thus the monitor only needs to check consistency proofs between the checkpoints, and check the log contents themselves once. Monitors MAY also rely on other parties in the transparency ecosystem to perform this check.

10.3. Log Availability

CAs and mirrors are expected to serve their log contents over HTTP. It is possible for the contents to be unavailable, either due to temporary service outage or because the log has been pruned (Section 5.6.1). If some resources are unavailable, they may not be visible to monitors.

As in CT, PKIs which deploy Merkle Tree certificates SHOULD establish availability policies, adhered to by trusted CAs and mirrors, and enforced by relying party vendors as a condition of trust. Exact availability policies for these services are out of scope for this document, but this section provides some general guidance.

Availability policies SHOULD specify how long an entry must be made available, before a CA or mirror is permitted to prune the entry. It is RECOMMENDED to define this using a _retention period_, which is some time after the entry has expired. In such a policy, an entry

could only be pruned if it, and all preceding entries, have already expired for the retention period. Policies MAY opt to set different retention periods between CAs and mirrors. Permitting limited log retention is analogous to the CT practice of temporal sharding [CHROME-CT], except that a pruned issuance log remains compatible with older, unupdated relying parties.

Such policies impact monitors. If the retention period is, e.g. 6 months, this means that monitors are expected to check entries of interest within 6 months. It also means that a new monitor may only be aware of a 6 month history of entries issued for a particular domain.

If historical data is not available to verify the retention period, such as information in another mirror or a trusted summary of expiration dates of entries, it may not be possible to confirm correct behavior. This is mitigated by the revocation process described in Section 7.5: if a CA were to prune a forward-dated entry and, in the 6 months when the entry was available, no monitor noticed the unusual expiry, an updated relying party would not accept it anyway.

The log pruning process simply makes some resources unavailable, so availability policies SHOULD constrain log pruning in the same way as general resource availability. That is, if it would be a policy violation for the log to fail to serve a resource, it should also be a policy violation for the log to prune such that the resource is removed, and vice versa.

PKIs that require mirror cosignatures (Section 7.3) can impose minimal to no availability requirements on CAs, all without compromising transparency goals. If a CA never makes some entry available, mirrors will be unable to update. This will prevent relying parties from accepting the undisclosed entries. However, a CA which is persistently unavailable may not offer sufficient benefit to be used by authenticating parties or trusted by relying parties.

However, if a mirror’s interface becomes unavailable, monitors may be unable to check for unauthorized issuance, if the entries are not available in another mirror. This does compromise transparency goals. As such, availability policies SHOULD set availability expectations on mirrors. This can also be mitigated by using multiple mirrors, either directly enforced in cosigner requirements, or by keeping mirrors up-to-date with each other.

In PKIs that do not require mirroring cosigners, the CA’s serving endpoint is more crucial for monitors. Such PKIs thus SHOULD set availability requirements on CAs.

In each of these cases, availability failures can be mitigated by revoking the unavailable entries by index, as described in Section 7.5, likely as a first step in a broader distrust.

10.4. Certificate Renewal

When an authenticating party requests a certificate, the signatureless certificate will not be available until the next landmark is ready. From there, the signatureless certificate will not be available until relying parties receive new trusted subtrees.

To maximize coverage of the signatureless certificate optimization, authenticating parties performing routine renewal SHOULD request a new Merkle Tree certificate some time before the previous Merkle Tree certificate expires. Renewing around 75% into the previous certificate's lifetime is RECOMMENDED. Authenticating parties additionally SHOULD retain both the new and old certificates in the certificate set until the old certificate expires. As the new subtrees are delivered to relying parties, certificate negotiation will transition relying parties to the new certificate, while retaining the old certificate for relying parties that are not yet updated.

The above also applies if the authenticating party is performing a routine key rotation alongside the routine renewal. In this case, certificate negotiation would pick the key as part of the certificate selection. This slightly increases the lifetime of the old key but maintains the size optimization continuously.

If the service is rotating keys in response to a key compromise, this option is not appropriate. Instead, the service SHOULD immediately discard the old key and request a full certificate and the revocation of the previous certificate. This will interrupt the size optimization until the new signatureless certificate is available and relying parties are updated.

10.5. Multiple CA Keys

The separation between issuance logs and CA cosigners gives CAs additional flexibility in managing keys. A CA operator wishing to rotate keys, e.g. to guard against compromise of older key material, or upgrade to newer algorithms, could retain the same issuance log and sign its checkpoints and subtrees with both keys in parallel, until relying parties are all updated. Older relying parties would verify the older signatures, while newer relying parties would verify the newer signatures. A cosignature negotiation mechanism in the application protocol (see Section 8) would avoid using extra bandwidth for the two signatures.

11. Privacy Considerations

The Privacy Considerations described in Section 9 of [I-D.ietf-tls-trust-anchor-ids] apply to its use with Merkle Tree Certificates.

In particular, relying parties that share an update process for trusted subtrees (Section 7.4) will fetch the same stream of updates. However, updates may reach different users at different times, resulting in some variation across users. This variation may contribute to a fingerprinting attack [RFC6973]. If the Merkle Tree CA trust anchors are sent unconditionally in trust_anchors, this variation will be passively observable. If they are sent conditionally, e.g. with the DNS mechanism, the trust anchor list will require active probing.

12. Security Considerations

12.1. Authenticity

A key security requirement of any PKI scheme is that relying parties only accept assertions that were certified by a trusted certification authority. Merkle Tree certificates achieve this by ensuring the relying party only accepts authentic subtree hashes:

* In full certificates, the relying party's cosigner requirements (Section 7.3) are expected to include some signature by the CA's cosigner. The CA's cosigner (Section 5.5) is defined to certify the contents of every checkpoint and subtree that it signs.

* In signatureless certificates, the cosigner requirements are checked ahead of time, when the trusted subtrees are predistributed (Section 7.4).

Given such a subtree hash, computed over entries that the CA certified, it then must be computationally infeasible to construct an entry not on this list, and some inclusion proof, such that inclusion proof verification succeeds. This requires using a collision-resistant hash in the Merkle Tree construction.

Log entries contain public key hashes, so it must additionally be computationally infeasible to compute a public key whose hash matches the entry, other than the intended public key. This also requires a collision-resistant hash.

12.2. Transparency

The transparency mechanisms in this document do not prevent a CA from issuing an unauthorized certificate. Rather, they provide comparable security properties as Certificate Transparency [RFC9162] in ensuring that all certificates are either rejected by relying parties, or visible to monitors and, in particular, the subject of the certificate.

Compared to Certificate Transparency, some of the responsibilities of a log have moved to the CA. All signatures generated by the CA in this system are assertions about some view of the CA's issuance log. However, a CA does not need to function correctly to ensure transparency properties. Relying parties are expected to require a quorum of additional cosigners, which together enforce properties of the log (Section 7.3) and prevent or detect CA misbehavior:

A CA might violate the append-only property of its log and present different views to different parties. However, each individual cosigner will only follow a single append-only view of the log history. Provided the cosigners are correctly operated, relying parties and monitors will observe consistent views between each other. Views that were not cosigned at all may not be detected, but they also will not be accepted by relying parties.

If the CA sends one view to some cosigners and another view to other cosigners, it is possible that multiple views will be accepted by relying parties. However, in that case monitors will observe that cosigners do not match each other. Relying parties can then react by revoking the inconsistent indices (Section 7.5), and likely removing the CA. If the cosigners are mirrors, the underlying entries in both views will also be visible.

A CA might correctly construct its log, but refuse to serve some unauthorized entry, e.g. by feigning an outage or pruning the log outside the retention policy (Section 10.3). If the relying party requires cosignatures from trusted mirrors, the entry will either be visible to monitors in the mirrors, or have never reached a mirror. In the latter case, the entry will not have been cosigned, so the relying party would not accept it. If the relying party accepts log views without a trusted mirror, the unauthorized entry may not be available. However, the existence of _some_ entry at that index will be visible, so monitors will know the CA is failing to present an entry. Relying parties can then react by revoking the undisclosed entries by index (Section 7.5), and likely removing the CA.

12.3. Public Key Hashes

Unlike Certificate Transparency, the mechanisms in this document do not provide the subject public keys, only the hashed values. This is intended to reduce log serving costs, particularly with large post-quantum keys. As a result, monitors look for unrecognized hashes instead of unrecognized keys. Any unrecognized hash, even if the preimage is unknown, indicates an unauthorized certificate.

This optimization complicates studies of weak public keys, e.g. [SharedFactors]. Such studies will have to retrieve the public keys separately, such as by connecting to the TLS servers, or fetching from the CA if it retains the unhashed key. This document does not define a mechanism for doing this, or require that CAs or mirrors retain unhashed keys. The transparency mechanisms in this protocol are primarily intended to allow monitors to observe certificate issuance.

12.4. Non-Repudiation

When a monitor finds an unauthorized certificate issuance in a log or mirror, it must be possible to prove the CA indeed certified the information in the entry. However, only the latest checkpoint signature is retained by the transparency ecosystem, so it may not be possible to reconstruct the exact certificate seen by relying parties.

However, per Section 5.5, any checkpoint signature is a binding assertion by the CA that it has certified every entry in the checkpoint. Thus, given _any_ signed checkpoint that contains the unauthorized entry, a Merkle inclusion proof (Section 2.1.3 of [RFC9162]) is sufficient to prove the CA issued the entry. This is analogous to how, in Section 3.2.1 of [RFC9162], CAs are held accountable for signed CT precertificates.

The transparency ecosystem does not retain unhashed public keys, so it also may not be possible to construct a complete certificate from the checkpoint signature and inclusion proof. However, if the log entry’s subjectPublicKeyInfoHash does not correspond to an authorized key for the subject of the certificate, the entry is still unauthorized. A Merkle Tree CA is held responsible for all log entries it certifies, whether or not the preimage of the hash is known.

12.5. New Log Entry Types

MerkleTreeCertEntry (Section 5.3) is extensible and permits protocol extensions to define new formats for the CA to certify. This means older CAs, cosigners, relying parties, and monitors might interact with new entries:

Section 5.3 and Section 5.5 forbid a CA from logging or signing entries that it does not recognize. A CA cannot faithfully claim to certify information if it does not understand it. This is analogous to how a correctly-operated X.509 can never sign an unrecognized X.509 extension.

External cosigners may or may not interact with the unrecognized entries. [TLOG-MIRROR] and [TLOG-WITNESS] describe cosigners whose roles do not interpret the contents of log entries. New entry types MAY be added without updating them. If a cosigner role does interpret a log entry, it MUST define how it interacts with unknown ones.

If a relying party trusts an issuance log, but the issuance log contains an unrecognized entry, the entry will not cause it to accept an unexpected certificate. In Section 7.2, the relying party constructs the MerkleTreeCertEntry that it expects. The unrecognized entry will have a different type value, so the proof will never succeed, assuming the underlying hash function remains collision-resistant.

If a monitor observes an entry with unknown type, it may not be able to determine if it is of interest. For example, it may be unable to tell whether it covers some relevant DNS name. Until the monitor is updated to reflect the current state of the PKI, the monitor may be unable to detect all misissued certificates.

This situation is analogous to the addition of a new X.509 extension. When relying parties add support for log entry types or new X.509 extensions, they SHOULD coordinate with monitors to ensure the transparency ecosystem is able to monitor the new formats.

12.6. Certificate Malleability

An ASN.1 structure like X.509's Certificate is an abstract data type that is independent of its serialization. There are multiple encoding rules for ASN.1. Commonly, protocols use DER [X.690], such as Section 4.4.2 of [RFC8446]. This aligns with Section 4.1.1.3 of [RFC5280], which says X.509 signatures are computed over the DER-encoded TBSCertificate. After signature verification, applications can assume the DER-encoded TBSCertificate is not malleable.

While the signature verification process in Section 7.2 first transforms the TBSCertificate into a TBSCertificateLogEntry, it preserves this non-malleability. There is a unique valid DER encoding for every abstract TBSCertificate structure, so malleability of the DER-encoded TBSCertificate reduces to malleability of the TBSCertificate value:

* The version, issuer, validity, subject, issuerUniqueID, subjectUniqueID, and extensions fields are copied from the TBSCertificate to the TBSCertificateLogEntry unmodified, so they are directly authenticated by the inclusion proof.

* serialNumber is omitted from TBSCertificateLogEntry, but its value determines the inclusion proof index, which authenticates it.

* The redundant signature field in TBSCertificate is omitted from TBSCertificateLogEntry, but Section 7.2 checks for an exact value, so no other values are possible.

* subjectPublicKeyInfo is hashed as subjectPublicKeyInfoHash in TBSCertificateLogEntry. Provided the underlying hash function is collision-resistant, no other values are possible for a given log entry.

X.509 implementations often implement Section 4.1.1.3 of [RFC5280] by equivalently retaining the original received DER encoding, rather than recomputing the canonical DER encoding TBSCertificate. This optimization is compatible with the assumptions above.

Some non-conforming X.509 implementations use a BER [X.690] parser instead of DER, and then apply this optimization to the received BER encoding. BER encoding is not unique, so this does not produce the same result. In such implementations, the BER-encoded TBSCertificate becomes also non-malleable, and applications may rely on this. To preserve this property in Merkle Tree Certificates, such non-conforming implementations MUST do the following when implementing Section 7.2:

* Reparse the initial identifier (the SEQUENCE tag) and length octets of the TBSCertificate structure with a conforming DER parser and fail verification if invalid.

* When copying the version, issuer, validity, subject, issuerUniqueID, subjectUniqueID, and extensions fields, either copy over the observed BER encodings, or reparse each field with a conforming DER parser and fail verification if invalid.

* Reparse the serialNumber field with a conforming DER parser and fail verification if invalid.

* Reparse the signature field with a conforming DER parser and fail verification if invalid. Equivalently, check for an exact equality with for the expected, DER-encoded value.

* When hashing subjectPublicKeyInfo, either hash the observed BER encoding, or reparse the structure with a conforming DER parser and fail verification if invalid.

These additional checks are redundant in X.509 implementations that use a conforming DER parser.

Section 5.3 requires that the TBSCertificateLogEntry in a MerkleTreeCertEntry be DER-encoded, so applying a stricter parser will be compatible with conforming CAs. While these existing non-conforming implementations may be unable to switch to a DER parser due to compatibility concerns, Merkle Tree Certificates is new, so there is no existing deployment of malformed BER-encoded TBSCertificateLogEntry structures.

The above only ensures the TBSCertificate portion is non-malleable. In Merkle Tree Certificates, similar to ECDSA X.509 signature, the signature value is malleable. Multiple MTCProof structures may prove a single TBSCertificate structure. Additionally, in all X.509-based protocols, a BER-based parser for the outer, unsigned Certificate structure will admit malleability in those portions of the encoding. Applications that derive a unique identifier from the Certificate MUST instead use the TBSCertificate, or some portion of it, for Merkle Tree Certificates.

13. IANA Considerations

13.1. Module Identifier

IANA is requested to add the following entry in the "SMI Security for PKIX Module Identifier" registry [RFC7299]:

Decimal	Description	References
TBD	id-mod-mtc-2025	[this-RFC]

Table 1

13.2. Algorithm

IANA is requested to add the following entry to the "SMI Security for PKIX Algorithms" registry [RFC7299]:

Decimal	Description	References
TBD	id-alg-mtcProof	[this-RFC]

Table 2

13.3. Relative Distinguished Name Attribute

IANA is requested to add the following entry to the "SMI Security for PKIX Relative Distinguished Name Attribute" registry [I-D.ietf-lamps-x509-alg-none]:

Decimal	Description	References
TBD	id-rdna-trustAnchorID	[this-RFC]

Table 3

14. References

14.1. Normative References

[FIPS186-5]
"Digital Signature Standard (DSS)", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.186-5, February 2023, <https://doi.org/10.6028/nist.fips.186-5>.

[FIPS204] "Module-lattice-based digital signature standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.204, August 2024, <https://doi.org/10.6028/nist.fips.204>.

[I-D.ietf-tls-trust-anchor-ids]
Beck, B., Benjamin, D., O'Brien, D., and K. Nekritz, "TLS Trust Anchor Identifiers", Work in Progress, Internet-Draft, draft-ietf-tls-trust-anchor-ids-02, 15 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-trust-anchor-ids-02>.

[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/rfc/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/rfc/rfc3629>.

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

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

[RFC5912] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, DOI 10.17487/RFC5912, June 2010, <https://www.rfc-editor.org/rfc/rfc5912>.

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

[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/rfc/rfc8174>.

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

[RFC8555] Barnes, R., Hoffman-Andrews, J., McCarney, D., and J. Kasten, "Automatic Certificate Management Environment (ACME)", RFC 8555, DOI 10.17487/RFC8555, March 2019, <https://www.rfc-editor.org/rfc/rfc8555>.

[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/rfc/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/rfc/rfc9162>.

[SHS] "Secure hash standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.180-4, 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[X.690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8824-1:2021 , February 2021.

14.2. Informative References

[APPLE-CT] Apple, "Apple’s Certificate Transparency policy", 5 March 2021, <https://support.apple.com/en-us/HT205280>.

[AuditingRevisited] Heimberger, L., Patton, C., and B. Westerbaan, "Private SCT Auditing, Revisited", 25 April 2025, <https://eprint.iacr.org/2025/556.pdf>.

[CABF-153] CA/Browser Forum, "Ballot 153 Short-Lived Certificates", 11 November 2015, <https://cabforum.org/2015/11/11/ballot-153-short-lived-certificates/>.

[CABF-SC081] CA/Browser Forum, "Ballot SC081v3: Introduce Schedule of Reducing Validity and Data Reuse Periods", 11 April 2025, <https://cabforum.org/2025/04/11/ballot-sc081v3-introduce-schedule-of-reducing-validity-and-data-reuse-periods/>.

[CHROME-CT] Google Chrome, "Chrome Certificate Transparency Policy", 17 March 2022, <https://googlechrome.github.io/CertificateTransparency/ct_policy.html>.

[CHROMIUM] Chromium, "Component Updater", 3 March 2022, <https://chromium.googlesource.com/chromium/src/+/main/components/component_updater/README.md>.

[FIREFOX] Mozilla, "Firefox Remote Settings", 20 August 2022, <https://wiki.mozilla.org/Firefox/RemoteSettings>.

[I-D.ietf-lamps-x509-alg-none]
Benjamin, D., "Unsigned X.509 Certificates", Work in Progress, Internet-Draft, draft-ietf-lamps-x509-alg-none-10, 5 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-lamps-x509-alg-none-10>.

[LetsEncrypt]
Let’s Encrypt, "Let’s Encrypt Stats", 7 March 2023, <https://letsencrypt.org/stats/>.

[MerkleTown]
Cloudflare, Inc., "Merkle Town", 7 March 2023, <https://ct.cloudflare.com/>.

[RFC4514] Zeilenga, K., Ed., "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", RFC 4514, DOI 10.17487/RFC4514, June 2006, <https://www.rfc-editor.org/rfc/rfc4514>.

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

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[RFC7299] Housley, R., "Object Identifier Registry for the PKIX Working Group", RFC 7299, DOI 10.17487/RFC7299, July 2014, <https://www.rfc-editor.org/rfc/rfc7299>.

[SCTNotAfter]
Adrian, D., "How to distrust a CA without any certificate errors", March 2025, <https://dadrian.io/blog/posts/sct-not-after/>.

[SharedFactors]
Vage, H. F. and University of Bergen, "Finding shared RSA factors in the Certificate Transparency logs", 13 May 2022, <https://bora.uib.no/bora-xmlui/bitstream/handle/11250/3001128/Masters_thesis__for_University_of_Bergen.pdf>.

[SIGNED-NOTE]
C2SP, "Note", April 2025, <https://c2sp.org/signed-note>.

[STH-Discipline]
Barnes, R., "STH Discipline & Security Considerations", 3 March 2017, <https://mailarchive.ietf.org/arch/msg/trans/Zm4NqyRc7LDsOtV56EchBIT9r4c/>.

[TLOG-CHECKPOINT]
C2SP, "Transparency Log Checkpoints", March 2024, <https://c2sp.org/tlog-checkpoint>.

[TLOG-MIRROR]
C2SP, "Transparency Log Mirrors", July 2025, <https://c2sp.org/tlog-mirror>.

[TLOG-TILES]
C2SP, "Tiled Transparency Logs", June 2025, <https://c2sp.org/tlog-tiles>.

[TLOG-WITNESS]
C2SP, "Transparency Log Witness Protocol", June 2025, <https://c2sp.org/tlog-witness>.

Appendix A. ASN.1 Module

```
MerkleTreeCertificates
  { iso(1) identified-organization(3) dod(6) internet(1)
    security(5) mechanisms(5) pkix(7) id-mod(0)
    id-mod-mtc-2025(TBD) }

DEFINITIONS IMPLICIT TAGS ::=
BEGIN

IMPORTS
  SIGNATURE-ALGORITHM
  FROM AlgorithmInformation-2009  -- in [RFC5912]
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-algorithmInformation-02(58) }
  Extensions{}, ATTRIBUTE
  FROM PKIX-CommonTypes-2009 -- in [RFC5912]
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkixCommon-02(57) }
  CertExtensions
  FROM PKIX1Implicit-2009 -- in [RFC5912]
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkix1-implicit-02(59) }
  Version, Name, Validity, UniqueIdentifier
```

```
    FROM PKIX1Explicit-2009 -- in [RFC5912]
      { iso(1) identified-organization(3) dod(6) internet(1)
        security(5) mechanisms(5) pkix(7) id-mod(0)
        id-mod-pkix1-explicit-02(51) }
    TrustAnchorID
    FROM TrustAnchorIDs-2025 -- in [I-D.ietf-tls-trust-ancohor-ids]
      { iso(1) identified-organization(3) dod(6) internet(1)
        security(5) mechanisms(5) pkix(7) id-mod(0)
        id-mod-trustAnchorIDs-2025(TBD) } ;

  TBSCertificateLogEntry  ::=  SEQUENCE  {
        version             [0]  EXPLICIT Version DEFAULT v1,
        issuer                   Name,
        validity                 Validity,
        subject                  Name,
        subjectPublicKeyInfoHash OCTET STRING,
        issuerUniqueID      [1]  IMPLICIT UniqueIdentifier OPTIONAL,
        subjectUniqueID     [2]  IMPLICIT UniqueIdentifier OPTIONAL,
        extensions          [3]  EXPLICIT Extensions{{CertExtensions}} OPTIONAL }

  id-alg-mtcProof OBJECT IDENTIFIER ::= {
      iso(1) identified-organization(3) dod(6) internet(1) security(5)
      mechanisms(5) pkix(7) algorithms(6) TBD}

  sa-mtcProof SIGNATURE-ALGORITHM ::= {
     IDENTIFIER id-alg-mtcProof
     PARAMS ARE absent
  }

  id-rdna-trustAnchorID OBJECT IDENTIFIER ::= {
      iso(1) identified-organization(3) dod(6) internet(1) security(5)
      mechanisms(5) pkix(7) rdna(25) TBD}

  at-trustAnchorID ATTRIBUTE ::= {
     TYPE TrustAnchorID
     IDENTIFIED BY id-rdna-trustAnchorID
  }

  END
```

Appendix B. Merkle Tree Structure

This non-normative section describes how the Merkle Tree structure relates to the binary representations of indices. It is included to help implementors understand the procedures described in Section 4.

B.1. Binary Representations

Within a Merkle Tree whose size is a power of two, the binary representation of an leaf's index gives the path to that leaf. The leaf is a left child if the least-significant bit is unset and a right child if it is set. The next bit indicates the direction of the parent node, and so on. Figure 12 demonstrates this in a Merkle Tree of size 8:

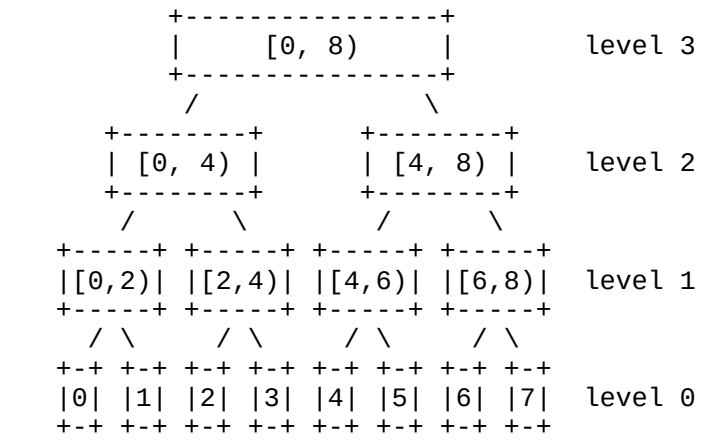


Figure 12: An example Merkle Tree of size 8

The binary representation of 4 is 0b100. It is the left (0) child of [4, 6), which is the left (0) child of [4, 8), which is the right (1) child of [0, 8).

Each level in the tree corresponds to a bit position and can be correspondingly numbered, with 0 indicating the least-significant bit and the leaf level, and so on. In this numbering, a node's level can be determined as follows: if the node is a root of subtree [start, end), the node's level is BIT_WIDTH(end - start - 1).

Comparing two indices determines the relationship between two paths. The highest differing bit gives the level at which paths from root to leaf diverge. For example, the bit representations of 4 and 6 are 0b100 and 0b110, respectively. The highest differing bit is bit 1. Bits 2 and up are the same between the two indices. This indicates that the paths from the root to leaves 4 and 6 diverge when going to level 2 to level 1.

This can be generalized to arbitrary-sized Merkle Trees. Figure 13 depicts a Merkle Tree of size 6:

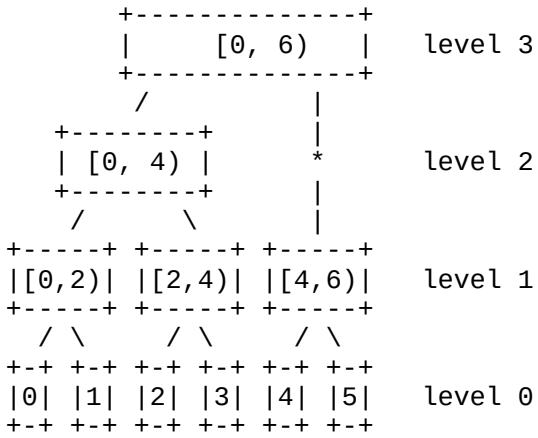


Figure 13: An example Merkle Tree of size 6

When the size of a Merkle Tree is not a power of two, some levels on the rightmost edge of the tree are skipped. The rightmost edge is the path to the last element. The skipped levels can be seen in its binary representation. Here, the last element is 5, which has binary representation 0b101. When a bit is set, the corresponding node is a right child. When it is unset, the corresponding node is skipped.

In a tree of the next power of two size, the skipped nodes in this path are where there _would_ have been a right child, had there been enough elements to construct one. Without a right child, the hash operation is skipped and a skipped node has the same value as its singular child. Figure 14 depicts this for a tree of size 6.

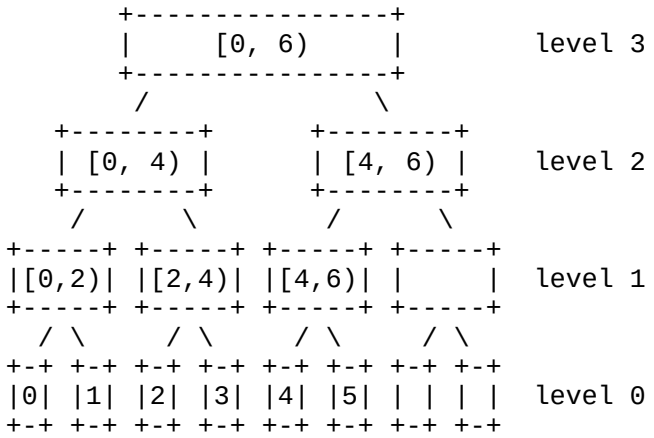


Figure 14: An example Merkle Tree of size 6, viewed as a subset of a tree of size 8

Zero bits also indicate skipped nodes in paths that have not yet diverged from the rightmost edge (i.e. the path to the last element), when viewed from root to leaf. In the example, the binary representation of 4 is 0b100. While bit 0 and bit 1 are both unset, they manifest in the tree differently. Bit 0 indicates that 4 is a right child. However, at bit 1, 0b100 has not yet diverged from the last element, 0b101. That instead indicates a skipped node, not a left child.

B.2. Inclusion Proof Evaluation

The procedure in Section 4.3.2 builds up a subtree hash in r by staring from entry_hash and iteratively hashing elements of inclusion_proof on the left or right. That means this procedure, when successful, must return _some_ hash that contains entry_hash.

Treating [start, end) as a Merkle Tree of size end - start, the procedure hashes by based on the path to index. Within this smaller Merkle Tree, it has index fn = index - start (first number), and the last element has index sn = end - start - 1 (second number).

Step 4 iterates through inclusion_proof and the paths to fn and sn in parallel. As the procedure right-shifts fn and sn and looks at the least-significant bit, it moves up the two paths, towards the root. When sn is zero, the procedure has reached the top of the tree. The procedure checks that the two iterations complete together.

Iterating from level 0 up, fn and sn will initially be different. While they are different, step 4.2 hashes on the left or right based on the binary representation, as discussed in Appendix B.1.

Once fn = sn, the remainder of the path is on the right edge. At that point, the condition in step 4.2 is always true. It only incorporates proof entries on the left, once per set bit. Unset bits are skipped.

Inclusion proofs can also be evaluated by considering these two stages separately. The first stage consumes l1 = BIT_WIDTH(fn XOR sn) proof entries. The second stage consumes l2 = POPCOUNT(fn >> l1) proof entries. A valid inclusion proof must then have l1 + l2 entries. The first l1 entries are hashed based on fn’s least significant bits, and the remaining l2 entries are hashed on the left.

B.3. Consistency Proof Structure

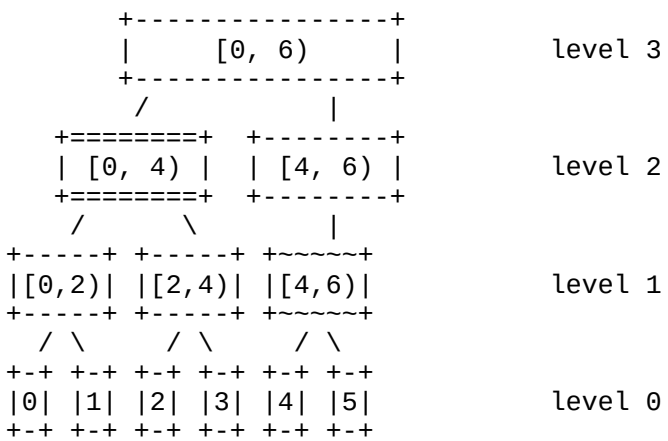
A subtree consistency proof for [start, end) and the tree of n elements is similar to an inclusion proof for element end - 1. If one starts from end - 1's hash, incorporating the whole inclusion proof should reconstruct root_hash and incorporating a subset of the inclusion proof should reconstruct node_hash. Thus end - 1's hash and this inclusion proof can prove consistency. A subtree consistency proof in this document applies two optimizations over this construction:

1. Instead of starting at level 0 with end - 1, the proof can start at a higher level. Any ancestor of end - 1 shared by both the subtree and the overall tree is a valid starting node to reconstruct node_hash and root_hash. Use the highest level with a commmon ancestor. This truncates the inclusion proof portion of the consistency proof.

2. If this starting node is the entire subtree, omit its hash from the consistency proof. The verifier is assumed to already know node_hash.

A Merkle consistency proof, defined in Section 2.1.4 of [RFC9162], applies these same optimizations.

Figure 15 depicts a subtree consistency proof between the subtree [0, 6) and the Merkle Tree of size 8. The consistency proof begins at level 1, or node [4, 6). The inclusion proof portion is similarly truncated to start at level 1: [6, 8) and [0, 4). If the consistency proof began at level 0, the starting node would be leaf 5, and the consistency proof would additionally include leaf 4.



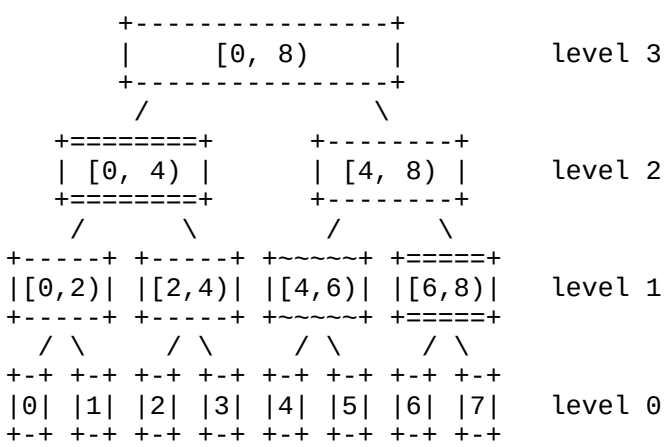
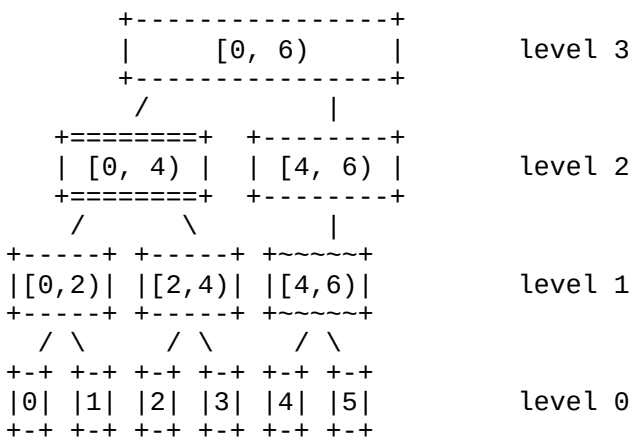


Figure 15: A subtree consistency proof that starts at level 1 instead of level 0

Note that the truncated inclusion proof may include nodes from lower levels, if the corresponding level was skipped on the right edge. Figure 16 depicts a subtree consistency proof between the subtree [0, 6) and the Merkle Tree of size 7. As above, the starting node is [4, 6) at level 1. The inclusion proof portion includes leaf 6 at level 0. This is because leaf 6 is taking the place of its skipped parent at level 1. (A skipped node can be thought of as a duplicate of its singular child.)



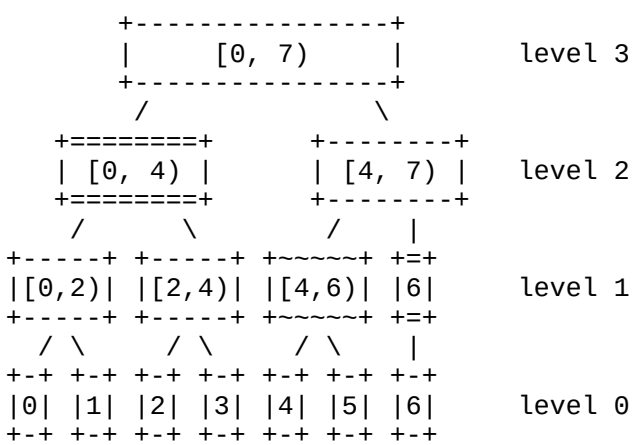


Figure 16: The interaction between inclusion proof truncation and skipped levels

B.4. Consistency Proof Verification

The procedure in Section 4.4.3 is structured similarly to inclusion proof evaluation (Appendix B.2). It iteratively builds two hashes, fr and sr, which are expected to equal node_hash and root_hash, respectively. Everything hashed into fr is also hashed into sr, so success demonstrates that root_hash contains node_hash.

Step 2 initializes fn (first number), sn (second number), and tn (third number) to follow, respectively, the paths to start, end - 1 (the last element of the subtree), and n - 1 (the last element of the tree).

Steps 3 and 4 then skip to the starting node, described in Appendix B.3. The starting node may be:

* The entire subtree [start, end) if [start, end) is directly contained in the tree. This will occur if end is n (step 3), or if [start, end) is full (exiting step 4 because fn is sn).

* Otherwise, the highest full subtree along the right edge of [start, end). This corresponds to the process exiting step 4 because LSB(sn) is not set.

Steps 5 and 6 initialize the hashes fr and sr:

* In the first case above, fn will equal sn after truncation. Step 5 will then initialize the hashes to node_hash because consistency proof does not need to include the starting node.

* In the second case above, fn is less than sn. Step 6 will then initialize the hashes to the first value in the consistency proof.

Step 7 incorporates the remainder of the consistency proof into fr and sr:

* All hashes are incorporated into sr, with hashing on the left or right determined the same as in inclusion proof evaluation.

* A subset of the hashes are incorporated into fr. It skips any hash on the right because those contain elements greater than end - 1. It also stops incorporating when fn and sn have converged.

This reconstructs the hashes of the subtree and full tree, which are then compared to expected values in step 8.

In the case when fn is sn in step 5, the condition in step 7.2.1 is always false, and fr is always equal to node_hash in step 8. In this case, steps 6 through 8 are equivalent to verifying an inclusion proof for the truncated subtree [fn, sn + 1) and truncated tree tn + 1.

Appendix C. Extensions to Tiled Transparency Logs (To Be Removed)

[[TODO: This section is expected to be removed. It is sketched here purely for illustrative purposes, until the features are defined somewhere else, e.g. in the upstream tlog documents.]]

C.1. Subtree Signed Note Format

A subtree, with signatures, can be represented as a signed note [SIGNED-NOTE]. Trust anchor IDs can be converted into log origins and cosigner names by concatenating the ASCII string oid/1.3.6.1.4.1. and the ASCII representation of the trust anchor ID. For example, the checkpoint origin for a log named 32473.1 would be oid/1.3.6.1.4.1.32473.1.

The note body is a sequence of the following lines, each terminated by a newline character (U+000A):

* The log origin

* Two space-separated, non-negative decimal integers, <start> <end>

* The subtree hash, as single hash encoded in base64

Each note signature has a key name of the cosigner name. The signature's key ID is computed using the reserved signature type in [SIGNED-NOTE], and a fixed string, as follows:

key ID = SHA-256(key name || 0x0A || 0xFF || "mtc-subtree/v1")[:4]

A subtree whose start is zero can also be represented as a checkpoint [TLOG-CHECKPOINT]. A corresponding subtree signature can be represented as a note signature using a key ID computed as follows:

key ID = SHA-256(key name || 0x0A || 0xFF || "mtc-checkpoint/v1")[:4]

The only difference between the two forms is the implicit transformation from the signed note text to the MTCSubtree structure.

C.2. Requesting Subtree Signatures

This section defines the sign-subtree cosigner HTTP endpoint for clients to obtain subtree signatures from non-CA cosigners, such as mirrors and witnesses. It may be used by the CA when assembling a certificate, or by an authenticating party to add a cosignature to a certificate that the CA did not themselves obtain.

The cosigner MAY expose this endpoint publicly to general authenticating parties, or privately to the CA. The latter is sufficient if the CA is known to automatically request cosignatures from this cosigner when constructing certificates. If private, authenticating the CA is out of scope for this document.

Clients call this endpoint as POST <prefix>/sign-subtree, where prefix is some URL prefix. For a mirror or witness, the URL prefix is the submission prefix. The client's request body MUST be a sequence of:

* The requested subtree as a signed note (Appendix C.1), with zero or more signatures. The endpoint MAY require signatures from the CA as a DoS mitigation, as described below.

* A blank line

* A checkpoint, signed by the requested cosigner. The checkpoint's tree size must be at least end.

* A blank line

* Zero or more subtree consistency proof (Section 4.4) lines. Each line MUST encode a single hash in base64 [RFC4648]. The client MUST NOT send more than 63 consistency proof lines.

Each line MUST terminate in a newline character (U+000A).

The cosigner performs the following steps:

1. Check that the checkpoint contains signatures from itself

2. Check that the subtree consistency proof proves consistency between the subtree hash and the checkpoint

3. If all checks pass, cosign the subtree, as described in Section 5.4

On success, the response body MUST be a sequence of one or more note signature lines [SIGNED-NOTE], each starting with an em dash character (U+2014) and ending with a newline character (U+000A). The signatures MUST be cosignatures from the cosigner key(s) on the subtree.

Instead of statelessly validating checkpoints by signature, the cosigner MAY statefully check the requested checkpoint against internal witness or mirror state. In this case, if the cosigner needs a newer checkpoint, it responds with a "409 Conflict" with its latest signed checkpoint. In this case, the subtree cosigning SHOULD remember and accept the last few signed checkpoints, to minimize conflicts.

If operating statefully, the subtree cosigner process only needs read access to the mirror or witness state and can freely operate on stale state without violating any invariants.

Mirrors MAY choose to check subtree hashes by querying their log state, instead of evaluating proofs.

Publicly-exposed subtree cosigning endpoints MAY mitigate DoS in a variety of techniques:

* Only cosigning recent subtrees, as old subtrees do not need to be co-signed
* Caching subtree signatures
* Requiring a CA signature on the subtree; CAs are only expected to sign two subtrees (Section 4.5) for each checkpoint
* Rate-limiting requests

Acknowledgements

This document stands on the shoulders of giants and builds upon decades of work in TLS authentication, X.509, and Certificate Transparency. The authors would like to thank all those who have contributed over the history of these protocols.

The authors additionally thank Bob Beck, Ryan Dickson, Aaron Gable, Nick Harper, Russ Housley, Dennis Jackson, Matt Mueller, Chris Patton, Ryan Sleevi, and Emily Stark for many valuable discussions and insights which led to this document, as well as feedback on the document itself. We wish to thank Mia Celeste in particular, whose implementation of an earlier draft revealed several pitfalls.

The idea to mint tree heads infrequently was originally described by Richard Barnes in [STH-Discipline]. The size optimization in Merkle Tree Certificates is an application of this idea to the certificate itself.

Change log

> *RFC Editor’s Note:* Please remove this section prior to publication of a final version of this document.

Since draft-davidben-tls-merkle-tree-certs-00

* Simplify hashing by removing the internal padding to align with block size. #72

* Avoid the temptation of floating points. #66

* Require lifetime to be a multiple of batch_duration. #65

* Rename window to validity window. #21

* Split Assertion into Assertion and AbridgedAssertion. The latter is used in the Merkle Tree and HTTP interface. It replaces subject_info by a hash, to save space by not serving large post-quantum public keys. The original Assertion is used everywhere else, including BikeshedCertificate. #6

* Add proper context to every node in the Merkle Tree. #32

* Clarify we use a single CertificateEntry. #11

* Clarify we use POSIX time. #1

* Elaborate on CA public key and signature format. #27

* Miscellaneous changes.

Since draft-davidben-tls-merkle-tree-certs-01

* Minor editorial changes

Since draft-davidben-tls-merkle-tree-certs-02

* Replace the negotiation mechanism with TLS Trust Anchor Identifiers.

Since draft-davidben-tls-merkle-tree-certs-03

* Switch terminology from "subscriber" to "authenticating party".

* Use <1..2^24-1> encoding for all certificate types in the CertificateEntry TLS message

* Clarify discussion and roles in transparency ecosystem

* Update references

Since draft-davidben-tls-merkle-tree-certs-04

Substantially reworked the design. The old design was essentially the landmark checkpoint and CA-built logs ideas, but targeting only the optimized and slow issuance path, and with a more bespoke tree structure:

In both draft-04 and draft-05, a CA looks like today's CAs except that they run some software to publish what they issue and sign tree heads to certify certificates in bulk.

In draft-04, the CA software publishes certificates in a bunch of independent Merkle Trees. This is very easy to do as a collection of highly cacheable, immutable static files because each tree is constructed independently, and never appended to after being built. In draft-05, the certificates are published in a single Merkle Tree. The [TLOG-TILES] interface allows such trees to also use highly cacheable, immutable static files.

In draft-04, there only are hourly tree heads. Clients are provisioned with tree heads ahead of time so we can make small, inclusion-proof-only certificates. In draft-05, the ecosystem must coordinate on defining "landmark" checkpoints. Clients are provisioned with subtrees describing landmark checkpoints ahead of time so we can make small, inclusion-proof-only certificates.

In draft-04, each tree head is independent. In draft-05, each landmark checkpoint contains all the previous checkpoints.

In draft-04, the independent tree heads were easily prunable. In draft-05, we define how to prune a Merkle Tree.

In draft-04, there is no fast issuance mode. In draft-05, frequent, non-landmark checkpoints can be combined with inclusion proofs and witness signatures for fast issuance. This is essentially an STH and inclusion proof in CT.

Since draft-davidben-tls-merkle-tree-certs-05

* Add some discussion on malleability

* Discuss the monitoring impacts of the responsibility shift from CA with log quorum to CA+log with mirror quorum

* Sketch out a more concrete initial ACME extension

Since draft-davidben-tls-merkle-tree-certs-06

* Fix mistyped reference

* Removed now unnecessary placeholder text

* First draft at IANA registration and ASN.1 module

* Added a prose version of the procedure to select subtrees

* Rename ’landmarks checkpoint’ to ’landmarks’
* Clarify and fix an off-by-one error in recommended landmark allocation scheme
* Add some diagrams to the Overview section

Since draft-davidben-tls-merkle-tree-certs-07

* Clarify landmark zero
* Clarify signature verification process
* Improve subtree consistency proof verification algorithm
* Add an appendix that explains the Merkle Tree proof procedures

Since draft-davidben-tls-merkle-tree-certs-08

* Improvements to malleability discussion
* Improvements to subtree definition
* Improvements to trust_anchors integration

Since draft-davidben-tls-merkle-tree-certs-09

* Editorial fixes
* Set a more accurate intended status
* Fixes to ASN.1 module
* Make log entry more friendly to single-pass verification

Authors’ Addresses

David Benjamin
Google LLC
Email: davidben@google.com

Devon O’Brien
Email: devon.obrien@gmail.com

Bas Westerbaan
Cloudflare

Email: bas@cloudflare.com

Luke Valenta
Cloudflare
Email: lvalenta@cloudflare.com

Filippo Valsorda
Geomys
Email: ietf@filippo.io