PKI, Logs, And Tree Signatures
Internet-Draft
Intended status: Standards Track
Expires: 25 November 2026

D. Benjamin
Google LLC
D. O'Brien
Apple Inc.
B. E. Westerbaan
L. Valenta
Cloudflare
F. Valsorda
Geomys

24 May 2026

Merkle Tree Certificates
draft-ietf-plants-merkle-tree-certs-04

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 existing X.509 constructions and Certificate Transparency. Merkle Tree certificates additionally admit an optional size optimization that avoids 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://ietf-plants-wg.github.io/merkle-tree-certs/draft-ietf-plants-merkle-tree-certs.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-plants-merkle-tree-certs/.

Discussion of this document takes place on the PKI, Logs, And Tree Signatures 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/ietf-plants-wg/merkle-tree-certs.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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 (MTCs), a new form of X.509 certificate that integrates logging with certificate issuance. Each CA maintains logs of everything it issues, signing views of its logs 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 logs. 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 "landmark-relative" 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 5.2 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]. It extends the numeric types defined in Section 3.3 of [RFC8446] with a big-endian, 48-bit integer:

```
uint8 uint48[6];
```

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, containing certification statements issued by that CA. A CA operates some number of issuance logs, which together contain all statements 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 landmark-relative certificates.

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

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

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

Directly-signed certificate: A certificate issued using the existing, non-MTC construction, where the TBSCertificate is passed directly to the private key’s signing operation.

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.

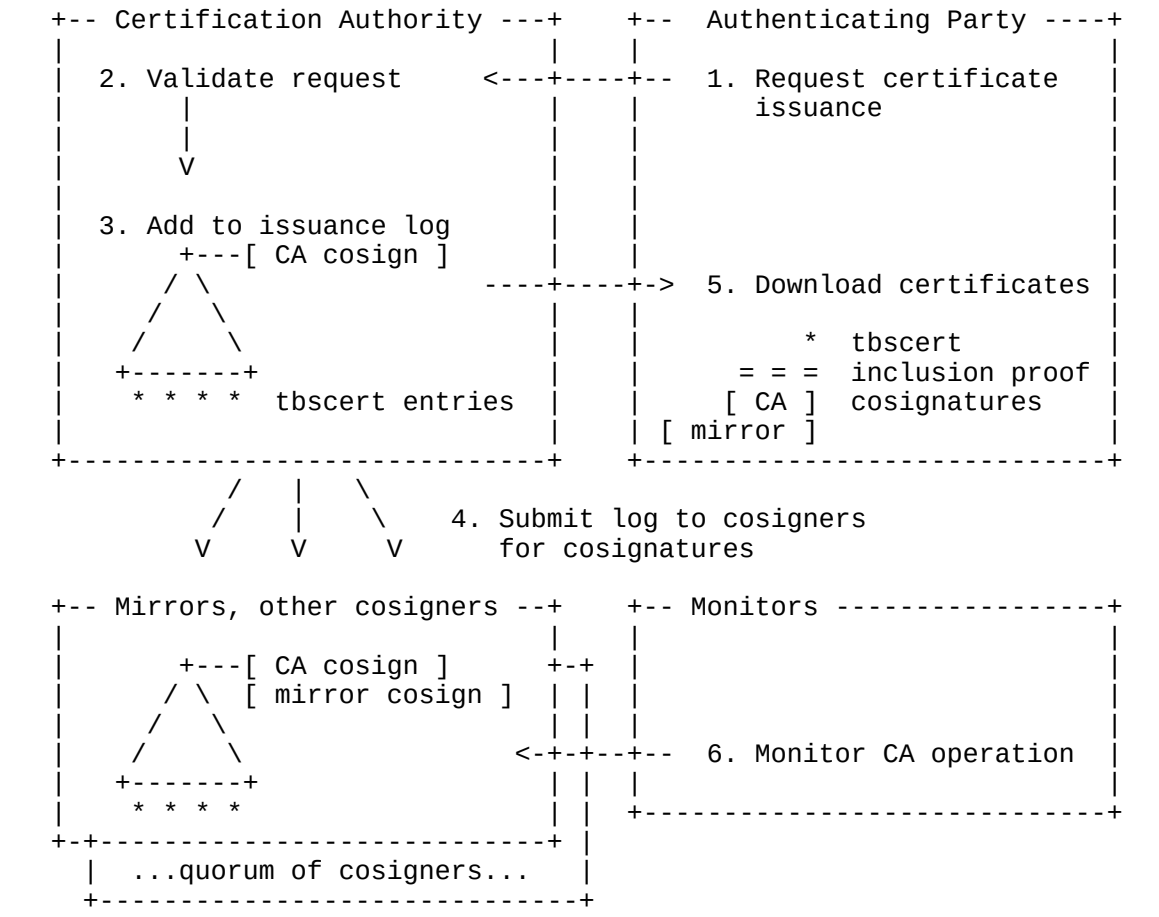


Figure 1: A diagram of the MTC issuance architecture, detailed below

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 a series of append-only _issuance logs_ (Section 5.2). Unlike a CT log, these logs only contain entries added by the CA:

   a. The CA adds a TBSCertificateLogEntry (Section 5.2.1, abbreviated "tbscert entries" in the diagram) to an issuance log, describing the information it is certifying.

   b. 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 5.4).

   c. 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 CA's issuance logs to ensure the CA is operated correctly.

A certificate with cosignatures is known as a _standalone 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 5.2.1) structures, which do not scale with public key or signature size.

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

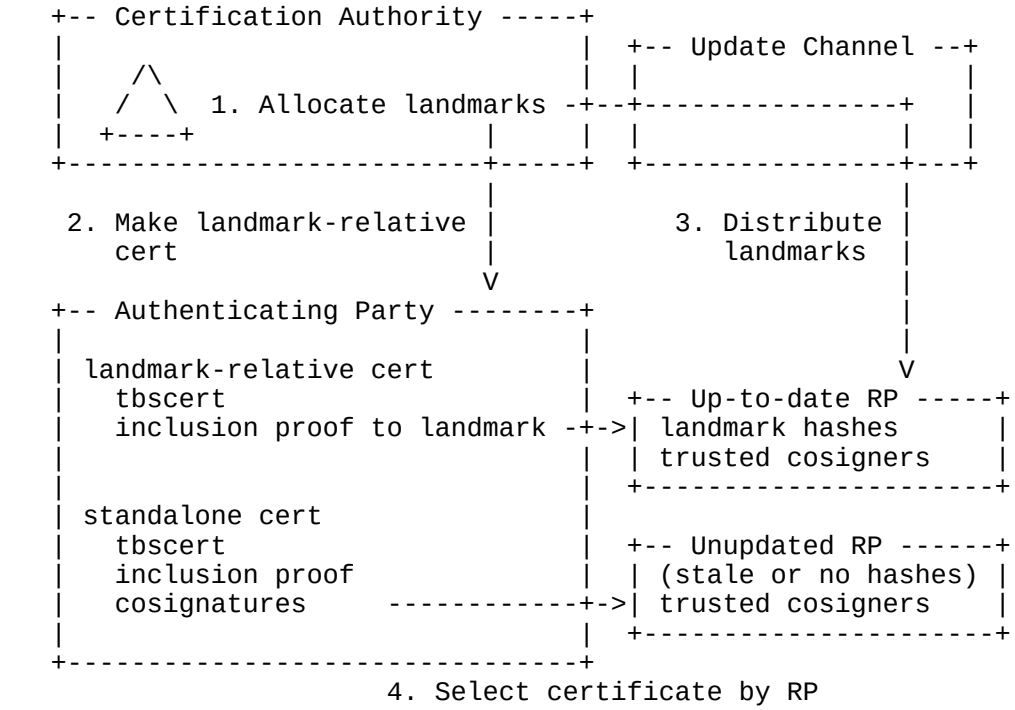


Figure 2: A diagram of landmark-relative certificate construction and usage, detailed below

Landmark-relative 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 landmark-relative certificates.

2. Once some landmark includes the TBSCertificate, the landmark-relative 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 landmark-relative certificate. Otherwise, it presents a standalone certificate. The authenticating party may also select between several landmark-relative 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 itself a Merkle Tree, built over an interval of entries from the original tree. Section 4.1 defines a subtree formally, including the constraints on those intervals.

As with Merkle Trees, a subtree inclusion proof, defined in Section 4.3, can prove an entry is contained in some subtree. Subtrees, and thus their inclusion proofs, are smaller than those of the original tree, so this document uses subtree inclusion proofs as a certificate size optimization.

Not all intervals can form subtrees. Subtrees are limited to intervals that can be efficiently proven consistent with the original tree, using subtree consistency proofs defined in Section 4.4. However, every interval of a Merkle Tree can be efficiently covered by two subtrees. Section 4.5 describes how to determine these subtrees.

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):

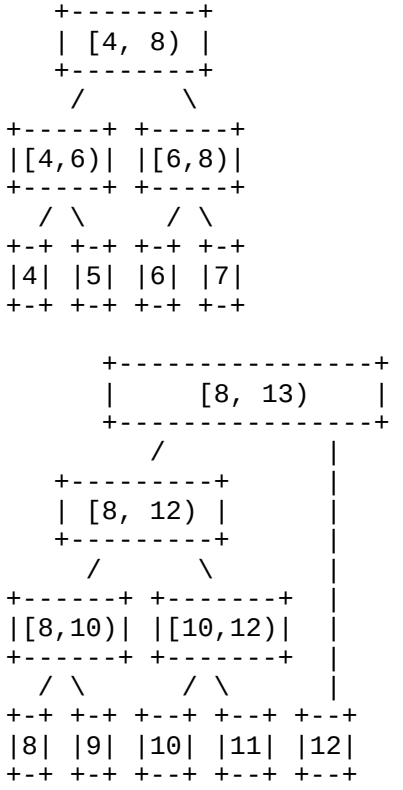


Figure 3: Two example subtrees, one full and one partial

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

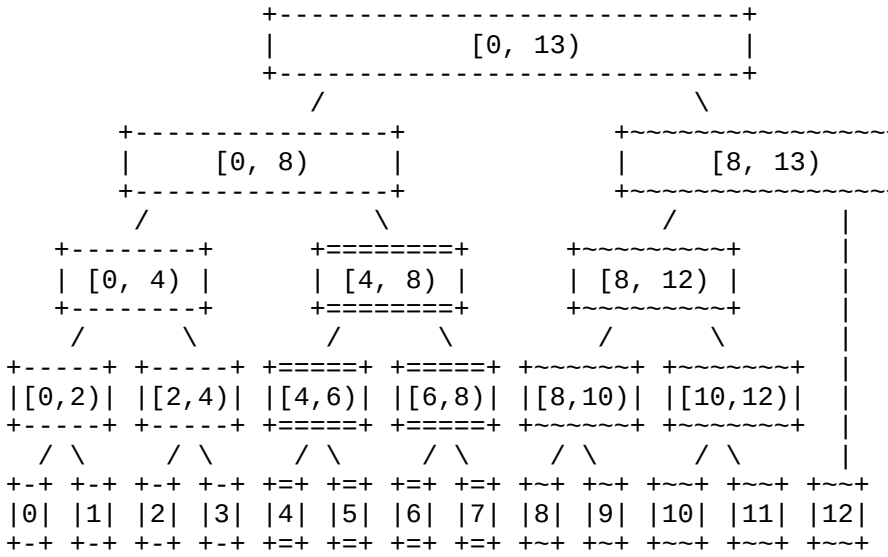


Figure 4: A Merkle Tree of size 13

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.

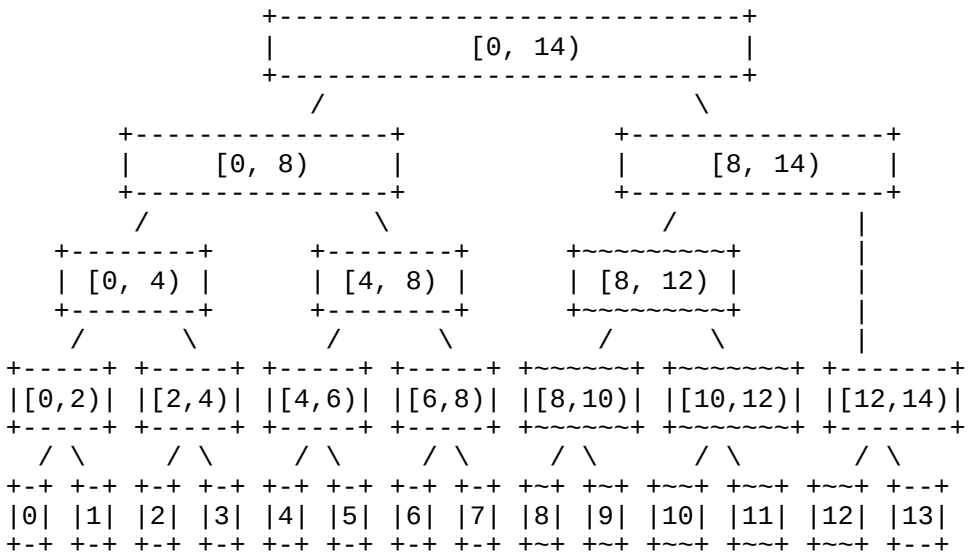


Figure 5: A Merkle Tree of size 14

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.

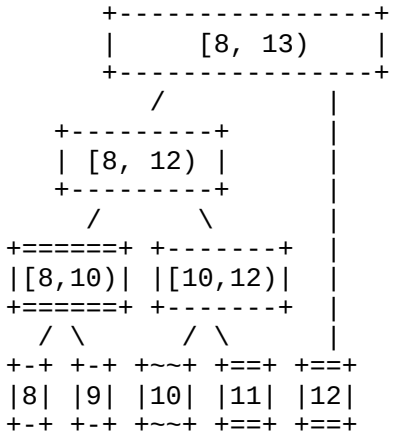


Figure 6: An example subtree inclusion proof

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.

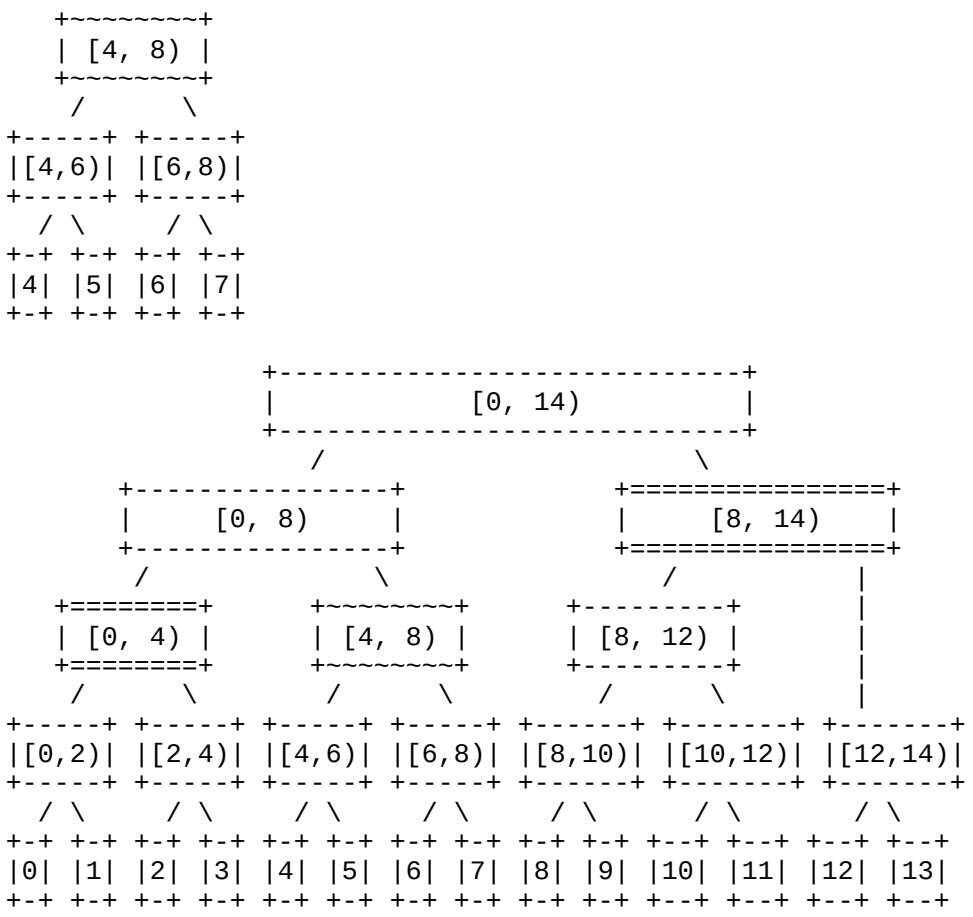
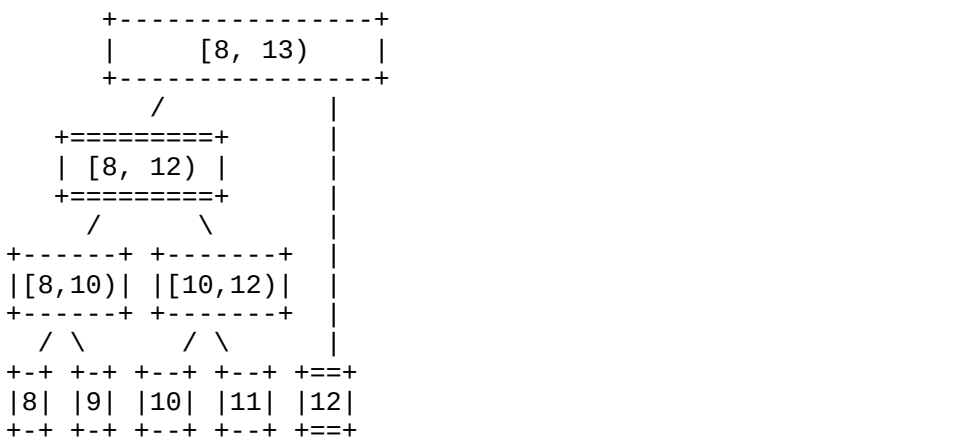


Figure 7: An example subtree consistency proof for a subtree that is directly contained in the full tree

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.



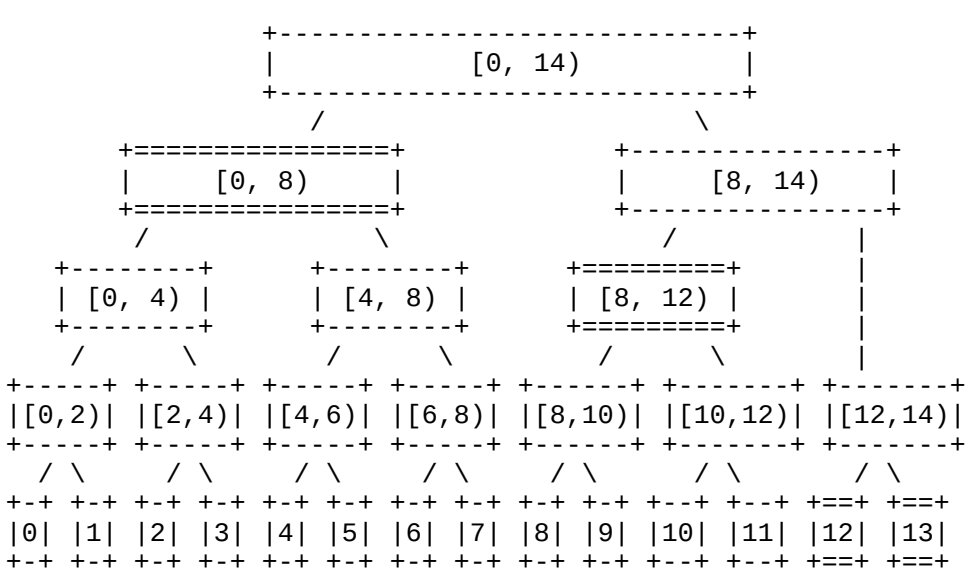


Figure 8: An example subtree consistency proof for a subtree that is not directly contained in the full tree

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).

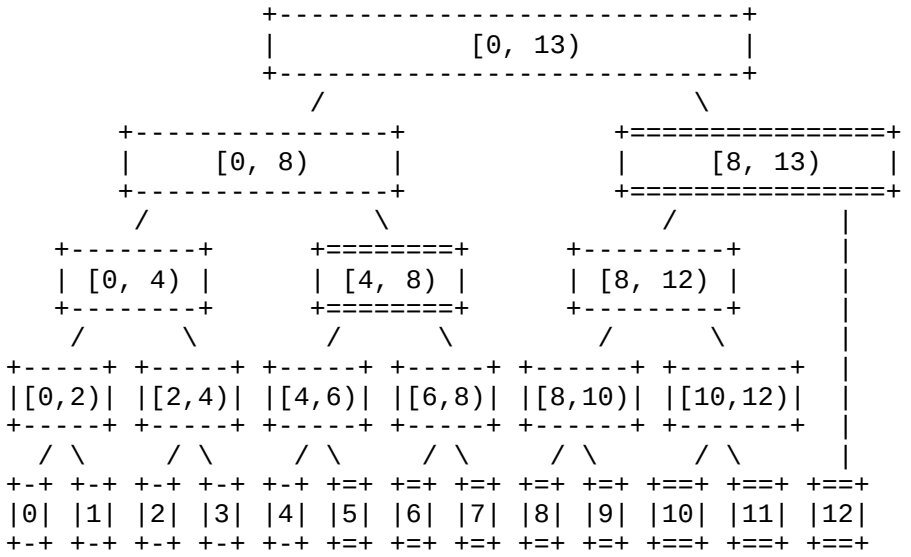


Figure 9: An example selection of subtrees to cover an interval

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).

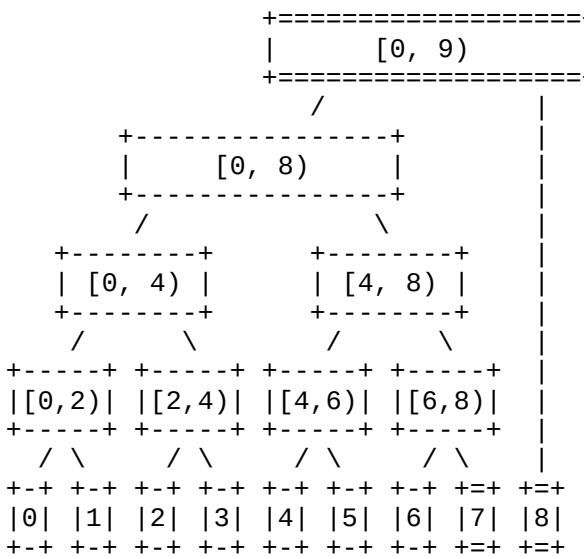


Figure 10: An example showing an inefficient choice of a single subtree

5. Certification Authorities

A CA consists of the following components:

* A CA ID (Section 5.1), which uniquely identifies the CA.

* A collision-resistant cryptographic hash function, used by the CA's issuance logs. SHA-256 [SHS] is RECOMMENDED. Throughout this document, this hash function is referred to as HASH, and the size of its output in bytes is referred to as HASH_SIZE.

* A series of issuance logs (Section 5.2), which contain all statements the CA has certified.

* A CA cosigner (Section 5.4), which signs subtrees of issuance logs to certify their contents.

* Optionally, a landmark sequence per log (Section 6.3.1), to support optimized landmark-relative certificates.

Section 5.5 defines an X.509 certificate representation of a CA.

5.1. Certification Authority Identifiers

Each Merkle Tree Certificate CA has a _CA ID_ to identify it. This CA ID is a trust anchor ID [I-D.ietf-tls-trust-anchor-ids].

Once allocated, the ID's entire object identifier (OID) arc is reserved by this protocol. Given a CA ID whose OID representation is caID, this document allocates the following OIDs:

* For each positive integer N, the OID {caID logs(0) N} represents the issuance log N (Section 5.2).

* For each positive integer N and L, the OID {caID landmarks(1) N L} represents landmark L (Section 6.3.1) of issuance log N. These OIDs may be used as trust anchor IDs, as described in Section 8.2. These OIDs are used when it is necessary to identify an individual landmark, e.g. as in the retry mechanism described Section 4.3 of [I-D.ietf-tls-trust-anchor-ids].

* For each positive integer N and L, the OID {caID landmarkGroups(2) N L} represents a trust anchor group (Section 5 of [I-D.ietf-tls-trust-anchor-ids]) containing landmark L of log N and earlier landmarks of that log, as defined in Section 8.2.1. These OIDs may be used to advertise a series of landmarks at once.

Future extensions to this protocol MAY define further allocations.

A CA ID determines a PKIX distinguished name (Section 4.1.2.4 of [RFC5280]) that can be used in the issuer or subject field of an X.509 TBSCertificate. This 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 CA with ID 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. 32473.1.

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 CA with ID 32473.1 would be represented in syntax of [RFC4514] as:

```
1.3.6.1.4.1.44363.47.1=#0c0733323437332e31
```

5.2. Issuance Logs

A CA operates a series of issuance logs, each identified by a positive integer _log number_. Log numbers are numbered consecutively starting from 1. Each log number MUST be at most 65535 (2^16-1).

Each issuance log has a _log ID_, which is a trust anchor ID constructed by concatenating the following OID components:

* The CA ID (Section 5.1)

* The constant 0

* The log number of the log

A log ID specifies both the CA and the log number in a single ID.

Each issuance log describes an append-only sequence of _entries_ (Section 5.2.1), 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].

Each log additionally maintains a _minimum index_ value, which is the index of the first log entry which is available. See Section 5.2.3. This value changes over the lifetime of the log.

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 committed 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.

5.2.1. 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 { (2^16-1) } MerkleTreeCertEntryExtensionType;

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} MerkleTreeCertEntryExtension;

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

struct {
    MerkleTreeCertEntryExtension extensions<0..2^16-1>;
    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;
```

Field extensions is the list of tag-length-value extensions associated with the log entry. The extensions list MUST be appear in ascending order by extension_type and MUST NOT contain two extensions with the same extension_type.

When type is null_entry, the entry does not represent any information. Entries at any index in the log MAY have type null_entry.

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,
    subjectPublicKeyAlgorithm AlgorithmIdentifier{PUBLIC-KEY,
                                  {PublicKeyAlgorithms}},
    subjectPublicKeyInfoHash  OCTET STRING,
    issuerUniqueID        [1] IMPLICIT UniqueIdentifier OPTIONAL,
    subjectUniqueID       [2] IMPLICIT UniqueIdentifier OPTIONAL,
    extensions            [3] EXPLICIT Extensions{{CertExtensions}}
                                       OPTIONAL
}
```

The fields of a TBSCertificateLogEntry are defined as follows:

* version, validity, subject, issuerUniqueID, subjectUniqueID, and extensions have the same semantics as the corresponding TBSCertificate fields, defined in Section 4.1.2 of [RFC5280].

* issuer is the CA ID as a PKIX distinguished name, as described in Section 5.1.

* subjectPublicKeyAlgorithm describes the algorithm of the subject's public key. It is constructed identically to the algorithm field of a SubjectPublicKeyInfo (Section 4.1.2.7 of [RFC5280]).

* subjectPublicKeyInfoHash contains the hash of subject's public key, encoded as a SubjectPublicKeyInfo. The hash uses the CA's hash function (Section 5) and is computed over the SubjectPublicKeyInfo's DER [X.690] encoding.

Note the subject's public key algorithm is incorporated into both subjectPublicKeyAlgorithm and subjectPublicKeyInfoHash.

MerkleTreeCertEntry is an extensible structure. Future documents may define new values for MerkleTreeCertEntryType or MerkleTreeCertEntryExtensionType, with corresponding semantics. See Section 5.4 and Section 12.5 for additional discussion.

A MerkleTreeCertEntry's size SHOULD NOT exceed 65535 (2^16-1) bytes. Doing so may exceed size limits in common log-serving protocols, such as [TLOG-TILES]. TBSCertificateLogEntry does not include signatures and hashes public keys, so post-quantum algorithms do not contribute to this size.

5.2.2. Publishing Logs

This protocol aims to enable monitors to detect misissued certificates by observing the issuance log. See Section 12.2.

This document does not prescribe a particular method of observing the issuance log. The access protocols do not affect certificate interoperability, and different applications may have different needs. For example, a PKI that authenticates public services might publicly serve issuance logs, while a PKI that authenticates a single organization's intranet services might keep the log private to the organization. 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.

5.2.3. Log Pruning

Over time, an issuance log's entries will expire and likely be replaced as certificates are renewed. 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 5.2). The minimum index is the index of the first log entry that the log publishes. (See Section 5.2.2.) 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 5.4)

* Any individual available log entry (Section 5.2.1)

* 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.

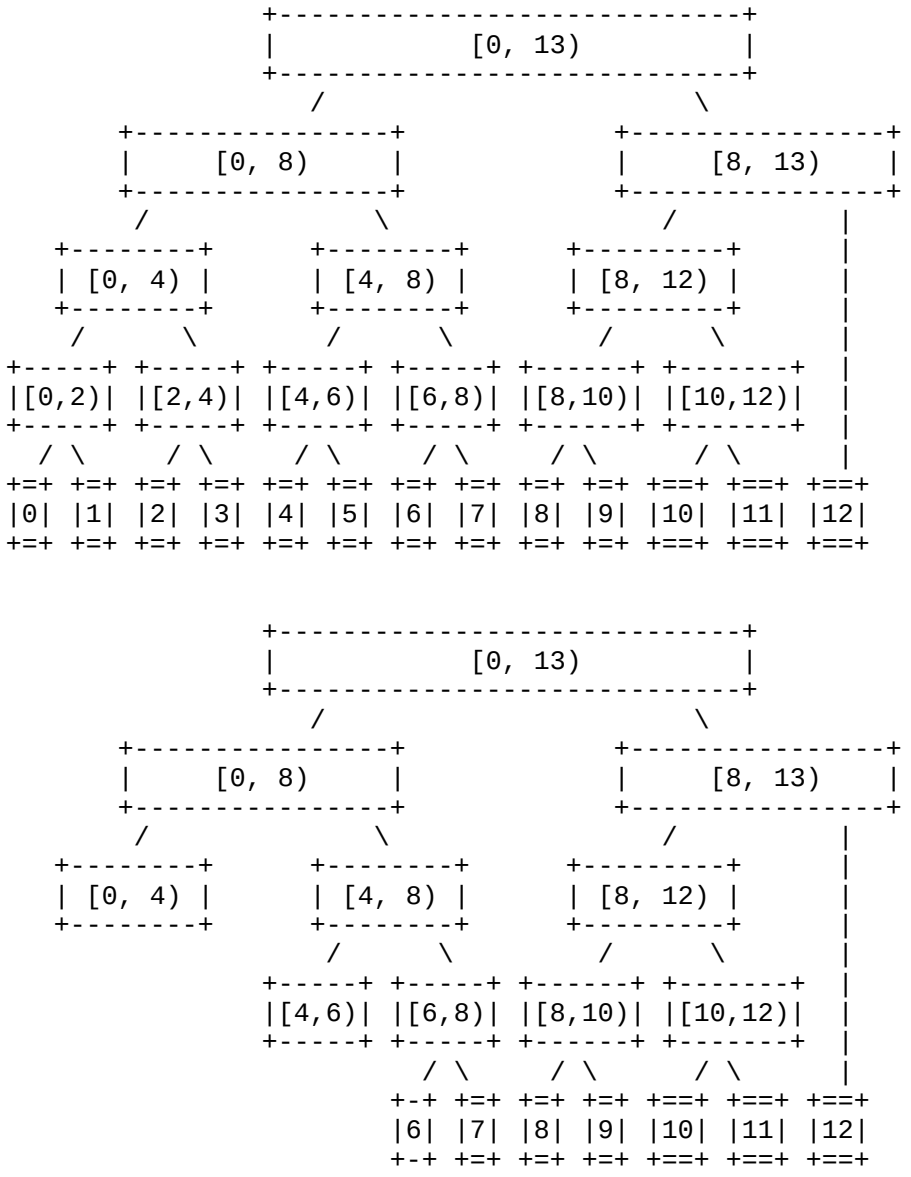


Figure 11: An example showing the minimum nodes that must be available after pruning

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.

5.3. 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 5.4), 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 the public key, signature algorithm, and any 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.

Following the principle of key separation [KeyReuse], cosigner keys SHOULD NOT be used for purposes outside this document. Additional uses MAY be defined but MUST NOT overlap with the signature format defined in Section 5.3.1. See Section 12.8 for additional discussion.

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.

5.3.1. Signature Format

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

```
opaque HashValue[HASH_SIZE];

struct {
    uint8 label[12] = "subtree/v1\n\0";
    opaque cosigner_name<1..2^8-1>;
    uint64 timestamp;
    opaque log_origin<1..2^8-1>;
    uint64 start;
    uint64 end;
    HashValue subtree_hash;
} CosignedMessage;
```

This signature format is designed to be compatible with the ML-DSA-44 signature construction in [TLOG-COSIGNATURE], but it supports signature algorithms other than ML-DSA-44 and tree hashes other than SHA-256.

label is a fixed prefix for domain separation. Its value MUST be the string subtree/v1, followed by a newline (U+000A), followed by a zero byte (U+0000).

cosigner_name and log_origin are computed from the cosigner ID and the issuance log's ID (Section 5.1), respectively. They contain the concatenation of:

* The 16-byte ASCII string oid/1.3.6.1.4.1.
* The trust anchor ID's ASCII representation (Section 3 of [I-D.ietf-tls-trust-anchor-ids])

This is equivalent to the concatenation of:

* The four-byte ASCII string oid/
* The trust anchor ID as a full OID, in dotted decimal notation

For example, the trust anchor ID 32473.1 would be encoded as the ASCII string oid/1.3.6.1.4.1.32473.1.

start and end MUST define a valid subtree of the log, and subtree_hash MUST be the subtree's hash value in the cosigner's view of the log.

If timestamp is non-zero, it MUST be the time that the signature was produced. This time is represented as seconds since the Epoch, as defined in Section 4.19 of Volume 1 of [POSIX]. Additionally, if timestamp is non-zero, the following MUST be true:

* start MUST be zero.

* end MUST be the size of the largest consistent tree that the cosigner has observed for the log.

timestamp MAY be zero, in which case no additional constraints are placed on start or end, and no statement is made about the signing time or largest observed tree.

5.3.2. Signature Semantics

Before signing a subtree of some log, the cosigner MUST ensure that subtree_hash is consistent with its view of the log. Different cosigner roles may obtain this assurance differently. For example:

* A cosigner may maintain a full copy of the log, e.g. if it's the log operator. The cosigner can then compute subtree_hash from this copy.

* A cosigner may maintain the hash of the largest consistent tree observed by the log. The cosigner can then check subtree_hash with a subtree consistency proof (Section 4.4).

In both cases, the cosigner MUST ensure that, as it updates its view of the log, the old and new views are consistent. For example, [TLOG-WITNESS] defines a cosigner that checks consistency proofs (Section 2.1.4 of [RFC9162]) between the two views.

When a cosigner signs a subtree, it is held separately responsible _both_ for the subtree being consistent with its other signatures, _and_ for the cosigner-specific additional statements. That is, if a cosigner signs an inconsistent subtree, it is held responsible for its additional statements on all entries in the inconsistent subtree, even if some other signed subtree exists that asserts different entries.

Subtree signatures can be used to sign timestamped log checkpoints with a non-zero timestamp. A signature with a non-zero timestamp asserts the complete state of the cosigner's view of the log at a given time. These signatures are not directly used in Merkle Tree Certificates (Section 6.1), but cosigners MAY generate them, subject to the rules above, as part of other functions in a PKI. This may include log serving or integrating an issuance log into a transparency ecosystem. For example, [TLOG-TILES] and [TLOG-WITNESS] use such signatures.

5.3.3. 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.

In this document, any PKIX signature algorithm MAY be used, such as the ML-DSA algorithms defined in [RFC9881]. The signature is generated as in PKIX, except that the input is the structure defined in Section 5.3.1. In particular, in ML-DSA algorithms, the context string MUST be an empty string, as in Section 3 of [RFC9881].

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.

5.4. Certification Authority Cosigners

A _CA cosigner_ is a cosigner (Section 5.3) that certifies the contents of a log. Each CA MUST operate a CA cosigner whose cosigner ID is the same as its CA ID (Section 5.1). A CA cosigner MUST NOT sign checkpoints or subtrees for logs not part of this CA instance.

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 5.2.1.

Entries are extensible. Future documents MAY define type and extension_type values and what it means to certify them. A CA MUST NOT sign a subtree if it contains an entry with type or extension_type that it does not recognize. Doing so would certify

that the CA has validated the information in some not-yet-defined format. Section 12.5 further discusses security implications of such extensions.

If the CA issues certificate revocation lists (CRLs) [RFC5280] or Online Certificate Status Protocol (OCSP) responses [RFC6960], the CA's cosigner key MAY be used to directly sign TBSCertList or OCSP ResponseData structures, respectively, but only for this CA instance. Such uses remain subject to other X.509 constraints, such as the key usage extension, which are out of scope for this document. See Section 12.8 for a discussion of domain separation.

If the CA operator additionally operates a directly-signing X.509 CA, that CA key MUST be distinct from any Merkle Tree CA cosigner keys. In particular, a CA cosigner key MUST NOT be used to directly sign TBSCertificate structures. A CA cosigner key issues certificates by signing subtrees.

5.5. Representing Certification Authorities

This section defines the X.509 Certificate [RFC5280] representation of a Merkle Tree Certificate CA. It identifies the CA cosigner (Section 5.4) and associated issuance logs. This information is encoded as follows:

* The subject field MUST be the CA ID as a PKIX distinguished name, as described in Section 5.1.

* The subjectPublicKeyInfo field MUST be the public key of the CA cosigner Section 5.4.

* The extensions field MUST contain a critical extension of type id-pe-mtcCertificationAuthority, defined below.

* The subject key identifier extension (Section 4.2.1.2 of [RFC5280]), if present, SHOULD be set to the CA ID Section 5.1. The CA ID is encoded in its binary representation, as defined in Section 3 of [I-D.ietf-tls-trust-anchor-ids].

Other fields and extensions in [RFC5280] apply unmodified. In particular:

* The key usage extension (Section 4.2.1.3 of [RFC5280]) MUST be present and assert at least the keyCertSign bit.

* The basic constraints extension (Section 4.2.1.9 of [RFC5280]) MUST be present and set the cA field to TRUE.

The id-pe-mtcCertificationAuthority extension is defined below. This extension indicates that the subject of the certificate is a CA that issues Merkle Tree Certificates. If present, it MUST be marked as critical.

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

ext-mtcCertificationAuthority EXTENSION ::= {
    SYNTAX MTCCertificationAuthority
    IDENTIFIED BY id-pe-mtcCertificationAuthority
    CRITICALITY TRUE
}

-- From draft-ietf-tls-trust-anchor-ids
TrustAnchorID ::= RELATIVE-OID

MTCCertificationAuthority ::= SEQUENCE {
    logHash   AlgorithmIdentifier{DIGEST-ALGORITHM, {...}},
    sigAlg    AlgorithmIdentifier{SIGNATURE-ALGORITHM, {...}},
    minSerial INTEGER
}
```

For initial experimentation, early implementations of this design will use the OID 1.3.6.1.4.1.44363.47.2 instead of id-pe-mtcCertificationAuthority.

The fields of a MTCCertificationAuthority structure are defined as follows:

* logHash describes the hash algorithm used by all logs operated by this CA. For example, if the hash is SHA-256, it would be mda-sha256 as defined in Section 8 of [RFC5912].

* sigAlg is the CA cosigner’s signature algorithm (Section 5.3.3).

* minSerial is an integer describing the minimum allowed serial number from this CA. Since the serial number encodes both the log number (Section 5.2) and the entry index into a specific log, it can be used to set a minimum allowed log number or a minimum allowed index in a particular log (Section 5.2.3).

If this extension is present, the key described in subjectPublicKeyInfo is a CA cosigner key and subject to the usage restrictions described in Section 5.4. In particular, it MUST NOT be used to directly sign TBSCertificate structures.

This extension indicates the subtree signature format defined in Section 5.3.1. If a later version of the protocol defines a new format, this SHOULD be represented in CA certificates with a new extension type.

A CA certificate using this format SHOULD NOT be self-signed by the Merkle Tree Certificate CA. Doing so would require writing the information in the issuance log. Instead, if used to represent a trust anchor, the certificate should be an unsigned certificate [RFC9925].

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 5.2.1) contained in the issuance log (Section 5.2)

* A subject public key whose hash matches the TBSCertificateLogEntry

* A subtree (Section 4) that contains the log entry

* Zero or more signatures (Section 5.3) 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, standalone certificates and landmark-relative 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 5.2.1, this means issuer MUST be the issuance log's CA ID as a PKIX distinguished name, as described in Section 5.1.

The TBSCertificate's serialNumber is constructed from the zero-based index of the TBSCertificateLogEntry in the log and the log's number (Section 5.2). The serialNumber MUST be equal to (log_number << 48) | index. All serial numbers constructed in this way will be positive and at most 2^64-1.

The TBSCertificate's subjectPublicKeyInfo contains the specified public key. Its algorithm field MUST match the TBSCertificateLogEntry's subjectPublicKeyAlgorithm. 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]):

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

opaque HashValue[HASH_SIZE];

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

struct {
    MerkleTreeCertEntryExtension extensions<0..2^16-1>;
    uint48 start;
    uint48 end;
    HashValue inclusion_proof<0..2^16-1>;
    MTCSignature signatures<0..2^16-1>;
} MTCProof;
```

extensions MUST contain the log entry's extensions value (Section 5.2.1).

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 5.3) in its binary representation (Section 3 of [I-D.ietf-tls-trust-anchor-ids]), and signature contains the signature value as described in Section 5.3.1. The timestamp field used when computing the signature MUST be zero.

Each element of the signatures field MUST have a unique cosigner_id. Elements MUST be ordered by cosigner_id as follows:

* Shorter byte strings are ordered before longer byte strings

* Byte strings of the same length are ordered lexicographically

An MTCProof parser MUST reject the input if there are duplicate cosigner_id values, or if they are not ordered correctly. This can be done by checking each cosigner_id value comes strictly after the previous one in the above order.

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. Standalone Certificates

A _standalone 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. Standalone 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 5.4).
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 5.3).
4. The CA requests sufficient checkpoint cosignatures (Section 5.3) from external cosigners to meet relying party requirements (Section 7.3).
5. The CA requests subtree cosignatures 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 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.

This document does not prescribe the specific cosigner roles, or a particular protocol for requesting cosignatures. Protocols for cosigners can vary depending on the needs of that cosigner. Some example protocols are described in [TLOG-WITNESS] and [TLOG-MIRROR]. It is RECOMMENDED that the CA collect cosignatures for the authenticating party, but the authenticating party MAY collect additional cosignatures and add them to the certificate.

6.3. Landmark-Relative Certificates

A _landmark-relative certificate_ is a Merkle Tree certificate which contains no signatures and instead assumes the relying party had predistributed information about which subtrees were trusted. Landmark-relative 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 standalone certificate alongside any landmark-relative 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. Landmark Tree Sizes

To issue landmark-relative certificates, a CA must additionally maintain a _landmark sequence_, which is a sequence of _landmarks_.

Each landmark specifies an agreed tree size, as a common point of reference across the ecosystem for optimizing certificates. Landmarks are numbered consecutively from zero. The first landmark, numbered zero, MUST have 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.

As the issuance log grows, CAs continuously allocate new landmarks. This allocation balances minimizing landmark-relative certificate delay with minimizing the size of the relying party’s predistributed state. To bound the latter, each CA sets a positive integer max_active_landmarks parameter, which is the maximum number of landmarks that may contain unexpired certificates at any time.

The most recent max_active_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_active_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_active_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_active_landmarks hashes (Section 7.4) per CA, so max_active_landmarks should be set to balance the delay between landmarks and the amount of state the relying party must maintain. Using the recommended procedure below, a CA with a maximum certificate lifetime of 7 days, allocating a landmark every hour, will have a max_active_landmarks of 169. The client state is then 338 hashes, or 10,816 bytes with SHA-256.

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_active_landmarks to ceil(max_cert_lifetime / time_between_landmarks) + 1, where max_cert_lifetime is the CA’s maximum certificate lifetime. The + 1 accounts for landmarks not allocated at the exact start of their time interval, which can push certificate expiry one interval further than ceil(max_cert_lifetime / time_between_landmarks) alone would bound.

6.3.3. Publishing Landmarks

CAs SHOULD publish their active landmarks, so that relying parties can configure trusted subtrees (Section 7.4). The following format can be used to describe this information. The format is the following sequence of 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_active_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.

It is RECOMMENDED that this format be published as an HTTP resource [RFC9110] with content type text/plain; charset=utf-8.

6.3.4. Constructing Landmark-Relative Certificates

Given a TBSCertificateLogEntry in the issuance log and a landmark sequence, a landmark-relative 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 an 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 standalone and landmark-relative 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, standalone certificate subtrees will span around 2,500 certificates, leading to 12 hashes in the inclusion proof, or 384 bytes. Standalone certificates additionally must carry a sufficient set of signatures to meet relying party requirements.

If a new landmark is allocated every hour, landmark-relative 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.

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. Relying Party Configuration

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

* The CA's ID (Section 5.1)
* The CA's log hash algorithm, e.g. SHA-256
* The CA cosigner, and any other 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 that are known to be consistent with the relying party's cosigner requirements (Section 7.4)
* A list of revoked ranges of serial numbers (Section 7.5)

This information may be obtained from a CA certificate structure, defined in Section 5.5:

* The CA ID is determined from the certificate's subject.
* The log hash algorithm is determined from the id-pe-mtcCertificationAuthority extension.
* The CA cosigner is determined from the certificate's subject public key and id-pe-mtcCertificationAuthority extension. The CA's cosigner ID is the same as its CA ID. The relying party incorporates this cosigner into its cosigner policy based on the guidance in Section 7.3.
* No trusted subtrees are directly represented by the CA certificate structure, but the relying party MAY incorporate trusted subtrees from out-of-band information.
* The revoked serial number ranges include the half-open range [0, minSerial), but the relying party MAY incorporate additional ranges from out-of-band information.

7.2. Verifying Certificate Signatures

When verifying the signature of 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 this check fails, abort this process and fail verification.

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

3. Let serial be the certificate's serial number. If serial is negative or greater than 2^64-1, abort this process and fail verification.

4. If serial is contained in one of the relying party's revoked ranges (Section 7.5), abort this process and fail verification.

5. Let index be the least significant 48 bits of serial and let log_number be serial >> 48. If log_number is zero, abort this process and fail verification.

6. Let log_id be the log ID constructed from the CA ID in issuer and the log_number (Section 5.2).

7. Construct a TBSCertificateLogEntry as follows:

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

    2. Set subjectPublicKeyAlgorithm to the algorithm field of the subjectPublicKeyInfo.

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

8. Construct a MerkleTreeCertEntry as follows:

    1. Set type to tbs_cert_entry.

    2. Set extensions to the MTCProof's extensions value.

    3. Set tbs_cert_entry_data to the TBSCertificateLogEntry, encoded as described in Section 5.2.1.

9. Let entry_hash be the hash of the entry, MTH({entry}) = HASH(0x00 || entry), as defined in Section 2.1.1 of [RFC9162].

10. 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.

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

12. 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 5.3.1. For each signature verification, the CosignedMessage structure is constructed as follows:

    1. Set the CosignedMessage's cosigner_name based on the cosigner ID as described in Section 5.3.1.

    2. Set the CosignedMessage's timestamp to zero.

    3. Set the CosignedMessage's log_origin based on log_id as described in Section 5.3.1.

    4. Set the CosignedMessage's start and end to the MTCProof's start and end, respectively.

    5. Set the CosignedMessage's subtree_hash 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 extensions field from the MTCProof to the hash.

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

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

5. Write the subjectPublicKeyInfo's algorithm field to the hash.

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

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

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

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

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

This is possible because the structure in Section 5.2.1 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 CA cosigner key. This is analogous to the signature in a directly-signed X.509 certificate. If the relying party obtains CA information from a CA certificate, the CA cosigner key is determined as in Section 7.1.

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 refusing 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 that cosigners 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 CAs.

In applications that do not enforce transparency requirements, a relying party MAY implement a policy that only checks for a signature from the CA cosigner. This fits the pattern of many existing X.509 applications, where CA information is determined directly from a CA certificate, with no additional out-of-band information. Unrecognized cosignatures are ignored, so such applications can interoperate with certificates issued for transparency-enforcing applications that require additional cosigners.

Cosigner roles are extensible without changes to certificate verification itself. Future specifications and individual deployments MAY define other cosigner roles to incorporate in 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 trusted subtrees from one or more of the CA's issuance logs. This allows the relying party to accept landmark-relative certificates (Section 6.3) constructed against those subtrees.

Each trusted subtree contains:

* The log number of the containing log
* The start and end values that define the subtree
* The hash of the subtree

Trusted subtrees for a given log are determined by its active landmark subtrees, as described in Section 6.3.1. 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 5.3) 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 landmark-relative 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. Revoked Ranges

For each supported Merkle Tree CA, the relying party maintains a list of revoked ranges of serial numbers. A serial number combines a log number and a log index. A relying party can thus efficiently revoke both ranges of entries of an issuance log, and ranges of issuance logs, 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 an issuance log, 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 5.2.3). 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 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.

The revocation mechanism in this section is complementary to certificate-level revocation mechanisms. log entries are uniquely identified by their serial number and issuer, existing revocation mechanisms like CRLs [RFC5280] and OCSP [RFC6960] apply unchanged.

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 any 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 directly-signing X.509 CA

* Whether the authenticating party should send a standalone or landmark-relative 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. Standalone Certificates

Authenticating and relying parties SHOULD use the trust_anchors extension to determine whether a standalone certificate would be acceptable. A standalone certificate has a trust anchor ID of the corresponding CA ID (Section 5.1). This trust anchor ID is additionally contained in the trust anchor groups defined in Section 8.2.1.

CA IDs MAY be incorporated into other trust anchor groups, following the guidance in Section 5 of [I-D.ietf-tls-trust-anchor-ids].

[[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 standalone 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.

8.2. Landmark-Relative Certificates

An authenticating party SHOULD NOT send a landmark-relative 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.

TLS implementations SHOULD use the trust_anchors extension to determine this. A landmark-relative certificate’s trust anchor ID is the concatenation of the following OID components:

* The CA ID Section 5.1 of the CA that issued the certificate
* The constant 1
* The log number of the log used to construct the certificate
* The landmark number of the landmark used to construct the certificate

For example, the trust anchor ID for landmark 42 of CA 32473.1 and log number 8 is 32473.1.1.8.42.

These trust anchor IDs are used when it is necessary to identify an individual landmark, e.g. as in the retry mechanism described Section 4.3 of [I-D.ietf-tls-trust-anchor-ids]. To more efficiently express a relying party’s complete landmark state, these IDs are contained in trust anchor groups defined in Section 8.2.1, which allow relying paries to express their landmark state with a single ID.

If both a landmark-relative and a standalone certificate are usable, an authenticating party SHOULD preferentially use the landmark-relative certificate. A landmark-relative certificate asserts the same information as its standalone counterpart, but is expected to be smaller.

8.2.1. Single-Log Landmark Groups

Relying parties support many landmarks per log at a time. To compactly represent this, each log ID implicitly defines series of trust anchor groups (Section 5 of [I-D.ietf-tls-trust-anchor-ids]) called _landmark groups_.

For each Merkle Tree Certificates CA, each log number N, and each landmark number L, a landmark group is defined. The group's ID is the concatenation of the following OID components:

* The CA ID Section 5.1 of the CA
* The constant 2
* The log number N
* The landmark number L

This group contains the following trust anchors:

* The CA ID itself (see Section 8.1)
* Each landmark of log N from L - max_active_landmarks + 1 to L, inclusive

Landmark-relative certificates SHOULD be configured with this information, as in Section 3.2 of [I-D.ietf-tls-trust-anchor-ids]. A relying party whose latest trusted subtree (Section 7.4) in log N is landmark L SHOULD configure the trust_anchors extension to advertise the above landmark group. This signals support for both standalone certificates and supported landmarks.

For example, a relying party which is up-to-date as of landmark 42 of log 8 of CA 32473.1 would send an ID of 32473.1.2.8.42.

8.2.2. Timestamped Landmark Groups

Landmark groups for an single CA, described above, allow relying parties to advertise one ID per supported CA. Depending on the number of trust anchors, this can be sufficient to efficiently represent relying party state.

When needed, Section 5 of [I-D.ietf-tls-trust-anchor-ids] describes how PKIs requiring further size savings can use trust anchor groups that span multiple CA instances. For example, a single ID may signal support for a group of CAs across one or more CA operators. This section describes how such groups can be applied to landmarks, using a variation of the versioning construction described in Section 5.1 of [I-D.ietf-tls-trust-anchor-ids].

Trust anchor groups containing landmarks SHOULD define versions predictably based on the time. For example, if the contained CAs allocate landmarks roughly hourly, the trust anchor group might increment the version component every hour. Each given version of the group SHOULD contain the active landmarks as of the corresponding timestamp.

This predictable cadence allows the CA to construct trust anchor group inclusions (Section 7.2 of [I-D.ietf-tls-trust-anchor-ids]) for issued certificates without additional coordination. Conversely, a relying party MAY send a version if its trusted subtrees (Section 7.4) are up-to-date for all contained CAs, as of the versions timestamp.

In some cases, the relying party's trusted subtrees may only be partially up-to-date. The relying party, or its update service, may be unable to reach one CA in the group, e.g. due to a transient outage. This complicates timestamp-based strategies:

* If the relying party sends the group with an older timestamp, it will not signal its up-to-date state for the reachable CAs. This means a single unreachable CA can disrupt service for certificates issued by unrelated CAs.

* If the relying party sends the group with a newer timestamp, the relying party may signal support for landmarks it does not have. This risks connection failures. If the unreachable CA issued recent landmark-relative certificates, those certificates will fail validation.

The relying party can mitigate this in a number of ways:

* If the trust anchor group consists of CAs from the same operator, waiting until all CAs are reachable will be minimally disruptive.

* The relying party can opt to send the group with an older timestamp, combined with other, smaller groups at newer timestamps to better describe its state.

* A client relying party can send the newer timestamp and, in the event the unreachable CA did issue recent landmark-relative certificates, rely on the retry mechanism described in Section 4.3 of [I-D.ietf-tls-trust-anchor-ids] to recover from any signaling failures.

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 7 of [I-D.ietf-tls-trust-anchor-ids]. Section 8 decribes the trust anchor ID assignments for standalone and landmark-relative 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 standalone certificate, constructed as described in Section 6.2.

The standalone certificate response SHOULD additionally carry an alternate URL for the landmark-relative certificate, as described Section 7.4.2 of [RFC8555]. Before the landmark-relative 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 landmark-relative 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 expect CAs to operate logs, the costs of these logs are expected to be much lower than a CT log from [RFC6962] or [RFC9162]:

Section 5.2.2 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. Costs are thus expected to scale with the CA's own issuance.

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 5.2.3) 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 a role comparable to 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 naturally batched and serialized.

10.1.3. Monitor Costs

In a CT-based PKI, every log carries a potentially distinct subset of active certificates. 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 inexpensive to run, but does not guarantee durable logging. 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, larger quorums result in larger, more expensive standalone certificates. The cost of standalone certificates will depend on how frequently the landmark optimization occurs in a given PKI. Conversely, relying parties that require smaller quorums have smaller standalone 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 5.2.3). If some resources are unavailable, they may not be visible to monitors.

As in CT, PKIs that deploy Merkle Tree certificates SHOULD establish availability policies. These policies SHOULD be 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. 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 without compromising transparency goals. If a CA never makes an entry available, mirrors will be unable to update. This will prevent relying parties from accepting the undisclosed entries. However, a CA that 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 SHOULD set availability requirements on CAs.

In each of these cases, the serial numbers of unavailable entries are known. Availability failures can thus be mitigated by revocation, 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 landmark-relative certificate will not be available until the next landmark is ready. From there, the landmark-relative certificate will not be available until relying parties receive new trusted subtrees.

To maximize coverage of landmark-relative certificates, authenticating parties performing routine renewal SHOULD request a new Merkle Tree certificate before the previous Merkle Tree certificate expires. Renewing around 75% of the way through 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 standalone certificate and the revocation of the previous certificate. This will interrupt the size optimization until the new landmark-relative certificate is available and relying parties are updated.

11. Privacy Considerations

The Privacy Considerations described in Section 9 of [I-D.ietf-tls-trust-anchor-ids] apply to their 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 standalone 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 5.4) is defined to certify the contents of every checkpoint and subtree that it signs.

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

Given a subtree hash computed over entries that the CA certified, it must be computationally infeasible to construct an entry not on this list, and an 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. 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. 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 range of inconsistent serials (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). The impact depends on the relying party's cosigner policy:

* 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. This is sufficient to determine the serial number, so relying parties can then react by revoking the undisclosed entries (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 signed checkpoint may be retained by the transparency ecosystem, so it may not be possible to reconstruct the exact certificate seen by relying parties.

However, per Section 5.4, any subtree signature is a binding assertion by the CA that it has certified every entry in the subtree. 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 signed checkpoint 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. Extensibility

MerkleTreeCertEntry (Section 5.2.1) contain several extension points:

* New X.509 extensions can be added to TBSCertificateLogEntry.
* New MerkleTreeCertEntryType values define new formats for the entry contents.
* New MerkleTreeCertEntryExtensionType values define new entry extension fields.

X.509 extensions apply to Merkle Tree Certificates without any modifications. The two entry-level extension points are new to this protocol. Older CAs, cosigners, relying parties, and monitors may encounter unrecognized entries:

Different cosigner roles interact with extensions differently. Some roles, e.g. [TLOG-MIRROR] and [TLOG-WITNESS], do not interpret entry contents. Unrecognized extensions do not impact these roles. Other roles, such as CA cosigners, have semantics that depend on the entry contents. If a cosigner role interprets log entry contents, it MUST define how it interacts with unrecognized types and extensions.

Section 5.4 forbids 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 CA can never sign an unrecognized X.509 extension.

Unrecognized entry types do not impact older relying parties. 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.

However, unrecognized entry extensions will be ignored by relying parties, analogously to a non-critical X.509 extension. Entry extensions thus SHOULD be defined so that this is safe.

If a monitor observes an entry with unknown type or entry extension, 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.

When 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 5.2.1 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 are 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 an 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.

12.7. Revocation

This document does not define a new certificate-level revocation mechanism. Existing mechanisms like CRLs and OCSP apply unchanged to Merkle Tree certificates. The sequential serial numbers assigned by issuance logs may enable future improvements to revocation, but such work is out of scope for this document.

12.8. Signature Domain Separation

The signature format defined in Section 5.3.1 includes a fixed label prefix to ensure domain separation. Provided other uses of the same key use a non-overlapping prefix, signatures in one context cannot be substituted for those in another.

Section 5.4 permits a CA cosigner key to be used to sign CRLs and OCSP resposes. These signatures do not include a domain separation prefix. Instead, X.509 relies on an undocumented assumption that the TBSCertificate, TBSCertList, and OCSP ResponseData structures do not overlap at the level of individual ASN.1 fields.

These ASN.1 structures all begin with a SEQUENCE tag, which is encoded in DER as 0x30 or the ASCII digit "0". The domain separation label used in Section 5.3.1, subtree/v1\n\0, does not begin with "0", so their inputs do not overlap. More generally, this label is not a prefix of any DER or BER encoding.

Domain separation analysis based on the structures themselves is fragile, particularly when individual ASN.1 fields must be analyzed. This document depends on a structure-level analysis for CRLs and OCSP responses due to how these legacy protocols were defined. Future uses of the key SHOULD use a more robust mechanism, namely a fixed label prefix or a context string parameter if the signature scheme supports it.

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. Certificate Extension

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

Decimal	Description	References
TBD	id-pe-mtcCertificationAuthority	[this-RFC]

Table 3

13.4. 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 4

14. References

14.1. Normative References

[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-04, 30 April 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-trust-anchor-ids-04>.

[POSIX] "IEEE/Open Group Standard for Information Technology--Portable Operating System Interface (POSIX) Base Specifications, Issue 8", IEEE, DOI 10.1109/ieeestd.2024.10555529, ISBN ["9798855707939"], June 2024, <https://doi.org/10.1109/ieeestd.2024.10555529>.

[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>.

[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>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/rfc/rfc6960>.

[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>.

[RFC9881] Massimo, J., Kampanakis, P., Turner, S., and B. E. Westerbaan, "Internet X.509 Public Key Infrastructure -- Algorithm Identifiers for the Module-Lattice-Based Digital Signature Algorithm (ML-DSA)", RFC 9881, DOI 10.17487/RFC9881, October 2025, <https://www.rfc-editor.org/rfc/rfc9881>.

[RFC9925] Benjamin, D., "Unsigned X.509 Certificates", RFC 9925, DOI 10.17487/RFC9925, February 2026, <https://www.rfc-editor.org/rfc/rfc9925>.

[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>.

[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>.

[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>.

[KeyReuse] Patton, C. and T. Shrimpton, "Security in the Presence of Key Reuse: Context-Separable Interfaces and their Applications", 2019, <https://eprint.iacr.org/2019/519>.

[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>.

[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>.

[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>.

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

[TLOG-COSIGNATURE] C2SP, "Transparency Log Cosignatures", April 2026, <https://c2sp.org/tlog-cosignature>.

[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, DIGEST-ALGORITHM, AlgorithmIdentifier{},
  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, PublicKeyAlgorithms
  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-anchor-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,
    subjectPublicKeyAlgorithm AlgorithmIdentifier{PUBLIC-KEY,
                                  {PublicKeyAlgorithms}},
    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
   }

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

   ext-mtcCertificationAuthority EXTENSION ::= {
       SYNTAX MTCCertificationAuthority
       IDENTIFIED BY id-pe-mtcCertificationAuthority
       CRITICALITY TRUE
   }

   MTCCertificationAuthority ::= SEQUENCE {
       logHash   AlgorithmIdentifier{DIGEST-ALGORITHM, {...}},
       sigAlg    AlgorithmIdentifier{SIGNATURE-ALGORITHM, {...}},
       minSerial INTEGER
   }

   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 a 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:

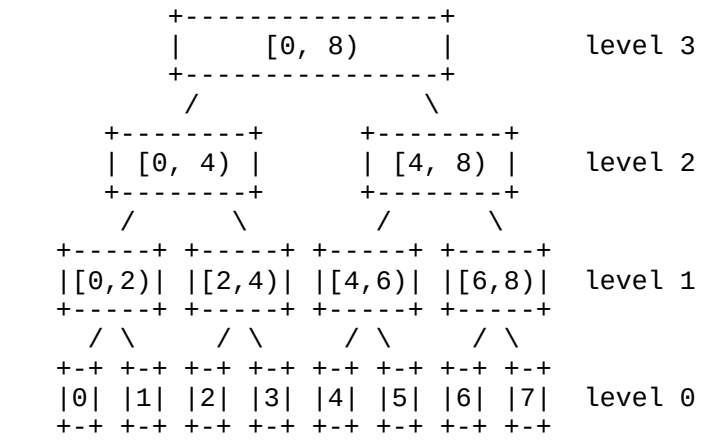


Figure 12: An example Merkle Tree of size 8

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 from level 2 to level 1.

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

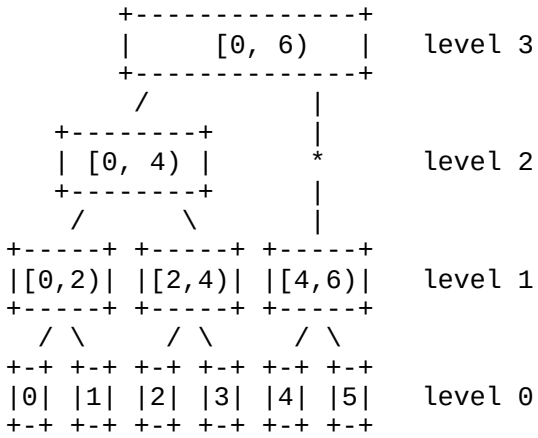


Figure 13: An example Merkle Tree of size 6

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.

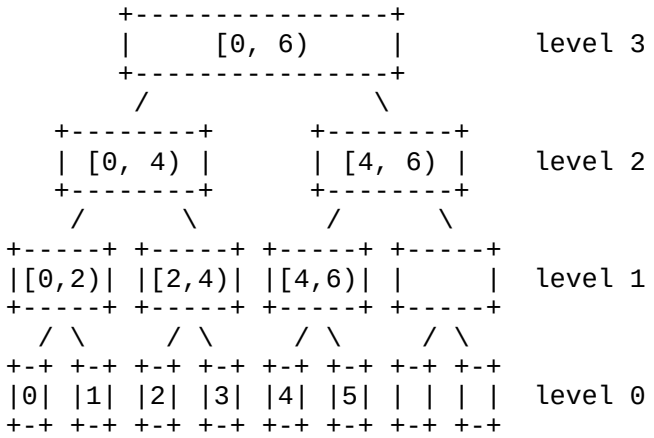


Figure 14: An example Merkle Tree of size 6, viewed as a subset of a tree of size 8

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 left 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 starting 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 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

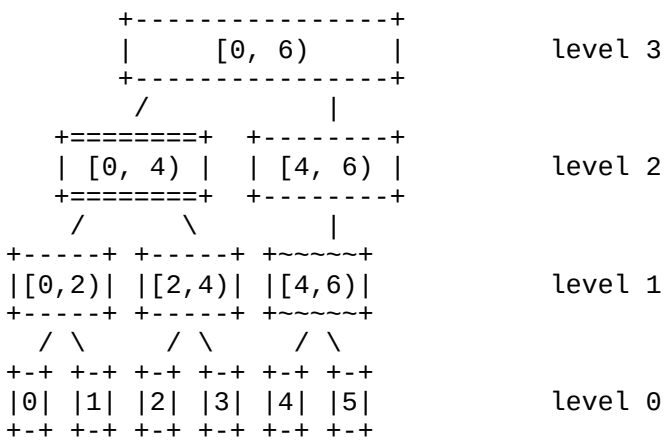
A subtree consistency proof for [start, end) and the tree of n elements is similar to an inclusion proof for element end - 1. If one starts from end - 1's hash, incorporating the whole inclusion proof should reconstruct root_hash and incorporating a subset of the inclusion proof should reconstruct node_hash. 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.



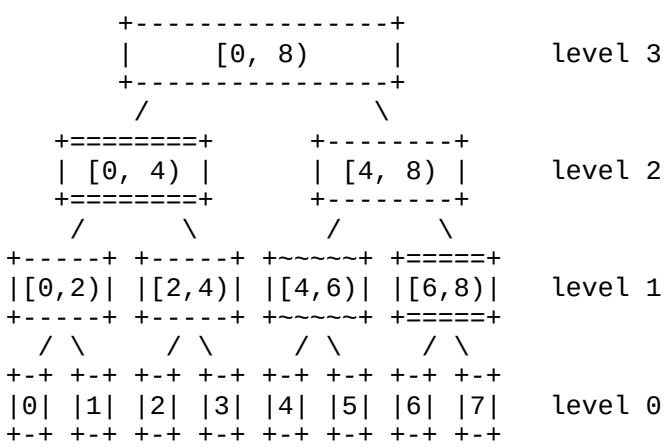


Figure 15: A subtree consistency proof that starts at level 1 instead of level 0

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.)

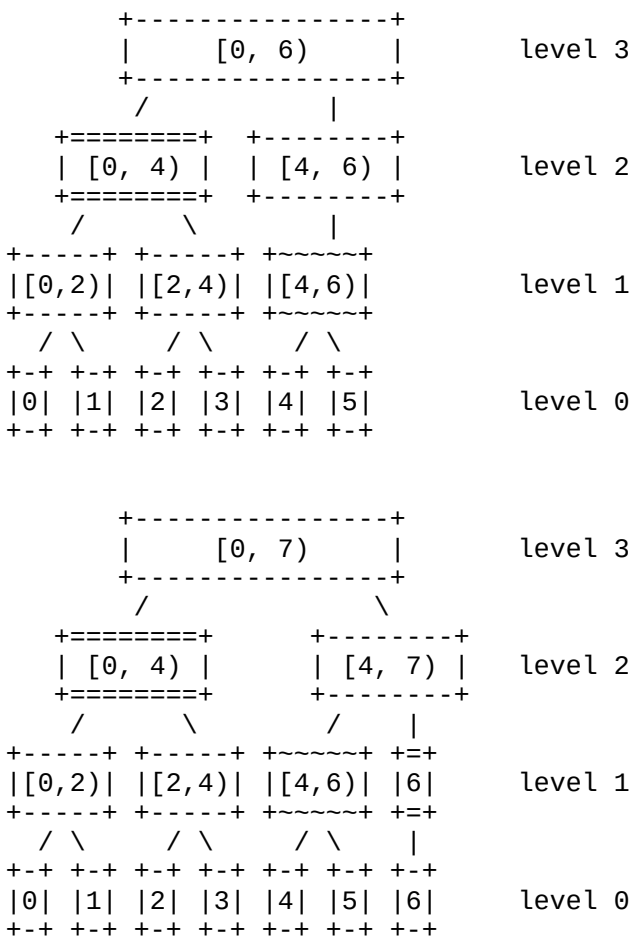


Figure 16: The interaction between inclusion proof truncation and skipped levels

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 the 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.

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, Corey Bonnell, Ryan Dickson, Aaron Gable, Nick Harper, Russ Housley, Dennis Jackson, Ilari Liusvaara, Sanketh Menda, Matt Mueller, Chris Patton, Michael Richardson, Ryan Sleevi, and Emily Stark for many valuable discussions and insights which led to this document, as well as feedback and contributions to 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

Since draft-davidben-tls-merkle-tree-certs-10

* Adopted by working group

Since draft-ietf-plants-merkle-tree-certs-00

* Address editorial comments from WG adoption call

Since draft-ietf-plants-merkle-tree-certs-01

* Renamed full certificate to standalone certificate, signatureless certificate to landmark certificate.
* Included subject public key algorithm in log entries

Since draft-ietf-plants-merkle-tree-certs-02

* Renamed landmark certificate to landmark-relative certificate
* Relaxed restrictions on null_entry
* Clarify that CRLs and OCSPs apply to MTCs unmodified

Since draft-ietf-plants-merkle-tree-certs-03

* Use a tlog-compatible signature scheme for ease of deployment
* Define a CA certificate representation
* Remove the one-to-many relationship between MTC CAs and CA cosigners
* Discuss domain separation for signatures

* Recommend a maximum log entry size for tlog compatibility

* Prescribe landmark OID allocation

* Update TLS integration now that trust anchor IDs extension has been moved to the base draft

* A single CA now operates a series of issuance logs, instead of a one-to-one correspondence

* Group components of a CA into a CA-specific section that enumerates the parts of a CA

* Canonicalize the order of cosignatures in MTCProofs

* Remove sketch of tlog subtree signer API in favor of https://github.com/C2SP/C2SP/pull/245 in [TLOG-WITNESS]

* Add an extensions block to log entries

Authors' Addresses

David Benjamin
Google LLC
Email: davidben@google.com

Devon O'Brien
Apple Inc.
Email: asymmetric@apple.com

Bas Westerbaan
Cloudflare
Email: bas@cloudflare.com

Luke Valenta
Cloudflare
Email: lvalenta@cloudflare.com

Filippo Valsorda
Geomys
Email: ietf@filippo.io