Workgroup: Transport Layer Security
Internet-Draft:
draft-davidben-tls-merkle-tree-certs-00
Published: 10 March 2023
Intended Status: Experimental
Expires: 11 September 2023
Authors: D. Benjamin (Google LLC)
D. O'Brien (Google LLC)
B. Westerbaan (Cloudflare)

Merkle Tree Certificates for TLS

Abstract

This document describes Merkle Tree certificates, a new certificate type for use with TLS. A relying party that regularly fetches information from a transparency service can use this certificate type as a size optimization over more conventional mechanisms with post-quantum signatures. Merkle Tree certificates integrate the roles of X.509 and Certificate Transparency, achieving comparable security properties with a smaller message size, at the cost of more limited applicability.

About This Document

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

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

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

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

Status of This Memo

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

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

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents

at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 11 September 2023.

Copyright Notice

Table of Contents

1. Introduction

Authors' Note: This is an early draft of a proposal with many parts. While we have tried to make it as concrete as possible, we anticipate that most details will change as the proposal evolves.

A typical TLS [RFC8446] handshake uses many signatures to authenticate the server public key. In a certificate chain with an end-entity certificate, an intermediate certificate, and an implicit trust anchor, there are two X.509 signatures [RFC5280]. Intermediate certificates additionally send an extra public key. If the handshake uses Certificate Transparency (CT) [RFC6962], each Signed Certificate Timestamp (SCT) also carries a signature. CT policies often require two or more SCTs per certificate [APPLE-CT] [CHROME-CT]. If the handshake staples an OCSP response [RFC6066] for revocation, that adds an additional signature.

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, Dilithium3 [Dilithium] uses 1,952 bytes per public key and 3,293 bytes per signature. A TLS Certificate message with, say, four Dilithum3 signatures (two X.509 signatures and two SCTs) and one intermediate CA's Dilithium3 public key would total 15,124 bytes of authentication overhead. Falcon-512 and

Falcon-1024 [Falcon] would, respectively, total 3,561 and 6,913 bytes.

This document introduces Merkle Tree Certificates, an optimization that authenticates a subscriber key using under 1,000 bytes. See Section 5.5. To achieve this, it reduces its scope from general authentication:

* Certificates are short-lived. The subscriber is expected to use an automated issuance protocol, such as ACME [RFC8555].

* Certificates are only usable with relying parties that have contacted a transparency service sufficiently recently. See Section 7.

* Certificates are issued after a significant processing delay of, in the recommended parameters (Section 5.1), about an hour. Subscribers that need a certificate issued quickly are expected to use a different mechanism.

* To mitigate the above, this document describes a certificate negotiation mechanism. This allows subscribers to send these more efficient certificates when available, while falling back to other mechanisms otherwise.

Merkle Tree Certificates are not intended to replace existing Public Key Infrastructure (PKI) mechanisms but, in applications where a significant portion of authentications meet the above requirements, complement them as an optional optimization. In particular, it is expected that, even within applications that implement it, this mechanism will not be usable for all TLS connections.

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].

3. Overview

3.1. Terminology and Roles

There are five roles involved in a Merkle Tree certificate deployment:

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

Merkle Tree certification authority (CA): The service that issues Merkle Tree certificates to the subscriber, and publishes logs of all certificates.

Relying party: The party authenticating the subscriber. In TLS, this is the side receiving the Certificate and CertificateVerify message.

Transparency service: The service that mirrors the issued certificates for others to monitor. It additionally summarizes the CA's activity for relying parties, in order for certificates to be accepted. This is conceptually a single service, but may be multiple services, run by multiple entities in concert. See Section 7. For example, if the relying party is a web browser, the browser vendor might run the transparency service, or it may trust a collection of third-party mirrors.

Monitors: Parties who monitor the list of valid certificates, published by the transparency service, for unauthorized certificates.

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.

Assertion: A protocol-specific statement that the CA is certifying. For example, in TLS, the assertion is that a TLS signing key can speak on behalf of some DNS name or other identity.

Certificate: A structure, generated by the CA, that proves to the relying party that the CA has certified some assertion. A certificate consists of the assertion itself accompanied by an associated proof string.

Batch: A collection of assertions certified at the same time. CAs in this proposal only issue certificates in batches at a fixed frequency.

Batch tree head: A hash computed over all the assertions in a batch, by building a Merkle Tree. The Merkle Tree construction and this hash are described in more detail in Section 5.4.1.

Inclusion proof: A structure which proves that some assertion is contained in some tree head. See Section 5.4.3.

Window: A range of consecutive batch tree heads. A relying party maintains a copy of the CA's latest window. At any time, it will accept only assertions contained in tree heads contained in the current window.

3.2. Certificate Lifecycle

The process of issuing and using a certificate is as follows:

1. The subscriber requests a certificate from the CA. Section 9 describes ACME [RFC8555] extensions for this.

2. The CA collects certificate requests into a batch (see Section 5.1) and builds the Merkle Tree and computes the tree head (see Section 5.4.1). It then signs the window ending at this tree head (see Section 5.4.2) and publishes (see Section 8) the result.

3. The CA constructs a certificate using the inclusion proof. It sends this certificate to the subscriber. See Section 5.4.3.

4. The transparency service downloads the Merkle Tree, validates the hashes, and mirrors it for monitors to observe. See Section 7.

5. The relying party fetches the latest window from the transparency service. This window will contain the new tree head.

6. In an application protocol such as TLS, the relying party communicates its window state to the subscriber.

7. If the relying party's window contains the subscriber's certificate, the subscriber negotiates this protocol and sends the Merkle Tree certificate. See Section 6.3 for details. If there is no match, the subscriber proceeds as if this protocol were not in use (e.g., by sending a traditional X.509 certificate chain).

Figure 1 below shows this process.

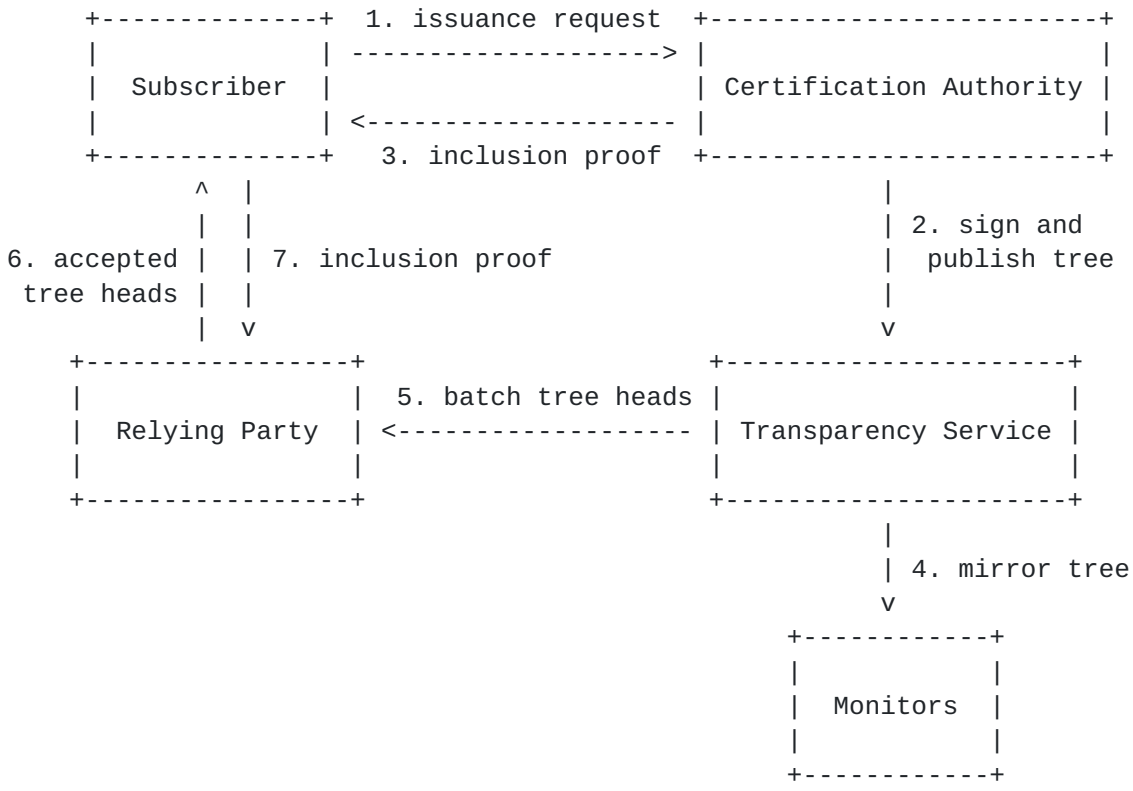

Figure 1: An overview of a Merkle Tree certificate deployment

The remainder of this document discusses this process in detail, followed by concrete instantions of it in TLS [RFC8446] and ACME [RFC8555].

4. Assertions

[[TODO: The protocol described in this document is broadly independent of the assertion format. We describe, below, one possible structure, but welcome feedback on how best to structure the encoding. The main aims are simplicity and to improve on handling cross-protocol attacks per Section 13.2.]]

TLS certificates associate some application-specific identifier with a TLS signing key. When TLS is used to authenticate HTTPS [RFC9110] servers, these identifiers specify DNS names or HTTP origins. Other protocols may require other kinds of assertions.

To represent this, this document defines an Assertion structure:

```
enum { tls(0), (2^16-1) } SubjectType;

enum {
    dns(0),
    dns_wildcard(1),
    ipv4(2),
    ipv6(3),
    (2^16-1)
} ClaimType;

struct {
    ClaimType claim_type;
    opaque claim_info<0..2^24-1>;
} Claim;

struct {
    SubjectType subject_type;
    opaque subject_info<0..2^24-1>;
    Claim claims<0..2^24-1>;
} Assertion;
```

An Assertion is roughly analogous to an X.509 TBSCertificate (Section 4.1.2 of [RFC5280]). It describes a series of claims about some subject. The subject_info field is interpreted according to the subject_type value. For TLS, the subject_type is tls, and the subject_info is a TLSSubjectInfo structure. TLSSubjectInfo is defined in full in Section 10.1 below, but as an illustrative example, it is reproduced below:

```
struct {
    SignatureScheme signature;
    opaque public_key<1..2^24-1>;
} TLSSubjectInfo;
```

This structure represents the public half of a TLS signing key. The semantics are thus that each claim in claims applies to the TLS client or server. This is analogous to X.509's SubjectPublicKeyInfo structure (Section 4.1.2.7 of [RFC5280]) but additionally incorporates the protocol. Protocols consuming an Assertion MUST check the subject_type is a supported value before processing subject_info. If unrecognized, the structure MUST be rejected.

Other protocols aiming to integrate with this structure allocate a SubjectType codepoint and describe how it is interpreted.

Likewise, a Claim structure describes some claim about the subject. The claim_info field is interpreted according to the claim_type. Each Claim structure in an Assertion's claims field MUST have a unique claim_type and all values MUST be sorted in order of increasing claim_type. Structures violating this constraint MUST be rejected.

When a relying party interprets an Assertion certified by the CA, it MUST ignore any Claim values with unrecognized claim_type. When a CA interprets an Assertion in a certification request from a subscriber, it MUST reject any Claim values with unrecognized claim_type.

This document defines claim types for DNS names and IP addresses, but others can be defined.

[[TODO: For now, the claims below just transcribe the X.509 GeneralName structure. Should these be origins instead? For HTTPS, it's a pity to not capture the scheme and port. We do mandate ALPN in Section 10.2, so cross-protocol attacks are mitigated, but it's unfortunate that subscribers cannot properly separate their HTTPS vs FTPS keys, or their port 443 vs port 444 keys. One option here is to have HTTPS claims instead, and then other protocols can have FTPS claims, etc.]]

4.1. DNS Claims

The dns and dns_wildcard claims indicate that the subject is authoritative for a set of DNS names. They use the DNSNameList structure, defined below:

```
opaque DNSName<1..255>;

struct {
    DNSName dns_names<1..2^16-1>;
} DNSNameList;
```

DNSName values use the "preferred name syntax" as specified by Section 3.5 of [RFC1034] and as modified by Section 2.1 of [RFC1123]. Alphabetic characters MUST additionally be represented in lowercase. IDNA names [RFC5890] are represented as A-labels. For example, possible values include example.com or xn--iv8h.example. EXAMPLE.COM and <U+1F50F>.example would not be permitted.

Names in a dns claim represent the exact DNS name specified. Names in a dns_wildcard claim represent wildcard DNS names and are processed as if prepended with the string "*." and then following the steps in Section 6.3 of [I-D.draft-ietf-uta-rfc6125bis].

4.2. IP Claims

The ipv4 and ipv6 claims indicate the subject is authoritative for a set of IPv4 and IPv6 addresses, respectively. They use the IPv4AddressList and IPv6AddressList structures, respectively, defined below. IPv4Address and IPv6Address are interpreted in network byte order.

```
uint8 IPv4Address[4];
uint8 IPv6Address[16];

struct {
    IPv4Address addresses<4..2^16-1>;
} IPv4AddressList;

struct {
    IPv6Address addresses<16..2^16-1>;
} IPv6AddressList;
```

5. Issuing Certificates

This section describes the structure of Merkle Tree certificates, and defines the process of how a Merkle Tree certification authority issues certificates for a subscriber.

5.1. Merkle Tree CA Parameters

A Merkle Tree certification authority is defined by the following values:

hash: A cryptographic hash function. In this document, the hash function is always SHA-256 [SHS], but others may be defined.

issuer_id: A short, opaque byte string that identifies the CA.

public_key: The public half of a signing keypair. The corresponding private key, private_key, is known only to the CA.

start_time: The issuance time of the first batch of certificates

lifetime: A duration of time which determines the lifetime of certificates issued by this CA.

batch_duration: A duration of time which determines how frequently the CA issues certificates. See details below.

window_size: An integer describing the maximum number of unexpired batches which may exist at a time. This value is determined from lifetime and batch_duration by floor(lifetime / batch_duration) + 1.

These values are public and known by the relying party and the CA. They may not be changed for the lifetime of the CA. To change these parameters, the entity operating a CA may deploy a second CA and either operate both during a transition, or stop issuing from the previous CA.

[[TODO: The signing key case is interesting. A CA could actually maintain a single stream of Merkle Trees, but then sign everything with multiple keys to support rotation. The CA -> Subscriber -> RP flow does not depend on the signature, only the CA -> Transparency Service -> RP flow. The document is not currently arranged to

capture this, but it probably should be. We probably need to decouple the signing half and the Merkle Tree half slightly.]]

Certificates are issued in batches. Batches are numbered consecutively, starting from zero. All certificates in a batch have the same issuance time, determined by start_time + batch_duration * batch_number. This is known as the batch's issuance time. That is, batch 0 has an issuance time of start_time, and issuance times increment by batch_duration. A CA can issue no more frequently than batch_duration. batch_duration determines how long it takes for the CA to return a certificate to the subscriber.

All certificates in a batch have the same expiration time, computed as lifetime past the issuance time. After this time, the certificates in a batch are no longer valid. Merkle Tree certificates uses a short-lived certificates model, such that certificate expiration replaces an external revocation signal like CRLs [RFC5280] or OCSP [RFC6960]. lifetime SHOULD be set accordingly. For instance, a deployment with a corresponding maximum OCSP [RFC6960] response lifetime of 14 days SHOULD use a value no higher than 14 days. See Section 13.3 for details.

The window_size parameter, computed from lifetime and batch_duration, determines relying party resource requirements, as described in Section 6.1. A relying party must maintain window_size hashes at a time. Parameters SHOULD be tuned to balance relying party resource requirements and issuance delay.

CAs are RECOMMENDED to use a batch_duration of one hour, and a lifetime of 14 days. This results in a window_size of 336, for a total of 10,752 bytes in SHA-256 hashes.

To prevent cross-protocol attacks, the key used in a Merkle Tree CA MUST be unique to that Merkle Tree CA. It MUST NOT be used in another Merkle Tree CA, or for another protocol, such as X.509 certificates.

5.2. Batch State

Each batch is in one of three states:

pending: The current time is before the batch's issuance time

ready: The current time is at or after the batch's issuance time, but the batch has not yet been issued

issued: Certificates have been issued for this batch

The CA also maintains a latest batch number, which is the number of the last batch in the "issued" state. As an invariant, all batches before this value MUST also be in the "issued" state.

For each batch in the "issued" state, the CA maintains the following batch state:

- The list of assertions certified in this batch.

- The tree head, a hash computed over this list, described in Section 5.4.1.

- A window signature computed as described in Section 5.4.2.

The CA exposes all of this information in an HTTP [RFC9110] interface described in Section 8.

5.3. Issuance Queue and Scheduling

The CA additionally maintains an issuance queue, not exposed via the HTTP interface.

When a subscriber requests a certificate for some assertion, the CA first validates it per its issuance policy. For example, it may perform ACME identifier validation challenges (Section 8 of [RFC8555]). Once validation is complete and the CA is willing to certify the assertion, the CA appends it to the issuance queue.

The CA runs a regularly-scheduled issuance job which converts this queue into certificates. This job runs the following procedure:

1. If no batches are in the "ready" state, do nothing and abort this procedure. Schedule a new job to run sometime after the earliest "pending" batch's issuance time.

2. For each batch in the "ready" state other than the latest one, run the procedure in Section 5.4 with an empty assertion list, in order of increasing batch number. Batches cannot be skipped.

3. Empty the issuance queue into an ordered list of assertions. Run the procedure in Section 5.4 using this list and the remaining batch in the "ready" state. This batch's issuance time will be at or shortly before the current time.

5.4. Certifying a Batch of Assertions

This section describes how to certify a given list of assertions at a given batch number. The batch MUST be in the "ready" state, and all preceding batches MUST be in the "issued" state.

5.4.1. Building the Merkle Tree

First, the CA then builds a Merkle Tree from the list as follows:

Let n be the number of input assertions. If n > 0, the CA builds a binary tree with l levels numbered 0 to l-1, where l is ceil(log_2(n)) + 1. Each node in the tree contains a hash value. Hashes in the tree are built from the following functions:

```
    HashEmpty() = hash(0x00)
    HashNode(left, right, level, index) = hash(HashNodeInput)
    HashAssertion(assertion) = hash(0x02 || assertion)
```

0x00 and 0x02 denote byte strings containing a single byte with value zero and two, respectively. || denotes concatenation. HashNodeInput is computed by encoding the structure defined below:

```
struct {
    uint8 distinguisher = 1;
    opaque issuer_id<1..32>;
    uint32 batch_number;
    opaque pad[N];
    opaque left[hash.length];
    opaque right[hash.length];
    uint8 level;
    uint64 index;
} HashNodeInput;
```

issuer_id and batch_number are set to the CA's issuer_id and the current batch number. pad is an array of zeros to pad up to the hash function's block size. For SHA-256, the block size is 64 bytes. The remaining fields are set to inputs of the function.

Tree levels are computed iteratively as follows:

1. Initialize level 0 with n elements. For i between 0 and n-1, inclusive, set element i to the output of HashAssertion(assertion[i]).

2. For i between 1 and l-1, inclusive, compute level i from level i-1 as follows:

    * If level i-1 has an odd number of elements, append HashEmpty() to the level.

    * Initialize level i with half as many elements as level i-1. For all j, set element j to the output of HashNode(left, right, i, j) where left is element 2*j of level i-1 and right is element 2*j+1 of level i-1. left and right are the left and right children of element j.

At the end of this process, level l-1 will have exactly one root element. This element is called the tree head. Figure 2 shows an example tree for three assertions. The tree head in this example is t20.

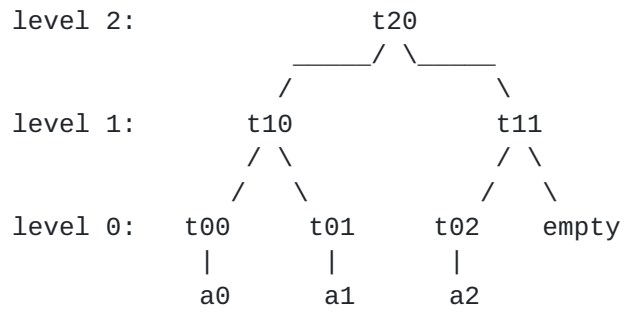

Figure 2: An example Merkle Tree for three assertions

If n is zero, the CA does not build a tree and the tree head is HashEmpty(). This value is constant for a given hash function. The value of HashEmpty() for SHA-256 is:

6e340b9cffb37a989ca544e6bb780a2c78901d3fb33738768511a30617afa01d

If n is one, the tree contains a single level, level 0, and has a tree head of HashAssertion(assertion).

5.4.2. Signing a Window

Batches are grouped into consecutive ranges of window_size batches, called windows. As window_size is computed to cover the full certificate lifetime, a window that ends at the latest batch number covers all certificates that may still be valid from a CA.

Windows are serialized into the following structure:

```
opaque TreeHead[hash.length];

struct {
    uint32 batch_number;
    TreeHead tree_heads[window_size];
} Window;
```

batch_number is the batch number of the highest batch in the window.

tree_heads value contains the last window_size tree heads, starting from batch_number, in decreasing batch number order. That is, tree_heads[0] is the tree head for batch batch_number, tree_heads[1] is the tree head for batch_number - 1, and so on. If batch_number < window_size - 1, any tree heads with numbers below zero are filled with HashEmpty().

After the CA builds the Merkle Tree for a batch, it constructs the Window structure whose batch_number is the number of the batch being issued. It then computes a signature over the following structure:

```
struct {
    uint8 label[31] = "Merkle Tree Certificate Window\0";
    opaque issuer_id<1..32>;
    Window window;
} LabeledWindow;
```

The label field is an ASCII string. The final byte of the string, "\0", is a zero byte, or ASCII NULL character. The issuer_id field is the CA's issuer_id. Other parties can verify the signature by constructing the same input and verifying with the CA's public_key.

The CA saves this signature as the batch's window signature. It then updates the latest batch to point to batch_number. If the CA's private key signs an input that can be interpreted as a LabeledWindow structure, the CA is considered to have certified every assertion contained in every value in the tree_heads list, with expiry determined by batch_number, the position of the tree head in the list, and the CA's input parameters as described in Section 5.1.

5.4.3. Certificate Format

[[TODO: BikeshedCertificate is a placeholder name until someone comes up with a better one.]]

For each assertion in the tree, the CA constructs a BikeshedCertificate structure containing the assertion and a proof. A proof is a message that allows the relying party to accept the associated assertion, provided it trusts the CA and recognizes the tree head. The structures are defined below:

```
enum { merkle_tree_sha256(0), (2^16-1) } ProofType;

struct {
    ProofType proof_type;
    opaque trust_anchor_data<0..2^8-1>;
} TrustAnchor;

struct {
    TrustAnchor trust_anchor;
    opaque proof_data<0..2^24-1>;
} Proof;

struct {
    Assertion assertion;
    Proof proof;
} BikeshedCertificate;
```

The proof_type identifies a type of proof. It determines the format of the trust_anchor_data and proof_data values. The mechanism defined in this document is merkle_tree_sha256, which uses trust_anchor_data and proof_data formats of MerkleTreeTrustAnchor and MerkleTreeProofSHA256, respectively:

```
struct {
    opaque issuer_id<1..32>;
    uint32 batch_number;
} MerkleTreeTrustAnchor;

opaque HashValueSHA256[32];

struct {
    uint64 index;
    HashValueSHA256 path<32..2^16-1>;
} MerkleTreeProofSHA256;
```

A trust anchor is a short identifier that identifies a source of certificates. It is analogous to an X.509 trust anchor's subject name. These are used for certificate selection, described in Section 6.3. In Merkle Tree certificates, each batch is a distinct trust anchor. The trust_anchor_data for merkle_tree_sha256 is a MerkleTreeTrustAnchor structure. The issuer_id field is the CA's issuer_id. The batch_number field is the number of the batch.

A relying party that trusts a trust anchor must know the batch's tree head. It then trusts any assertion which can be proven to be in the corresponding Merkle Tree, as described in Section 6.2.

The proof_data for merkle_tree_sha256 is a MerkleTreeProofSHA256. After building the tree, the CA constructs a MerkleTreeProofSHA256 for each assertion as follows. For each index i in the batch's assertion list:

1. Set index to i. This will be a value between 0 and n-1, inclusive.

2. Set path to an array of l-1 hashes. Set element j of this array to element k of level j, where k is (i >> j) ^ 1. >> denotes a bitwise right-shift, and ^ denotes a bitwise exclusive OR (XOR) operation. This element is the sibling of an ancestor of assertion i in the tree. Note the tree head is never included.

For example, the path value for the third assertion in a batch of three assertions would contain the marked nodes in Figure 3, from bottom to top.

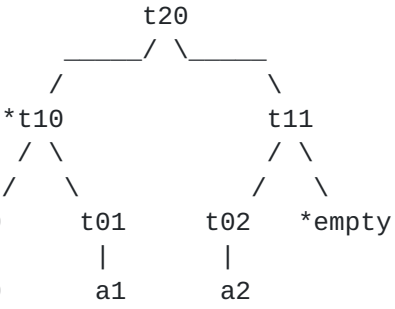

Figure 3: An example Merkle Tree proof for the third of three assertions

If the batch only contained one assertion, path will be empty and index will be zero.

For each assertion, the CA assembles a BikeshedCertificate structure and sends it to the subscriber. It SHOULD also send the additional information described in Section 6.3.

This certificate can be presented to supporting relying parties as described in Section 6. It is valid until the batch expires.

5.5. Size Estimates

Merkle Tree proofs scale logarithmically in the batch size. Section 11.2 recommends subscribers renew halfway through the previous certificate's lifetime. Batch sizes will thus, on average, be subscriber_count * 2 / window_size, where subscriber_count is a CA's active subscriber count. The recommended parameters in Section 5.1 give an average of subscriber_count / 168.

Some organizations have published statistics which can estimate batch sizes for the Web PKI. On March 7th, 2023, [LetsEncrypt] reported around 330,000,000 active subscribers for a single CA. [MerkleTown] reported around 3,800,000,000 unexpired certificates in Certificate Transparency logs, and an issuance rate of around 257,000 per hour. Note the numbers from [MerkleTown] represent, respectively, all Web PKI CAs combined and issuance rates for longer-lived certificates and may not be representative of a Merkle Tree certificate deployment.

These three estimates correspond to batch sizes of, respectively, around 2,000,000, around 20,000,000, and 257,000. The corresponding path lengths will be 20, 24, and 17, given proof sizes of, respectively, 640 bytes, 768 bytes, and 544 bytes.

For larger batch sizes, 32 hashes, or 1024 bytes, is sufficient for batch sizes up to 2^33 (8,589,934,592) certificates.

6. Using Certificates

This section describes how subscribers present and relying parties verify Merkle Tree certificates.

6.1. Relying Party State

For each Merkle Tree CA it trusts, a relying party maintains a copy of the most recent window from the CA. This structure determines which certificates the relying party will accept. It is regularly updated from the transparency service, as described in Section 7.

6.2. Certificate Verification

When a subscriber presents a BikeshedCertificate whose proof_type field is merkle_tree_sha256, the relying party runs the following procedure to verify it. This procedure's error conditions are described with TLS alerts, defined in Section 6.2 of [RFC8446]. Non-TLS applications SHOULD map these error conditions to the corresponding application-specific errors. When multiple error conditions apply, the application MAY return any applicable error.

1. Decode the trust_anchor_data and proof_data fields as MerkleTreeTrustAnchor and MerkleTreeProofSHA256 structures, respectively. If they cannot be decoded, abort this procedure with a bad_certificate error.

2. Check if the certificate's issuer_id corresponds to a trusted Merkle Tree CA with a saved window. If not, abort this procedure with an unknown_ca error.

3. Check if the certificate's batch_number is contained in the saved window. If not, abort this procedure with a unknown_ca error.

4. Compute the expiration time of the certificate's batch_number, as described in Section 5.1. If this value is before the current time, abort this procedure with a certificate_expired error.

5. Set hash to the output of HashAssertion(assertion). Set remaining to the certificate's index value.

6. For each element v at zero-based index i of the certificate's path field, in order:

   * If remaining is odd, set hash to the output of HashNode(v, hash, i + 1, remaining >> 1). Otherwise, set hash to the output of HashNode(hash, v, i + 1, remaining >> 1)

   * Set remaining to remaining >> 1.

7. If remaining is non-zero, abort this procedure with an error.

8. If hash is not equal to the corresponding tree head in the saved window, abort this procedure with a bad_certificate error.

9. Optionally, perform any additional application-specific checks on the assertion and issuer. For example, an HTTPS client might constrain an issuer to a particular DNS subtree.

10. If all the preceding checks succeed, the certificate is valid and the application can proceed with using the assertion.

6.3. Certificate Negotiation

Merkle Tree certificates can only be presented to up-to-date relying parties, so this document describes a mechanism for subscribers to select certificates. This section describes the general negotiation mechanism. Section 10.3 describes it as used in TLS.

Subscribers maintain a certificate set of available BikeshedCertificates. The TrustAnchor value in each BikeshedCertificate is known as the primary TrustAnchor. Each BikeshedCertificate is also associated with the following values:

- A set of additional TrustAnchor values which also match this certificate
- An expiration time, after which the certificate is no longer usable

These values can be computed from the BikeshedCertificate, given knowledge of the ProofType value and the CA's parameters. However, CAs are RECOMMENDED to send this information to subscribers in a ProofType-independent form. See Section 9 for how this is represented in ACME. This simplifies subscriber deployment and improves ecosystem agility, by allowing subscribers to use certificates without precise knowledge of their parameters.

For Merkle Tree certificates, the expiration time is computed as described in Section 5.1. There are window_size - 1 additional TrustAnchor values: for each i from 1 to window_size - 1, make a copy of the primary TrustAnchor with the batch_number value replaced with batch_number + i.

Each relying party maintains a set of TrustAnchor values, which describe the certificates it accepts. This set is sent to the subscriber to aid in certificate selection. The ProofType code point defines how the relying party determines the TrustAnchor values. For Merkle Tree certificates, the proof_type is merkle_tree_sha256, the issuer_id is the CA's issuer_id, and the batch_number is the batch_number of the relying party's window.

The subscriber compares this set with its certificate set. A certificate is eligible if all of the following are true:

- The current time is before the certificate's expiration time
- Either the certificate's primary TrustAnchor value or one of the additional TrustAnchor values appears in the relying party's TrustAnchor set.
- Any additional application-specific constraints hold. For example, the TLS signature_algorithms (Section 4.2.3 of [RFC8446]) extension constrains the types of keys which may be used.

The subscriber SHOULD select the smallest available certificate where the above checks succeed. When two comparably-sized certificates are available, the subscriber SHOULD select the one with the later expiration time, to reduce clock skew risks. If no certificate is available, the subscriber SHOULD fallback to another PKI mechanism, such as X.509.

7. Transparency Services

This section describes the role of the transparency service. The transparency service ensures all certificates accepted by the relying party are consistently and publicly logged. It performs three functions:

- Mirror all assertions certified by the CA and present them to monitors
- Validate all tree heads and windows produced by the CA
- Provide the latest valid window to relying parties

In doing so, the transparency service MUST satisfy the following requirements:

- The mirrored CA state is append-only. That is, the hashes, signatures, and assertions for a given batch number MUST NOT change.
- All windows sent to relying parties MUST be reflected in the mirrored CA state. That is, each tree hash in a window MUST correspond to a mirrored tree hash, consistent with the corresponding mirrored assertions.

The transparency service publishes the mirrored CA state using the same interface as Section 8. The protocol between the relying party and transparency service is out of scope of this document. The relying party MAY use the interface defined here, or an existing application-specific authenticated channel.

As discussed in Section 13.1, relying parties MUST ensure that any windows obtained were asserted by the CA. This SHOULD be done by having the transparency service forward the CA's signature, with the relying party verifying it. However, if the transparency service already maintains a trusted, authenticated channel to the relying parties (e.g. a software or root store update channel), relying parties MAY rely on the transparency service to validate the CA windows on their behalf, only sending valid windows over this channel.

Although described as a single service for clarity, the transparency service may be implemented as a combination of services run by multiple entities, depending on security goals. For example deployments, this section first describes a single trusted service,

then it describes other possible models where trust is divided between entities.

7.1. Single Trusted Service

Some relying parties regularly contact a trusted update service, either for software updates or to update individual components, such as the services described in [CHROMIUM] and [FIREFOX]. Where these services are already trusted for the components such as the trust anchor list or certificate validation software, a single trusted transparency service may be a suitable model.

The transparency service maintains a mirror of the CA's latest batch number, and batch state. Roughly once every batch_duration, it polls the CA's HTTP interface (see Section 8) and runs the following steps:

1. Fetch the CA's latest batch number. If this fetch fails, abort this procedure with an error.

2. Let new_latest_batch be the result and old_latest_batch be the currently mirrored value. If new_latest_batch equals old_latest_batch, finish this procedure without reporting an error.

3. If new_latest_batch is less than old_latest_batch, abort this procedure with an error.

4. If the issuance date for batch new_latest_batch is in the future (see Section 5.1), abort this procedure with an error.

5. For all i such that old_latest_batch < i <= new_latest_batch:

    1. Fetch the signature, tree head, and assertion list for batch i. If this fetch fails, abort this procedure with an error.

    2. Compute the tree head for the assertion list, as described in Section 5.4.1. If this value does not match the fetched tree head, abort this procedure with an error.

    3. Compute the Window structure and verify the signature, as described in Section 5.4.2. Set tree_heads[0] to the tree head fetched above. Set the other values in tree_heads to the previously mirrored values. If signature verification fails, abort this procedure with an error.

    4. Set the mirrored latest batch number to i and save the fetched batch state.

[[TODO: If the mirror gets far behind, if the CA just stops publishing for a while, it may suddenly have to catch up on many batches. Should we allow the mirror to catch up to the latest window

and skip the intervening batches? The intervening batches are guaranteed to have been expired]]

7.2. Single Update Service with Multiple Mirrors

If the relying party has a trusted update service, but the update service does not have the resources to mirror the full batch state, the transparency service can be composed of this update service and several, less trusted mirrors. In this model, the mirrors are not trusted to serve authoritative trust anchor information to relying parties, but the update service trusts at least half of them to faithfully and consistently mirror the batch state.

Each mirror follows the procedure in Section 7.1 to maintain and publish a mirror of the CA's batch state.

The update server maintains the latest window validated to appear in all mirrors. It updates this by polling the mirrors and running the following steps:

1. For each mirror, fetch the latest batch number.

2. Let new_latest_batch be the highest batch number that is bounded by the value fetched from at least half of the mirrors. Let old_latest_batch be the batch number of the currently stored window.

3. If new_latest_batch equals old_latest_batch, finish this procedure without reporting an error.

4. If new_latest_batch is less than old_latest_batch, abort this procedure with an error.

5. If the issuance date for batch new_latest_batch is in the future (see Section 5.1), abort this procedure with an error.

6. Fetch the window with new_latest_batch from each mirror that returned an equal or higher latest batch number. If any fetches fail, or if the results do not match across all mirrors, abort this procedure with an error.

7. Verify the window signature, as described in Section 5.4.2. If the signature is invalid, abort this procedure with an error.

8. If the old and new windows contain overlapping batch numbers, verify that the tree hashes match. If not, abort this procedure with an error.

9. Update the saved window with the new value.

Compared to Section 7.1, this model reduces trust in the mirror services, but can delay certificate usability if some of the mirrors consume CA updates too slowly. This can be tuned by adjusting the threshold in step 2.

In a transparency service using this model, each mirror independently publishes the batch state via Section 8.

7.3. Multiple Transparency Services

Relying parties without a trusted update service can fetch from mirrors directly. Rather than relying on the update service to fetch the window state, the relying party runs the procedure described in Section 7.2, and uses the saved window to verify certificates.

7.4. Monitors

Monitors in this document are analogous to monitors in [RFC6962]. Monitors watch an implementation of the HTTP APIs in Section 8 to verify correct behavior and watch for certificates of interest. This is typically the transparency service. A monitor needs to, at least, inspect every new batch. It may also maintain a copy of the batch state.

It does so by following the procedure in Section 7.1, fetching from the service being monitored. If the procedure fails for a reason other than the service availability, this should be viewed as misbehavior on the part of the service. If the procedure fails due to service availability and the service remains unavailable for an extended period, this should also be viewed as misbehavior. If the monitor is not maintaining a copy of the batch state, it skips saving the assertions.

[RFC6962] additionally defines the role of auditor, which validates that Signed Certificate Timestamps (SCTs) and Signed Tree Heads (STHs) in Certificate Transparency are correct. There is no analog to SCTs in this document. The signed window structure (Section 5.4.2) is analogous to an STH, but consistency is checked simply by ensuring overlapping tree heads match, so this document does not define this as an explicit role. If two inconsistent signed windows are ever observed from a Merkle Tree CA, this should be viewed as misbehavior on the part of the CA.

8. HTTP Interface

[[TODO: This section hasn't been written yet. For now, this is just an informal sketch. The real text will need to define request/response formats more precisely, with MIME types, etc.]]

CAs and transparency services publish state over an HTTP [RFC9110] interface described below.

CAs and any components of the transparency service that maintain window information implement the following interfaces:

* GET {prefix}/latest returns the latest batch number.

* GET {prefix}/window/latest returns the Window structure and signature (see Section 5.4.2) for the latest batch number.

*GET {prefix}/window/{number} returns the Window structure and signature (see Section 5.4.2) for batch number, if it is in the "issued" state, and a 404 error otherwise.

*GET {prefix}/batch/{number}/info returns the window signature and tree head for batch number, if batch number is in the "issued" state, and a 404 error otherwise.

CAs and any components of the transparency service that mirror the full assertion list additionally implement the following interface:

*GET {prefix}/batch/{number}/assertions returns the assertion list for batch number, if number is in the issued state, and a 404 error otherwise.

If the interface is implemented by a distributed service, with multiple servers, updates may propagate to servers at different times, which will cause temporary inconsistency. This inconsistency can impede this system's transparency goals (Section 13.4).

Services implementing this interface SHOULD wait until batch state is fully propagated to all servers before updating the latest batch number. That is, if any server returns a latest batch number of N in either of the first two HTTP endpoints, batch numbers N and below SHOULD be available under the last three batch-number-specific HTTP endpoints in all servers. If this property does not hold at any time, it is considered a service unavailability.

Individual servers in a service MAY return different latest batch numbers. Individual servers MAY also differ on whether a batch number has a response available or return a 404 error. Provided the above consistency property holds, these two inconsistencies do not constitute service unavailability.

Section 11.5 discusses service availability requirements.

[[TODO: Once a batch has expired, do we allow a CA to stop publishing it? The transparency service can already log it for as long, or as little, as it wishes. We effectively have CT log temporal sharding built into the system.]]

[[TODO: If we have the window endpoint, do we still need to separate "info" and "assertions"?]]

9. ACME Extensions

[[TODO: This section hasn't been written yet. Instead, what follows is an informal sketch and design discussion.]]

See Section 11.4 for the overall model this should target.

Define ACME [RFC8555] extensions for requesting these. We probably need to add a new field in the Order object (Section 9.7.2 of [RFC8555]) to request this. Also a new MIME type for the thing being

fetched Section 7.4.2 of [RFC8555]. This format should capture additional metadata per Section 6.3.

Otherwise, the long issuance time is already modeled by the allowance for the "processing" state taking a while. The ACME server should use the Retry-After header so the subscriber knows when to query again.

Also use [I-D.draft-ietf-acme-ari] to move the renewal logic in Section 11.2 from the subscriber to the ACME server.

Per Section 11.4, a subscriber may need multiple certificates. That should be a service provided by the ACME server. Come up with a scheme to mint multiple orders from a single newOrder request, or request multiple certificates off of a single order. (Note different certificates may have different processing time. It seems an ACME order only transitions from the "processing" state to the "valid" state once, so the former is probably better.)

We should also define a certificate request format, though it is broadly just reusing the Assertion structure. If the CA wishes to check possession of the private key, it'll need to come with a signature or do some online operation (e.g. if it's a KEM key). This is inherently protocol-specific, because the mechanism needs to coexist with the target protocol. (Signed CSRs implicitly assume the target protocol's signature payloads cannot overlap with that of a CSR.)

10. Use in TLS

10.1. TLS Subjects

This section describes the SubjectType for use with TLS [RFC8446]. The SubjectType value is tls, and the subject_info field contains a TLSSubjectInfo structure, defined below:

```
enum { tls(0), (2^16-1) } SubjectType;

struct {
    SignatureScheme signature;
    opaque public_key<1..2^24-1>;
    /* TODO: Should there be an extension list? */
} TLSSubjectInfo;
```

A TLSSubjectInfo describes a TLS signing key. The signature field is a SignatureScheme Section 4.2.3 of [RFC8446] value describing the key type and signature algorithm it uses for CertificateVerify.

The public_key field contains the subscriber's public key. The encoding is determined by the signature field as follows:

RSASSA-PSS algorithms: The public key is an RSAPublicKey structure [RFC8017] encoded in DER [X.690]. BER encodings which are not DER MUST be rejected.

ECDSA algorithms:
The public key is a UncompressedPointRepresentation structure defined in Section 4.2.8.2 of [RFC8446], using the curve specified by the SignatureScheme.

EdDSA algorithms: The public key is the byte string encoding defined in [RFC8032]

This document does not define the public key format for other algorithms. In order for a SignatureScheme to be usable with TLSSubjectInfo, this format must be defined in a corresponding document.

[[TODO: If other schemes get defined before this document is done, add them here. After that, it's on the other schemes to do it.]]

10.2. The Bikeshed Certificate Type

[[TODO: Bikeshed is a placeholder name until someone comes up with a better one.]]

This section defines the Bikeshed TLS certificate type, which may be negotiated with the client_certificate_type, server_certificate_type [RFC7250], or cert_type [RFC6091] extensions. It can only be negotiated with TLS 1.3 or later. Servers MUST NOT negotiate it in TLS 1.2 or below. If the client receives a ServerHello that negotiates it in TLS 1.2 or below, it MUST abort the connection with an illegal_parameter alert.

[[TODO: None of these three extensions is quite right for client certificates because the negotiation isn't symmetric. See discussion in Section 10.4. We may need to define a third one.]]

When negotiated, the CertificateEntry structure in the Certificate message is updated as follows:

```
enum { Bikeshed(TBD), (255) } CertificateType;

struct {
    select (certificate_type) {
        // certificate type defined in this document.
        case Bikeshed:
            BikeshedCertificate certificate;

        case RawPublicKey:
            /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
            opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
            opaque cert_data<1..2^24-1>;

        /* Additional certificate types based on the
          "TLS Certificate Types" registry */
    };
    Extension extensions<0..2^16-1>;
} CertificateEntry;
```

The subject_type field in the certificate MUST be of type tls (Section 10.1). The CertificateVerify message is computed and processed as in [RFC8446], with the following modifications:

- The signature is computed and verified with the key described in the TLSSubjectInfo. The relying party uses the key decoded from the public_key field, and the subscriber uses the corresponding private key.

- The SignatureScheme in the CertificateVerify MUST match the signature field in the TLSSubjectInfo.

The second modification differs from [RFC8446]. Where [RFC8446] allowed an id-rsaEncryption key to sign both rsa_pss_rsae_sha256 and rsa_pss_rsae_sha384, TLSSubjectInfo keys are specific to a single algorithm. Future documents MAY relax this restriction for a new SignatureScheme, provided it was designed to be used concurrently with the value in TLSSubjectInfo. In particular, the underlying signature algorithm MUST match, and there MUST be appropriate domain separation between the two modes. For example, [I-D.draft-ietf-tls-batch-signing] defines new SignatureSchemes, but the same keypair can be safely used with one of the new values and the corresponding base SignatureScheme.

If this certificate type is used for either the client or server certificate, the ALPN [RFC7301] extension MUST be negotiated. If no application protocol is selected, endpoints MUST close the connection with a no_application_protocol alert.

[[TODO: Suppose we wanted to introduce a second SubjectType for TLS, either to add new fields or capture a new kind of key. That would need to be negotiated. We could use another extension, but defining

a new certificate type seems most natural. That suggests this certificate type isn't about negotiating BikeshedCertificate in general, but specifically SubjectType.tls and TLSSubjectInfo. So perhaps the certificate type should be TLSSubjectInfo or BikeshedTLS.]]

10.3. The Trust Anchors Extension

The TLS trust_anchors extension which implements certificate negotiation (see Section 6.3). The extension body is a TrustAnchors structure, defined below:

```
enum { trust_anchors(TBD), (2^16-1) } ExtensionType;

struct {
    TrustAnchor trust_anchors<1..2^16-1>;
} TrustAnchors;
```

This extension carries the relying party's trust anchor set, computed as described in Section 6.3. When the client is the relying party for a server certificate, the extension is sent in the ClientHello. When the server is the relying party for a client certificate, the extension is sent in the CertificateRequest message. This extension is only defined for use with TLS 1.3 and later. It MUST be ignored when negotiating TLS 1.2.

When the subscriber receives this extension, selects a certificate from its certificate set, as described in Section 6.3. If none match, it does not negotiate the Bikeshed type and selects a different certificate type. [[TODO: This last step does not work. See Section 10.4]]

10.4. Certificate Type Negotiation

[[TODO: We may need a new certificate types extension, either in this document or a separate one. For now, this section just informally describes the problem.]]

The server certificate type is negotiated as follows:

- The client sends server_certificate_type in ClientHello with accepted certificate types.
- The server selects a certificate type to use, It sends it in server_certificate_type in EncryptedExtensions.
- The server sends a certificate of the server-selected type in Certificate.

This model allows the server to select its certificate type based on not just server_certificate_type, but also other ClientHello extensions like certificate_authorities or trust_anchors (Section 10.3). In particular, if there is no match in

trust_anchors, it can fallback to X.509, rather than staying within the realm of BikeshedCertificate.

However, the client certificate type is negotiated differently:

- The client sends client_certificate_type in ClientHello with certificates it can send
- The server selects a certificate type to request. It sends it in client_certificate_type in EncryptedExtensions.
- The server requests a client certificate in CertificateRequest
- The client sends a certificate of the server-selected type in Certificate.

Here, the client (subscriber) does not select the certificate type. The server (relying party) does. Moreover, this selection is made before the client can see the server's certificate_authorities or trust_anchors value, in CertificateRequest. There is no opportunity for the client to fallback to X.509.

The cert_types extension behaves similarly, but additionally forces the client and server types to match. These extensions were defined when TLS 1.2 was current, but TLS 1.3 aligns the client and server certificate negotiation. Most certificate negotiation extensions, such as certificate_authorities or compress_certificate [RFC8879] can be offered in either direction, in ClientHello or CertificateRequest. They are then symmetrically accepted in the Certificate message.

A more corresponding TLS 1.3 negotiation would be to defer the client certificate type negotiation to CertificateRequest, with the server offering the supported certificate types. The client can then make its selection, taking other CertificateRequest extensions into account, and indicate its selection in the Certificate message.

Two possible design sketches:

10.4.1. Indicate in First CertificateEntry

We can have the subscriber indicate the certificate type in an extension of the first CertificateEntry. One challenge is the extensions come after the certificate, so the relying party must seek to the extensions field independent of the certificate type. Thus all certificate types must be updated to use a consistent opaque cert_data<0..2^24> syntax, with any type-specific structures embedded inside.

RawPublicKey and X509 already meet this requirement. OpenPGP and Bikeshed need an extra length prefix.

10.4.2. Change Certificate Syntax

Alternatively, we can negotiate an extension that changes the syntax to Certificate to:

```
struct {
    CertificateType certificate_type;
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

The negotiation can be:

- Client sends its accepted certificate types in ClientHello. Offering this new extension also signatures it is willing to accept the new message format. Unlike the existing extensions, an X.509-only client still sends the extension with just X509 in the list.

- Server, if it implements the new syntax, acknowledges the syntax change with an empty extension in EncryptedExtensions. (It doesn't indicate its selection yet.)

- If both of the above happen, Certificate's syntax has changed. Server indicates its selection with the certificate_type field

- Server can also send this extension in CertificateRequest to offer non-X.509 certificate types

- Client likewise indicates its selection with the certificate_type field.

This is a bit cleaner to parse, but the negotiation is more complex.

11. Deployment Considerations

11.1. Fallback Mechanisms

Subscribers using Merkle Tree certificates SHOULD additionally provision certificates from another PKI mechanism, such as X.509. This ensures the service remains available to relying parties that have not recently fetched window updates, or lack connectivity to the transparency service.

If the pipeline of updates from the CA to the transparency service to relying parties is interrupted, certificate issuance may halt, or newly issued certificates may no longer be usable. When this happens, the optimization in this document may fail, but fallback mechanisms ensure services remain available.

11.2. Rolling Renewal

When a subscriber requests a certificate, the CA cannot fulfill the request until the next batch is ready. Once published, the

certificate will not be accepted by relying parties until the batch state is mirrored by their respective transparency services, then pushed to relying parties.

To account for this, subscribers SHOULD request a new Merkle Tree certificate significantly before the previous Merkle Tree certificate expires. Renewing halfway into the previous certificate's lifetime is RECOMMENDED. Subscribers additionally SHOULD retain both the new and old certificates in the certificate set until the old certificate expires. As the new tree hash is delivered to relying parties, certificate negotiation will transition relying parties to the new certificate, while retaining the old certificate for clients that are not yet updated.

11.3. Deploying New Keys

Merkle Tree certificates' issuance delays make them unsuitable when rapidly deploying a new service and reacting to key compromise.

When a new service is provisioned with a brand new Merkle Tree certificate, relying parties will not yet have received a window containing this certificate from the transparency service and can therefore not validate this certificate until receiving them. The subscriber SHOULD, in parallel, also provision a certificate using another PKI mechanism (e.g. X.509). Certificate negotiation will then switch over to serving the Merkle Tree certificate as relying parties are updated.

If the service is performing a routine key rotation, and not in response to a known compromise, the subscriber MAY use the process described in Section 11.2, allowing certificate negotiation to also switch the private key used. 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 available. Instead, the service SHOULD immediately discard the old key and request a more immediate issuance mechanism. As in the initial deployment case, it SHOULD request a Merkle Tree certificate in parallel, which will restore the size optimization over time.

11.4. Agility and Extensibility

Beyond negotiating Merkle Tree certificates, certificate negotiation can also handle variations in which CAs a relying party trusts. With a single certificate, the subscriber is limited to the intersection of these sets. Instead, Section 6.3 allows a subscriber to maintain multiple certificates that, together, encompass the relying parties it supports.

This improves trust agility. If a relying party distrusts a CA, a subscriber can include certificates from both the distrusted CA and a replacement CA. This allows the distrusting relying party to

request the replacement CA, while existing relying parties, which may not trust the replacement CA, can continue to use the distrusted CA. Likewise, an entity operating a CA may deploy a second CA to rotate key material. The certificate set can include both the new and old CA to ensure a smooth transition.

Moreover, Section 6.3 allows subscribers to select certificates without recognizing either the CA or the ProofType. Only the Assertion structure directly impacts the application protocol on the subscriber's side. This allows for a more flexible deployment model where ACME servers, or other certificate management services, assemble the certificate set:

Instead of each subscriber being individually configured with the CAs to use, the ACME server can provide multiple certificates, covering all supported relying parties. As relying party requirements evolve, CAs rotate keys, or new ProofTypes are designed, the ACME server is updated to incorporate these into certificate sets. As the PKI evolves, subscribers are automatically provisioned appropriately.

11.5. Batch State Availability

CAs and transparency services serve an HTTP interface defined in Section 8. This service may be temporarily unavailable, either from service outage or if the service does not meet the consistency condition mid-update. Exact availability requirements for these services are out of scope for this document, but this section provides some general guidance.

If the CA's interface becomes unavailable, the transparency service will be unavailable to update. This will prevent relying parties from accepting new certificates, so subscribers will need to use fallback mechanisms per Section 11.1. This does not compromise transparency goals per Section 13.4.2. However, a CA which is persistently unavailable may not offer sufficient benefit to be used by subscribers or trusted by relying parties.

However, if the transparency service's interface becomes unavailable, monitors will be unable to check for unauthorized certificates. This does compromise transparency goals. Mirrors of the batch state partially mitigate this, but service unavailability may prevent mirrors from replicating a batch that relying parties accept.

11.6. Trust Anchor List Size

Section 6.3 and Section 10.3 involve the relying party sending a list of TrustAnchor values to aid the subscriber in selecting certificates. A sufficiently large list may be impractical to fit in a ClientHello and require alternate negotiation mechanisms or a different PKI structure. To reduce overhead, issuer_id values SHOULD be short, no more than eight bytes long.

12. Privacy Considerations

The negotiation mechanism described in Section 6.3 and Section 10.3 presumes the relying party's trust anchor list is not sensitive. In particular, information sent in a TLS ClientHello is unencrypted without the Encrypted ClientHello extension [I-D.draft-ietf-tls-esni].

This mechanism SHOULD NOT be used in contexts where the list reveals information about an individual user. For example, a web browser may support both a common set of trust anchors configured by the browser vendor, and a set of user-specified trust anchors. The common trust anchors would only reveal which browser is used, while the user-specified trust anchors may reveal information about the user. In this case, the trust anchor list SHOULD be limited to the common trust anchors.

Additionally, even if all users are served the same updates, individual users may fetch from the transparency service at different times, resulting in variation in the trust anchor list. Like other behavior changes triggered by updates, this may, when combined with other sources of user variation, lead to a fingerprinting attack [RFC6973].

13. Security Considerations

13.1. Authenticity

A key security requirement of any PKI scheme is that relying parties only accept assertions that were certified by a trusted certification authority. This is achieved by the following two properties:

* The relying party MUST NOT accept any window that was not authenticated as coming from the CA.

* For any tree head computed from a list of assertions as in Section 5.4.1, it is computationally infeasible to construct an assertion not this list, and some inclusion proof, such that the procedure in Section 6.2 succeeds.

Section 7 discusses achieving the first property.

The second property is achieved by using a collision-resistant hash in the Merkle Tree construction. The HashEmpty, HashNode, and HashAssertion functions use distinct initial bytes when calling the hash function, to achieve domain separation.

13.2. Cross-protocol attacks

Using the same key material in different, incompatible ways risks cross-protocol attacks when the two uses overlap. To avoid this, Section 5.1 forbids the reuse of Merkle Tree CA private keys in another protocol.

To reduce the risk of attacks if this guidance is not followed, the LabeledWindow structure defined in Section 5.4.2 includes a label string, and the CA's issuer_id. Extensions of this protocol MAY be defined which reuse the keys, but any that do MUST use a different label string and analyze the security of the two uses concurrently.

Likewise, key material included in an assertion (Section 4) MUST NOT be used in another protocol, unless that protocol was designed to be used concurrently with the original purpose. The Assertion structure is designed to facilitate this. Where X.509 uses an optional key usage extension (see Section 4.2.1.3 of [RFC5280]) and extended key usage extension (see Section 4.2.1.12 of [RFC5280]) to specify key usage, an Assertion is always defined first by a SubjectType value. Subjects cannot be constructed without first specifying the type, and subjects of different types cannot be accidentally interpreted as each other.

The TLSSubjectInfo structure additionally protects against cross-protocol attacks in two further ways:

* A TLSSubjectInfo specifies the key type not with a SubjectPublicKeyInfo Section 4.1.2.7 of [RFC5280] object identifier, but with a SignatureScheme structure. Where [RFC8446] allows an id-rsaEncryption key to sign both rsa_pss_rsae_sha256 and rsa_pss_rsae_sha384, this protocol specifies the full signature algorithm parameters.

* To mitigate cross-protocol attacks at the application protocol [ALPACA], this document requires connections using it to negotiate the ALPN [RFC7301] extension.

13.3. Revocation

Merkle Tree certificates avoid sending an additional signature for OCSP responses by using a short-lived certificates model. Per Section 5.1, Merkle Tree CA's certificate lifetime MUST be set such that certificate expiration replaces revocation. Existing revocation mechanisms like CRLs and OCSP are themselves short-lived, signed messages, so a low enough certificate lifetime provides equivalent revocation capability.

Relying parties with additional sources of revocation such as [CRLite] or [CRLSets] SHOULD provide a mechanism to express revoked assertions in such systems, in order to opportunistically revoke assertions in up-to-date relying parties sooner. It is expected that, in most deployments, relying parties can fetch this revocation data and Merkle Tree CA windows from the same service.

[[TODO: Is it worth defining an API for Merkle Tree CAs to publish a revocation list? That would allow automatically populating CRLite and CRLSets. Maybe that's a separate document.]]

13.4. Transparency

The transparency service does not prevent unauthorized certificates, but it aims to provide comparable security properties to Certificate Transparency [RFC6962]. If a subscriber presents an acceptable Merkle Tree certificate to a relying party, the relying party should have assurance it was published in some form that monitors and, in particular, the subject of the certificate will be able to notice.

13.4.1. Unauthorized Certificates

If a Merkle Tree certificate was unauthorized, but seen and mirrored by the transparency service, the relying party may accept it. However, provided the transparency service is operating correctly, this will be detectable. Unlike Certificate Transparency, Merkle Tree certificates achieve this property without a Maximum Merge Delay (MMD). Certificates are fully mirrored by the transparency service before the relying party will accept them. However, this comes at the cost of immediate issuance, as described in Section 11.

If the unauthorized certificate was not seen by the transparency service, the relying party will reject it. In order to accept a certificate, the relying party must have been provisioned with the corresponding tree head. A correctly operating transparency service will never present relying parties with tree heads unless the corresponding certificates have all been mirrored.

13.4.2. Misbehaving Certification Authority

Although CAs in this document publish structures similar to a Certificate Transparency log, they do not need to function correctly to provide transparency.

A CA could violate the append-only property of its batch state, and present differing views to different parties. Unlike a misbehaving Certificate Transparency log, this would not compromise transparency. Whichever view is presented to the transparency service at the time of updates determines the canonical batch state for both relying parties and monitors. Certificates that are consistent with only the other view will be rejected by relying parties. If the transparency service observes both views, the procedures in Section 7 will prevent the new, conflicting view from overwriting the originally saved view. Instead, the update process will fail and further certificates will not be accepted.

A CA could also sign a window containing an unauthorized certificate and feign an outage when asked to serve the corresponding assertions. However, if the assertion list was never mirrored by the transparency service, the tree head will never be pushed to relying parties, so the relying party will reject the certificate. If the assertion list was mirrored, the unauthorized certificate continues to be available to monitors.

As a consequence, monitors MUST use the transparency service's view of the batch state when monitoring for unauthorized certificates. If the transparency service is a collection of mirrors, as in Section 7.2 or Section 7.3, monitors MUST monitor each mirror. Monitors MAY optionally monitor the CA directly, but this alone is not sufficient to avoid missing certificates.

13.4.3. Misbehaving Transparency Service

This document divides CA and transparency service responsibilities differently from how [RFC6962] divides CA and Certificate Transparency log. The previous section describes the implications of a failure to meet the log-like responsibilities of a CA, provided the transparency service is operating correctly.

For the remainder of log-like responsibilities, the relying party trusts its choice of transparency service deployment to ensure the windows it uses are consistent with what monitors observe. Otherwise, a malicious transparency service and CA could collude to cause a relying party to accept an unauthorized certificate not visible to monitors. Where a single trusted service is not available, the Section 7 discusses possible deployment structures where the transparency service is a collection of mirrors, all or most of whom must collude instead.

13.5. Security of Fallback Mechanisms

Merkle Tree certificates are intended to be used as an optimization over other PKI mechanisms. More generally, Section 6.3 and Section 11.4 allow relying parties to support many kinds of certificates, to meet different goals. This document discusses the security properties of Merkle Tree certificates, but the overall system's security properties depend on all of a relying party's trust anchors.

In particular, in relying parties that require a publicly auditable PKI, the supported fallback mechanisms must also provide a transparency property, either with Certificate Transparency [RFC6962] or another mechanism.

14. IANA Considerations

IANA is requested to create the following entry in the TLS ExtensionType registry [RFC8447]. The "Reference" column should be set to this document.

Value	Extension Name	TLS 1.3	DTLS-Only	Recommended
TBD	trust_anchors	CH, CR	N	TBD

Table 1: Additions to the TLS ExtensionType Registry

IANA is requested to create the following entry in the TLS Certificate Types registry [RFC8447]. The "Reference" column should be set to this document.

Value	Name	Recommended
TBD	Bikeshed	TBD

Table 2: Additions to the TLS Certificate Types Registry

TODO: Define registries for the enums introduced in this document:

*SubjectType

*ClaimType

*ProofType

15. References

15.1. Normative References

[I-D.draft-ietf-uta-rfc6125bis] Saint-Andre, P. and R. Salz, "Service Identity in TLS", Work in Progress, Internet-Draft, draft-ietf-uta-rfc6125bis-11, 2 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-uta-rfc6125bis-11>.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/rfc/rfc1034>.

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <https://www.rfc-editor.org/rfc/rfc1123>.

[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>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/rfc/rfc5890>.

[RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC

6091, DOI 10.17487/RFC6091, February 2011, <https://www.rfc-editor.org/rfc/rfc6091>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/rfc/rfc7250>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/rfc/rfc7301>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

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

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

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

[RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/rfc/rfc8447>.

[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>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 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.

15.2. Informative References

[ALPACA]
Brinkmann, M., Dresen, C., Merget, R., Poddebniak, D., Müller, J., Somorovsky, J., Schwenk, J., and S. Schinzel, "ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication", August 2021, <https://www.usenix.org/conference/usenixsecurity21/presentation/brinkmann>.

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

[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>.

[CRLite] Larisch, J., Choffnes, D., Levin, D., Maggs, B., Mislove, A., and C. Wilson, "CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers", 2017 IEEE Symposium on Security and Privacy (SP), DOI 10.1109/sp.2017.17, May 2017, <https://doi.org/10.1109/sp.2017.17>.

[CRLSets] Chromium, "CRLSets", 4 August 2022, <https://www.chromium.org/Home/chromium-security/crlsets/>.

[Dilithium]
Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., and D. Stehlé, "CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation", 8 February 2021, <https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf>.

[Falcon]
Fouque, P., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., and Z. Zhang, "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU", 10 January 2020, <https://falcon-sign.info/falcon.pdf>.

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

[I-D.draft-ietf-acme-ari]
Gable, A., "Automated Certificate Management Environment (ACME) Renewal Information (ARI) Extension", Work in

Progress, Internet-Draft, draft-ietf-acme-ari-01, 8 February 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-acme-ari-01>.

[I-D.draft-ietf-tls-batch-signing] Benjamin, D., "Batch Signing for TLS", Work in Progress, Internet-Draft, draft-ietf-tls-batch-signing-00, 13 January 2020, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-batch-signing-00>.

[I-D.draft-ietf-tls-esni] Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-15, 3 October 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-15>.

[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/>.

[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>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/rfc/rfc6066>.

[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>.

[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>.

[RFC8879] Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", RFC 8879, DOI 10.17487/RFC8879, December 2020, <https://www.rfc-editor.org/rfc/rfc8879>.

Acknowledgements

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

The authors additionally thank Bob Beck, Ryan Dickson, Nick Harper, Dennis Jackson, Ryan Sleevi, and Emily Stark for many valuable discussions and insights which led to this document.

Authors' Addresses

David Benjamin
Google LLC

Email: davidben@google.com

Devon O'Brien
Google LLC

Email: asymmetric@google.com

Bas Westerbaan
Cloudflare

Email: bas@cloudflare.com