Merkle Tree Certificates

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

David Benjamin

Devon O'Brien

Bas Westerbaan

Motivation

Size

Post-quantum signatures are no longer cheap

Time to revisit past design decisions

Classical signature sizes

P-256	65-byte pubkey, 70-byte sig

Post-quantum signature sizes

Falcon-512	897-byte pubkey, 666-byte sig
Falcon-1024	1793-byte pubkey, 1280-byte sig
Dilithium2	1312-byte pubkey, 2420-byte sig
Dilithium3	1952-byte pubkey, 3293-byte sig
Dilithium5	2592-byte pubkey, 4595-byte sig

X.509

Post-quantum transition requires updating the *entire* ecosystem: CAs, servers, clients, etc.

Ideal time to revisit where X.509 has and hasn't been a good fit for TLS

- Poor trust agility in practice ("one size fits all" forces lowest common denominator)
- Keys not tightly bound to use (cross-protocol attacks)
- Complexity (path-building, X.509 names, parsing bugs)

Merkle Tree Certificates

- Very early draft of some ideas in this space
  - Interested in feedback on the design and general direction
  - Initial focus on browsers / HTTPS, and cases with similar needs
- New PKI mechanism, comparable to X.509 with Certificate Transparency
- Avoid large, post-quantum signatures at the cost of limited scope
  - Assumes short-lived certificates and automated issuance
  - Issuance is delayed by may take up to ~1 hour
  - Assumes relying parties have frequent access to an online transparency service
- A supporting certificate negotiation mechanism
- An optimization — used in conjunction with other PKI mechanisms
  - Subscribers fall back to other mechanisms, like X.509, when not applicable
  - *Not*, in itself, an X.509 replacement

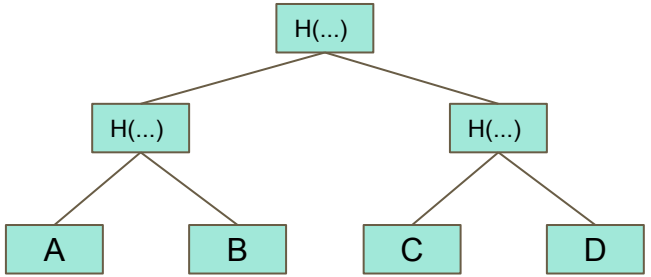
Merkle Tree Primer

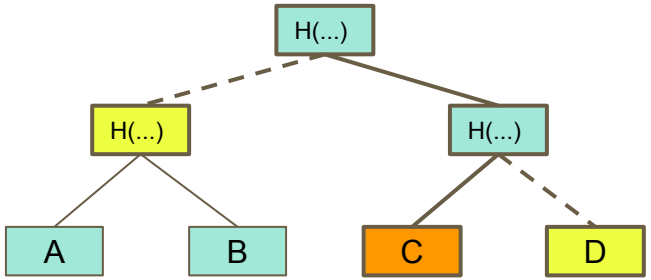
Build a tree over inputs

Each node contains hash of children

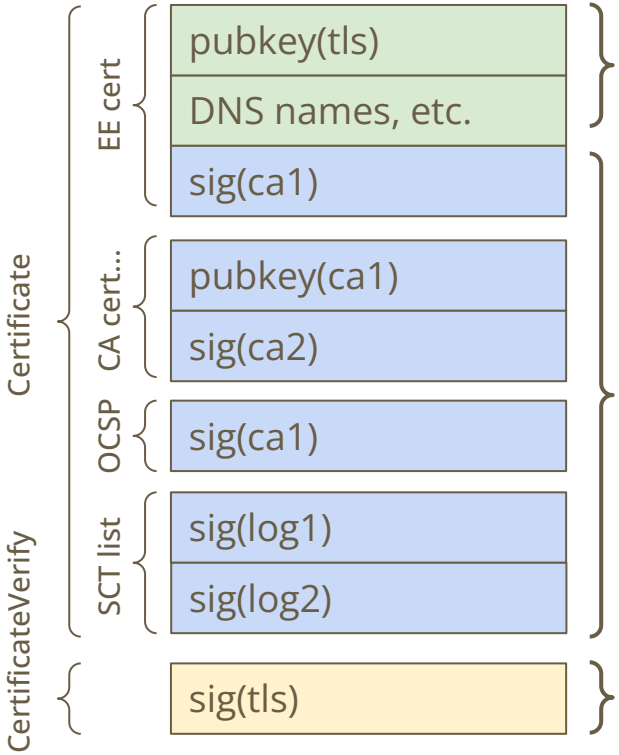
Root hash (“tree head”) commits to entire tree

Need O(log n) hashes prove that an input is contained in the tree





What's in a Certificate?



Certificate and CertificateVerify contain:

A TLS key and application identifier

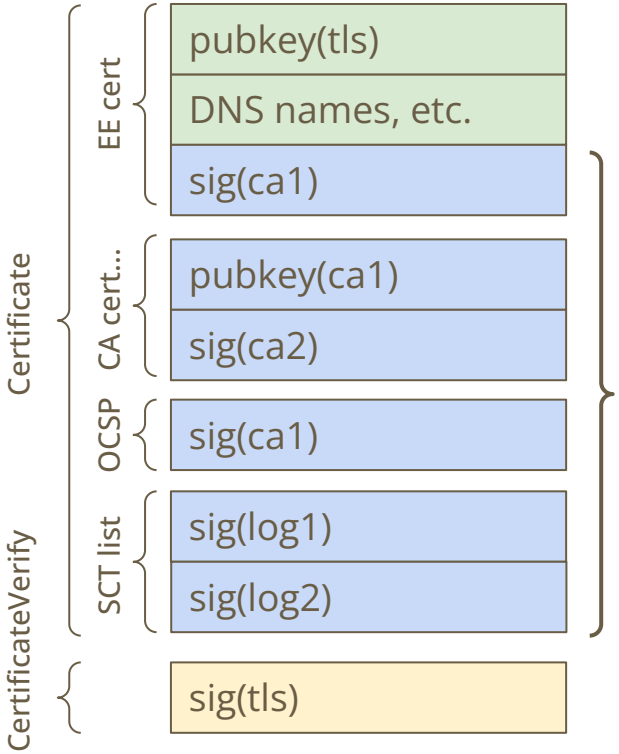
Proof from the PKI that identifiers match the key

Using the TLS key

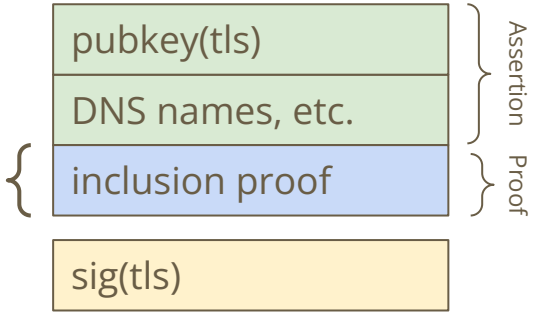
Proof determined by PKI — the draft addresses this portion

Key, identifiers, and usage determined by protocol — the draft *does not* address this. See AuthKEM for an example of changing it.

Merkle Tree Certificates

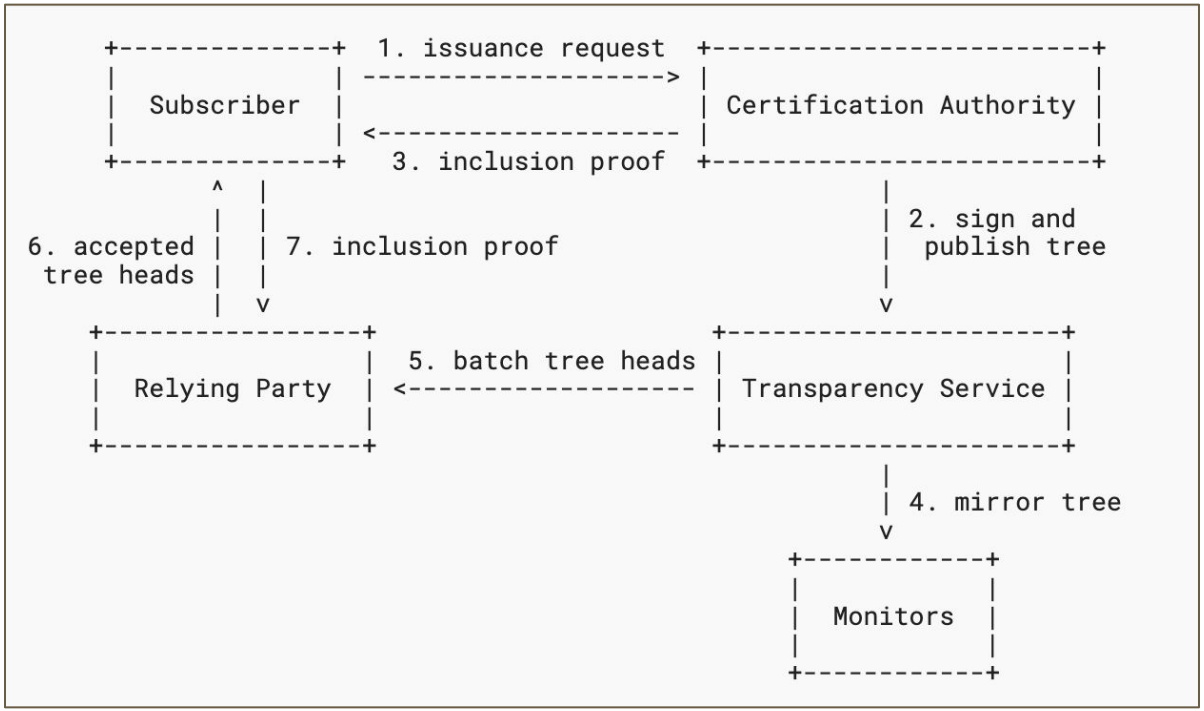


Replace *entire* PKI proof with *one* Merkle Tree inclusion proof. Estimate 640 bytes for the Web PKI (Section 5.5)



Falcon-512	897-byte pubkey, 666-byte sig
Falcon-1024	1793-byte pubkey, 1280-byte sig
Dilithium2	1312-byte pubkey, 2420-byte sig
Dilithium3	1952-byte pubkey, 3293-byte sig
Dilithium5	2592-byte pubkey, 4595-byte sig

Certificate Lifecycle



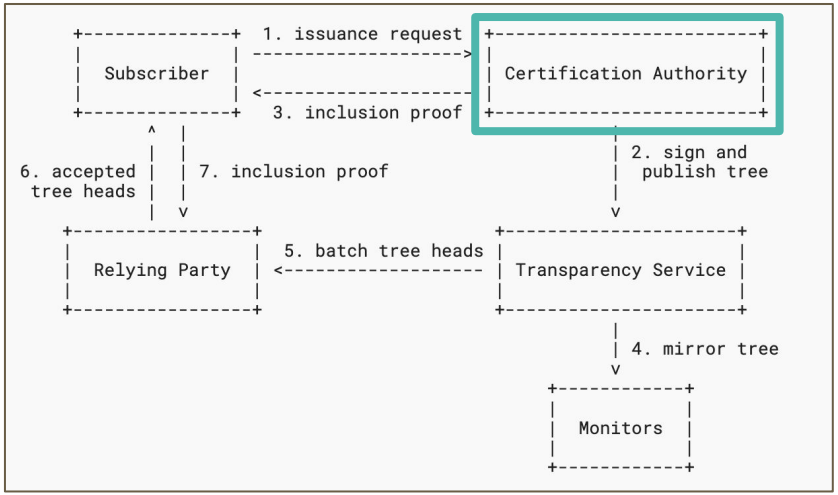
Merkle Tree CAs

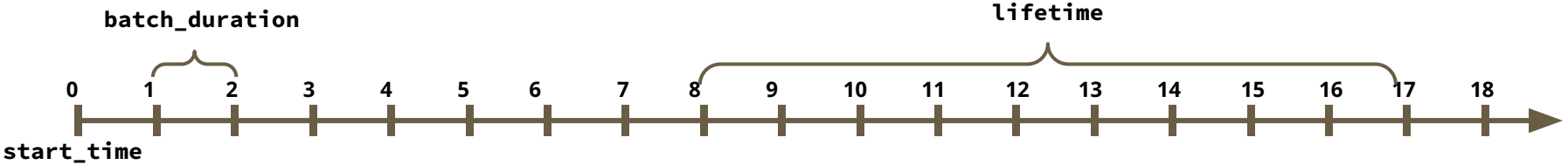
Entities that make assertions for relying parties, but differently from X.509 CAs

> *start_time*, *batch_duration*: defines periodic batches at which CAs issue certificates, e.g. 1 hour
>
> *lifetime*: lifetime for *all* certificates issued by CA, e.g. 14 days

Parameters are fixed for lifetime of the CA; run multiple CAs if needed

CAs wait for available batch, and build a Merkle Tree out of all pending requests. All certs in a batch expire together.





Merkle Tree CAs

CA signs a "window", a range of unexpired tree heads, to attest to contents

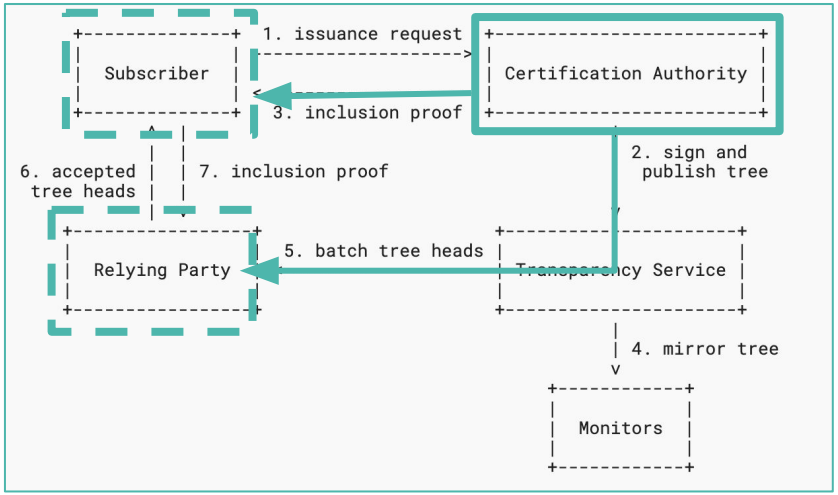
CA publishes *entire* tree

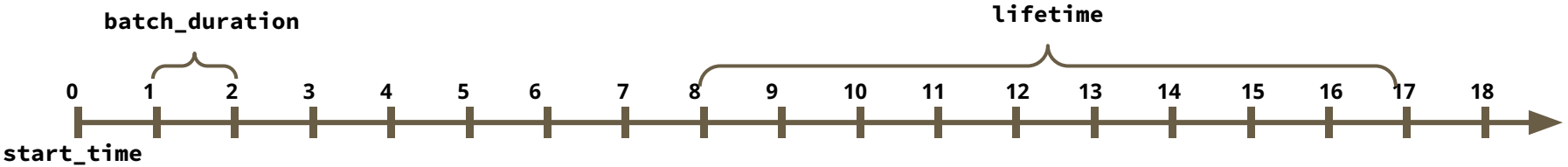
Subscriber gets inclusion proof (no signature) for its tree head — the “certificate” (issuance)

Relying party gets signed window (out-of-band)

Subscriber presents inclusion proof in TLS

Relying party trusts if inclusion proof matches a trusted tree head





Transparency Service

Intermediary between CAs and relying parties:

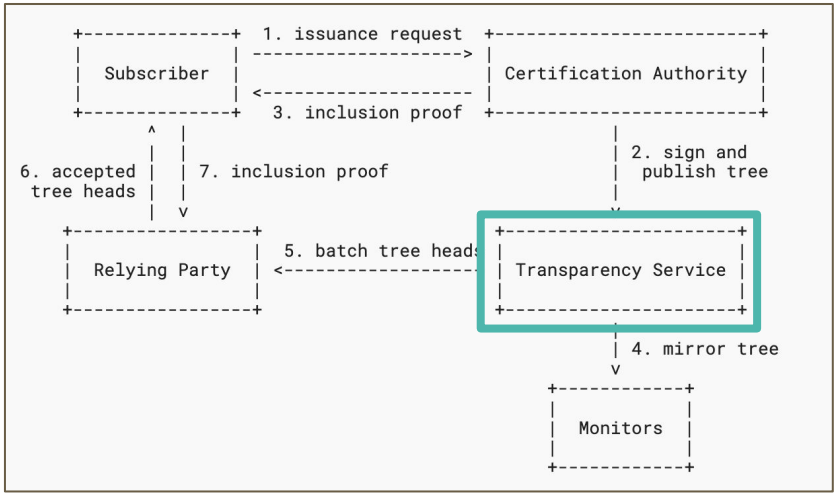
- Mirrors CA assertions for monitors to audit
- Provides the latest valid window to relying parties

Key responsibility: If a relying party trusts a tree head, the corresponding tree is available to monitors

Provides log availability, canonical consistent log state

Still OK if CA publishes split view or goes down)

Described as a single entity, but many trust models are possible (Section 7)



Transparency Service Examples

Single trusted service

- Update service from software vendor, etc.

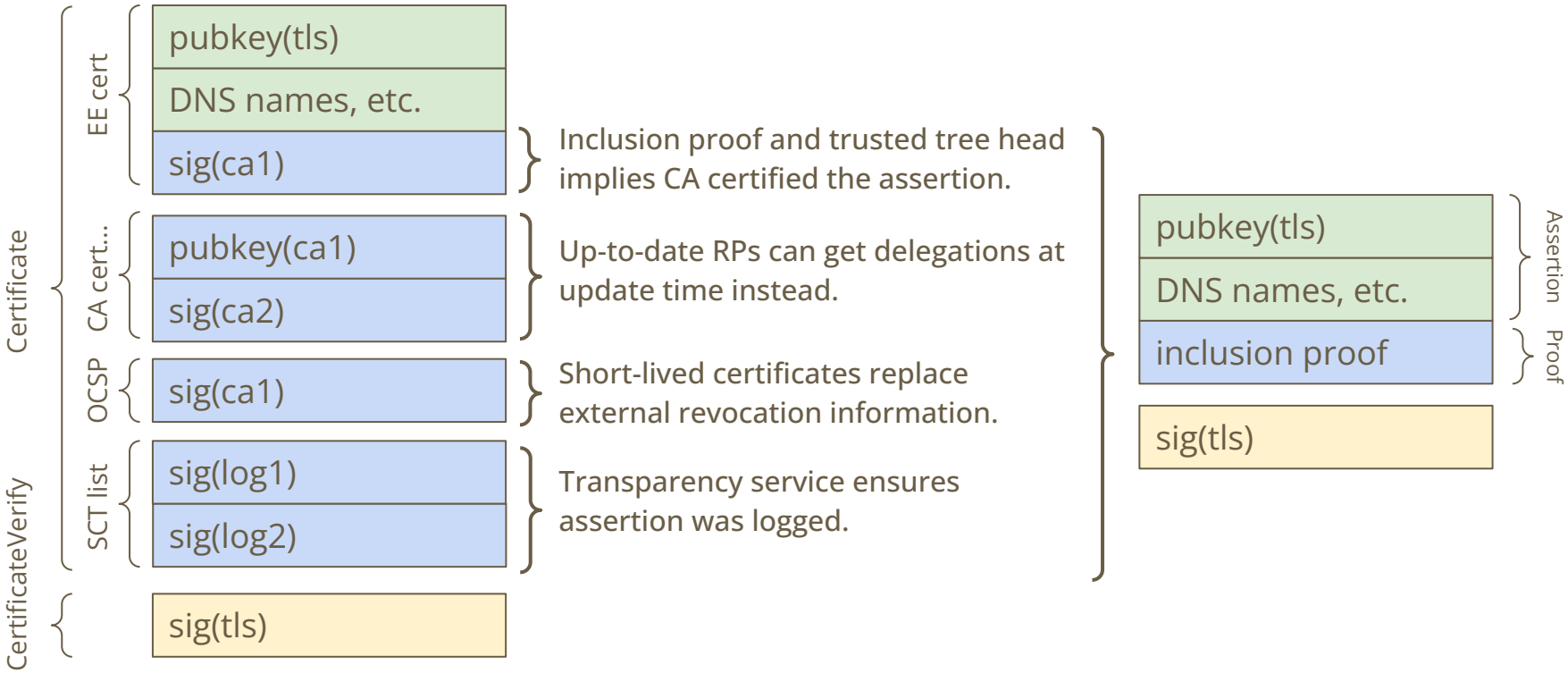
Single update service with multiple mirrors

- Update service defers mirrors to N mirrors, forwards tree heads once M mirrors agree

Multiple mirrors

- Relying party directly contacts N mirrors, accepts tree heads once M mirrors agree

Security Comparison



Format Goals

A single “one size fits all” certificate limits to lowest component denominator

- Unupdatable relying party
- Limit to intersection of root stores
- Poor agility — Trust changes, root key rotation are difficult

Merkle Tree certificates need certificate negotiation

- Trade off issuance time and reach for size
- Only work if relying party updated *after* certificate was issued
- *This cannot be your only certificate*

Secondary goals:

- Bound keys to usage (cross-protocol attacks)
- Easy to parse
- Extensibility in the right places

Assertion Syntax

First pass at assertion syntax:

Assertion: subject and list of claims

Subject: who the CA is talking about, e.g. holder of a TLS key

Claims: facts about the subject, e.g. “this TLS key is authorized for example.com”

Protocols like TLS allocate SubjectTypes and define the contents (keys, associated metadata)

Unlike X.509 SPKIs, subjects are inherently tied to usage — avoid cross-protocol attacks

Allocate ClaimTypes to define new kinds of identifiers (emails, user IDs, etc.). Just transcribed SAN for now.

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

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

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

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;
```

Certificate Syntax

Designed for certificate negotiation

Certificate: Assertion and proof

Proof: Some message that allows the relying party to believe the assertion. Analogy: X.509 signatures + delegation chain

Trust anchor: Identifies a set of proofs that an relying party accepts. Analogy: X.509 root's subject names.

Syntax defined by ProofType. In Merkle Tree certs, the proof is an inclusion proof, and the trust anchor is a batch number.

Negotiation: find a match between relying party's trust anchors and subscriber's certificates

```
struct {
    Assertion assertion;
    Proof proof;
} BikeshedCertificate;
/* blatant placeholder name */

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

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

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

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

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

Certificate Negotiation

Subscribers maintain *certificate set* + extra info from CA

- Trust anchor, e.g. (A, 1000)
- Matching aliases, e.g. (A, 1001), ..., (A, 1099) if window size is 100
- Expiration time

RP sends supported trust anchors

- RPs use aliases to send a single trust anchor
- Here, (A, 1050) means “A’s validity window ending at batch 1050”

Subscriber picks smallest unexpired certificate, no need to understand proof type or contents

```
struct {
    Assertion assertion;
    Proof proof;
} BikeshedCertificate;
/* blatant placeholder name */

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

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

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

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

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

Use in TLS

`tls` SubjectType carries TLS signing keys

Negotiate with `Bikeshed` CertificateType:

- Single CertificateEntry with BikeshedCertificate
- Require ALPN if negotiated
- TODO: Fix client certificate type negotiation (Section 10.4)

New `trust_anchors` extension (CH+CR) for relying party TrustAnchor list

If no trust anchors match, fallback to X.509 or other mechanism

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

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

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

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

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

Deployment Model

Subscribers maintain certificate set to cover different relying parties

Rolling renewal halfway through lifetime

Trust agility: Make this ACME server's responsibility, transparent to subscriber

- Provision different certs as rely party's trust requirements change
- Rotate CA keys transparently
- Deploy new ProofTypes transparently
- ARI to control renewal time

Use other cert or proof types for...

- Emergency rekeys
- Relying parties with stale data
- First few hours of a newly deployed site
- Existing relying parties

Next Steps

Looking for feedback (both design and general direction)

Flesh out missing details

Refine negotiation mechanism?

Worth making an X.509-style signature-based ProofType?

Your ideas here?

https://github.com/davidben/merkle-tree-certs

https://datatracker.ietf.org/doc/draft-davidben-tls-merkle-tree-certs/