Post-Quantum

Cryptography Conference

Making PQ Signatures work in the WebPKI

Post-quantum signatures are not easily deployable in the WebPKI. Using the signature algorithms recently standardized by NIST as drop-in replacements for existing classical algorithms on the Web would incur significant performance degradations, making this approach infeasible unless a cryptographically-relevant quantum computer (CRQCs) is imminent. There's a real risk that post-quantum signatures do not see widespread adoption before CRQCs become a reality, unless we make changes to how signatures are used in the WebPKI. This talk dives into several of the more promising proposals for making post-quantum signatures deployable, from TLS extensions to reduce the number of transmitted signatures, to using key agreement as an authentication mechanism, to complete overhauls of the WebPKI. We discuss ongoing work to evaluate the feasibility of each of these proposals and to address known unknowns. _(this is a 60 minute session)_

Luke Valenta
Research Engineer at Cloudflare

SSL.com PQSHIELD HID KEYFACTOR ENTRUST

January 15 and 16, 2025 - Austin, TX (US) | Online

PKI Consortium

Technical breakout: Making PQ signatures work in the WebPKI

Luke Valenta, Cloudflare Research
lvalenta@cloudflare.com

PKI Consortium Post-Quantum Cryptography Conference, January 15, 2025

This technical break-out

Start with a brief recap of the many signatures in the WebPKI.

Then we take a look at NIST's standardized post-quantum signature schemes and those being considered in the on-ramp.

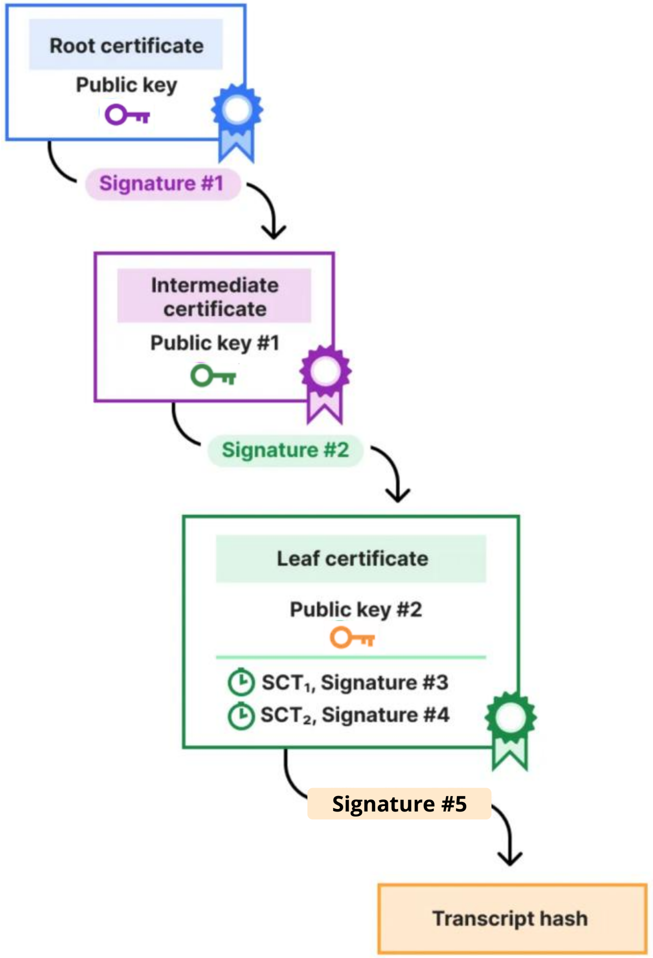
And we give updates[1] on three promising strategies for making PQ signatures work in the WebPKI

- Leaving out intermediates
- KEMTLS: using key agreement as authentication
- Merkle Tree Certificates

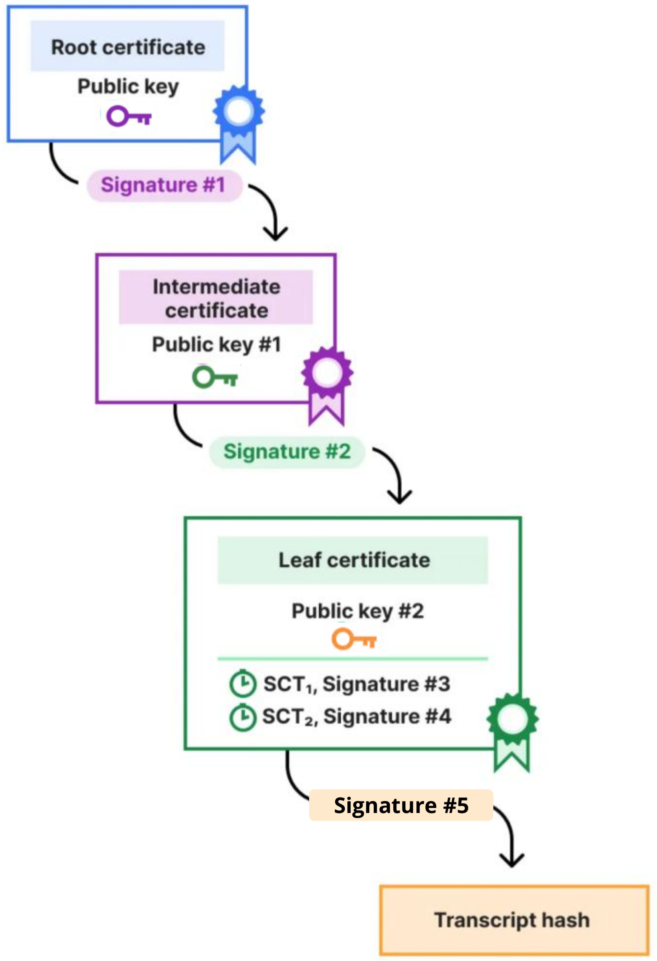
1. Bas Westerbaan, Coping with post-quantum certificates in the WebPKI, PKIC PQ Conference, AMS, Nov. 2023

There are many signatures on the Web

Typically 5 signatures
and 2 public keys
when visiting a website.

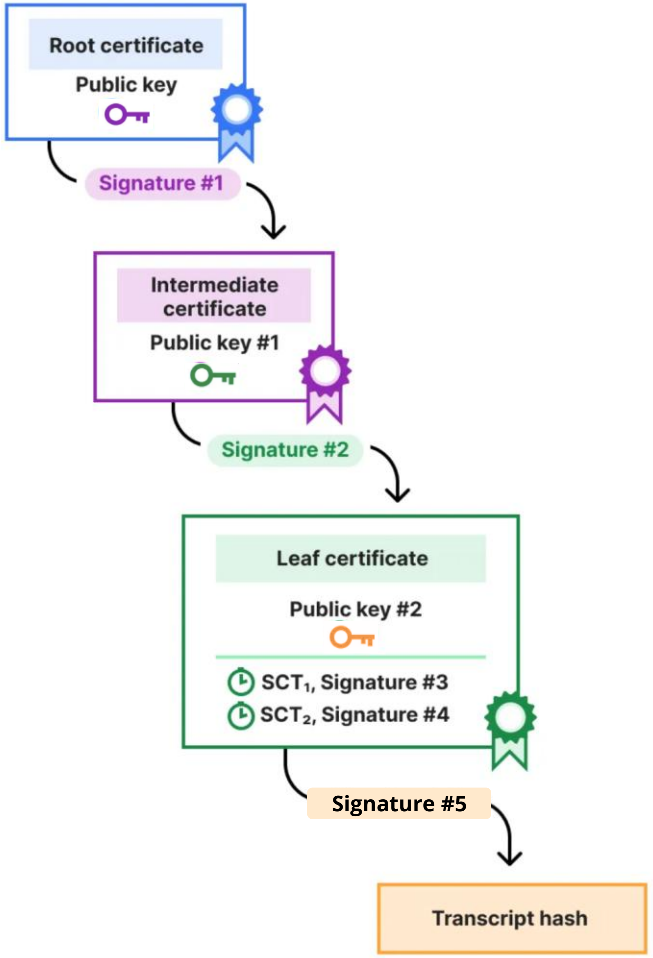


Tailoring for TLS	Priorities
Signature #1 (root on intermediate)	Short signature (big public key OK) Fast verification (slow signing OK)
Public key #1 (intermediate)	Balanced signature/public key Fast verification
Signature #2 (intermediate on leaf)	
Public key #2 (leaf)	Balanced signature/public key Balanced signing/verification
Signature #5 (leaf on transcript)	
Signature #3 (signed certificate timestamp)	Short signature Fast verification
Signature #4 (signed certificate timestamp)	Short signature Fast verification



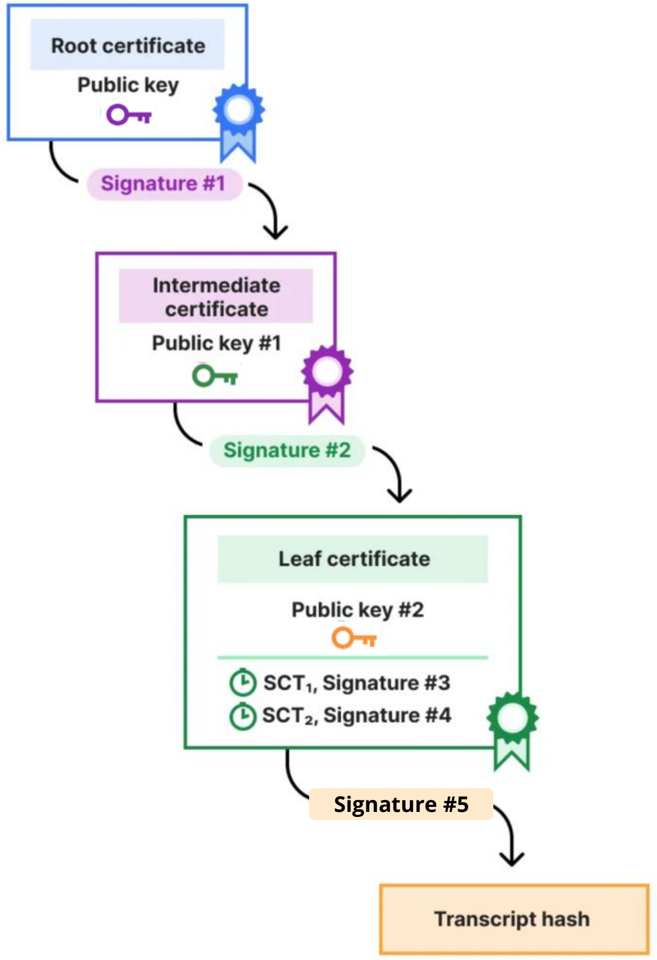
Signature schemes of all shapes and sizes			Sizes (bytes)		CPU time (lower is better)	
		PQ	Public key	Signature	Signing	Verification
Standardised	Ed25519	❌	32	64	0.15	1.3
	RSA$_{2048}$	❌	256	256	80	0.4
NIST standards	ML-DSA$_{44}$	✅	1,312	2,420	1 (baseline)	1 (baseline)
	Falcon$_{512}$ (soon FN-DSA)	✅	897	666	3 ⚠	0.7
	SLH-DSA$_{128s}$	✅	32	7,856	14,000	40
	SLH-DSA$_{128f}$	✅	32	17,088	720	110
	LMS$_{M4\_H20\_W8}$	✅	48	1,112	2.9 ⚠	8.4
Sample from round 2 signatures on-ramp	MAYO$_{one}$	✅	1,168	321	1.4	1.4
	MAYO$_{two}$	✅	5,488	180	1.7	0.8
	SQISign$_{I}$	✅	64	177	17,000	900
	UOV$_{Is\text{-}pkc}$	✅	66,576	96	0.3	2.3
	HAWK$_{512}$	✅	1,024	555	0.25	1.2

Classical ❌	Algorithm	Size (bytes)
Signature #1 (root on intermediate)	RSA_{4096}	512
Public key #1 (intermediate)	RSA_{2048}	256
Signature #2 (intermediate on leaf)	RSA_{2048}	256
Public key #2 (leaf)	P-256	32
Signature #5 (leaf on transcript)	P-256	64
Signature #3 (signed certificate timestamp)	P-256	64
Signature #4 (signed certificate timestamp)	P-256	64
	Total	1,248

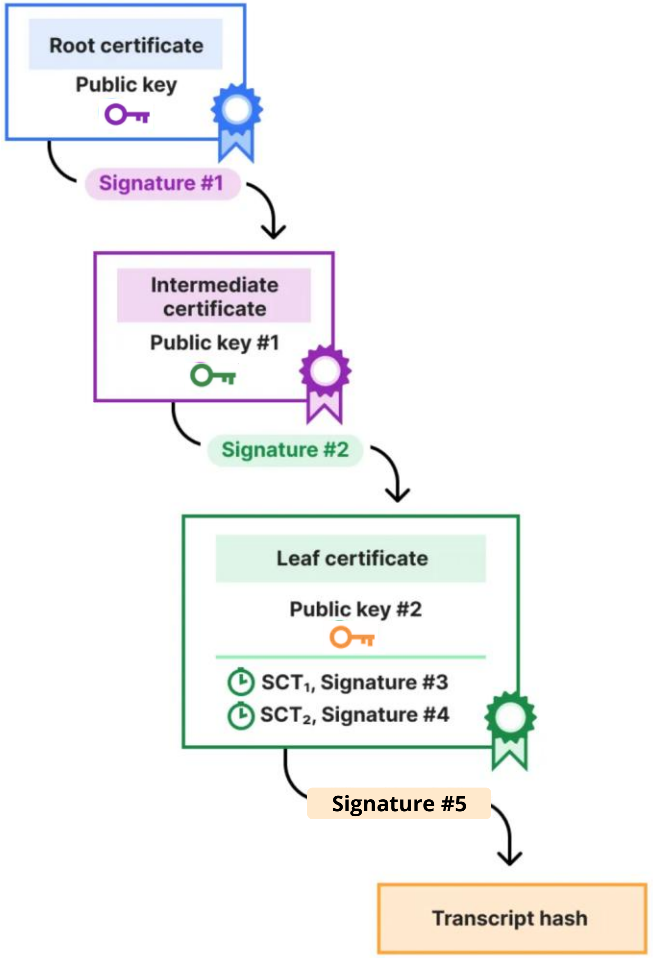


All SLH-DSA ✅	Algorithm	Size (bytes)
Signature #1 (root on intermediate)	SLH-DSA$_{128s}$	7,856
Public key #1 (intermediate)	SLH-DSA$_{128s}$	32
Signature #2 (intermediate on leaf)	SLH-DSA$_{128s}$	7,856
Public key #2 (leaf)	SLH-DSA$_{128f}$	32
Signature #5 (leaf on transcript)	SLH-DSA$_{128f}$	17,088
Signature #3 (signed certificate timestamp)	SLH-DSA$_{128s}$	7,856
Signature #4 (signed certificate timestamp)	SLH-DSA$_{128s}$	7,856
	Total	48,576

Most conservative choice. Order of magnitude slower signing than RSA.
Use 128f for handshake for more reasonable signing performance.



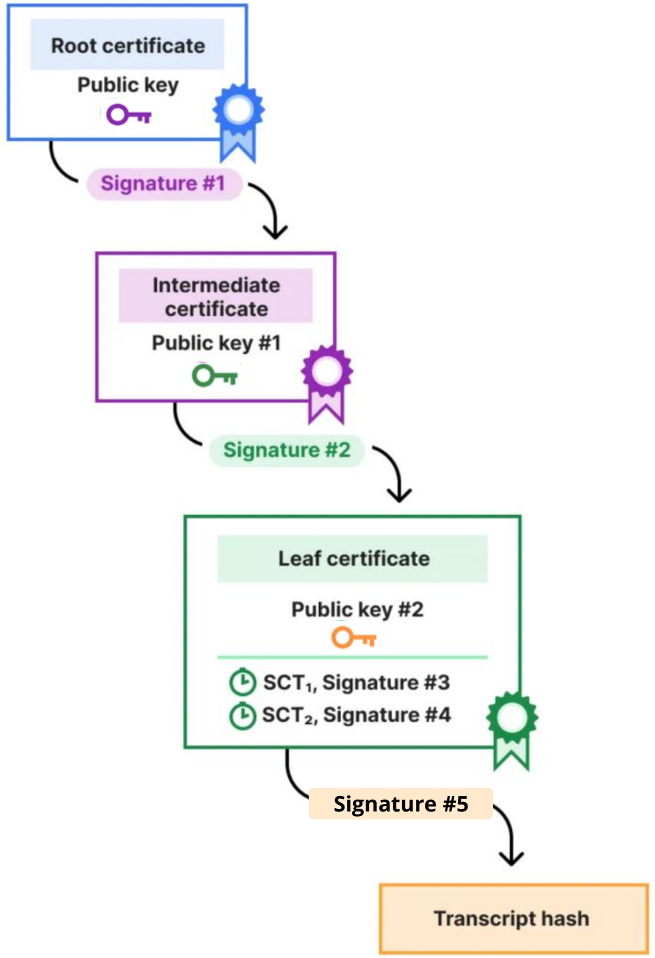
All ML-DSA ✅	Algorithm	Size (bytes)
Signature #1 (root on intermediate)	ML-DSA$_{44}$	2,420
Public key #1 (intermediate)	ML-DSA$_{44}$	1,312
Signature #2 (intermediate on leaf)	ML-DSA$_{44}$	2,420
Public key #2 (leaf)	ML-DSA$_{44}$	1,312
Signature #5 (leaf on transcript)	ML-DSA$_{44}$	2,420
Signature #3 (signed certificate timestamp)	ML-DSA$_{44}$	2,420
Signature #4 (signed certificate timestamp)	ML-DSA$_{44}$	2,420
	Total	14,724



General purpose. Good performance. Using ML-DSA$_{65}$ instead adds 20kB.

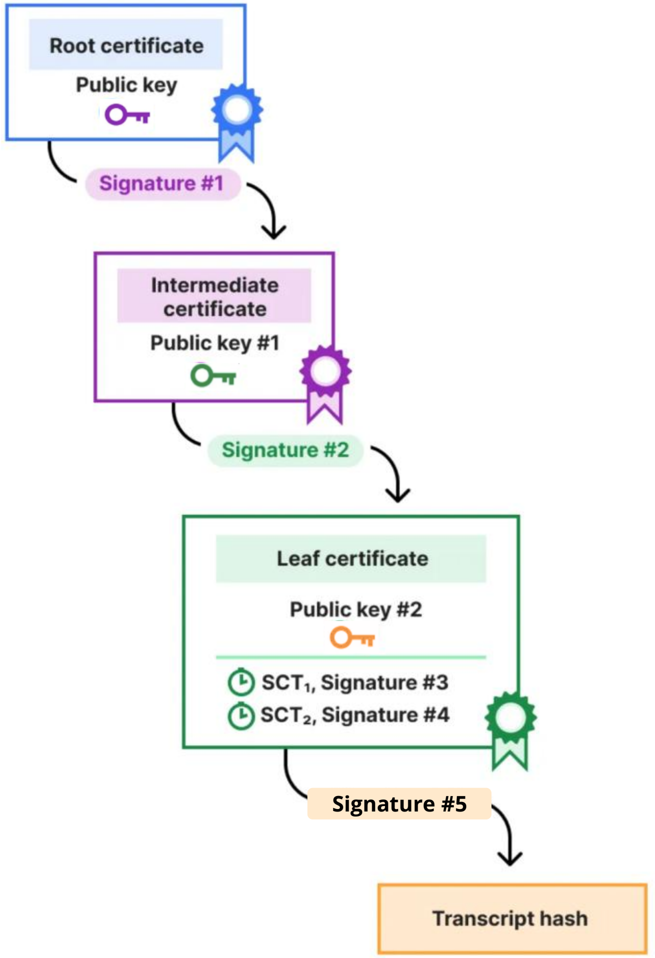
Falcon+ML-DSA ✅ ⚠️	Algorithm	Size (bytes)
Signature #1 (root on intermediate)	$Falcon_{512}$	666
Public key #1 (intermediate)	$Falcon_{512}$	897
Signature #2 (intermediate on leaf)	$Falcon_{512}$	666
Public key #2 (leaf)	$ML\text{-}DSA_{44}$	1,312
Signature #5 (leaf on transcript)	$ML\text{-}DSA_{44}$	2,420
Signature #3 (signed certificate timestamp)	$Falcon_{512}$	666
Signature #4 (signed certificate timestamp)	$Falcon_{512}$	666
	Total	7,293

⚠️ Fast and secure Falcon512 signing is hard to implement.



Stateful hash-based ✅ ⚠️	Algorithm	Size (bytes)
Signature #1 (root on intermediate)	$XMSS_{h16\_w256\_n128}$	544
Public key #1 (intermediate)	$XMSS^{MT}_{h16,16\_w256\_n128}$	32
Signature #2 (intermediate on leaf)	$XMSS^{MT}_{h16,16\_w256\_n128}$	1,088
Public key #2 (leaf)	$ML\text{-}DSA_{44}$	1,312
Signature #5 (leaf on transcript)	$ML\text{-}DSA_{44}$	2,420
Signature #3 (signed certificate timestamp)	$XMSS^{MT}_{h16,16\_w256\_n128}$	1,088
Signature #4 (signed certificate timestamp)	$XMSS^{MT}_{h16,16\_w256\_n128}$	1,088
	Total	7,572

⚠️ n=128 and w=256 instances are not standardised.
⚠️ We lose non-repudiation.
⚠️ Large precomputations/storage required for efficient signing.
⚠️ Challenging to keep state.
Estimates from https://westerbaan.name/~bas/hashcalc/



Ideally, we have something that outperforms Falcon/ML-DSA and is easier to deploy.

We also want a backup with different cryptographic assumptions than structured lattices.

			Sizes (bytes)		CPU time (lower is better)	
		PQ	Public key	Signature	Signing	Verification
Standardise						
NIST standa						)
	SLH-DSA$_{128f}$	✓	32	17,088	720	110
	LMS$_{M4\_H20\_W8}$	✓	48	1,112	2.9 ⚠	8.4
Sample from round 2 signatures on-ramp	MAYO$_{one}$	✓	1,168	321	1.4	1.4
	MAYO$_{two}$	✓	5,488	180	1.7	0.8
	SQISign$_{I}$	✓	64	177	17,000	900
	UOV$_{Is\text{-}pkc}$	✓	66,576	96	0.3	2.3
	HAWK$_{512}$	✓	1,024	555	0.25	1.2

Concrete instances with on-ramp candidates

Using MAYO *one* for leaf/intermediate, and *two* for the rest, adds 3.5kB. Signing time between ECC/RSA. ⚠️ Security uncertain.

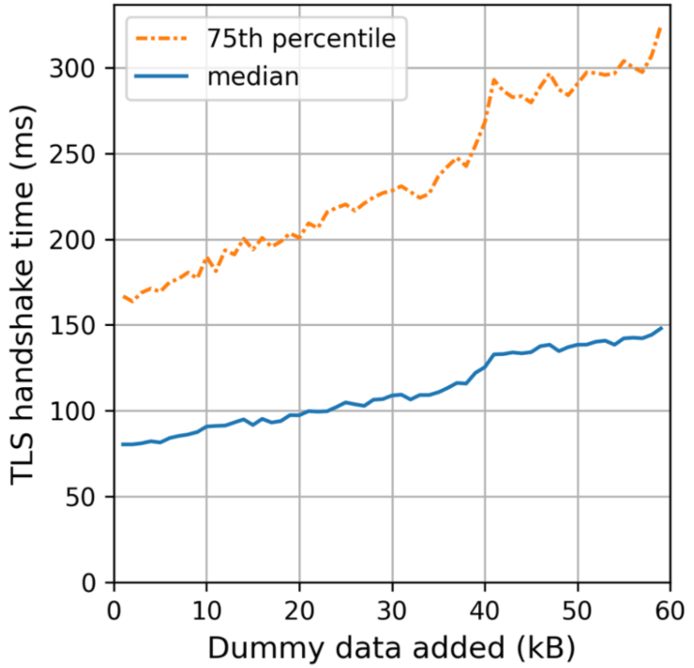
Using UOV ls-pkc for root and SCTs, and HAWK512 for the rest, adds 3.2kB. 66kB for stored UOV public keys. HAWK relies on Falcon assumptions and then some more.

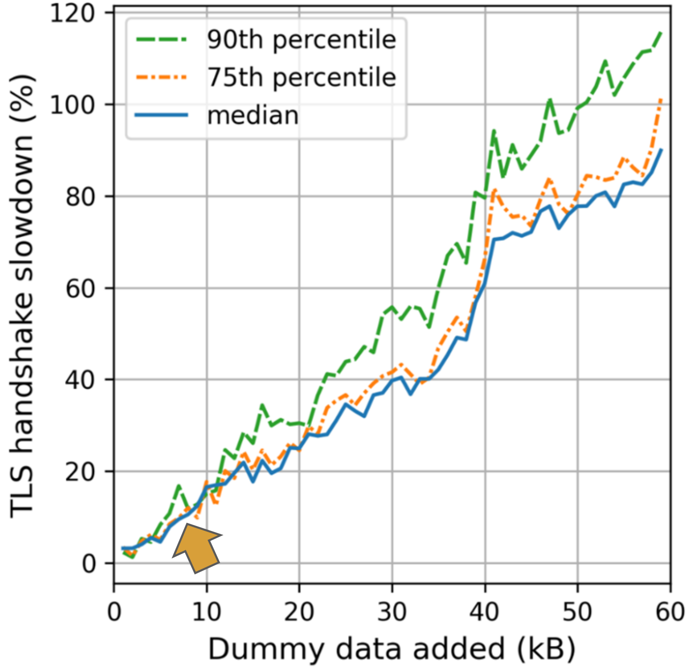
Using UOV ls-pkc again, but combined with ML-DSA$_{44}$. Adds 7.4kB. Relatively conservative choice.

SQIsign only. Adds 0.5kB. Signing time >1s (not constant-time), and verification time >35ms. 🐢

How many (bytes) is too many?

Sizing up post-quantum signatures, 2021: We found that every 1kB added to the TLS handshake slows it down by about 1.5% at the median.





How many (bytes) is too many?

Sizing up post-quantum signatures, 2021: We found that every 1kB added to the TLS handshake slows it down by about 1.5% at the median.

Chromium Security Design Principles, 2024: “Adding ~7kB is implausible unless a cryptographically relevant quantum computer (CRQC) is tangibly imminent.”

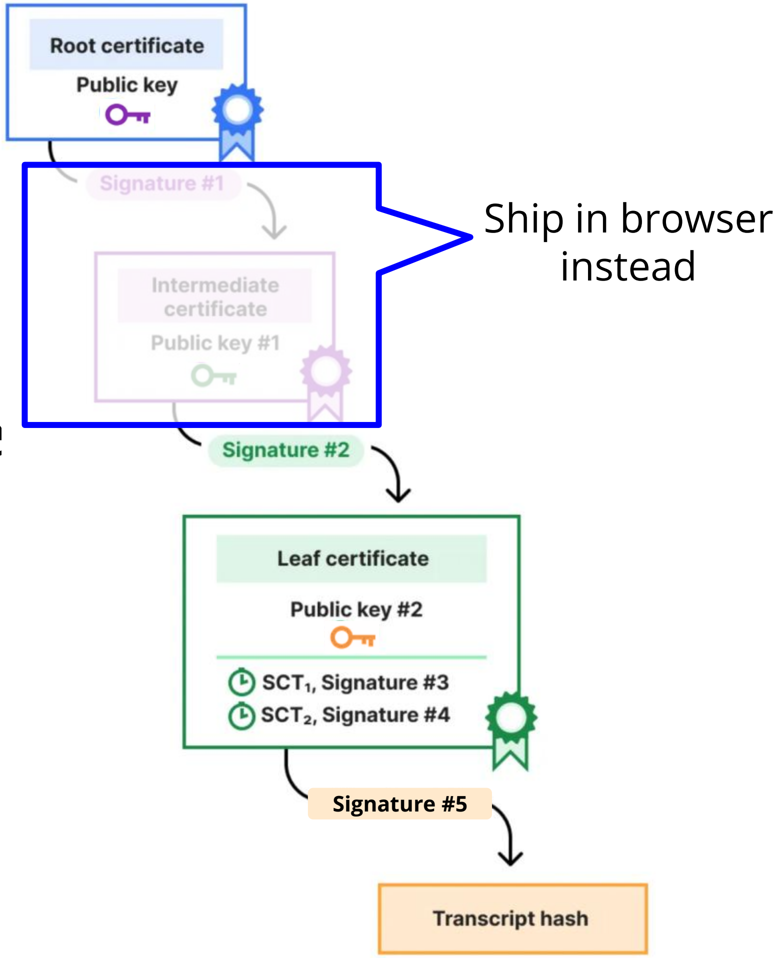
Another look at PQ signatures, 2024: Median bytes transferred from server to client for the lifetime of non-resumed QUIC connections to Cloudflare is 4.4kB.

- Classical signatures and public keys ***already*** account for about 25% of all bytes transferred on over half the connections!

If PQ signatures are too expensive, the real risk is that we deploy too late!

Leaving out intermediates

Most browsers ship intermediate so why bother sending them?

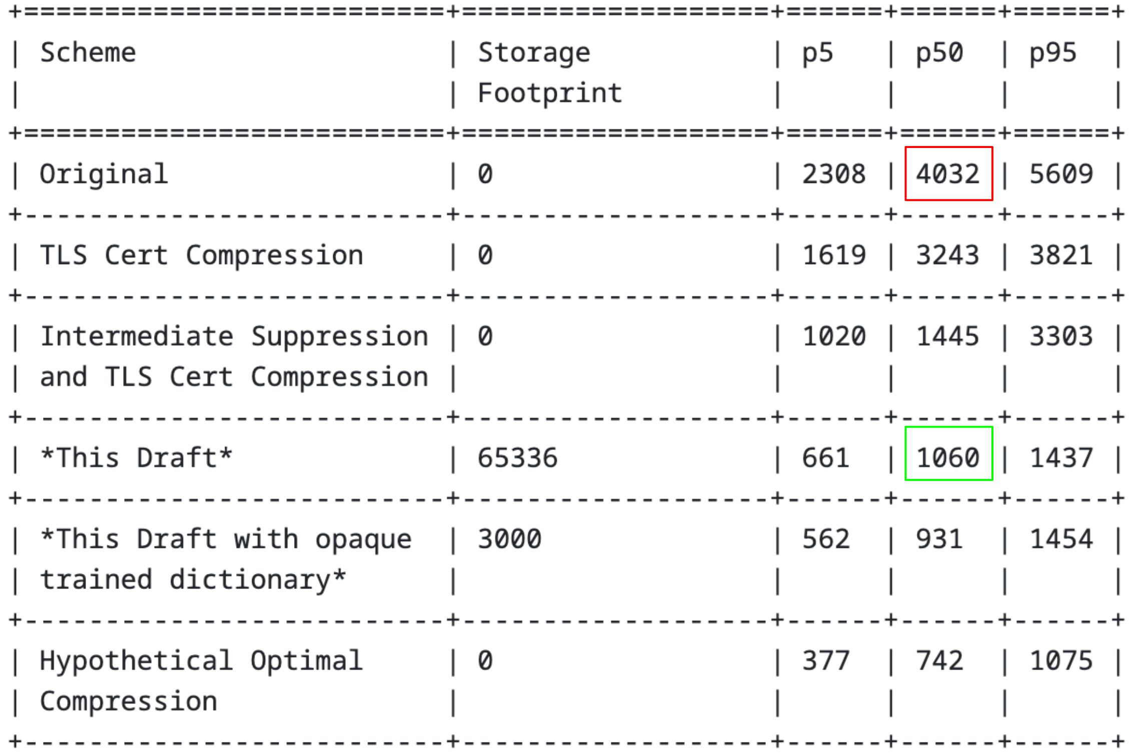


Leaving out intermediates

Three proposals:

- 2019, draft-kampanakis-tls-scas, send flag to indicate server should only return leaf. Simple but error prone.
- 2022, draft-ietf-tls-cert-abridge, replaces intermediates with identifiers from yearly updated central list from CCADB. Client sends version of latest list. Also proposes tailored compression.
- 2023, draft-davidben-tls-trust-expr. Simplified: client sends which trust store it uses, and the version it has. CA adds as metadata to a certificate, in which trust store (version) it's included. Trust stores can then add intermediates as roots.

Savings leaving out intermediates: median 3kB



Scheme	Storage Footprint	p5	p50	p95
Original	0	2308	4032	5609
TLS Cert Compression	0	1619	3243	3821
Intermediate Suppression and TLS Cert Compression	0	1020	1445	3303
This Draft	65336	661	1060	1437
This Draft with opaque trained dictionary	3000	562	931	1454
Hypothetical Optimal Compression	0	377	742	1075

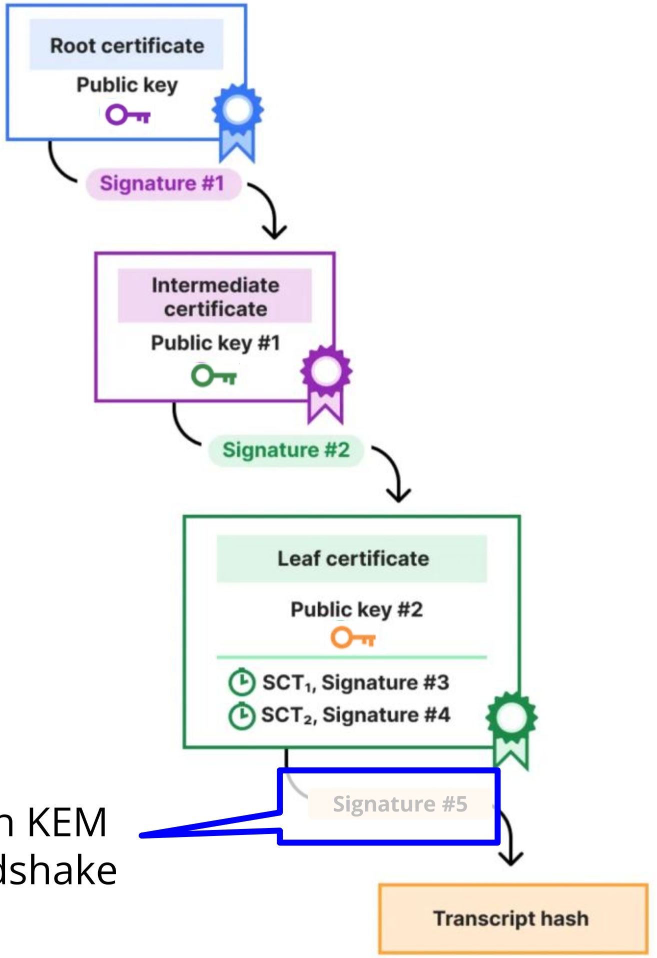
From Dennis Jackson's draft-ietf-tls-cert-abridge-00

Savings apply even before PQ!

We will be experimenting with leaving out intermediates in 2025.

KEMTLS (aka. Authkem)

Use KEM instead of signature for handshake authentication.



Replace with KEM later in handshake

KEMTLS

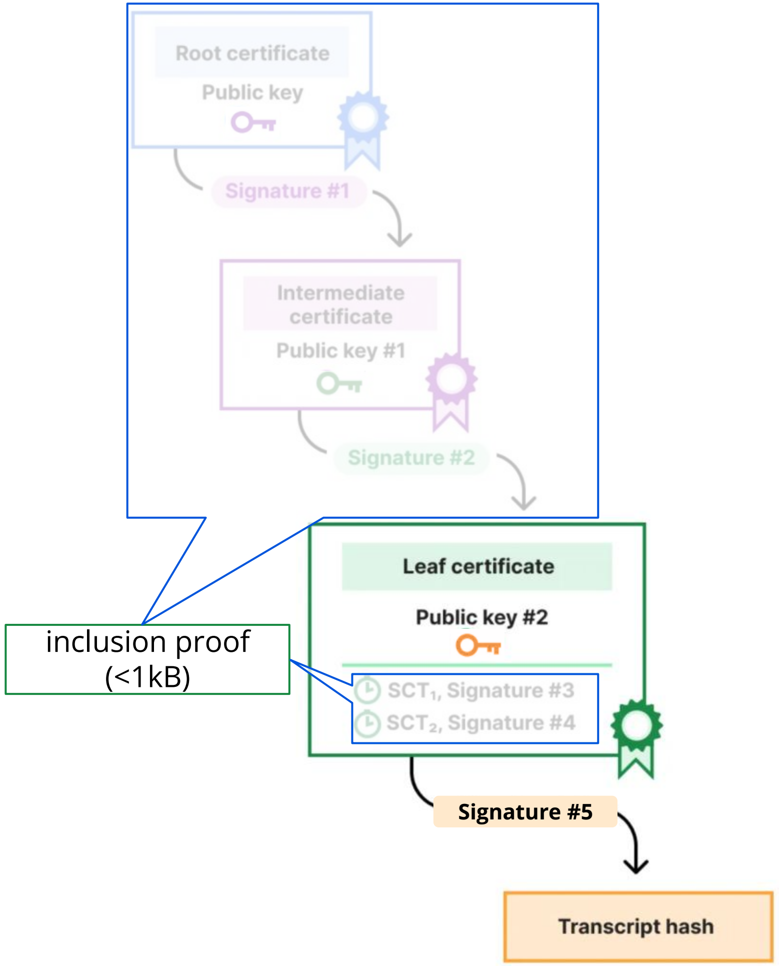
Replacing ML-DSA$_{44}$ handshake signature with ML-KEM$_{768}$ saves 2.9kB server $\longrightarrow$ client, but adds 768B in the second flight client $\longrightarrow$ server.

At the moment gains are modest. Interesting for embedded, to reduce code size by eliminating primitive. Client authentication with KEM requires extra roundtrip.

Large change to TLS. Subtle changes in security guarantees. We contributed to a formal analysis.

Proof-of-possession unclear. Could be done with lattice-based zero-knowledge proofs or challenge-response.

Merkle Tree Certificates



Pain-points of current WebPKI

OCSP is expensive to run, whereas majority of users don’t use it, but rely on CRL instead (via eg. CRLite).

Too many signatures (designed assuming signatures cheap).

Certificate Transparency is difficult to run.

Many sharp edges: path building, punycode, constraint validation, etc.

(Domain control validation is imperfect — not addressed.)

Changing the WebPKI

With the post-quantum migration, the marginal cost of changing the WebPKI is lower than ever.

There is a huge design space, with many trade offs.

Merkle Tree Certificates (MTC) is a concrete, ambitious, but early draft. We're looking for feedback on the design and general direction.

Not a complete replacement for current WebPKI: it makes the common case fast and falls back to X.509+CT.

Merkle Tree Certificates tl;dr

Goal: Efficiently authenticate a TLS key, making the common case fast.

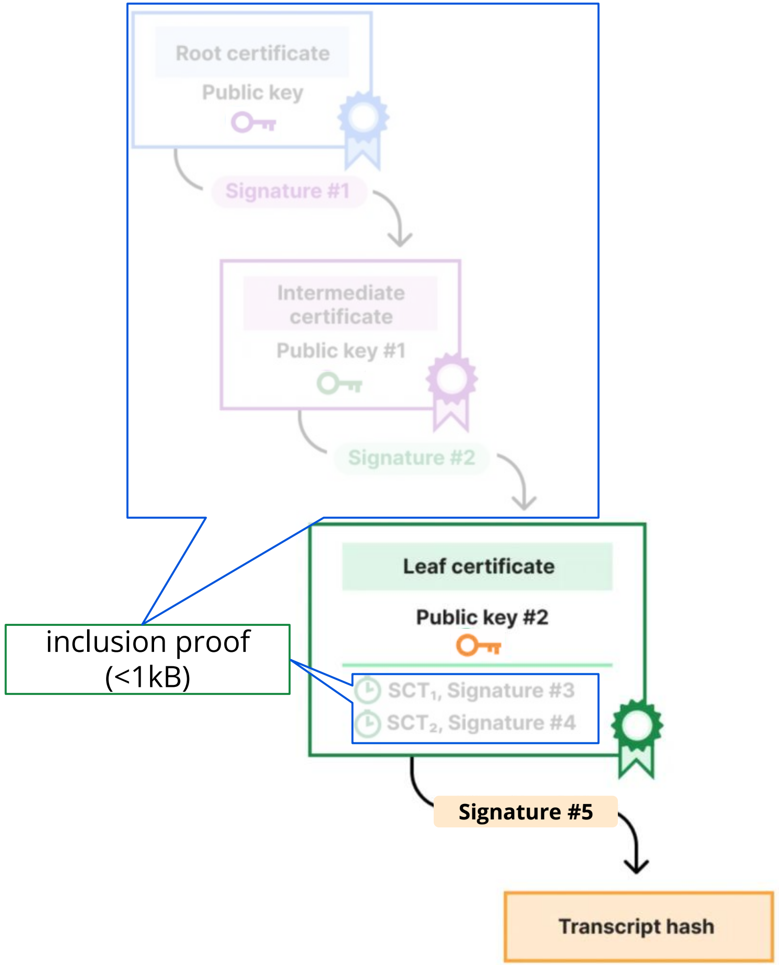
Strategy: Replace entire PKI proof with a single <1kB Merkle Tree inclusion proof.

Scope:

- Short-lived certificates (14 days, ACME)
- Up-to-date relying parties (browsers)
- Significant processing delay (1 hour)

Fall back[1] to X.509+CT for everything else (new domain registration, overlooked cert renewal, unplanned domain move).

1. Fallbacks needed <0.1% of the time: SCT Auditing Revisited

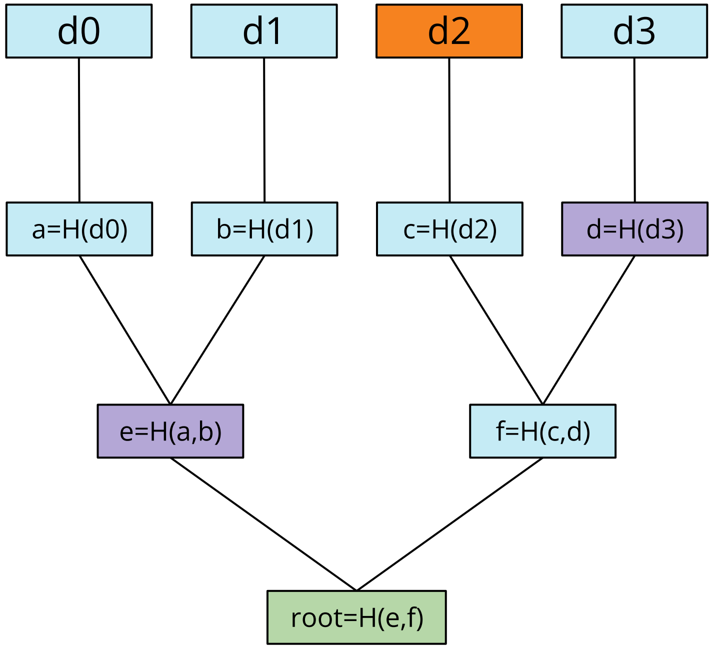


Merkle Tree Primer

Build a tree over inputs (d0, d1, d2, d3).

Each node is hash of children, and root hash ("tree head") commits to entire tree.

Need O(log n) hashes to prove input is in tree.
Inclusion proof for d2 is [d, e].



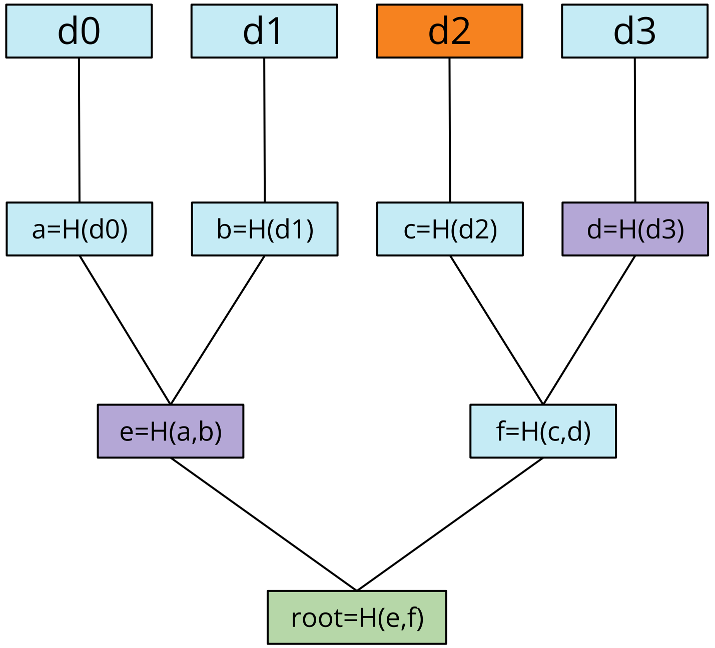
Merkle Tree Certificates (simplified)

Inputs are assertions: TLS key ⇔ DNS name.

CAs periodically build Merkle trees from batches of assertions, and publish tree + signature on tree head.

Clients periodically fetch latest tree heads.

In TLS, server presents a certificate consisting of an assertion + inclusion proof. Client validates proof against tree head.



Size estimates

There are currently about 2 billion unexpired certificates in CT.

If reissued every 7 days by one MTC CA, we'd have hourly batches of 12 million assertions.

That amounts to authentication paths of 768 bytes, and with a ML-DSA$_{44}$ public key a typical Merkle tree certificate will be well below 2.5kB, smaller than only the median compressed classical intermediate certificate of 3.2kB.

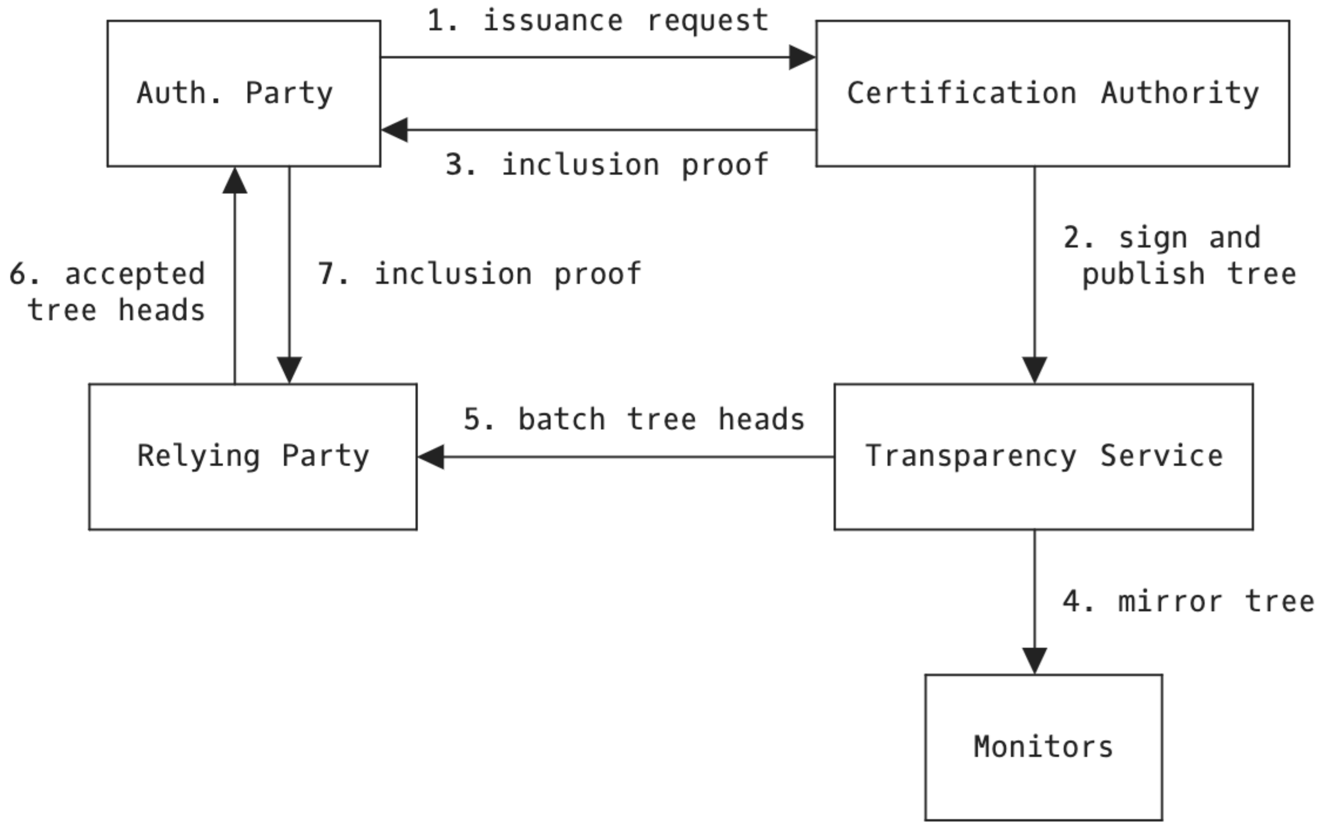


Figure 1: An overview of a Merkle Tree certificate deployment

Certificate Authorities

Similar to X.509 CAs, MTC CAs make assertions on behalf of authenticating parties (servers).

At set time, eg. hourly, CA publishes:

- The batch of assertions they certify. All assertions in a batch are implicitly valid for the same window, eg. 14 days. For each batch, the CA builds a Merkle tree on top.
- A signature on the tree heads of all currently valid batches.

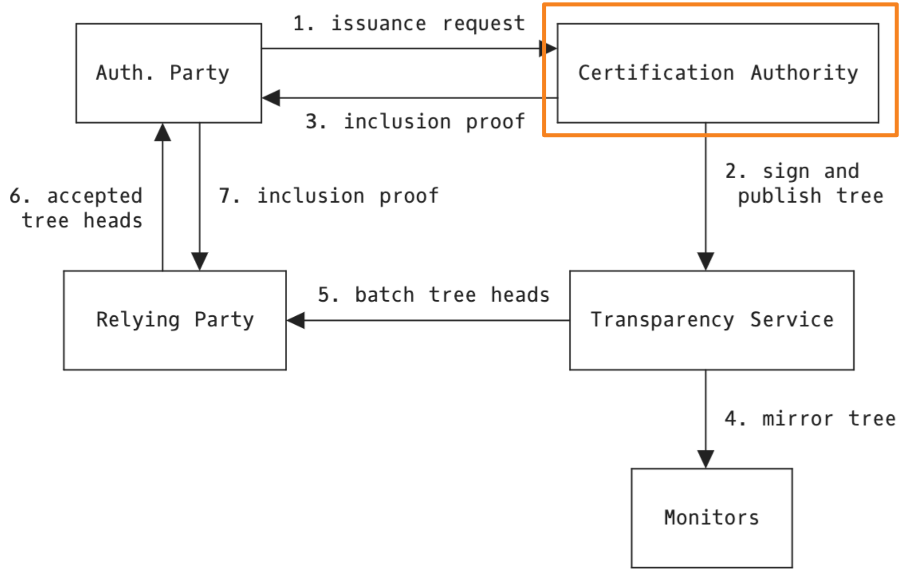


Figure 1: An overview of a Merkle Tree certificate deployment

TLS authentication

Server (auth. party) gets inclusion proof, and combines with assertion to get MTC.

- With 14 day lifetimes, keep two MTCs 7 days apart and a backup X.509 cert.

Client (relying party) gets trusted tree heads.

In TLS, client sends sequence number of latest batches it knows for each MTC CA. If client is sufficiently up-to-date, server returns a MTC or otherwise falls back to X.509.

Client trusts the MTC if the inclusion proof validates for one of its trusted tree heads.

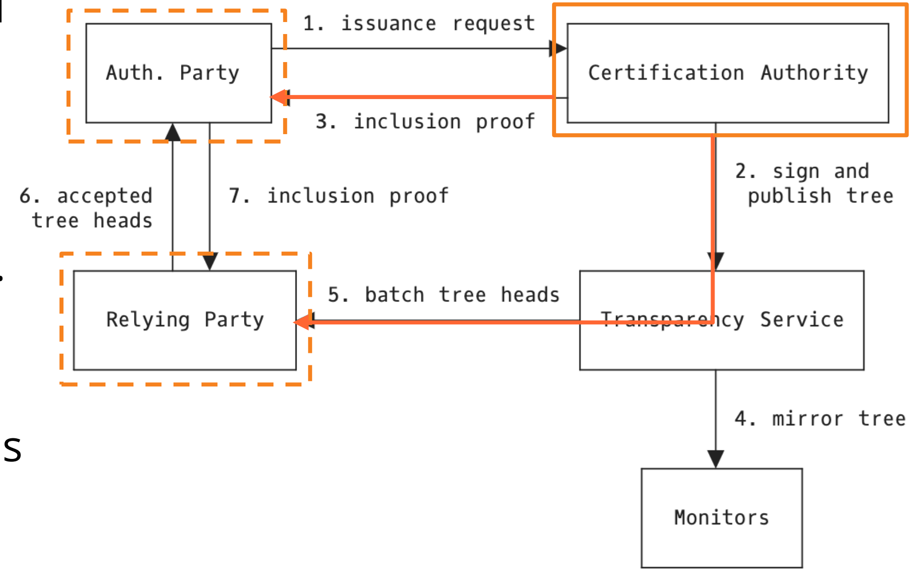


Figure 1: An overview of a Merkle Tree certificate deployment

Transparency Service

Intermediary in front of CAs that ensures all assertions are publicly logged and auditable.

- Mirrors CA assertions for monitors to audit.
- Provides window of valid batch tree heads to relying parties.

Key responsibility: if a relying party sees a tree head, the corresponding tree is available to monitors.

Transparency service makes MTC robust against CA failures (split views, downtime).

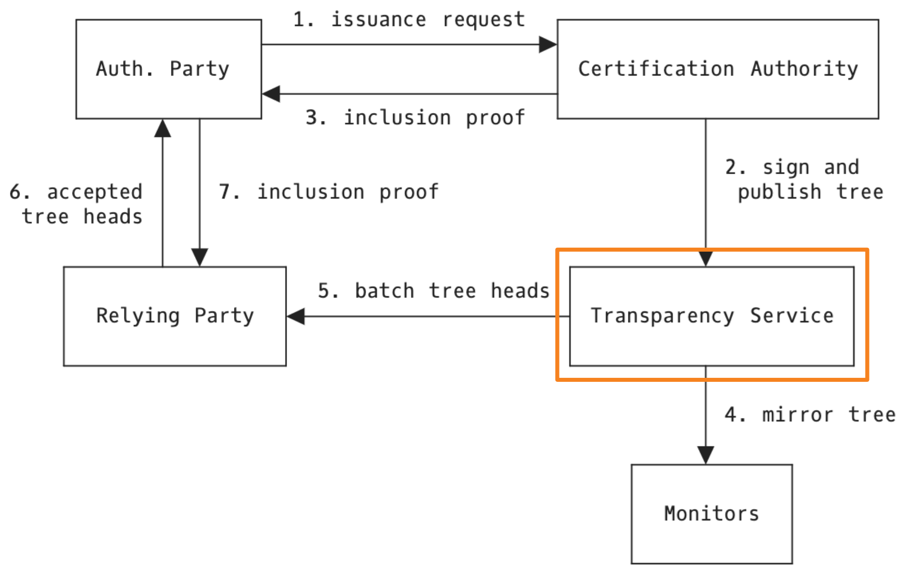


Figure 1: An overview of a Merkle Tree certificate deployment

MTC Demo - Create a CA

```
# Create a CA.
$ mtc ca new --batch-duration 5m --lifetime 1h 123.4.5 ca.example.com

# See which files were created.
./signing.key
./www/mtc/v1/ca-params
./queue

# Inspect CA parameters.
$ mtc inspect ca-params www/mtc/v1/ca-params
issuer                 123.4.5
start_time             1736361423 2025-01-08 13:37:03 -0500 EST
batch_duration         300        5m0s
life_time              3600       1h0m0s
storage_window_size    24         2h0m0s
validity_window_size   12 <- this is just 1h / 5m
http_server            ca.example.com
public_key fingerprint ml-dsa-87:8a217229497d6b0ef2911c60da284df59a6543b785a8e307c8239e6c94d585f2
```

https://github.com/bwesterb/mtc

MTC Demo - Queue an assertion

```
# Create an assertion (similar to a CSR)
$ openssl ecparam -name prime256v1 -genkey -out p256.priv
$ openssl ec -in p256.priv -pubout -out p256.pub
$ mtc new-assertion --tls-pem p256.pub --dns example.com -o my-assertion
checksum: a5036ed9faf56d72d36ba7fc6ce2de19323b2ec4da88cc848cebb42e5f2e841c
$ mtc inspect assertion my-assertion
subject_type     TLS
signature_scheme p256
public_key_hash  a8530c7f69c85989c60a8a40c34975e7a8e2610bbd39bc8e51784e0f19726b17
dns              [example.com]

# Submit it to the CA.
$ mtc ca queue -i my-assertion
$ mtc ca show-queue
checksum         c79d5fc55790456de2c08c7f6cfb7adebaf24acd7393826899d3621c0d9d4caa
subject_type     TLS
signature_scheme p256
public_key_hash  a8530c7f69c85989c60a8a40c34975e7a8e2610bbd39bc8e51784e0f19726b17
dns              [example.com]

Total number of assertions in queue: 1
```

MTC Demo - Issue a batch of assertions

```
# Try to issue the batch. Oops! Too early. (Usually this would be a cron job.)
$ mtc ca issue
2025/01/10 09:29:52 INFO Starting issuance time=2025-01-10T09:29:52.563-05:00
2025/01/10 09:29:52 INFO Current state expectedStored=∅ expectedActive=∅ existingBatches=∅
2025/01/10 09:29:52 INFO No batches were ready to issue. Next batch ready in 54s.

# Wait 54s (since we have 5m batch intervals) and try again... it worked!
$ mtc ca issue
2025/01/10 09:31:17 INFO Starting issuance time=2025-01-10T09:31:17.695-05:00
2025/01/10 09:31:17 INFO Current state expectedStored=0 expectedActive=0 existingBatches=∅
2025/01/10 09:31:17 INFO To issue batches=0

# Batch 0 has been created!
$ find . -type f
./signing.key
./www/mtc/v1/ca-params
./www/mtc/v1/batches/0/tree
./www/mtc/v1/batches/0/abridged-assertions
./www/mtc/v1/batches/0/signed-validity-window
./www/mtc/v1/batches/0/index
./queue
```

MTC Demo - Issue a batch of assertions

```
# Batch 0 has been created!
$ find . -type f
./signing.key
./www/mtc/v1/ca-params
./www/mtc/v1/batches/0/tree
./www/mtc/v1/batches/0/abridged-assertions
./www/mtc/v1/batches/0/signed-validity-window
./www/mtc/v1/batches/0/index
./queue

# Inspect the issued assertions.
$ mtc inspect abridged-assertions www/mtc/v1/batches/0/abridged-assertions
key                 5401b10a7e8dfa8cedaa1178e4576c240b0087629d5e4fe0ae178e019f2e26ef
subject_type        TLS
signature_scheme    p256
public_key_hash     a8530c7f69c85989c60a8a40c34975e7a8e2610bbd39bc8e51784e0f19726b17
dns                 [example.com]

Total number of abridged assertions: 1
```

MTC Demo - Issue a batch of assertions

```
# Batch 0 has been created!
$ find . -type f
./signing.key
./www/mtc/v1/ca-params
./www/mtc/v1/batches/0/tree
./www/mtc/v1/batches/0/abridged-assertions
./www/mtc/v1/batches/0/signed-validity-window
./www/mtc/v1/batches/0/index
./queue

# Inspect the tree.
$ mtc inspect tree www/mtc/v1/batches/0/tree
number of leaves 1
number of nodes  1
root             87463d48df7bd1420625cc21dadeb2d9eaf85c10db293eeadfc9d01e65ccd574
```

MTC Demo - Issue a batch of assertions

```
# Batch 0 has been created!
$ find . -type f
./signing.key
./www/mtc/v1/ca-params
./www/mtc/v1/batches/0/tree
./www/mtc/v1/batches/0/abridged-assertions
./www/mtc/v1/batches/0/signed-validity-window
./www/mtc/v1/batches/0/index
./queue

# Inspect the signed validity window (signature over all valid batch tree heads).
$ mtc inspect -ca-params www/mtc/v1/ca-params signed-validity-window www/mtc/v1/batches/0/signed-
validity-window
signature       ✅
batch_number    0
tree_heads[-11] 481f8e05ff34cb6314159cb5756c5f536b209442b99b4ea88157fc1aa6ec2341 <- placeholder
tree_heads[-10] 481f8e05ff34cb6314159cb5756c5f536b209442b99b4ea88157fc1aa6ec2341
<snip>
tree_heads[-1]  481f8e05ff34cb6314159cb5756c5f536b209442b99b4ea88157fc1aa6ec2341
tree_heads[0]   87463d48df7bd1420625cc21dadeb2d9eaf85c10db293eeadfc9d01e65ccd574
```

MTC Demo - Issue a batch of assertions

```
# Batch 0 has been created!
$ find . -type f
./signing.key
./www/mtc/v1/ca-params
./www/mtc/v1/batches/0/tree
./www/mtc/v1/batches/0/abridged-assertions
./www/mtc/v1/batches/0/signed-validity-window
./www/mtc/v1/batches/0/index
./queue

# Inspect the index, which maps hash of assertion to index in batch.
$ mtc inspect index www/mtc/v1/batches/0/index
                                                             key   seqno  offset
5401b10a7e8dfa8cedaa1178e4576c240b0087629d5e4fe0ae178e019f2e26ef       0       0

total number of entries: 1
```

Merkle Tree Certificates next steps

We are working with Chrome and others to experiment in 2025.
More feedback and collaboration is welcome!

Learn more:

- David Benjamin's TLS working group IETF 116 presentation
- Draft specification: https://davidben.github.io/merkle-tree-certs/draft-davidben-tls-merkle-tree-certs.html
- MTC CA implementation: https://github.com/bwesterb/mtc

Wrapping up

We saw several different approaches to cope with large post-quantum signatures, from simple to ambitious.

There are still many unknowns: among others, compliance requirements; cryptanalytic breakthroughs; ecosystem ossification; stakeholder constraints; etc.

Which approach to take? Best to try them all and see which ones gain traction.

Thank you, questions?

Please reach out if you want to collaborate on testing these approaches @ ask-research@cloudflare.com