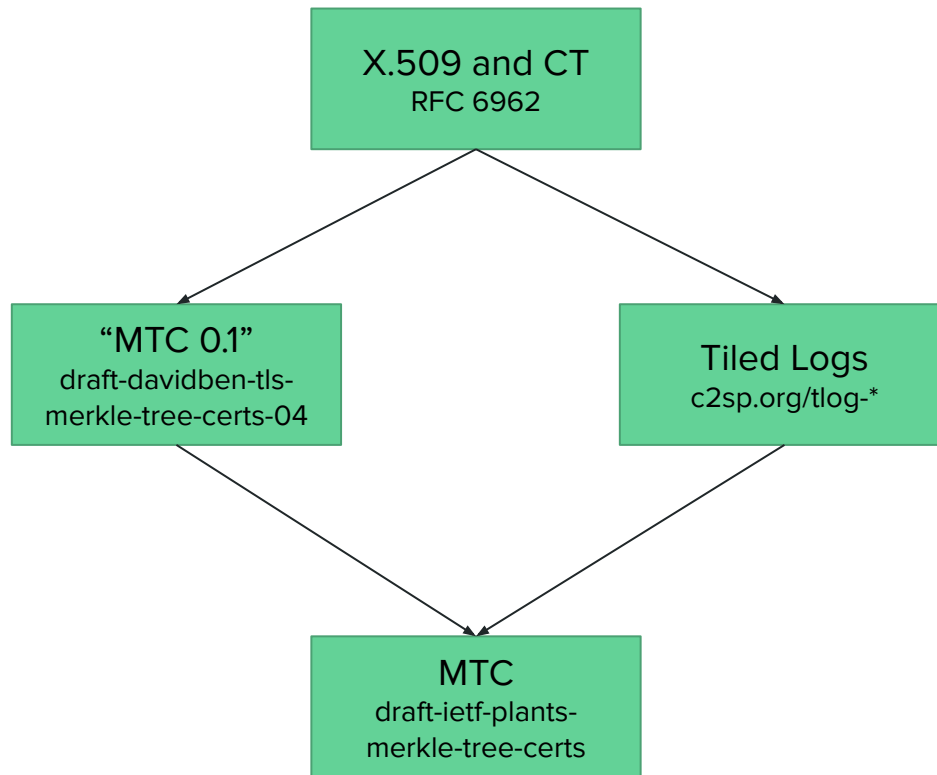


Transparency Log Developments from MTCs

David Benjamin

Transparency.dev Summit 2026





Parallel evolution and convergence

Looking back to MTC 0.1

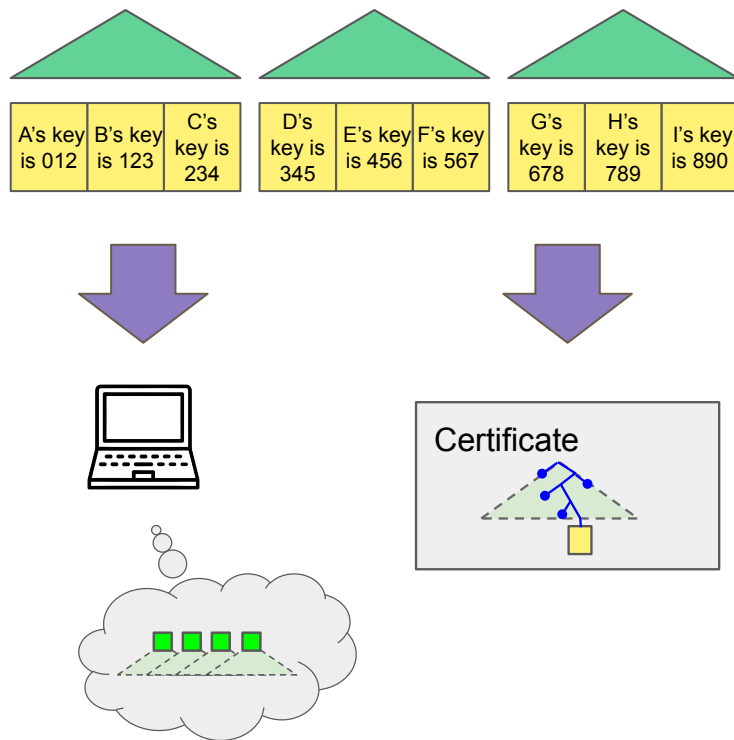
CAs collected certificate requests into hourly *batches*

- CAs built trees over each batch
- CA signs window of tree hashes to certify
- Separate trees dodge 2023 scaling pains
- certificate = entry + inclusion proof

Predistribute window of tree hashes to clients

Use fallback cert for out-of-date clients

Targeted handshake size, but log size also improved



One very productive Filippo conversation later...

Rebuild MTC with tlog

Keep the CA-built trees

Keep the small entries

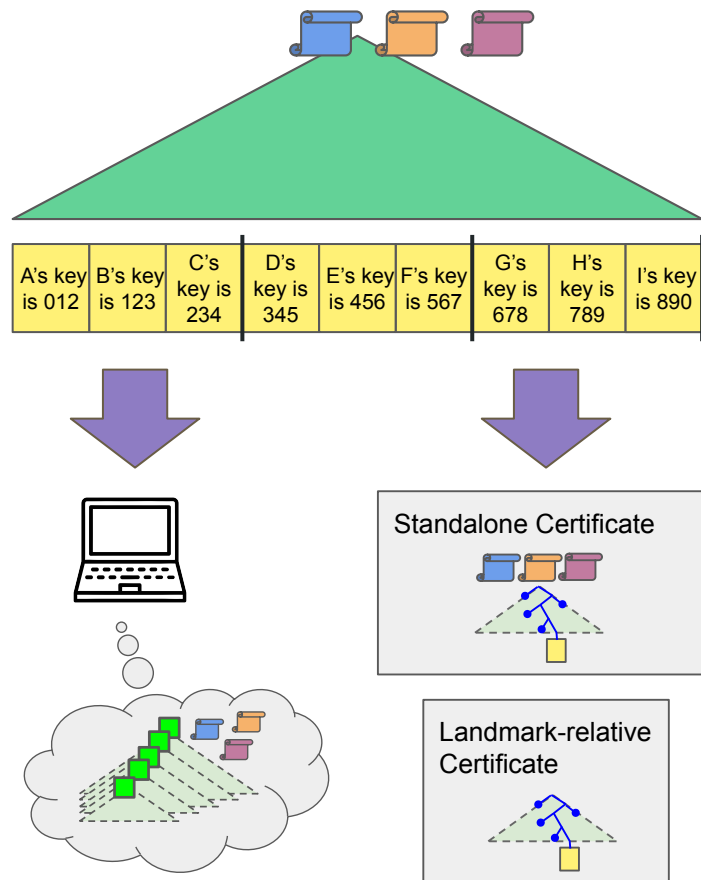
Back to one big tree—tlog solved tree serving!

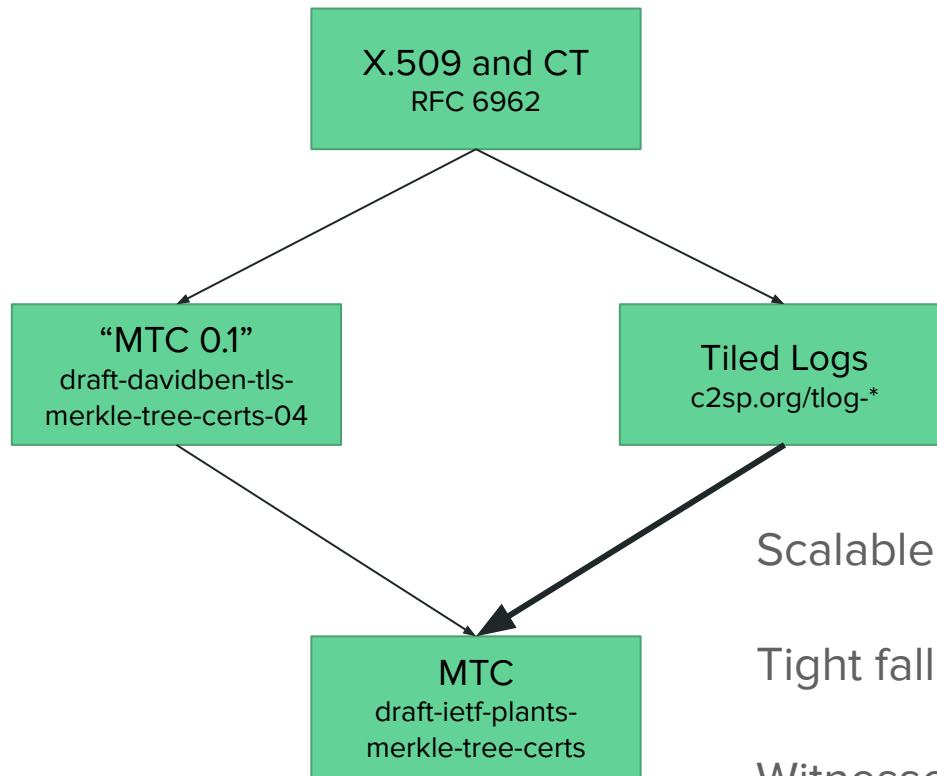
A single CA checkpoint signature certifies everything

Batches became predistributed hourly *landmark checkpoints*¹

- Standalone: inclusion proof and checkpoint¹ (co)signatures
- Landmark-relative: only inclusion proof

¹ For now...





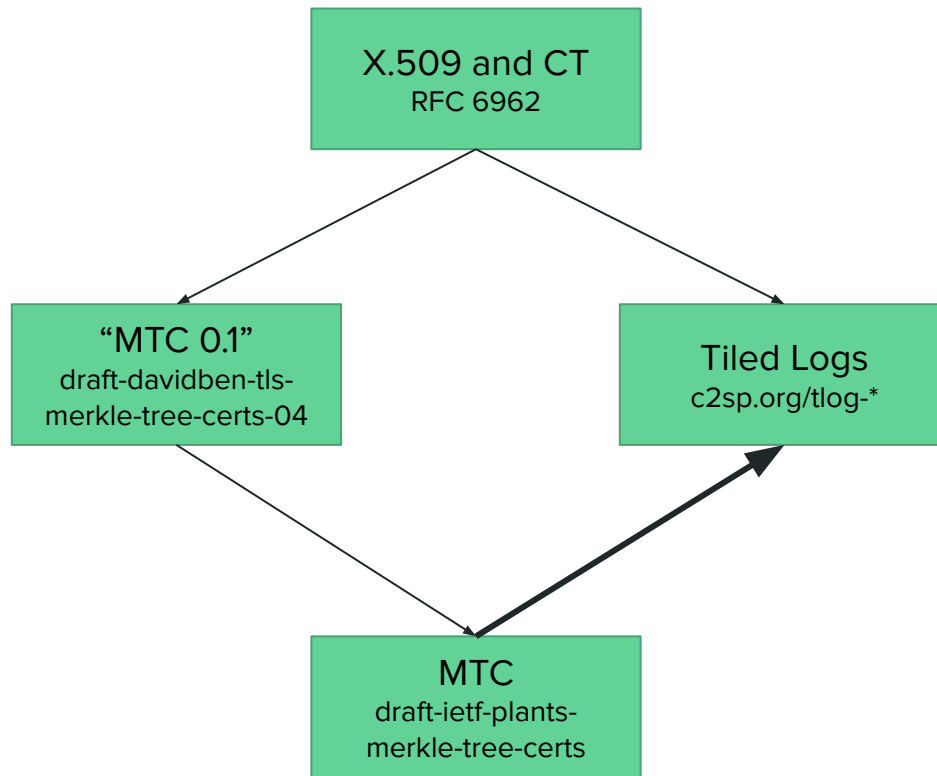
Scalable log serving

Tight fallback story

Witnessed checkpoints

Tons of existing infrastructure

What tlog gave MTC...



What MTC can give back to tlog?

What's left?

Inclusion proofs got bigger

Ecosystem role comparison

Handling log growth

Subtrees



Inclusion proofs got bigger

MTC 0.1: landmarks contain one hour's worth of certificates...

$\log_2(\text{certs_per_hour})$ hashes

MTC + tlog: checkpoints contain all certificates...

$\log_2(\text{total_certs})$ hashes

If you remember your logarithms...

$$\log_2(\text{total_certs}) - \log_2(\text{certs_per_hour}) = \log_2(\text{total_hours})$$

Difference for two-year old CA is 14 hashes or 448 bytes *per TLS handshake*

Build small trees from big trees?

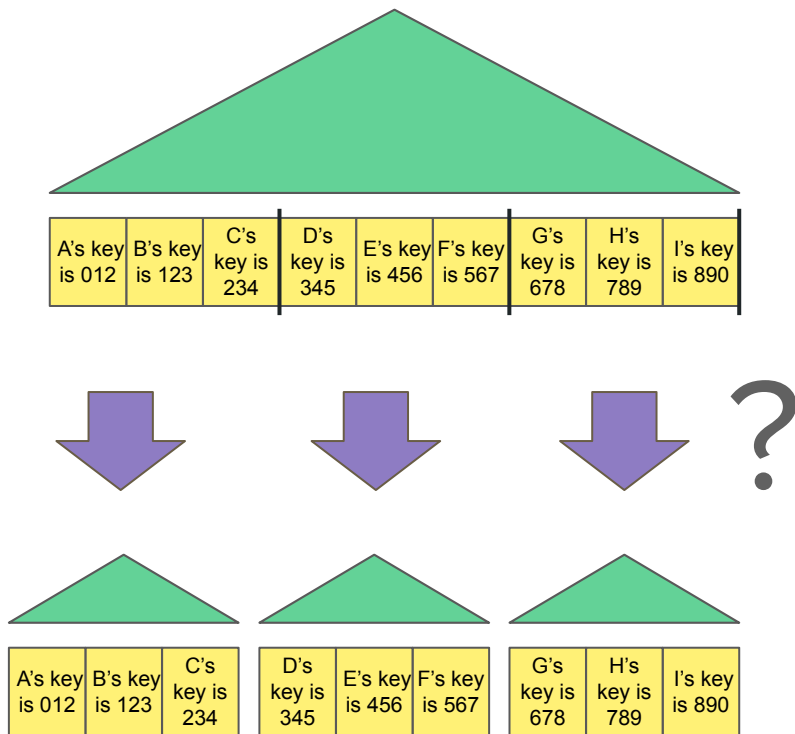
What if we extracted hourly trees again?

$\text{MTH}(D_n) \rightarrow \text{MTH}(D[\text{start}:\text{end}])$

Client / monitor mismatch

- Clients see $\text{MTH}(D[\text{start}:\text{end}])$
- Monitors see $\text{MTH}(D_n)$

Can we prove them consistent?



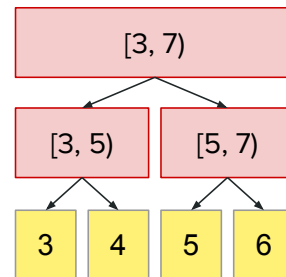
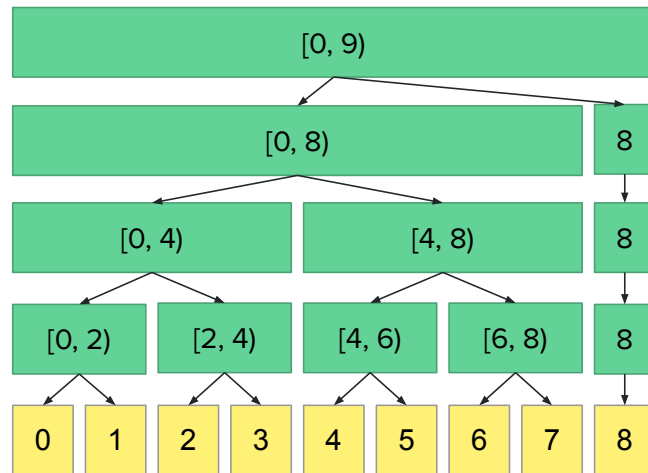
Misaligned trees

Consistency proofs work because trees share common nodes

Merkle Trees work by multiples of 2^N

An arbitrary range may be misaligned

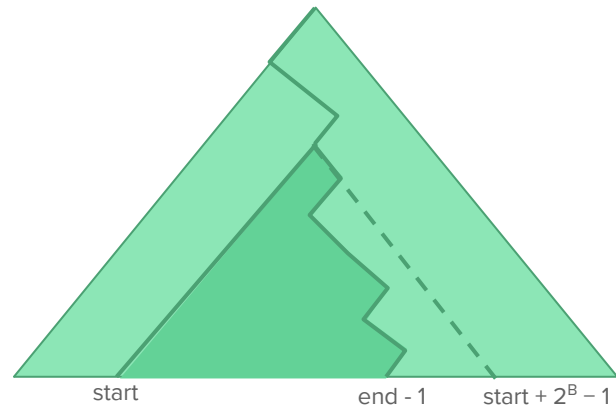
No hope for efficient consistency proof



Subtrees

Limit *subtrees* to power-of-two-aligned ranges

- $\text{start} = A \times 2^B$ for some A and B
- $A \times 2^B \leq \text{end} \leq (A+1) \times 2^B$



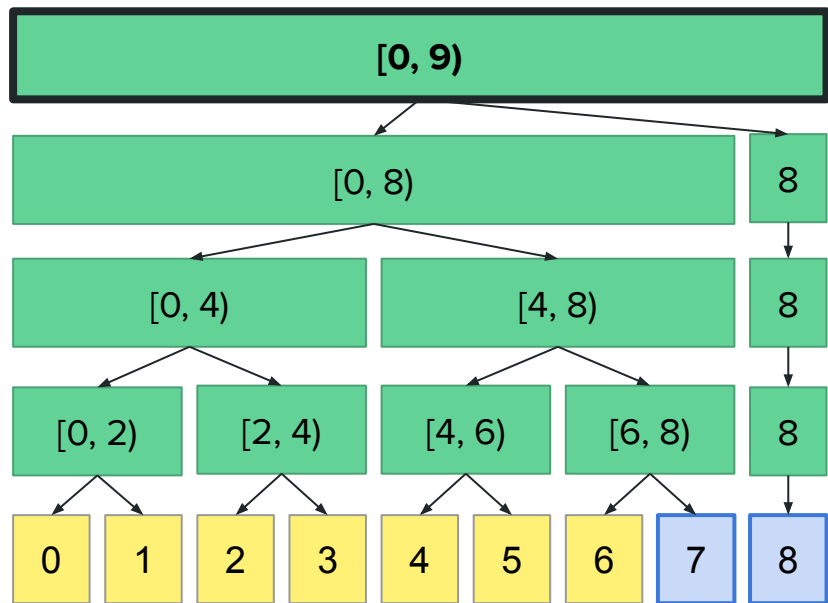
Then consistency proof between $\text{MTH}(D[\text{start}:\text{end}])$ and $\text{MTH}(D_n)$ is possible

Our ranges aren't subtrees...

Round up to the smallest subtree containing the range?

Oops, we accidentally the whole tree

Rounding $[7, 9)$ up to a subtree gives the entire tree $[0, 9)$



Apr 26, 2025

Okay, solved it! The answer is 2. I need to go write down how to pick these nodes and why it works, but...

Given a tree of size N and a batch of the rightmost B nodes in the tree, we can constructively pick up to 2 interior nodes such that:

1. The disjoint union of their subtrees is the rightmost B' nodes of the tree
2. $B \leq B' < 2B$

This means we multiply client state by a factor of 2, but the max height of the two subtrees is at most $\text{ceil}(\log_2(B))$, which is exactly the bound MTCs achieved.

Brought to you by "the reason David woke up at 5:30am and can't go back to sleep"

06:07 

Then I didn't get enough sleep....

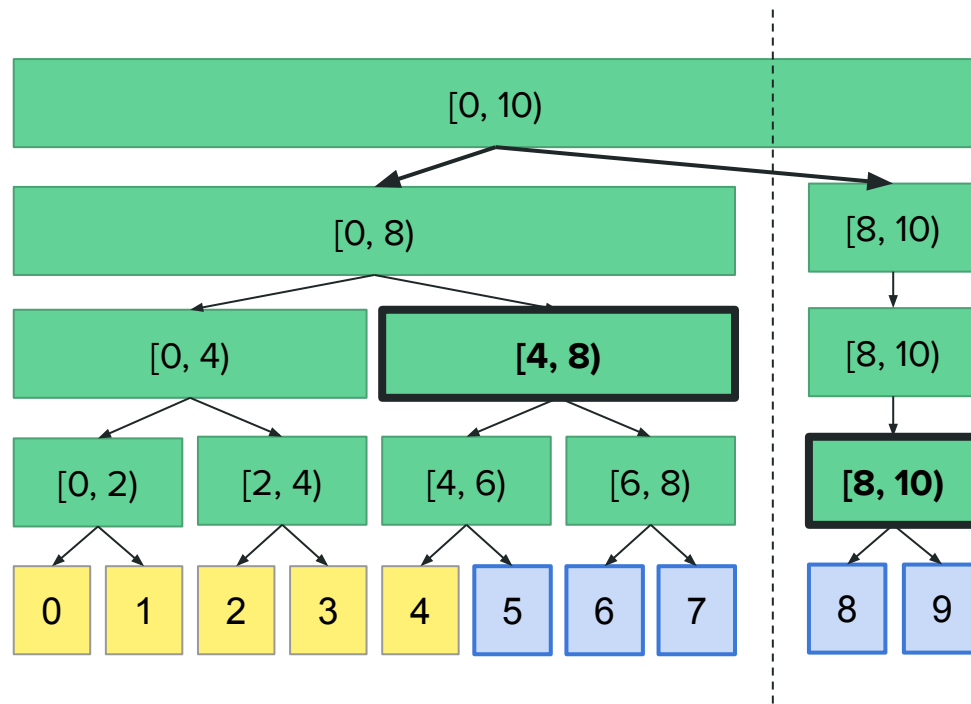
Two nodes are efficient

Find where first and last nodes diverge

Split the range there

Find smallest subtree per half

- Left half is full, so rounding up is efficient
- Right half is already a valid subtree



What does this mean?

You can “hash” any range of any Merkle Tree

(Really two hashes)

Supports smaller inclusion proofs for everything in the range

Can be proven consistent with overall Merkle Tree

Spec updates

- draft-ietf-plants-merkle-tree-certs defines subtrees
- tlog-cosignature’s ML-DSA algorithm can sign subtrees or checkpoints
- tlog-witness has a sign-subtree endpoint to request subtree signatures

What's left?

~~Inclusion proofs got bigger~~

Ecosystem role comparison

Handling log growth

Mirrors



Ecosystem role comparison

Web PKI today uses quorum of independent logs (plus CA)

tlog-checkpoint uses quorum of independent witnesses (plus log)

Map CT logs to MTC witnesses?

Log quorum protects against split views *and* unavailability

Witness quorum *only* protects against split views

MTC 0.1: trees could be mirrored by other services

Generalizing witnesses

A witness cosignature says:

“This checkpoint is consistent with everything I’ve seen”

A mirror cosignature says:

“This checkpoint is consistent with everything I’ve seen *and* I mirrored the log up to this point”

c2sp.org/tlog-mirror

Spec for mirror role and APIs

Independent of MTCs, any tlog application can use it

Two-stage update process

- Avoid large atomic transactions
- Always stay consistent, even if origin log is not

Cosigns checkpoints when mirrored...

Additional cosignature statements

Witness and mirror checkpoint signatures have different semantics

No need for different cosignature type per role

- If you know the cosigner, you know the role

- If you don't know the cosigner, you don't care

tlog-cosignature@main:

- “A cosigner MAY make additional statements about a checkpoint. These additional statements need to be communicated out of band to those defining trust policies based on tuples of (public key, supported cosignature version).”

Bonus: this unify log and witnesses signatures

Extensible

Applications can define cosigner roles *without changing tlog protocol or libraries*

“This checkpoint is consistent with everything I’ve seen and...”

- “...this checkpoint is the state of my log” (log owner)
- “...I mirrored the log up to this point” (mirrors)
- “...I certify all the name/key bindings in these entries” (MTC CAs)

Or even:

- “...I checked `entry.foo_header.bar_counter` fields are all consistent”
- “...I checked all entries contain a square number”
- “...I think all the entries are really, really cool!”

What's left?

~~Inclusion proofs got bigger~~

~~Witnesses aren't quite the same as CT logs~~

Handling log growth

Pruning



Handling log growth

Most of the volume is expired and stale

CT temporally shards logs

Every 6 months or 1 year, reset with new logs

This assumes stale clients disable transparency enforcement

CA rotation is harder—stale clients cannot disable *all* certificate verification

MTC 0.1: old landmarks stop mattering and could be discarded...

Pruning the tree

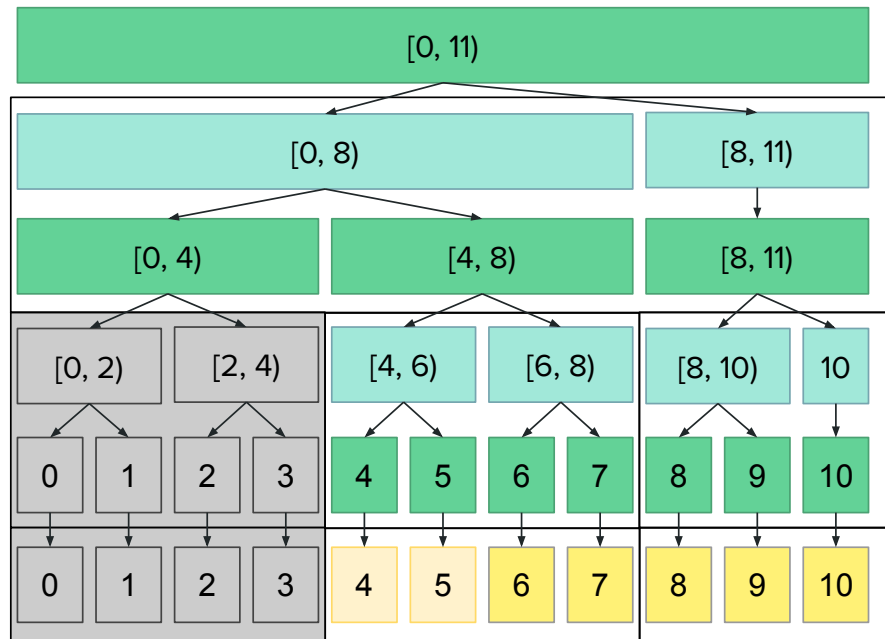
Just don't serve the old stuff!

Stop serving a tile if every descendant is below some *minimum index*

No change to tree structure—all proofs checkout

Unpruned nodes still available

Proofs involving unpruned nodes still available



min index = 6
tile size 4 instead of 256

Log availability policy

Pruning really an availability policy tool

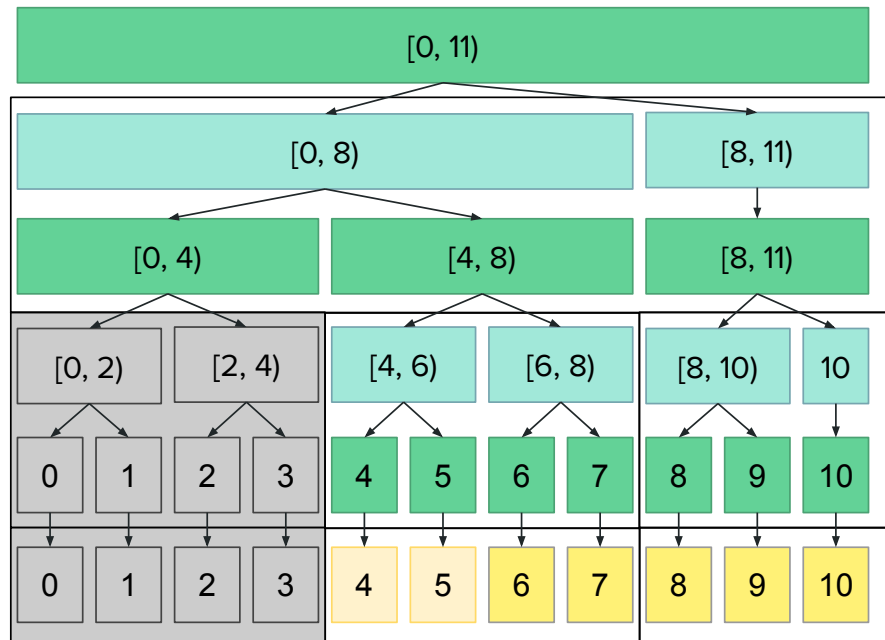
Log *already* might fail to serve a tile

- Clients *already* must have policy

What is pruning, really?

- Some tiles do not matter for new entries
- Clients *could* decide to tolerate worse availability on obsolete tiles

tlog-tiles@main#pruning



min index = 6
tile size 4 instead of 256

Client-side enforcement

If entries are pruned, clients *shouldn't accept them*

Clients configure a minimum index (minSerial) with MTC CAs

Availability policies should integrate this:

tlog-tiles@main#pruning

“Log applications that support pruning SHOULD additionally define mechanisms for applicable log clients to reject entries below some per-log lower bound.”

mtc-tlog@main:

“Relying parties SHOULD set restrictions on pruning, such as requiring that the log's minimum index be at most the minimum trusted index in up-to-date copies of the relying party's trust anchors”

What's left?

~~Inclusion proofs got bigger~~

~~Ecosystem role comparison~~

~~Handling log growth~~

Questions?

Subtrees

`draft-ietf-plants-merkle-tree-certs#name-subtrees`

`tlog-witness@main#sign-subtree`

`tlog-cosignature@main`

Additional statements and mirrors

`tlog-cosignature@main`

`tlog-mirror`

Pruning

`tlog-tiles@main#pruning`

