Architecture Document

Brendan McMillion
IETF 124 / November 5, 2025

No changes since IETF 123

Next two PRs:

1. Editorial (update references to protocol details that have changed)
2. “Maximum Lifetime” feature was never discussed wrt pruning or privacy law compliance (oops)

Then last call?

Protocol Document

Fixed Version Search

Fixed-Version Search Ladder	**Greatest-Version Search Ladder**	**Monitoring Ladder**
Stops at first: - Non-inclusion proof for a version < target - Inclusion proof for a version >= target	Stops at first: - Non-inclusion proof for a version < target - Inclusion proof for a version > target	Expects inclusion proofs for all requested versions

Fixed Version Search

Fixed-Version Search Ladder	**~~Greatest-Version~~ Search Ladder**	**Monitoring Ladder**
Stops at first: - Non-inclusion proof for a version < target - Inclusion proof for a version >= target	Stops at first: - Non-inclusion proof for a version < target - Inclusion proof for a version > target	Expects inclusion proofs for all requested versions

Now only one ladder type used for both types of search.

Fixed Version Search

Previous Algorithm:

Binary search for very first log entry where target version of label exists.

New Algorithm:

Binary search for very first log entry where target version of label exists

BUT stop as soon as any log entry is found where the target version is the **greatest** that exists

Fixed Version Search

Previous Algorithm:

Binary search for very first log entry where target version of label exists.

New Algorithm:

Binary search for very first log entry where target version of label exists

BUT stop as soon as any log entry is found where the target version is the **greatest** that exists

- Each search step may be more expensive, but there are fewer search steps => No real impact on efficiency
- Fewer binary ladder types => Less code, easier to reason about security

Greatest Version Search

No changes

Update

Previous Algorithm:

There wasn’t one!

Important things to remember:

- Labels may change without the user initializing it (in the background)
- Multiple new versions of a label may be created in a single log entry

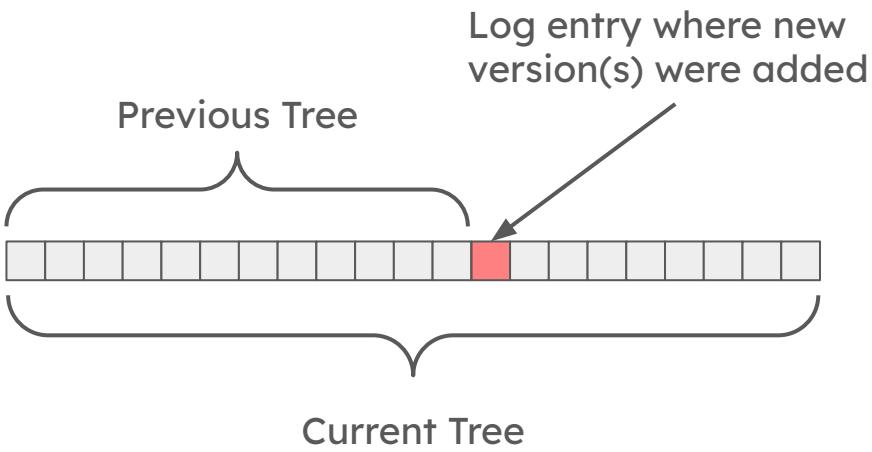
Update

Previous Algorithm:

There wasn't one!

Current Algorithm:

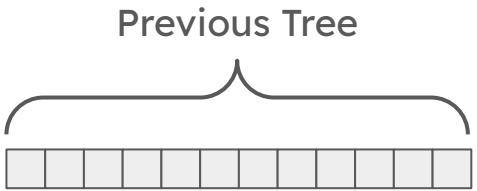
1. Divide the tree into "previous" and "current"



Update

Previous Algorithm:

There wasn't one!



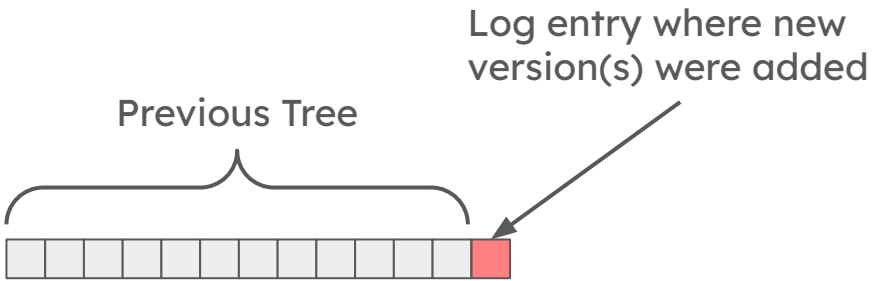
Current Algorithm:

1. Divide the tree into "previous" and "current"
2. Perform a Greatest Version Search in the previous tree; verify no unexpected versions are found.

Update

Previous Algorithm:

There wasn't one!



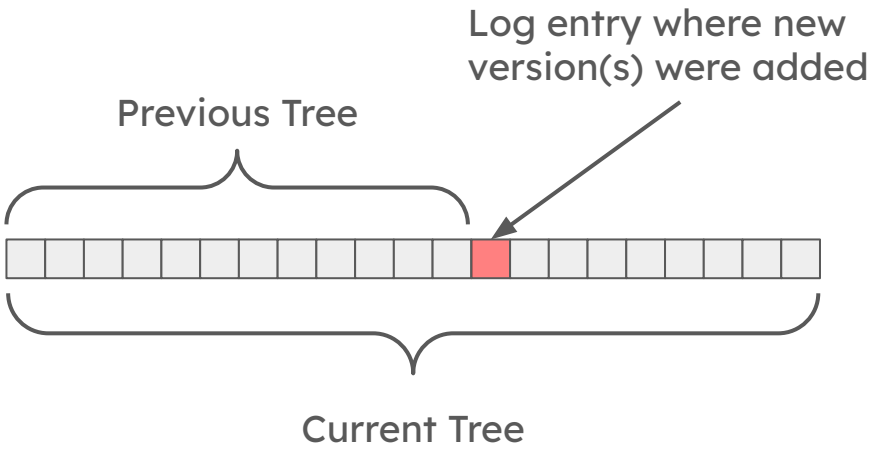
Current Algorithm:

1. Divide the tree into "previous" and "current"
2. Perform a Greatest Version Search in the previous tree; verify no unexpected versions are found.
3. Verify the correct values were inserted into the Prefix Tree of the new log entry

Update

Previous Algorithm:

There wasn't one!



Current Algorithm:

1. Divide the tree into "previous" and "current"
2. Perform a Greatest Version Search in the previous tree; verify no unexpected versions are found.
3. Verify the correct values were inserted into the Prefix Tree of the new log entry
4. Contact Monitoring algorithm takes over going forward

Contact Monitoring

No changes

Owner Monitoring

Previous Algorithm:

Short fluffy description: “Users check each distinguished log entry for correctness in left-to-right order”

Current Algorithm:

Main body of algorithm specified as depth-first search. For a given log entry:

1. Recurse left if that makes sense
2. Monitor the current log entry
3. Right right if that makes sense

\+ Specified how to *initialize* owner state

Credentials

Previously:

Assumed credentials would basically just be a serialized Search

Current:

They basically still are, but:

- Redundant information is reduced
- Correct verification is specified
- State for verifying credentials MUST be sandboxed

Credentials

```
enum {
  reserved(0),
  standard(1),
  provisional(2),
  (255)
} CredentialType;

struct {
  CredentialType credential_type;

  uint32 version;
  opaque opening[Nc];
  UpdateValue value;

  BinaryLadderStep binary_ladder<0..2^8-1>;
  select (Credential.credential_type) {
    case standard:
      uint64 tree_size;
      PrefixProof distinguished;
    case provisional:
      FullTreeHead full_tree_head;
      CombinedTreeProof search;
  };
} Credential;
```

Two credential types:
standard or provisional

Credentials

```
enum {
  reserved(0),
  standard(1),
  provisional(2),
  (255)
} CredentialType;

struct {
  CredentialType credential_type;
  uint32 version;
  opaque opening[Nc];
  UpdateValue value;

  BinaryLadderStep binary_ladder<0..2^8-1>;
  select (Credential.credential_type) {


    case provisional:
      FullTreeHead full_tree_head;
      CombinedTreeProof search;
  };
} Credential;
```

= provisional

Exactly the same as a Search

Credentials

```
enum {
  reserved(0),
  standard(1),
  provisional(2),
  (255)
} CredentialType;

struct {
  CredentialType credential_type; = standard

  uint32 version;
  opaque opening[Nc];
  UpdateValue value;

  BinaryLadderStep binary_ladder<0..2^8-1>;
  select (Credential.credential_type) {
    case standard:
      uint64 tree_size;
      PrefixProof distinguished;



  };
} Credential;
```

Only one step of a Search (much smaller)

Hot Swapping Auditors

Change to arch doc was made before IETF 123. This is implementation:

Users verify an `AuditorTreeHead` with the following steps:

1. If the user advertised a previously observed tree head, verify that the `tree_size` of the `AuditorTreeHead` structure in the previous tree head (which may be from a different auditor) is greater than or equal to `auditor_start_pos` for the current auditor.

To Do

- Implement more of draft => fix things that come up from that https://github.com/Bren2010/katie

The End