Key Transparency Auditor Spec

May 1, 2025

The big picture

In an end-to-end encrypted messaging ecosystem, users must have some way to obtain the public keys of the users they wish to message. This typically happens by users uploading a mapping of their public identifier (for example, their phone number) to their public key to a central service, and other users querying this service. However, this requires users to trust that the service operator will behave honestly and not tamper with the log.

Key transparency ensures that a service operator cannot do this without being detected. It does so by sequentially recording updates to the mappings, where an update is either an insert of a new mapping from a search key to a value, or an update of an existing mapping. In this context, "search key" refers to something like "e164:+18005550101", and the value it maps to could be the public key associated with that phone number.

Every update—also referred to as a log entry—is appended to the key transparency log, a globally consistent, cryptographically-protected, append-only log. Although the key transparency log is an append-only construct, callers will need to update existing search keys from time to time (e.g. if they lose their phone and wind up generating a new public key). The key transparency log allows for updates by appending new log entries that supersede previous entries for the same search key.

The key transparency log consists of two types of data structures: a *log tree* and *prefix trees*. Log entries are stored as leaves in the *log tree*, and *prefix trees* are used to facilitate efficient lookups of a specific mapping in the large and ever-growing log tree. The transparency log uses various cryptographic techniques to append to the log tree and update prefix trees in a verifiable manner. However, this alone is not sufficient to detect whether the service operator has forked the log because the service operator can continue to serve views that are consistent per user but not globally. Instead, some party must also monitor the log and verify that the service operator is serving the same view of the log to all users. Signal has chosen to do this monitoring via *third-party auditing*; other options include *contact monitoring* and *third-party management*.

Third-party auditors operate by requesting batches of updates from the key transparency service, cryptographically verifying those updates, and then returning an "auditor tree head" back to the key transparency service, verifying that its view of the transparency log agrees with the main service's view. Auditors keep a much, much smaller subset of information than the main key transparency service (in our production environment, our reference auditor implementation maintains less than 3 KiB of persistent data). The two main things auditors need to hold onto are the previous "prefix tree root hash" and just enough "log tree node hashes" to calculate new "log tree root hashes".

A quick note

The term “key” is overloaded in this space. It can refer to a “search key” as described above, or a user’s public key which is the value stored in a mapping, or a cryptographic key used to perform various encryption and hashing operations. We did our best to disambiguate each usage by prepending with a descriptor.

The “key” in “key transparency” refers to the system’s goal of transparently publishing users’ public keys in a way that is cryptographically verifiable.

Glossary

This section is a quick introduction to some important terms and data structures used throughout the rest of this document. Where applicable, there are links to sections with more detailed explanations.

- **Search key:** The raw input value for a lookup in the log. For example, this might be something like “e164:+18005550123”. The auditor never sees or processes raw search keys for privacy reasons. Instead, it works with something called a “commitment index”.
- **Commitment index:** A 256-bit (16-byte) value deterministically computed by a Verifiable Random Function (VRF) from the search key. See Verifiable Random Functions for more details.
- **Log entry:** A record of a change to a mapping. It is represented as a leaf in the log tree. The auditor always works with a *commitment* instead of the log entry itself.
- **Commitment:** A cryptographic hash computed from the updated mapping. See Commitments for more details.
- **Prefix tree:** A 256-level binary Merkle tree. Every log entry has its own prefix tree. See Prefix tree for more details.
- **Log tree:** A left-balanced, append-only binary Merkle tree. Each leaf in the log tree corresponds to a log entry and new log entries are appended to the rightmost edge. See Log tree for more details.
- **Tree head:** A representation of the state of the log tree at a given point in time. Contains the size of the log tree, a timestamp, and a signature over the size, timestamp, log tree root hash, and some configuration data. See Signatures for more details.
    - **Auditor tree head:** The auditor’s view of the state of the key transparency log.
- **Root hash:** The cryptographic hash of the root node of a tree.
    - **Prefix tree root hash**: The root hash of the prefix tree for a given log entry.
- **Node hash:** The cryptographic hash of a parent or leaf node in a tree.
    - **Log tree node hash:** The cryptographic hash of a parent or leaf node in the log tree.

Tree terms

- **Subtree:** The tree formed by all descendants of a given node.

- **Leaf node:** A node with no children.
- **Parent node**: A node with either a left or right child node or both.
- **Root node:** A node with no parents.
- **Intermediate node:** A node with both parent and child nodes.
- **Direct path:** The path to a given node. The direct path of the root node is an empty list.
- **Copath:** The copath of a given node is the node's sibling concatenated with siblings of all the nodes in its direct path, excluding the root node. See Copaths for more details.

Operational prerequisites

A third-party auditor requires some cryptographic keys and a persistent storage solution to operate.

Cryptographic Keys

An auditor will need all of the following cryptographic keys:

1. Its own Ed25519 key pair for producing signatures
2. The VRF public key used by the main key transparency service
3. The signature public key used by the main key transparency service

An auditor will also need to provide the public part of its Ed25519 key pair to the key transparency service so that the key transparency service can verify the auditor's signature.

A Signal-issued client certificate

The auditor is the only party external to the key transparency service that can write data to the transparency log and therefore, it must be authenticated in order to do so. The key transparency service is set up behind an AWS application load balancer that performs mutual TLS with Signal's private certificate authority configured as a trusted root certificate authority.

The auditor should send a certificate signing request to Signal's private certificate authority to get a client certificate. The auditor must provide this certificate in order to pass mutual TLS with the load balancer and talk to the key transparency service.

The load balancer's certificate is issued from AWS's certificate manager for the domains `audit.kt.signal.org` (production) and `audit.kt.staging.signal.org` (staging).

Storage

An auditor must persist some data about its own state so that it can pick up auditing the key transparency's log where it last left off. Specifically, the auditor needs to persist:

- How many updates the auditor has processed so far

- The prefix tree root hash associated with the number of updates processed so far
- The log tree node data necessary to recompute the log tree root hash after the given number of updates processed. See the Maintaining the auditor's log tree section for more details.

This is the same set of data that the auditor should have on hand after each update (see Processing updates for more details).

In addition, the auditor should ensure that a malicious party cannot tamper with the stored data to trick the auditor into accepting a bad update. One possible way to achieve this would be to store and verify a signature over the persisted data.

API

A third-party auditor communicates with the key transparency service via gRPC, specifically the `Audit` and `SetAuditorHead` methods:

```
None
service KeyTransparencyService {
  /**
   * Auditors use this endpoint to request a batch of key transparency service
   * updates to audit.
   */
  rpc Audit(AuditRequest) returns (AuditResponse) {}
  /**
   * Auditors use this endpoint to return a signature on the log tree root hash
   * corresponding to the last audited update.
   */
  rpc SetAuditorHead(AuditorTreeHead) returns (google.protobuf.Empty) {}
}
```

Requesting updates

The auditor sends an `AuditRequest` to `kt.signal.org`'s `Audit` method requesting up to `x` updates starting from the `total_updates_processed` index. The auditor may set the `limit` field up to 1000; the key transparency service will reject the request if `limit` is set higher.

```
None
message AuditRequest {
  /**
   * The index of the next update to process.
```

```
   */
  uint64 start = 1;
  /**
   * The maximum number of updates to return for auditing, starting from the
   * given index.
   */
  uint64 limit = 2;
}
```

The key transparency service's response looks like this:

```
None
message AuditResponse {
  /**
   * A list of updates for the auditor to audit.
   */
  repeated AuditorUpdate updates = 1;
  /**
   * Whether there are additional updates for the auditor to audit.
   */
  bool more = 2;
}
```

While `more = true`, the auditor should request another batch of updates, setting the `start` field of the next `AuditRequest` to that of the previous `AuditRequest` plus the number of updates in the response. If `more = false`, the auditor should stop requesting updates.

Each `AuditorUpdate` protobuf contains the following fields:

```
None
message AuditorUpdate {
  bool real = 1;
  bytes index = 2;
  bytes seed = 3;
  bytes commitment = 4;
  AuditorProof proof = 5;
}
```

- `real` - whether this is a “real” or “fake” update.
- `index` - the `commitmentIndex` (i.e. VRF output, see the VRF section). This is a randomly generated value if the update is fake.
- `seed` - a pseudo-random value that gets hashed with a prefix tree level to produce a stand-in hash value for an unpopulated node in a level of the prefix tree.
- `commitment` - the `commitment` described in the Commitments section; this is a function of the contents of the log tree entry, and is used to calculate log tree leaf hashes. This is a randomly generated value if the update is fake.
- `proof` - a cryptographic proof that this update is a valid extension of the previous prefix tree and log tree; there are three possible types of proofs, each detailed in the AuditorProofs section below.

Fake updates

In order to obscure the true update rate of the key transparency log from outside observers, the key transparency service generates “fake updates”.

Fake updates use randomly generated values for the commitment index and the commitment where real updates would use cryptographically-derived values. Fake updates are also represented differently than real updates in the prefix tree; see Fake update to prefix tree for more details and Processing a fake update for a concrete example. However, fake update log entries are generated and appended to the log tree in the same way as real updates.

Data structures and cryptographic components

Before we discuss how the auditor processes and verifies each update, it’s important to understand the data structures and cryptographic components that the main key transparency service uses to preserve and prove the integrity of the log.

Verifiable Random Functions (VRF)

To preserve the privacy of the actual search keys that exist in the key transparency log, the key transparency service uses a verifiable random function, also known as a VRF, to deterministically map the search key to a 256-bit pseudorandom value known as the `commitmentIndex`. The key transparency service generates an Ed25519 public/private key pair — where the public key is known by clients and the private key is known only by the key transparency server — and computes the `commitmentIndex` as:

```
commitmentIndex, commitmentVrfProof = VRF(vrfPrivateKey, searchKey)
```

The commitment index is used to traverse the prefix tree.

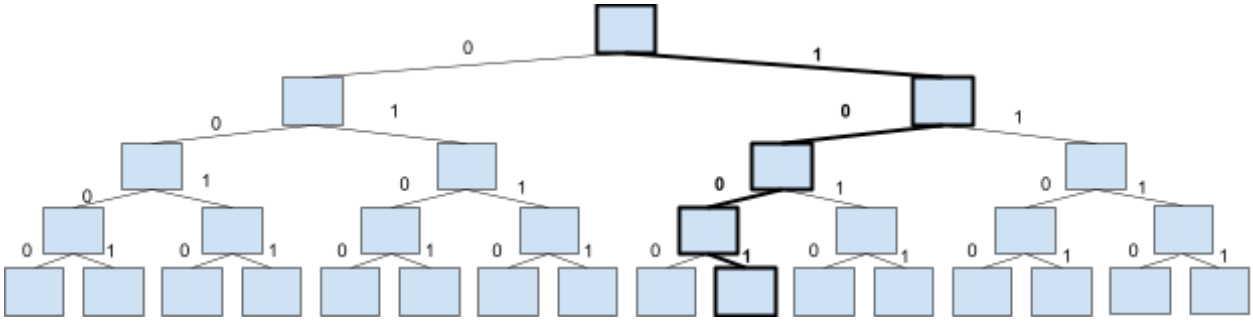
Prefix tree

In the context of key transparency, a prefix tree is a 256-level binary Merkle tree where each leaf corresponds to a commitment index (defined above) which itself corresponds to a given search key. The prefix tree is traversed with the `commitmentIndex`; that is, we go left or right at each node in the tree based on whether the `n`th bit in the commitment index is set.

Example

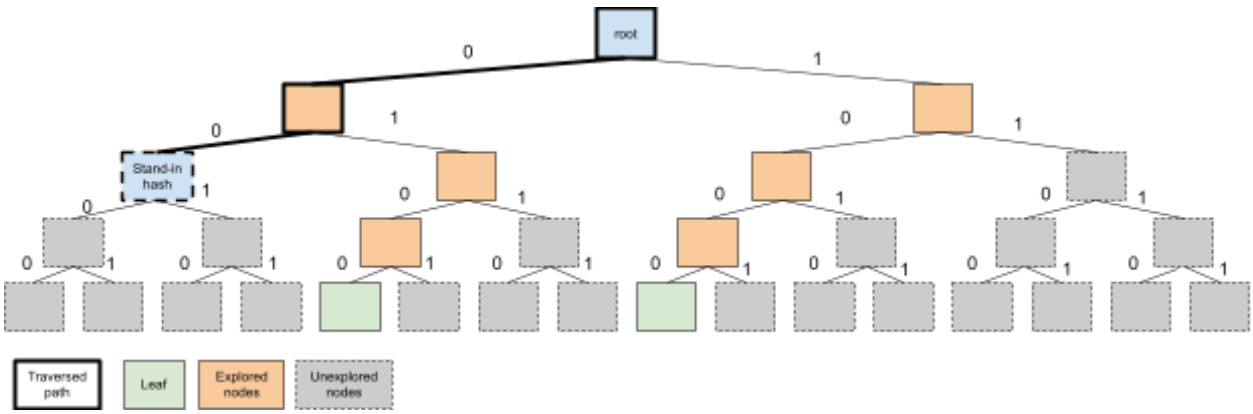
Real update to prefix tree

Assuming that the `commitmentIndex` is computed as 4 bits rather than 256, and we have `commitmentIndex=0b1001` for a real update, the prefix tree would look like this, and the bolded nodes would be the path taken to find the leaf corresponding to `0b1001`.



Fake update to prefix tree

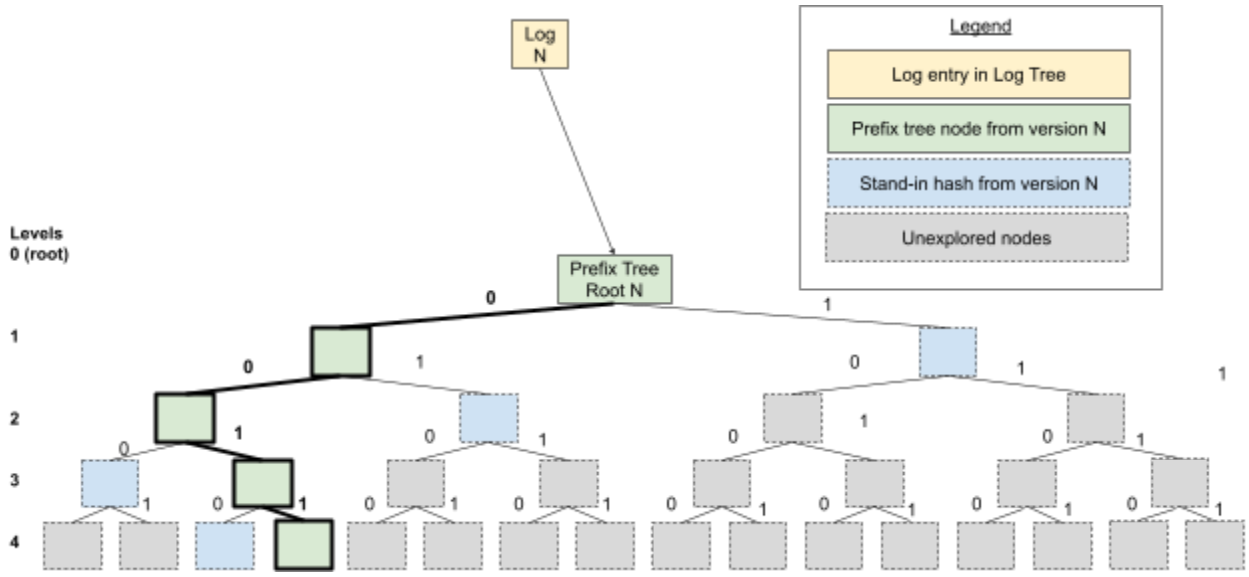
Only real updates create a leaf in the prefix tree. Fake updates generate a new [“stand-in hash”] at the first “unexplored node” they encounter — a node whose subtree has no leaves. For example, if the tree already had 2 real updates, and the next update is fake with `commitmentIndex=0b0010`, its traversed path would look like this:



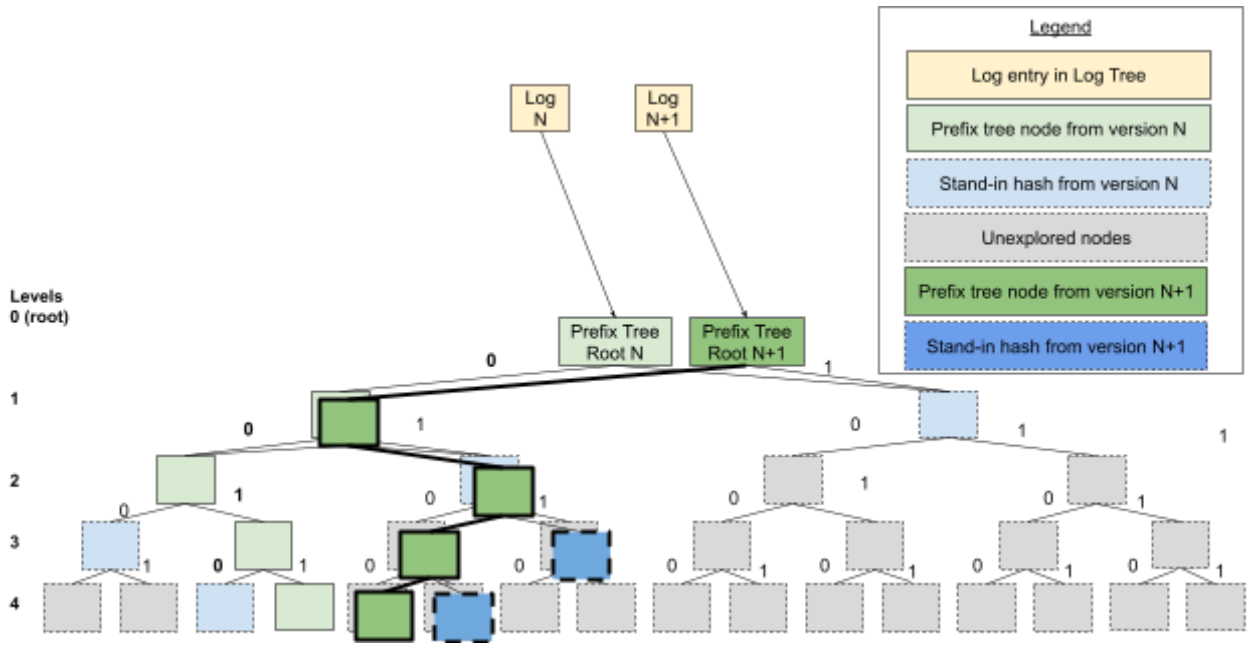
Every log entry has a prefix tree

Previously, we mentioned that there was one log tree but multiple prefix trees, and introduced the notion of a “previous prefix tree root hash” and a “new prefix tree root hash”. This all stems from the concept that every log entry has a new and different prefix tree. This is best demonstrated via a concrete example: consider a log entry at index N that changes the value of `commitmentIndex=0b0011` and a log entry at index N+1 that changes the value of `commitmentIndex=0b0100`.

The prefix tree for log N looks like this:



… and the prefix tree for log entry N+1 looks like this:



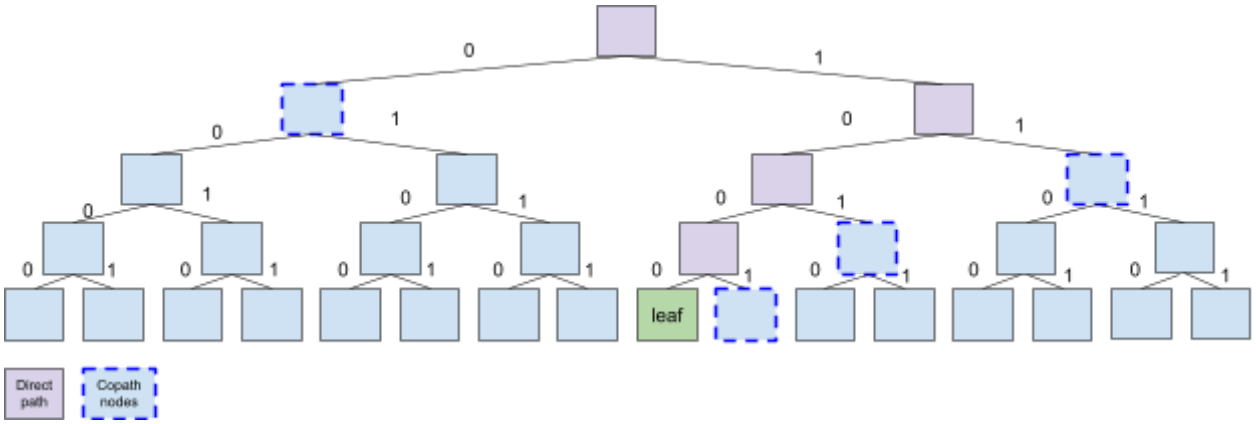
Log entry N+1…

- Overwrites the root hash and the node hash at level 1 from the previous log entry
- References the *same* stand-in hash value at level 1 as log N
- Creates new node hash values for levels 2, 3, and 4
- Generates its own stand-in hashes for levels 3 and 4, which were previously unexplored.

While there is a "new tree" for each log entry, that actually means there's a new prefix tree root hash, not that all nodes in the prefix tree have been duplicated. Since each log entry affects exactly one prefix tree leaf, a new log entry only changes the nodes in the direct path from that leaf to the new prefix tree root, generating stand-in hashes where needed. The log entry references existing nodes for all other parts of the tree.

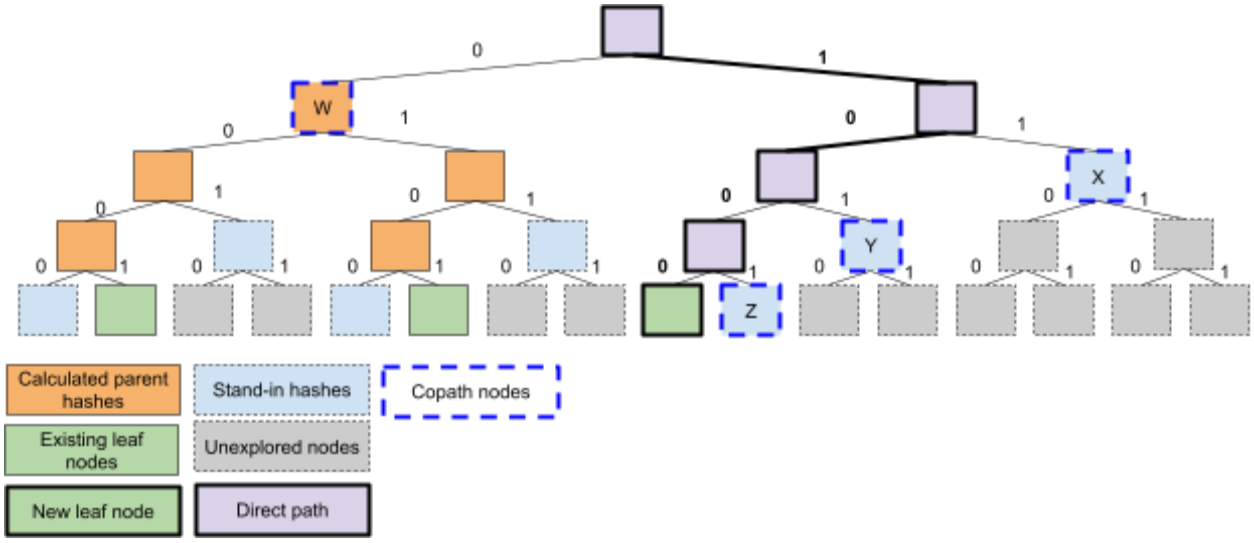
Copaths

Copaths are only relevant in the context of a prefix tree leaf. The copath of a prefix tree leaf is the sibling of each node in the leaf's direct path. In the diagram below, the blue dotted squares are the copath for the leaf node corresponding to `0b1000`:



The key transparency service provides a prefix tree copath in `SameKey` and `DifferentKey` auditor proofs, but only for "explored" nodes — nodes whose subtrees contain a real leaf. The value at index `i` in the copath list corresponds to the copath hash at level `i+1`. The values of the remaining nodes in a leaf's copath are stand-in hashes, which the auditor can recompute on demand.

In the example below, the `DifferentKey.copath` list for the new leaf node, `0b1000`, only contains the hash for node W because node W is the only node in the copath hwhose subtree contains real leaves. The auditor computes nodes X, Y, and Z's values as stand-in hashes.



Prefix tree leaves

The leaves of the prefix tree store three pieces of data that are concatenated together and then hashed to create a leaf hash (see the next section):

1. `firstLogTreePositionForThisSearchKey`: The log position in which the search key/commitment index was first inserted into the log tree
2. `updateCount`: A counter of how many times the search key's mapped value has been updated
3. `commitmentIndex`: A 256-bit pseudorandom value computed from a verifiable random function

The key transparency service uses the position data to narrow down the range of log entries that it must binary search to find the most recent log entry that updated the given search key. This is not relevant for the auditor, but it is relevant for clients who wish to search or monitor a given search key.

Parent and leaf hashes

A parent hash in the prefix tree is computed as:

```
None
parentHash = sha256(0x01 || leftChildHash || rightChildHash)
```

…where the child hashes can be a leaf hash, a stand-in hash, or another parent hash.

A leaf hash is computed as:

```
None
leafHash = sha256(0x00 || commitmentIndex || updateCount ||
firstLogTreePositionForThisSearchKey)
```

Stand-in hashes

As mentioned above, every log entry corresponds to the update of a search key, and has a corresponding prefix tree. For unpopulated nodes in that prefix tree, the key transparency service uses a *seed* to generate stand-in hashes. The seed is computed as:

```
None
seed =
NewAES(signal.prefixAesKey).Encrypt(firstLogTreePositionForThisSe
archKey)
```

…where `prefixAesKey` is a randomly generated 32-byte AES key known only by the key transparency service. The auditor does not have to recompute this; instead, the key transparency service provides the necessary `seed` value(s) in each `AuditorUpdate`.

In the prefix tree, the root node is level 0 and leaves are level 256. However, because the stand-in hash calculation for the prefix tree requires the level index to fit within a byte, and because we have no need to calculate stand-in hashes for the root node, we calculate the stand-in hash for a given `level` as:

```
None
standInHash = sha256(0x02 || seed || level - 1)
```

Log tree

The log tree is a left-balanced, binary Merkle tree, where the leaves correspond to log entries, and new log entries are added to the rightmost edge along with a parent node. The “log tree size” refers to the number of entries in the log or the number of leaves in the log tree (not the number of nodes).

A parent hash in the log tree is computed as:

```
None
parentHash = sha256(leftChildHash || rightChildHash)
```

…where the child hash is prepended with a `0x00` if it is a leaf hash and a `0x01` if it is a parent hash. So for example, the parent hash for 2 leaf hashes would look like:

```
None
sha256(0x00 || leftLeafHash || 0x00 || rightLeafHash)
```

A leaf hash is computed as:

```
None
leafHash = sha256(prefixTreeRootHash || commitment)
```

Commitments

A commitment, within the context of the *log tree*, is a byte-string computed for each log entry and constructed from the search key-value mapping updated in that log entry. A “search key” in this context might be something like “e164:+18005550123” with a corresponding value being the Signal-generated UUID associated with that phone number.

To calculate the commitment, the key transparency service uses the following cryptographic keys:

- an `openingKey`, a randomly generated 32-byte AES key known only by the key transparency server
- a `commitmentKey`, a random HMAC key that's publicly known. This key is actually hard-coded into the key transparency server.

The commitment is computed as:

- `commitmentOpening = NewAES(openingKey).Encrypt(logPosition)`
- `commitment = HMAC(publiclyKnown.commitmentKey, commitmentOpening || len(searchKey) || searchKey || len(value) || value)`

…where `logPosition` is the number of entries in the log when the new entry was inserted (including the given update). The key transparency service uses SHA256 as the hashing algorithm, so the resulting commitment is 256 bits (16 bytes) long. The commitment is

concatenated with the prefix tree root hash and then hashed to generate the leaf hash in the log tree (see this section).

Processing updates

For each update, the auditor's job is:

1. Decide whether or not to accept the update; the auditor should only accept the update if it looks like the new update is using the auditor's current prefix tree root hash as a starting point. The precise mechanics for this check vary by proof type. See Calculating previous prefix tree root hashes for more details.
    a. If the auditor chooses to reject the update, it should stop processing updates and no longer send auditor tree heads back to the main key transparency service. The key transparency service should continue to serve its last verified auditor tree head to clients; clients will eventually reject the last valid tree head after it expires at some point in the future.
2. If the new update is acceptable, calculate a new prefix tree root hash based on the changes from the update. The precise mechanics for this step depend on the proof/update type. See Calculating new prefix tree root hashes for more details..
3. With the newly-calculated prefix tree root hash and the given `commitment` from the new entry, calculate a leaf hash for the new log tree entry.
4. With the new log tree leaf hash and the auditor's previously-calculated log tree node hashes, calculate a new root hash for the log tree. See Maintaining the auditor's log tree for more details.
5. With the updated log tree, return a signature for the current state of the transparency log to the key transparency service; note that this doesn't have to happen for every individual update, especially under high load. See Sending signatures to the key transparency service for more details.

The appendix walks through concrete examples of these steps.

AuditorProofs

The key transparency service can send two types of updates ("real" or "fake") and three types of proofs to a third-party auditor.

The types of `AuditorProof`s are:

- `newTree` - This will only happen once for the very first update. There's no previous prefix tree hash, so the auditor just accepts the update unconditionally. "New tree" proofs may only be applied to real updates.

- `differentKey` - A `differentKey` proof corresponds to a log entry that inserted a new commitment index into the prefix tree. It can be for either a real or fake update, and follows another pre-existing entry's path in the prefix tree until it reaches "unexplored" territory, or a node whose subtree has no leaf nodes. A "different key" proof provides a `copath` up to the point of divergence and an `old_seed` value that the auditor can use to calculate the previous prefix tree root hash.
- `sameKey` - A `sameKey` proof corresponds to a log entry that updated an existing leaf in the prefix tree. By extension, it is only allowed for real updates. The proof includes a `copath` (up to the point of entering "unexplored" territory) and both the commitment index and update count for the *existing* leaf node. Auditors can use the copath, stand-in values, and the real leaf node hash to verify the previous prefix tree root hash.

The details of calculating previous prefix tree root hashes vary by proof type, and the details of calculating a new prefix tree root hash vary by both update type and proof type.

Calculating previous prefix tree root hashes

Here is the relevant code snippet of this section in the Java reference implementation.

`newTree` proofs

```
None
message NewTree {}
```

In a `newTree` proof, no previous prefix tree root hash exists, and so the auditor simply accepts the associated update (as long as the auditor agrees that the tree is, indeed, empty).

`differentKey` proofs

```
None
message DifferentKey {
      /**
      * The list of sibling hashes in the prefix tree, up to and including
      * the sibling of the stand-in hash value.
      * This list is returned in root to leaf order.
      */
      repeated bytes copath = 1;
      /**
      * Used to calculate the stand-in hash value where the search ended.
      */
```

```
        bytes old_seed = 2;
    }
```

In a `differentKey` proof, auditors calculate the previous prefix tree root by following the given `copath` to its end, then calculating the stand-in hash of the last copath entry's sibling using the given `old_seed`. From there the auditor recursively hashes its way back up to the root to calculate the root hash.

`sameKey` proofs

```
None
message SameKey {
        /**
        * Hashes of the siblings of nodes in the direct path to the given leaf.
        * This list only contains hashes that are in the direct path of another
        * leaf (the "explored part" of the prefix tree).
        * Use AuditorUpdate.seed to calculate stand-in hashes in the
        * "unexplored" part of the prefix tree.
        */
        repeated bytes copath = 1;
        /**
        * The number of times that the value associated with the search key has
        * been updated.
        */
        uint32 counter = 2;
        /**
        * The position of the first instance of the search key in the log tree.
        */
        uint64 position = 3;
    }
```

In a `sameKey` proof, auditors must calculate the full hash from the previous leaf node to the root of the tree to verify the previous prefix tree root hash. The proof includes a `copath` that covers some (likely) or all (highly unlikely) of the route to the leaf and a `counter` and `position` value for the existing leaf node. Critically, the `seed` in the update is the *same* seed for both the new and old entries because updating an existing search key doesn't overwrite any existing stand-in nodes. Auditors calculate previous prefix tree root hash from bottom to top, using the `counter`

and `position` values to calculate the previous leaf hash, `seed` to calculate stand-in copath hashes, and `copath` entries wherever possible.

Calculating new prefix tree root hashes

Here is the relevant code snippet of this section in the Java reference implementation.

`newTree` proofs

Auditors calculate the new prefix tree root hash by calculating the leaf hash for the update (using its own count of log tree entries for the log tree position and assuming an update count of 0), then recursively hashing with stand-in hashes derived from the `seed` in the `AuditorUpdate` up to the root of the tree.

The key transparency service requires at least one real update before applying fake updates, so the auditor should reject any `newTree` proofs provided with a fake update.

`differentKey` proofs

For real updates, auditors calculate the new prefix tree root as they would for a real update in the `newTree` case, but use the given `copath` entries instead of stand-in hashes when provided.

For fake updates, auditors follow the `copath` to its end, then calculate a stand-in hash on the new entry's *direct* path (this is notable because we most commonly use stand-in hashes on copaths) as a sibling to the last copath entry. From there the usual recursive hashing dance takes place until we hash our way back to the root of the tree.

`sameKey` proofs

"Same key" proofs may only be used with real updates. To calculate a new prefix tree root, auditors first calculate a new leaf hash using the same `position` value given in the proof and `counter + 1` for the update count. From there, auditors calculate hashes along the whole path using stand-in hashes and `copath` entries as needed.

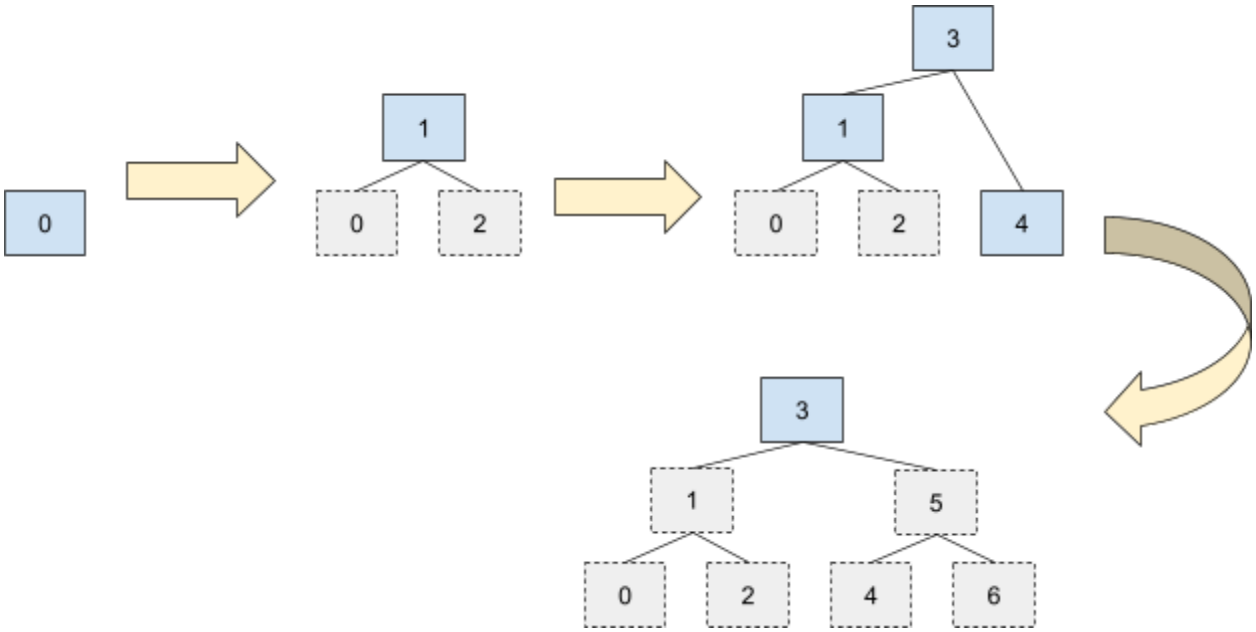
Maintaining the auditor's log tree

The auditor needs to maintain enough of a view of the log tree to be able to calculate new log tree root hashes for each new update. It *could* retain every leaf hash it has encountered in a naive implementation. Unlike the main key transparency service, which will need to produce log tree hashes at various points in history, auditors will only ever need to produce hashes from the

current state of the tree. That means the auditor can “garbage-collect” nodes as its tree grows and store only a small subset of all of the nodes it has seen.

In particular, the auditor only needs to store hashes for roots of complete subtrees. Once a subtree is “complete,” no nodes will ever be added to it, and its hash will never change. Once a node becomes the root of a complete subtree, all the hashes of its descendants can be discarded.

As an example, consider the following sequence of updates:



1. First, a single node (0) is added to the empty tree. The node’s leaf hash and the log tree’s root hash are one and the same.
2. A second node (2) is added to the tree. The root hash (1) is now the combined hash of nodes 0 and 2. Node 1 is the root of a complete subtree, and so the hashes for nodes 0 and 2 can be discarded.
3. Node 4 is added to the tree. The root hash (3) can be calculated using just the hash from node 1 (which already includes 0 and 2) and 4.
4. Node 6 is added to the tree. Node 3 is now the root of a complete subtree, and once its hash has been calculated, all the stored hashes from its child nodes can be discarded. The auditor is again retaining only a single hash value even though the tree contains seven nodes.

Signatures

When the auditor has verified an update, it may send an `AuditorTreeHead` back to the main key transparency service's `SetAuditorHead` method. An `AuditorTreeHead` has the following fields:

```
None
message AuditorTreeHead {
  uint64 tree_size = 1;
  int64 timestamp = 2;
  bytes signature = 3;
}
```

The `signature` is an Ed25519 signature over a `TreeHeadTBS` structure (the TBS stands for "to be signed"). The `TreeHeadTBS` structure gets serialized to a byte array as:

```
None
{0x00, 0x00} (cipher suite identifier)
|| 0x03 (third-party auditing mode)
|| [2 bytes of key transparency service's public signing key length]
|| [key transparency service's public signing key bytes]
|| [2 bytes of VRF public key length]
|| [VRF public key bytes]
|| [2 bytes of auditor public key length]
|| [auditor public key]
|| [8-byte log tree size]
|| [8-byte timestamp]
|| [32-byte log tree root hash]
```

…and so the `AuditorTreeHead` is effectively built as:

- `tree_size`: the number of leaf nodes in the auditor's view of the log tree
- `timestamp`: the current time, in milliseconds since the epoch
- `signature`: `ed25519Sign(privateSigningKey, buildTreeHeadBtsArray(tree_size, timestamp, root_hash))`

Sending signatures to the key transparency service

While the auditor does not need to send a signed tree head back to the key transparency service after every update, it *does* need to meet certain requirements or else the auditor's signed tree head will be rejected:

- The auditor's tree head must be no more than 10,000,000 updates behind the key transparency's tree head.
- The auditor's tree head timestamp must be no more than 7 days behind the current time.
- The auditor's tree head can never "go back" in time: it must always send tree heads with a `tree_size` and `timestamp` greater than the ones sent previously.

Given these requirements, the auditor should send signed tree heads to the key transparency service on both a time-based interval and a number-of-updates-processed interval.

Handling errors

Possible error responses fall into 3 broad categories:

- `InvalidArgument`: This can happen if the auditor tree head doesn't meet the requirements in the "Sending signatures to the key transparency service" portion of the spec. In general, the recommended action is to continue processing updates and attempting to send auditor tree heads.
- `FailedPrecondition`: This happens if the key transparency service cannot verify the auditor's signature for whatever reason (likely wrong auditor public key or data marshaling). In steady state, we should never see this error.
- `Unavailable`: We cannot have more than one key transparency replica running at a time, so brief (<2 min) downtime is expected whenever we deploy new changes. The recommended action here is to wait a minute or two and attempt to query the key transparency service again.

Test vectors

Here is a test vector protobuf and some generated data that can be plugged in to verify that the auditor works as expected.

Appendix

In this section, we'll work through two examples of processing a real update and then a fake one.

To make things easier, we'll work with a 4-bit commitment index instead of a 256-bit one.

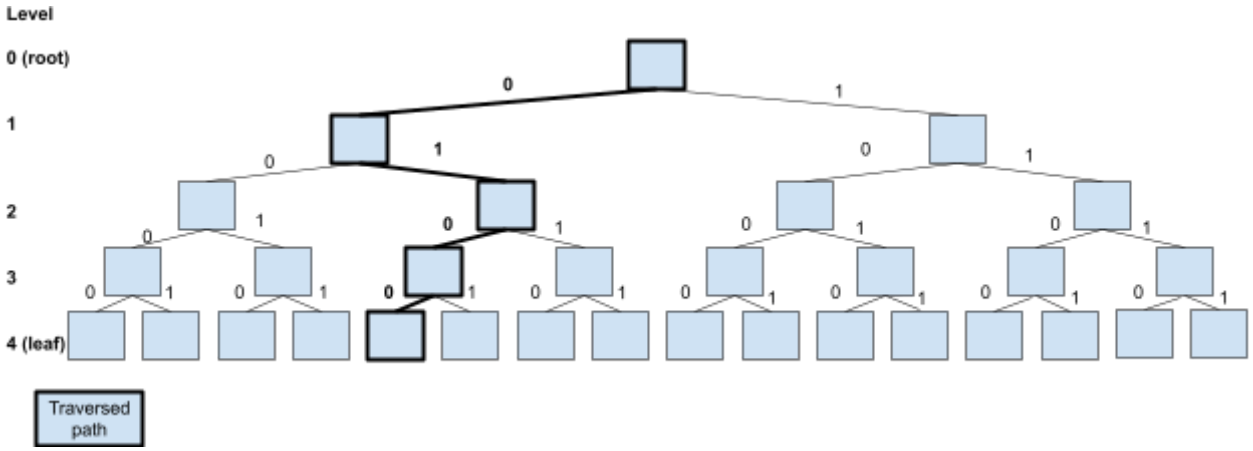
Processing a real update

Let’s assume that the auditor processed 5 updates already, and the next update looks like this:

```
None
message AuditorUpdate {
  bool real = true
  bytes index = 0b0100;
  bytes seed = 0xe7183b94e014d8d64ffd1392e8a1828e;
  bytes commitment =
0x2f4cc81752926443cd67662a1d69e8dcd733b217b3184ee3f87ea11e04d71fb4;
  AuditorProof proof =
       // Let’s assume we’re updating an existing search key
       message SameKey {
              repeated bytes copath = [
              “24fff154fc91194731c251d6767ffadd5460b8a3eee875b2a3080c2a69845058”
              ,
              “be95b336e6a14b795ddbb2e99fc80f730b194e9fdb7aadb9506180fa05384bfb”
              ];
              // Assume that this is the first time the key transparency
              // service is updating this search key
              uint32 counter = 0;
              // The first time that the key transparency service inserted this
              // search key was at position 2
              uint64 position = 2;
  };
}
```

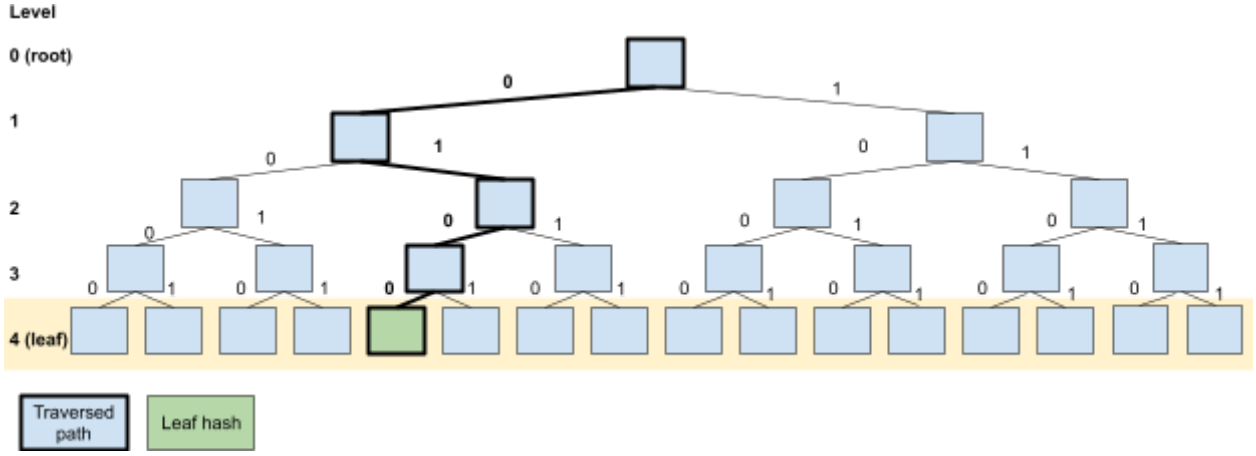
Prefix tree path

The following diagram illustrates the prefix tree path that will be traversed for the index `0b0100`:



Part 1: Calculate the previous prefix tree root hash

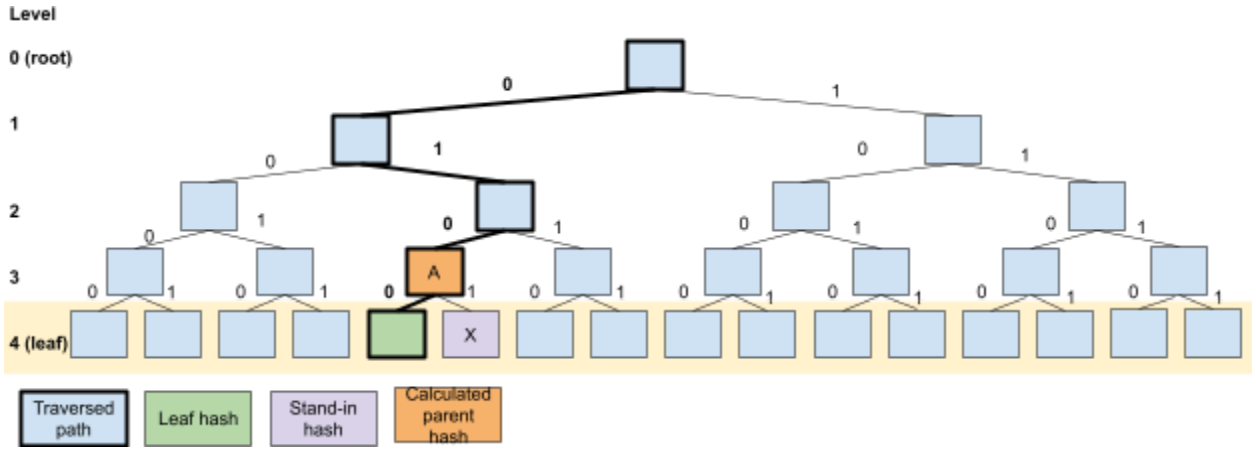
1. Calculate the [prefix tree leaf hash](#).



None

```
prefixLeafHash = Sha256(0x00 || AuditorUpdate.index || SameKey.counter||
SameKey.position)
 = Sha256(0x00 || 0b1000 || 0 || 2)
```

2. In our specific example, the leaf `level` of the prefix tree is 4. We only have 2 values in the provided copath, so we have to calculate a stand-in hash for node X to get the hash of node A.



None

```
// We subtract one from the level because we need it to fit within a
// byte and will never calculate a stand-in hash for level 0 (the root node).
```

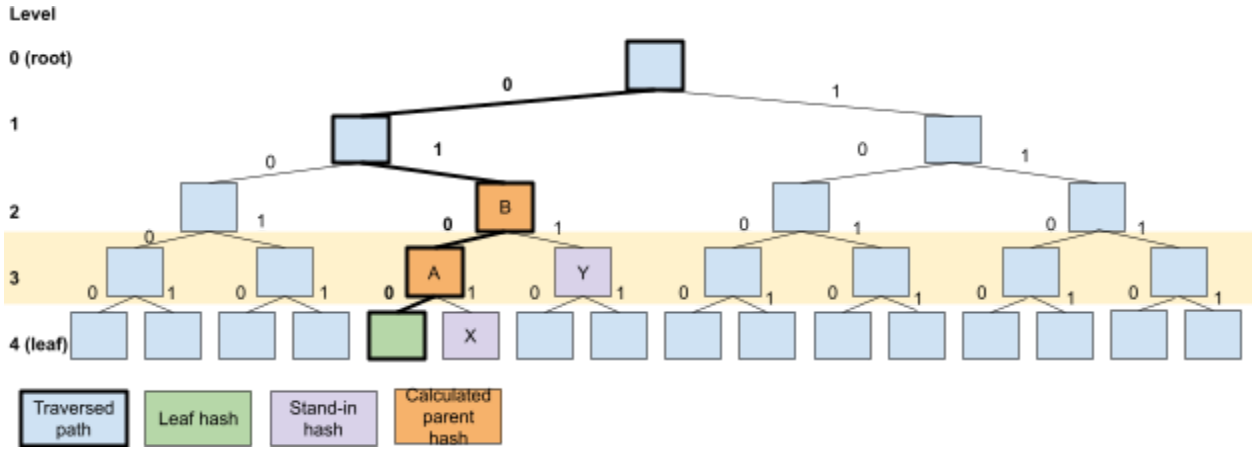
```
standInHashForNodeX = Sha256(0x02 || AuditorUpdate.seed || level - 1) =
Sha256(0x02 || 0xe7183b94e014d8d64ffd1392e8a1828e || 4 - 1)
```

And then hash that with `prefixLeafHash` to get the hash for node A:

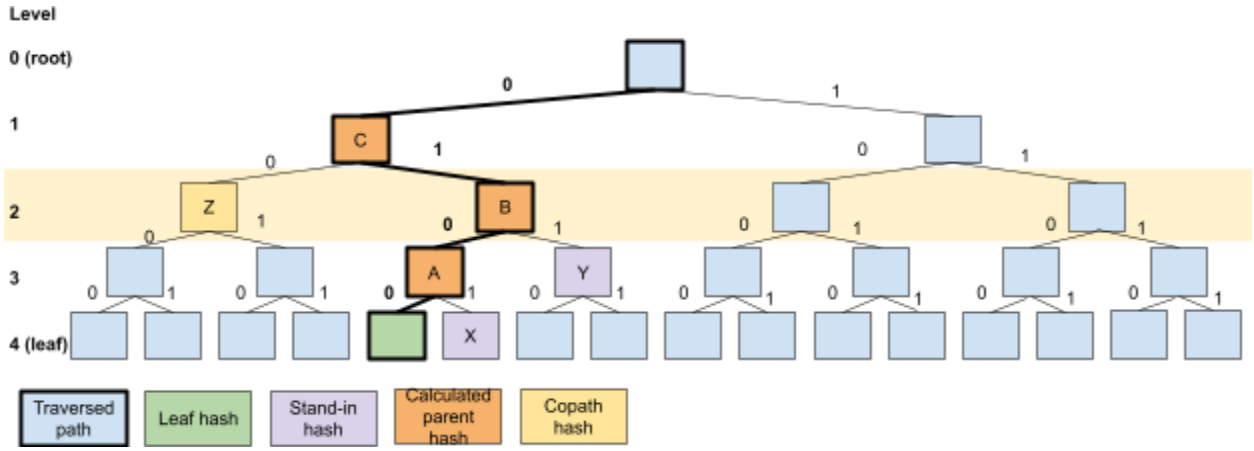
```
None
nodeA = Sha256(0x01 || leftChildHash || rightChildHash) =
Sha256(0x01 || prefixLeafHash || standInHashForLeafSibling)
```

Note that the prefix leaf hash is the *left* child because the bit corresponding to level 4 is 0.

3. We decrement `level` to 3, and calculate another stand-in hash for node Y, then compute node B.



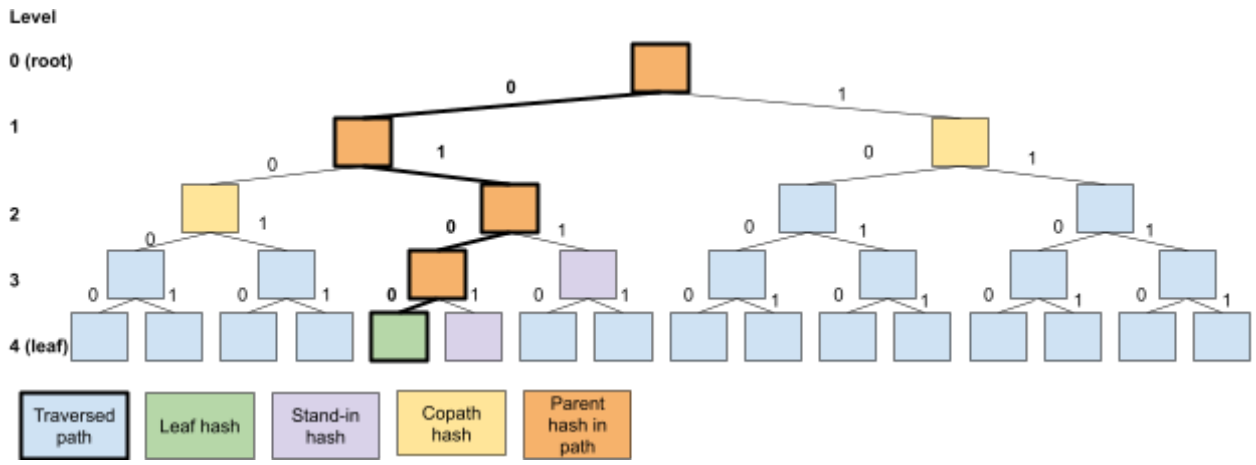
4. Decrement `level` to 2. The provided copath has 2 values, so we now grab the value at index `level - 1` and use that for node Z:



```
None
nodeZ = AuditorUpdate.SameKey.copath[level-1] = AuditorUpdate.SameKey.copath[2]
= cfade7e194dc82245a37cc8c42d338529187456ecd211b3710b2c454a7345a96

nodeC = Sha256(0x01 || nodeA || nodeB)
```

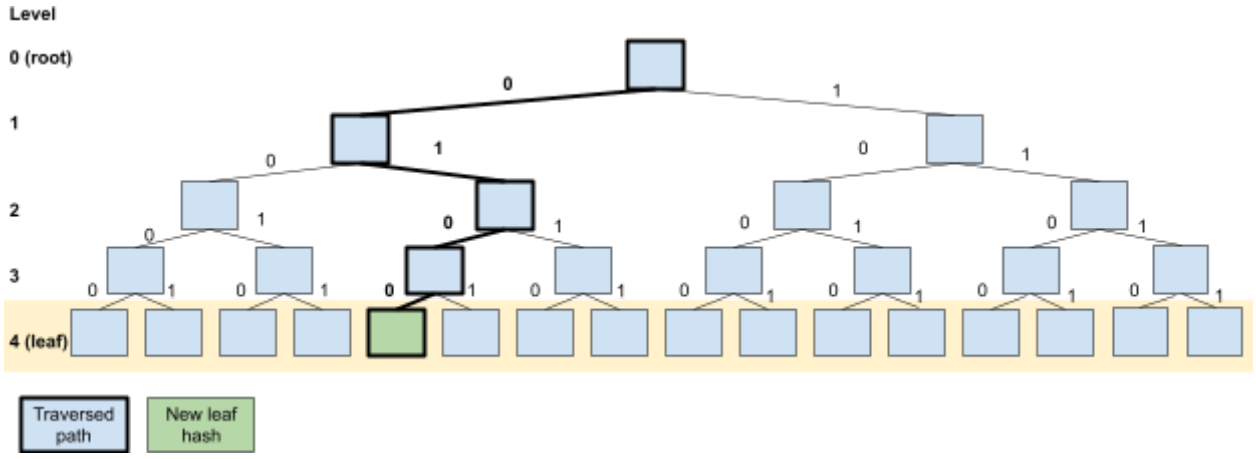
5. And so on, until we reach the root node. In total, the following nodes will be used or computed:



The auditor will then compare the computed root node hash with the value it has stored, making sure that the two match.

Part 2: Calculate the new prefix tree root hash

1. Calculate the new prefix tree leaf hash



```
None
newPrefixLeafHash = Sha256(0x00 || AuditorUpdate.index || SameKey.counter + 1
|| SameKey.position)
 = Sha256(0x00 || 0b1000 || 1 || 4)
```

2. Do the same hashing steps as in part 1, calculating stand-in hashes and using copath values as necessary. Use the same `AuditorUpdate.seed` value.
3. The auditor can now discard its previous prefix tree root hash in favor of the newly calculated one.

Part 3: Calculate the log tree leaf hash

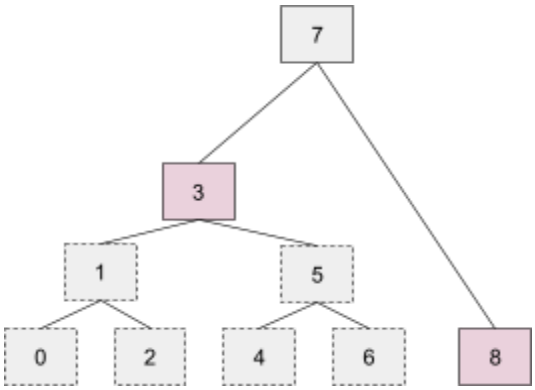
Using the newly calculated prefix tree root hash and the `commitment` in the `AuditorUpdate`, the auditor can calculate the log tree leaf hash as

```
None
logTreeLeafHash = sha256(newPrefixTreeRootHash || AuditorUpdate.commitment)
```

Part 4: Calculate the new log tree nodes to store

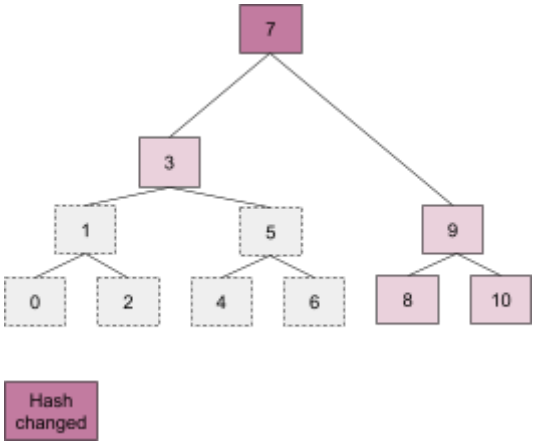
We started out with the assumption that the auditor had already processed 5 updates. Recall that the log tree is a left-balanced Merkle tree where new leaves are always added to the rightmost edge along with a parent node and once a node is the root of a full subtree, its hash will remain the same.

That implies that the auditor's stored log tree nodes would look like this:

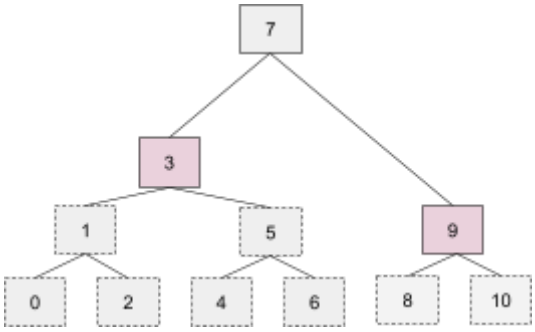


where nodes 0, 2, 4, 6, and 8 correspond to the first 5 updates, and nodes 3 and 8 are stored. Since the root hash is always changing, the auditor recomputes it on demand instead of storing it.

With the addition of the sixth update as node 10 and the creation of parent node 9, the log tree would look like this:



And because node 9 forms a full subtree, the auditor can discard the hashes for nodes 8 and 10 to reach a final state of storing nodes 3 and 9.



Processing a fake update

Let’s say that our next update is a fake one. Fake updates do *not* create leaf nodes in the prefix tree but they do generate a new stand-in hash at the same level where their copath terminates.

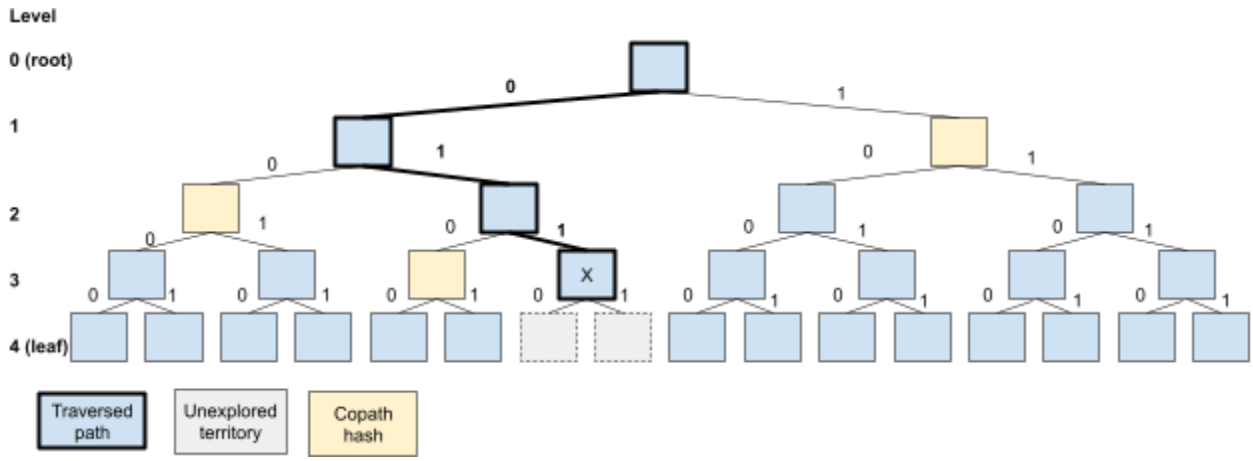
```
None
message AuditorUpdate {
  bool real = false
  bytes index = 0b0110;
  bytes seed = 0x0365c865d3bfd0dd5573e6e028029fff;
  bytes commitment =
0x81db5ded09ea509bed9bfa295cc079ddd285bc580a7e5de9f26a167176df1a22;
  AuditorProof proof =
       message DifferentKey {
```

```
            repeated bytes copath = [
            “24fff154fc91194731c251d6767ffadd5460b8a3eee875b2a3080c2a69845058”
            ,
            “be95b336e6a14b795ddbb2e99fc80f730b194e9fdb7aadb9506180fa05384bfb”
            ,
            "af227f9656b3f833fbd5c30cce5049fc6c52ddba9a6d66c4ca02fbecb4a6b0ac"
            ];
            bytes old_seed = 0x81b7e5aaf89b1e46a4620bc2d0e16c2d;
  };
}
```

A `DifferentKey` proof for a fake update means that the index follows another pre-existing entry’s path in the prefix tree until it reaches “unexplored” territory, a subtree where there are no leaf nodes. In our case, this fake update index shares the first two bits — and therefore the first two copath values — as the previous update for `0b0100`. The last copath value is generated by that previous update.

Prefix tree path

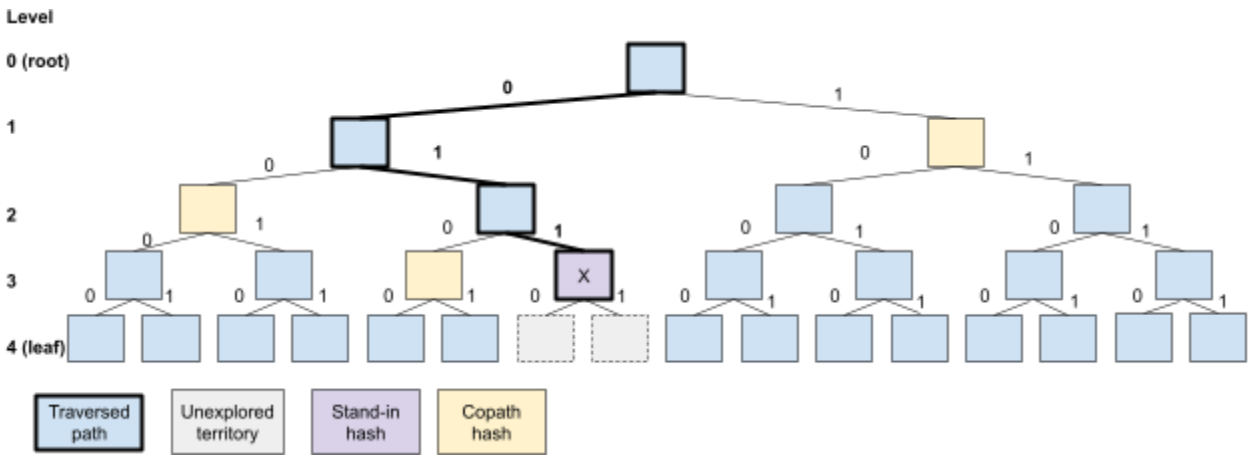
The following diagram illustrates the prefix tree path we will traverse for the index `0b0110`:



Since the copath is three hashes long, we traverse the prefix tree for 3 levels before we “stop” at node X, which has no real leaves in its subtree.

Part 1: Calculate the previous prefix tree root hash

1. Calculate the stand-in hash value for node X, the sibling of the last copath entry, using `DifferentKey.old_seed`.



None

```
nodeY = Sha256(0x02 || DifferentKey.old_seed || level - 1) = Sha256(0x02 ||
0x81b7e5aaf89b1e46a4620bc2d0e16c2d || 2 - 1)
```

2. Hash our way back up to the root using the values in `DifferentKey.copath`, and compare the computed root hash with what the auditor has stored.

Part 2: Calculate the new prefix tree root hash

1. Calculate the stand-in hash for node X, this time using `AuditorUpdate.seed`. While a fake index does not create a prefix tree leaf, it *does* generate a new seed value which in turn creates a new stand-in hash.
2. Hash our way back up to the root.

Part 3: Calculate the log tree leaf hash

Same deal as Part 3 for the real update above.

Part 4: Calculate the new log tree nodes to store

Same deal as Part 4 for the real update above.

gRPC Specification

The full gRPC specification can be found in the signalapp/key-transparency-auditor repo. If you do not have access to that repo, please let a Signal point of contact know.

Other references

- Java implementation
- Draft Key Transparency RFC