KEYTRANS Working Group
Internet-Draft
Intended status: Standards Track
Expires: 8 January 2026

B. McMillion

F. Linker

7 July 2025

Key Transparency Protocol
draft-ietf-keytrans-protocol-02

Abstract

While there are several established protocols for end-to-end encryption, relatively little attention has been given to securely distributing the end-user public keys for such encryption. As a result, these protocols are often still vulnerable to eavesdropping by active attackers. Key Transparency is a protocol for distributing sensitive cryptographic information, such as public keys, in a way that reliably either prevents interference or detects that it occurred in a timely manner.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://ietf-wg-keytrans.github.io/draft-protocol/draft-ietf-keytrans-protocol.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-keytrans-protocol/.

Discussion of this document takes place on the Key Transparency Working Group mailing list (mailto:keytrans@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/keytrans/. Subscribe at https://www.ietf.org/mailman/listinfo/keytrans/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-keytrans/draft-protocol.

Status of This Memo

This Internet-Draft will expire on 8 January 2026.

Copyright Notice

Table of Contents

1. Introduction

End-to-end encrypted communication services rely on the secure exchange of public keys to ensure that messages remain confidential. It is typically assumed that service providers correctly manage the public keys associated with each user's account. However, this is not always true. A service provider that is compromised or malicious can change the public keys associated with a user's account without their knowledge, thereby allowing the provider to eavesdrop on and impersonate that user.

This document describes a protocol that enables a group of users to ensure that they all have the same view of the public keys associated with each other's accounts. Ensuring a consistent view allows users to detect when unauthorized public keys have been associated with their account, indicating a potential compromise.

More detailed information about the protocol participants and the ways the protocol can be deployed can be found in [ARCH].

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the TLS presentation language [RFC8446] to describe the structure of protocol messages, but does not require the use of a specific transport protocol. As such, implementations do not necessarily need to transmit messages according to the TLS format and can choose whichever encoding method best suits their application. However, cryptographic computations MUST be done with the TLS presentation language format to ensure the protocol's security properties are maintained.

3. Tree Construction

A Transparency Log is a verifiable data structure that maps a _label-version pair_ to some unstructured data such as a cryptographic public key. Labels correspond to user identifiers, and a new version of a label is created each time the label's associated value changes.

KT uses a _prefix tree_ to store a mapping from each label-version pair to a commitment to the label's value at that version. Every time the prefix tree changes, its new root hash and the current timestamp are stored in a _log tree_. The benefit of the prefix tree is that it is easily searchable and the benefit of the log tree is that it can easily be verified to be append-only. The data structure powering KT combines a log tree and a prefix tree, and is called the _combined tree_.

This section describes the operation of prefix trees, log trees, and the combined tree structure, at a high level. More precise algorithms for computing the intermediate and root values of the trees are given in Section 9.

3.1. Terminology

Trees consist of _nodes_, which have a byte string as their _value_. A node is either a _leaf_ if it has no children, or a _parent_ if it has either a _left child_ or a _right child_. A node is the _root_ of a tree if it has no parents, and an _intermediate_ if it has both children and parents. Nodes are _siblings_ if they share the same parent.

The _descendants_ of a node are that node, its children, and the descendants of its children. A _subtree_ of a tree is the tree given by the descendants of a particular node, called the _head_ of the subtree.

The _direct path_ of a root node is the empty list, and of any other node is the concatenation of that node’s parent along with the parent’s direct path. The _copath_ of a node is the node’s sibling concatenated with the list of siblings of all the nodes in its direct path, excluding the root.

The _size_ of a tree or subtree is defined as the number of leaf nodes it contains.

3.2. Log Tree

Log trees store information in the chronological order that it was added, and are constructed as _left-balanced_ binary trees.

A binary tree is _balanced_ if its size is a power of two and for any parent node in the tree, its left and right subtrees have the same size. A binary tree is _left-balanced_ if for every parent, either the parent is balanced, or the left subtree of that parent is the largest balanced subtree that could be constructed from the leaves present in the parent’s own subtree. Given a list of n items, there is a unique left-balanced binary tree structure with these elements as leaves. Note also that every parent always has both a left and right child.

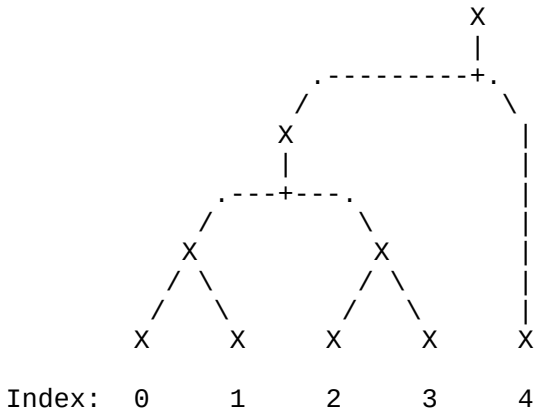

Figure 1: A log tree containing five leaves.

Log trees initially consist of a single leaf node. New leaves are added to the right-most edge of the tree along with a single parent node to construct the left-balanced binary tree with n+1 leaves.

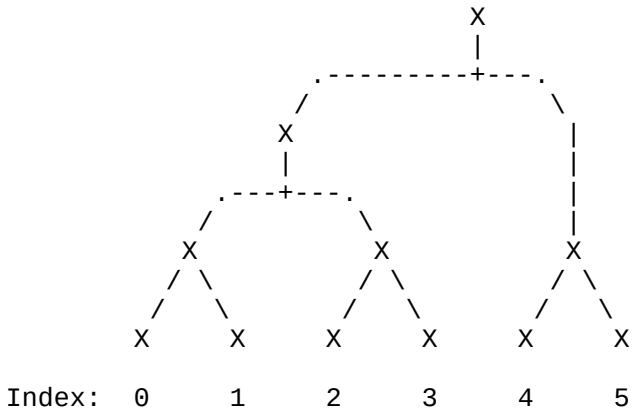

Figure 2: Example of inserting a new leaf with index 5 into the previously depicted log tree. Observe that only the nodes on the path from the new root to the new leaf change.

Leaves can have arbitrary data as their value, and are frequently referred to as "log entries" later in the document. The value of a parent node is always the hash of the combined values of its left and right children.

Log trees are powerful in that they can provide both _inclusion proofs_, which demonstrate that a leaf is included in a log, and _consistency proofs_, which demonstrate that a new version of a log is an extension of a previous version.

Inclusion and consistency proofs in KT differ from similar protocols in that proofs only ever contain the values of nodes that are the head of a balanced subtree. Whenever the value of the head of a non-balanced subtree is needed by a verifier, the prover breaks down the non-balanced subtree into the smallest-possible number of balanced subtrees and provides the value of the head of each. This allows verifiers to cache a larger number of intermediate values than would otherwise be possible, reducing the size of subsequent responses.

As a result, an inclusion proof for a leaf is given by providing the copath values of the leaf with any non-balanced subtrees broken down as mentioned. The proof is verified by hashing the leaf value together with the copath values, re-computing the head values of non-balanced subtrees where needed, and checking that the result equals the root value of the log.

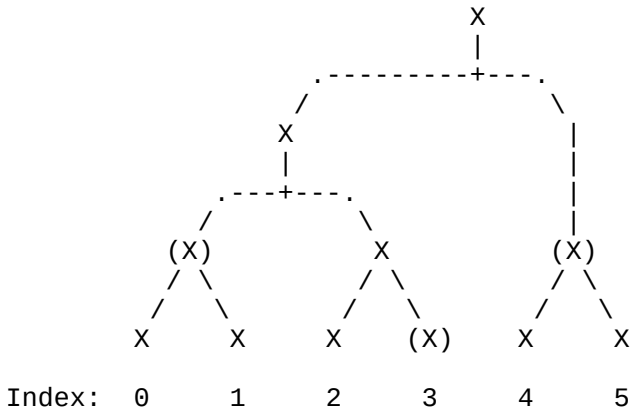

Figure 3: Illustration of an inclusion proof. To verify that leaf 2 is included in the tree, the prover provides the verifier with the values of leaf 2's copath. These nodes are marked by (X).

When requesting a consistency proof, verifiers are expected to have retained the head values of the largest-possible balanced subtrees (these will later be defined as the "full subtrees") of the previous version of the log. A consistency proof then consists of the minimum set of node values that are necessary to compute the root value of the new version of the log from the values that the verifier retained.

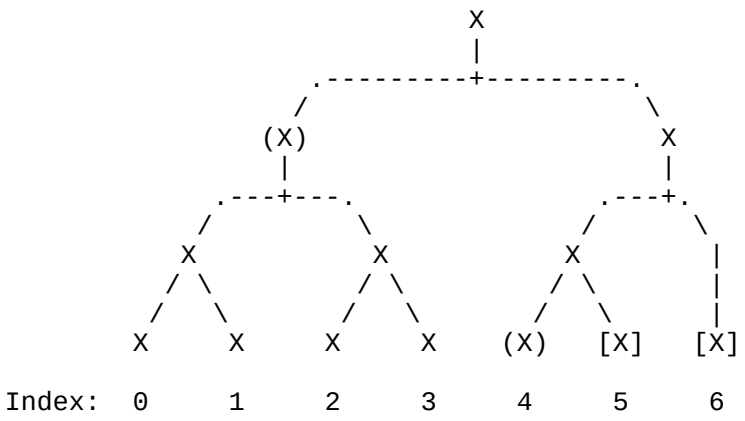

Figure 4: Illustration of a consistency proof between a log with 4 and with 6 leaves respectively. The verifier is expected to already have the values (X), so the prover provides the verifier with the values of the nodes marked [X]. By combining these, the verifier is able to compute the new root value of the log.

3.3. Prefix Tree

Prefix trees store a mapping between search keys and their corresponding values, with the ability to efficiently prove that a search key's value was looked up correctly.

Each leaf node in a prefix tree represents a specific mapping from search key to value, while each parent node represents some prefix which all search keys in the subtree headed by that node have in common. The subtree headed by a parent's left child contains all search keys that share its prefix followed by an additional 0 bit, while the subtree headed by a parent's right child contains all search keys that share its prefix followed by an additional 1 bit.

The root node, in particular, represents the empty string as a prefix. The root's left child contains all search keys that begin with a 0 bit, while the right child contains all search keys that begin with a 1 bit.

A prefix tree can be searched by starting at the root node and moving to the left child if the first bit of a search key is 0, or the right child if the first bit is 1. This is then repeated for the second bit, third bit, and so on until the search either terminates at a leaf node (which may or may not be for the desired value), or a parent node that lacks the desired child.

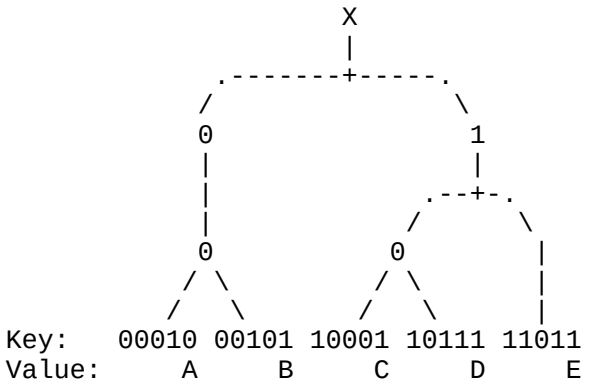

Figure 5: A prefix tree containing five entries.

New key-value pairs are added to the tree by searching it according to the same process. If the search terminates at a parent without a left or right child, a new leaf is simply added as the parent's missing child. If the search terminates at a leaf for the wrong search key, one or more intermediate nodes are added until the new leaf and the existing leaf would no longer reside in the same place. That is, until we reach the first bit that differs between the new search key and the existing search key.

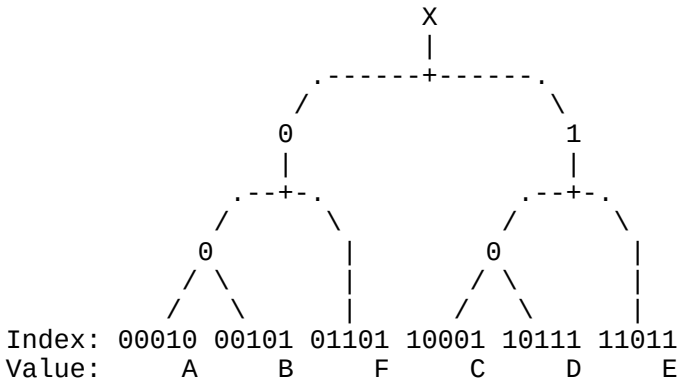

Figure 6: The previous prefix tree after adding the key-value pair: 01101 -> F.

The value of a leaf node is the encoded key-value pair, while the value of a parent node is the hash of the combined values of its left and right children (or a stand-in value when one of the children doesn't exist).

A proof of membership is given by providing the leaf value, along with the value of each copath entry along the search path. A proof of non-membership is given by providing an abridged proof of membership that follows the path for the intended search key, but ends either at a stand-in node or a leaf for a different search key. In either case, the proof is verified by hashing together the leaf with the copath hash values and checking that the result equals the root hash value of the tree.

3.4. Combined Tree

Log trees are desirable because they can provide efficient consistency proofs to convince verifiers that nothing has been removed from a log that was present in a previous version. However, log trees can't be efficiently searched without downloading the entire log. Prefix trees are efficient to search and can provide inclusion proofs to convince verifiers that the returned search results are correct. However, it's not possible to efficiently prove that a new version of a prefix tree contains the same data as a previous version with only new values added.

In the combined tree structure, based on [Merkle2], each label-version pair stored by a Transparency Log corresponds to a search key in a prefix tree. This prefix tree maps the label-version pair's search key to a commitment to the label's contents at that version. To allow users to track changes to the prefix tree, a log tree contains a record of each version of the prefix tree along with the timestamp of when it was published. With some caveats, this combined structure supports both efficient consistency proofs and can be efficiently searched.

Note that, although the Transparency Log maintains a single logical prefix tree, each modification of the prefix tree results in a new root value which is then stored in the log tree. As part of the protocol, the Transparency Log is often required to perform lookups in different versions of the prefix tree. Different versions of the prefix tree are identified by the log entry where their root value was stored.

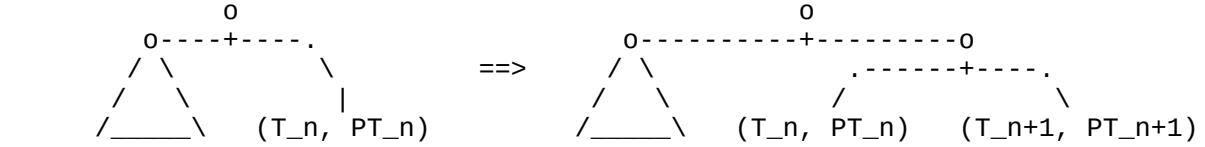

Figure 7: An example evolution of the combined tree structure. Every new log entry added contains the timestamp T_n of when it was created and the new prefix tree root hash PT_n.

4. Updating Views of the Tree

As users interact with the Transparency Log over time, they will see many different root hashes as the contents of the log changes. It's necessary for users to guarantee that the root hashes they observe are consistent with respect to two important properties:

* If root hash B is shown after root hash A, then root hash B contains all the same log entries as A with any new log entries added to the rightmost edge of A.

* All log entries in the range starting from the rightmost log entry of A and ending at the rightmost log entry of B, have monotonically increasing timestamps.

The first property is necessary to ensure that the Transparency Log never removes a log entry after showing it to a user, as this would allow the Transparency Log to remove evidence of its own misbehavior. The second property ensures that all users have a consistent view of when each portion of the tree was created. As will be discussed in later sections, users rely on log entry timestamps to decide whether to continue monitoring certain labels and which portions of the tree to skip when searching. Disagreement on when portions of the tree were created can cause users to disagree on the value of a label-version pair, introducing the same security issues as a fork.

Proving the first property, that the log tree is append-only, can be done by providing a consistency proof from the log tree. Proving the second property, that newly added log entries have monotonically increasing timestamps, requires establishing some additional structure on the log's contents.

4.1. Implicit Binary Search Tree

Intuitively, the leaves of the log tree can be considered a flat array representation of a binary tree. This structure is similar to the log tree, but distinguished by the fact that not all parent nodes have two children.

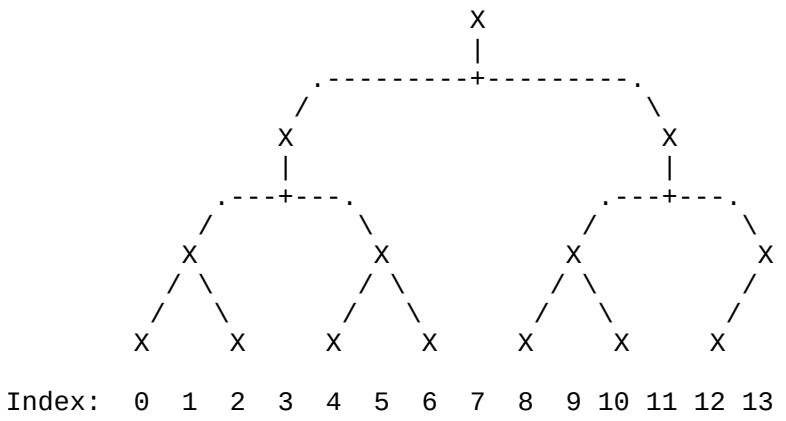

Figure 8: A binary tree constructed from 14 entries in a log

The implicit binary search tree containing n entries can be defined inductively. The index of the root log entry in the implicit binary search tree is the greatest power of two, minus one, that is less than the size of the implicit binary search tree. That is i_root = 2^floor(log2(n)) - 1. The left subtree is the implicit binary search tree of size i_root, i.e., the implicit binary search tree for all elements with a smaller index than the root. The right subtree is the implicit binary search tree of size n-i_root-1, but offset with i_root+1. Initially, these will be all indices larger than the root.

Users ensure that log entry timestamps are monotonic by enforcing that the structure of this search tree holds. That is, users check that any timestamp they observe in the root's left subtree is less than or equal to the root's timestamp, and that any timestamp they observe in the root's right subtree is greater than or equal to the root's timestamp, and so on recursively. Following this tree structure ensures that users can detect misbehavior quickly while minimizing the number of log entries that need to be checked.

As an example, consider a log with 50 entries. Instead of having the root be the typical "middle" entry of 50/2 = 25, the root would be entry 31. As new log entries are added to the tree's right edge, all users that interact with the Transparency Log will require log entries to the right of entry 31 to have timestamps that are greater than or equal to that of entry 31, regardless of how much or how little the tree grows.

Because we are often looking at the rightmost log entry, it is frequently useful to refer to the *frontier* of the log. The frontier consists of the root log entry, followed by the entries produced by repeatedly moving right until reaching the last entry of the log. Using the same example of a log with 50 entries, the frontier would be entries: 31, 47, 49.

Example code for efficiently navigating the implicit binary search tree is provided in Appendix A.

4.2. Algorithm

Users retain the following information about the last tree head they've observed:

1. The size of the log tree (that is, the number of leaves it contained).

2. The head values of the log tree's *full subtrees*. The full subtrees are the balanced subtrees which are as large as possible, meaning that they do not have another balanced subtree as their parent.

3. The timestamps of the log entries along the frontier.

When users make queries to the Transparency Log, they advertise the size of the last tree head they observed. If the Transparency Log responds with an updated tree head, it first provides a consistency proof to show that the new tree head is an extension of the previous one. It then also provides the following:

* In the new implicit binary search tree, compute the direct path of the log entry with index size-1, where size is the tree size advertised by the user. Provide the timestamp of each log entry in the direct path whose index is greater than or equal to size.

* Exactly one of these log entries will lie on the new tree's frontier. From this log entry, compute the remainder of the frontier. That is, compute the log entry's right child, the right child's right child, and so on. Provide the timestamps for these log entries as well.

Users verify that the first timestamp is greater than or equal to the timestamp of the rightmost log entry they retained, and that each subsequent timestamp is greater than or equal to the one prior. While this only requires users to verify a logarithmic number of the newly added log entries' timestamps, it guarantees that two users with overlapping views of the tree will detect any violations. While

retaining only the rightmost log entry's timestamp would be sufficient for this purpose, users retain the timestamps of all log entries along the frontier. The additional timestamps are retained to make later parts of the protocol more efficient.

Additionally, the Transparency Log defines two durations: how far ahead and how far behind the current time the rightmost log entry's timestamp may be. Users verify this against their local clock.

For users which have never interacted with the Transparency Log before and don't have a previous tree head to advertise, the Transparency Log simply provides the timestamps of the log entries on the frontier. The user verifies each timestamp is greater than or equal to the one prior, as above.

5. Binary Ladder

A *binary ladder* is a series of lookups, producing a series of inclusion and non-inclusion proofs, from a single log entry's prefix tree. The purpose of a binary ladder varies depending on the exact context in which it is provided, but it is generally to establish some bound on the greatest version of a label that existed as of a particular log entry. All binary ladders are variants of the following series of lookups, which exactly determines the greatest version of a label that exists:

1. First, version x of the label is looked up, where x is a consecutively higher power of two minus one (0, 1, 3, 7, ...). This is repeated until the first non-inclusion proof is produced.

2. Once the first non-inclusion proof is produced, a binary search is conducted between the greatest version that was proved to be included, and the version that was proved to not be included. Each step of the binary search produces either an inclusion or non-inclusion proof, which guides the search left or right until it terminates.

As an example, if the greatest version of a label that existed in a particular log entry was version 6, that would be established by the following: inclusion proofs for versions 0, 1, 3, a non-inclusion proof for version 7, then followed by inclusion proofs for versions 5 and 6. This series of lookups uniquely identifies 6 as the greatest version that exists, in the sense that the Transparency Log would be unable to prove a different greatest version to any user.

While the description above may imply that the series of lookups is interactive, this is not the case in practice. Users may receive one or more binary ladders, corresponding to the same or different log

entries, in a single query response. The Transparency Log's query response always contains sufficient information to allow users to predict the outcome of each lookup (inclusion or non-inclusion of a particular version) in the binary ladder.

Example code for computing the versions of a label that go in a binary ladder is provided in Appendix B.

6. Fixed-Version Searches

When searching the combined tree structure described in Section 3.4, users perform a binary search for the first log entry where the prefix tree at that entry contains the target label-version pair. Users reuse the implicit binary search tree from Section 4.1 for this purpose. This ensures that all users will check the same or similar entries when searching for the same label, allowing for efficient user monitoring of the Transparency Log.

6.1. Binary Ladder

To perform a binary search, users need to be able to inspect individual log entries and determine whether their search should continue to the left of the current log entry or the right. Specifically, they need to be able to determine if the greatest version of their label that was present in some version of the prefix tree was greater than, equal to, or less than their *target version*.

This is accomplished by having the Transparency Log provide a binary ladder from each log entry in the user's search path. Binary ladders provided for the purpose of a fixed-version search follow the series of lookups described in Section 5, but with two optimizations:

First, the series of lookups ends after the first inclusion proof for a version greater than or equal to the target version, or the first non-inclusion proof for a version less than the target version. The additional lookups are unnecessary, since the user only needs to know whether the greatest version of the label that existed as of a particular log entry is greater than or less than their target version -- not its exact value.

Second, the Transparency Log omits inclusion proofs for any versions of the label where another inclusion proof for the same version was already provided in the same query response for a log entry to the left. Similarly, the Transparency Log omits non-inclusion proofs for any versions of the label where another non-inclusion proof for the same version was already provided in the same query response for a log entry to the right. Providing these proofs is unnecessary since the only possible outcome they could have on the user's binary search would be to cause it to fail.

6.2. Maximum Lifetime

A Transparency Log operator MAY define a maximum lifetime for log entries. If defined, it MUST be greater than zero milliseconds. Whether a log entry has surpassed its maximum lifetime is determined by subtracting the timestamp of the rightmost log entry from the timestamp of the log entry in question and checking if the result is greater than or equal to the defined duration.

A user executing a search may arrive at a log entry which is past its maximum lifetime by either of two ways: The user may have inspected a log entry which is *not* expired and decided to recurse to the log entry's left child, which is expired. Alternatively, the root log entry may be expired, in which case the user would've started their search at an expired root log entry.

When a user's search proceeds from a log entry which is not expired to a log entry which is expired, the user is provided with a binary ladder from the expired log entry as usual. If the user's search would recurse further into the expired portion of the tree (to the log entry's left child), the search is aborted. If the user's search would recurse away from the expired portion of the tree (to the log entry's right child), the user continues as normal.

When the root and potentially multiple frontier log entries are expired, the user skips to the furthest-right expired frontier log entry without receiving binary ladders from any of its parents. Similar to the previous case, the user is provided with a binary ladder from this log entry. If the user determines that its search would recurse to the left (further into the expired portion of the tree), it aborts; to the right (into the unexpired portion of the tree), it continues.

This allows the Transparency Log to prune data which is sufficiently old, as only a small amount of the log tree and prefix tree outside of the maximum lifetime need to be retained. Specifically, users will still need only a logarithmic number of log entries that have passed their maximum lifetime, meaning the rest can be discarded. Pruning is explained in more detail in [ARCH].

6.3. Algorithm

The algorithm for performing a fixed-version search (a search for a specific version of a label) is described below as a recursive algorithm. It starts with the root log entry, as defined by the implicit binary search tree, and then recurses to left or right children, each time starting back at step 1.

1. Verify that the log entry's timestamp is consistent with the timestamps of all ancestor log entries. That is, if the log entry is in the ancestor's left subtree, then its timestamp is less than or equal to the ancestor's. If the log entry is in the ancestor's right subtree, then its timestamp is greater than or equal to the ancestor's.

2. If the log entry has surpassed its maximum lifetime and is on the frontier, determine whether its right child has also surpassed its maximum lifetime. If so, recurse to the right child; otherwise, continue to step 3. Note that a right child always exists, as the rightmost log entry cannot exceed its maximum lifetime by definition.

3. Obtain a binary ladder from the current log entry for the target version. Verify that the binary ladder terminates in a way that is consistent with previously inspected log entries. Specifically, verify that it indicates a maximum version greater than or equal to any log entries to the left, and less than or equal to any log entries to the right.

4. If the binary ladder was terminated early due to a non-inclusion proof for a version less than or equal the target version, recurse to the log entry's right child. Otherwise, check if the log entry has surpassed its maximum lifetime. If so, abort the search with an error indicating that the desired version of the label has expired and is no longer available. If not, recurse to the log entry's left child. If, in either case, recursion isn't possible because the search is at a leaf node:

5. This largely concludes the search. However, there are some additional technicalities to address. First, it's possible for the binary search to conclude even if the label-version pair that

the user is interested in doesn't exist or is expired. Out of the log entries touched by the binary search, identify which log entry was first to contain the desired label-version pair. If there is no such log entry, or if it is past its maximum lifetime, abort the search and return an error to the user.

6. It's also possible at this point that a commitment to the contents of the desired label-version pair has not been provided by the Transparency Log. This can happen, for example, if multiple versions of a label were inserted in the same log entry and the binary ladder was terminated early due to an inclusion proof for a version greater than the target version. If this has happened, obtain a search proof for the target label-version pair from the prefix tree in the first log entry to contain it (identified in step 5). If the search proof shows non-inclusion rather than inclusion, return an error to the user.

The most important goal of this algorithm is correctly identifying the first log entry that contains the target label-version pair. The purpose of doing this is to make monitoring more efficient for the label owner. If a label has a large number of versions, it can become prohibitively expensive for its owner to repeatedly check that every single version is represented correctly in multiple log entries. Instead, the label owner can check that the version was created correctly in the one log entry where it was first added and then enforce that binary searches for that version always converge back to that same log entry.

7. Monitoring the Tree

As new entries are added to the log tree, the search path that's traversed to find a specific version of a label may change. New intermediate nodes may be established between the search root and the log entry, or a new search root may be created. The goal of monitoring a label is to efficiently ensure that, when these new parent nodes are created, they're created correctly such that searches for the same versions of a label continue converging to the same entries in the log.

Monitoring is performed both by the users that own a label, meaning they are the authoritative source for the label's content, and the users that lookup a label. Owners monitor their labels to ensure that past (expected) versions of a label are still correctly stored in the log and that no new (unexpected) versions have been added. Users that looked up a label may sometimes need to monitor it afterwards to ensure that the version they observed isn't later concealed by the Transparency Log.

7.1. Reasonable Monitoring Window

Label owners MUST monitor their labels regularly, ensuring that past versions of the label are still correctly represented in the log and that any new versions of the label are permissible (alerting the user if not). Transparency Logs define a duration, referred to as the *Reasonable Monitoring Window* (RMW), which is the frequency with which the Transparency Log generally expects label owners to perform monitoring. The log entry maximum lifetime, if defined, MUST be greater than the RMW.

Distinguished log entries are chosen according to the algorithm below, such that there is roughly one per every interval of the RMW. If a user looks up a label (either through a fixed-version or greatest-version search) and finds that the first log entry that contains the desired label-version pair is to the right of the rightmost distinguished log entry, and the Transparency Log is deployed in Contact Monitoring mode, the user MUST regularly monitor the label-version pair until its monitoring path intersects a distinguished log entry. That is, until a new distinguished log entry is established to its right and the two log entries are verified to be consistent. The purpose of this monitoring is to ensure that the label-version pair is not removed or obscured by the Transparency Log before the label owner has had an opportunity to detect it. If the Transparency Log is deployed with a Third-Party Auditor or Third-Party Manager, this monitoring is not necessary if the third party is honest. However, the user MAY still perform it to detect collusion between the Transparency Log and the third party.

If a user looks up a label and finds that the first log entry containing the label-version pair is either a distinguished log entry or to the left of any distinguished log entry, they do not need to monitor it afterwards. The only state that would be retained from the query would be the tree head, as discussed in Section 4.

"Regular" monitoring SHOULD be performed at least as frequently as the RMW and MUST, if at all possible, happen more frequently than the log entry maximum lifetime.

7.2. Distinguished Log Entries

Distinguished log entries are chosen according to the following recursive algorithm:

1. Take as input: a log entry, the timestamp of a log entry to its left, and the timestamp of a log entry to its right.

2. If the right timestamp minus the left timestamp is less than the Reasonable Monitoring Window, terminate the algorithm. Otherwise, declare that the given log entry is distinguished.

3. If the given log entry has a left child in the implicit binary search tree, then recurse to its subtree by executing this algorithm with: the given log entry's left child, the given left timestamp, and the timestamp of the given log entry.

4. If the given log entry has a right child, then recurse to its right subtree by executing this algorithm with: the given log entry's right child, the timestamp of the given log entry, and the given right timestamp.

The algorithm is initialized with these parameters: the root node in the implicit binary search tree, the timestamp 0, and the timestamp of the rightmost log entry. Note that step 2 is specifically "less than" and not "less than or equal to"; this ensures correct behavior when the RMW is zero.

This process for choosing distinguished log entries ensures that they are *regularly spaced*. Having irregularly spaced distinguished log entries risks either overwhelming label owners with a large number of them, or delaying consensus between users by having arbitrarily few. Distinguished log entries must reliably occur at roughly the same interval as the Reasonable Monitoring Window regardless of variations in how quickly new log entries are added.

This process also ensures that distinguished log entries are *stable*. Once a log entry is chosen to be distinguished, it will never stop being distinguished. This is important because it means that, if a user looks up a label and checks consistency with some distinguished log entry, this log entry can't later avoid inspection by the label owner by losing its distinguished status.

7.3. Binary Ladder

Similar to the algorithm for searching the tree, the algorithm for monitoring the tree requires a way to prove that the greatest version of a label stored in a particular log entry's prefix tree is greater than or equal to a *target version*. The target version in this case is the version of the label that the user is monitoring. Unlike in a search though, users already know that the target version of the label exists and only need proof that there has not been an unexpected downgrade.

Binary ladders provided for the purpose of monitoring follow the series of lookups that would be made by the algorithm in Section 5 if the target version of the label was the greatest that existed. Note that this means the series of lookups performed is always the same for the same target version, regardless of whatever the actual greatest version of the label is. From this series of lookups, two optimizations are made:

First, any lookup for a version greater than the target version is omitted. As a result, all lookups in the binary ladder will result in an inclusion proof if the Transparency Log is behaving honestly.

Second, any lookup that would be omitted from a binary ladder for the log entry when executing a fixed-version or greatest-version search for the label-version pair is also omitted here. That is, when preparing a binary ladder for a log entry, the Transparency Log considers the log entries that are in its direct path and to its left. If, during a search for the label-version pair being monitored, the user would receive an inclusion proof for some version v from one of these log entries, then the lookup for version v is omitted.

7.4. Algorithm

To monitor a given label, users maintain a small amount of state: a map from a position in the log to a version counter. The version counter is the greatest version of the label that’s been proved to exist at that log position. Users initially populate this map by setting the position of the first log entry to contain the label-version pair they’ve looked up to map to that version. A map may track several different versions of a label simultaneously if a user has been shown different versions of the same label.

To update this map, users receive the most recent tree head from the server and follow these steps for each entry in the map, from rightmost to leftmost log entry:

1. Determine if the log entry is distinguished. If so, leave the position-version pair in the map and move on to the next map entry.

2. Compute the ordered list of log entries to inspect:

    1. Initialize the list by setting it to be the log entry’s direct path in the implicit binary search tree based on the current tree size.

    2. Remove all entries that are to the left of the log entry.

3. If any of the remaining log entries are distinguished, terminate the list just after the first distinguished log entry.

3. If the computed list is empty, leave the position-version pair in the map and move on to the next map entry.

4. For each log entry in the computed list, from left to right:

    1. Check if a binary ladder for this log entry was already provided in the same query response. If so:

        1. If the previously provided binary ladder had a greater target version than the current map entry, then this version of the label no longer needs to be monitored. Remove the position-version pair with the the lesser version from the map and move on to the next map entry.

        2. If it had a version less than or equal to that of the current map entry, terminate and return an error to the user.

    2. Receive and verify a binary ladder from this log entry where the target version is the version currently in the map. This proves that, at the indicated log entry, the greatest version present is greater than or equal to the previously observed version.

    3. If the above check fails, terminate and return an error to the user. Otherwise, remove the current position-version pair from the map and replace it with a new one for the position of the log entry that the binary ladder came from.

Once the map entries are updated according to this process, the final step of monitoring is to remove all mappings where the position corresponds to a distinguished log entry. All remaining entries will be non-distinguished log entries lying on the log’s frontier.

In summary, monitoring works by progressively moving up the tree as new intermediate/root nodes are established and verifying that they’re constructed correctly. Once a distinguished log entry is reached and successfully verified, monitoring is no longer necessary and the relevant entry is removed from the map.

Users will often be able to execute the monitoring process, at least partially, with the output of a fixed-version or greatest-version search for the label. This may reduce the need for monitoring-specific requests. It is also worth noting that the work required to

monitor several versions of the same label scales sublinearly because the direct paths of the different versions will often intersect. Intersections reduce the total number of entries in the map and therefore the amount of work that will be needed to monitor the label from then on.

7.4.1. Owner Algorithm

If the user owns the label being monitored, they will additionally need to retain the rightmost distinguished log entry where they've verified that the greatest version of the label is correct. Users advertise this log entry's position in their Monitor request. For a number of subsequent distinguished log entries, the Transparency Log provides the greatest version of the label that the log entry's prefix tree contains, along with a binary ladder (according to the rules stated in Section 8.1) to prove that this is correct.

Users verify that the version has not unexpectedly increased or decreased. Importantly, users also verify that they receive a binary ladder for the distinguished log entry immediately following the one they've advertised, the distinguished log entry immediately following that one, and so on. The Transparency Log provides whichever intermediate timestamps are necessary to demonstrate that this is the case. To avoid excessive load, the Transparency Log SHOULD limit the number of distinguished log entries it provides binary ladders for in a single response.

If a user is monitoring the label for the first time since it was created, they advertise the first log entry to contain the label even if it is not known to be distinguished. The Transparency Log provides binary ladders for subsequent distinguished log entries.

8. Greatest-Version Searches

Users often wish to search for the "most recent" version, or the greatest version, of a label. Unlike searches for a specific version, label owners regularly verify that the greatest version is correctly represented in the log. This enables a simpler, more efficient approach to searching.

Section 7.2 defines the concept of a distinguished log entry, which is any log entry that label owners are required to check for correctness. As a result, users can start their search at the rightmost distinguished log entry and only consider new versions which have been created since then. The rightmost distinguished log entry will always be on the frontier of the log and will never be past its maximum lifetime.

8.1. Binary Ladder

One special consideration for a greatest-version search is that the Transparency Log must prove that it is revealing the absolute greatest version of a label that exists, referred to as the *target version*. This differs from the binary ladders described for fixed-version searches (Section 6.1) and monitoring (Section 7.3), which only aim to prove a lower bound on the greatest version.

Binary ladders provided for the purpose of a greatest-version search follow the series of lookups described in Section 5, with two optimizations:

First, the series of lookups ends after the first non-inclusion proof for a version less than the target version. This differs from Section 6.1 in that the binary ladder algorithm will continue even after receiving an inclusion proof for a version equal to the target version. This is often necessary to demonstrate that there are no versions greater than the target version.

Second, depending on whether the binary ladder is for a distinguished or non-distinguished log entry:

* If the log entry is non-distinguished:
  - An inclusion proof for a version is omitted if an inclusion proof for the same version has already been provided in the same query response from a log entry to the left.
  - A non-inclusion proof for a version is omitted if a non-inclusion proof for the same version has already been provided in the same query response from a log entry to the right.
* If the log entry is distinguished:
  - An inclusion or non-inclusion proof for a version is omitted only if it has previously been provided in the same query response for the same log entry. This may happen if the binary ladder is provided in a Monitor query response and the user owns the label being monitored.

8.2. Algorithm

The algorithm for performing a greatest-version search (a search for the greatest version of a label) is described below as a recursive algorithm. It starts at the rightmost distinguished log entry, or the root of the implicit binary search tree if there are no distinguished log entries, and then recurses down the remainder of the frontier, each time starting back at step 1:

1. Obtain a binary ladder from the current log entry for the target version. If this is not the starting log entry, verify that the binary ladder indicates a maximum version greater than or equal to that of its parent log entry.

2. If this is the rightmost log entry, verify the binary ladder terminates in a way that proves the target version to be the greatest that exists. This means that it does not terminate early, all lookups for versions less than or equal to the target version produce inclusion proofs, and all lookups for versions greater than the target version produce non-inclusion proofs.

3. If this is not the rightmost log entry, recurse to the current log entry’s right child.

If the starting log entry was not distinguished or if the starting log entry did not contain the greatest version of the label, note that the user may be obligated to monitor the label in the future per Section 7.1.

9. Cryptographic Computations

9.1. Cipher Suites

Each Transparency Log uses a single fixed cipher suite, chosen when it is initially created, that specifies the following primitives and parameters for cryptographic computations:

* A hash algorithm

* A signature algorithm

* A Verifiable Random Function (VRF) algorithm

* Nc: The size in bytes of commitment openings

* Kc: A fixed string of bytes used in the computation of commitments

The hash algorithm is used to calculate intermediate and root values of hash trees. The signature algorithm is used for signatures from both the service operator and the third party, if one is present. The VRF is used for preserving the privacy of labels. One of the VRF algorithms from [RFC9381] must be used.

Cipher suites are represented with the CipherSuite type. The cipher suites are defined in Section 14.1.

9.2. Tree Head Signature

The head of a Transparency Log, which represents its most recent state, is encoded as:

```
struct {
  uint64 tree_size;
  opaque signature<0..2^16-1>;
} TreeHead;
```

where tree_size is the number of log entries. If the Transparency Log is deployed with Third-Party Management, then the public key used to verify the signature belongs to the Third-Party Manager; otherwise the public key used belongs to the Service Operator.

The signature itself is computed over a TreeHeadTBS structure, which incorporates the log’s current state as well as long-term log configuration:

```
enum {
  reserved(0),
  contactMonitoring(1),
  thirdPartyManagement(2),
  thirdPartyAuditing(3),
  (255)
} DeploymentMode;

struct {
  CipherSuite ciphersuite;
  DeploymentMode mode;
  opaque signature_public_key<0..2^16-1>;
  opaque vrf_public_key<0..2^16-1>;

  select (Configuration.mode) {
    case contactMonitoring:
    case thirdPartyManagement:
      opaque leaf_public_key<0..2^16-1>;
    case thirdPartyAuditing:
      uint64 max_auditor_lag;
      uint64 auditor_start_pos;
      opaque auditor_public_key<0..2^16-1>;
  };

  uint64 max_ahead;
  uint64 max_behind;
  uint64 reasonable_monitoring_window;
  optional<uint64> maximum_lifetime;
} Configuration;

struct {
  Configuration config;
  uint64 tree_size;
  opaque root[Hash.Nh];
} TreeHeadTBS;
```

The ciphersuite field contains the cipher suite for the Transparency Log, chosen from the registry in Section 14.1. The mode field specifies whether the Transparency Log is deployed in Contact Monitoring mode or with a Third-Party Manager or Auditor. The signature_public_key field contains the public key to use for verifying signatures on the TreeHeadTBS structure. The vrf_public_key field contains the VRF public key to use for evaluating the VRF proofs provided in the BinaryLadderStep.proof field described in Section 11.1.

If the deployment mode specifies a Third-Party Manager, a public key is provided in leaf_public_key. This public key is used to verify the Service Operator's signature on modifications to the Transparency Log, as described in Section 9.5.

If the deployment mode specifies a Third-Party Auditor, the maximum amount of time in milliseconds that the auditor may lag behind the most recent version of the Transparency Log is provided in max_auditor_lag. The position of the first log entry that the auditor started processing is provided in auditor_start_pos. A public key for verifying the auditor's signature on views of the Transparency Log is provided in auditor_public_key.

The max_ahead and max_behind fields contain the maximum amount of time in milliseconds that a tree head may be ahead of or behind the user's local clock without being rejected. The reasonable_monitoring_window contains the Reasonable Monitoring Window, defined in Section 7.1, in milliseconds. If the Transparency Log has chosen to define a maximum lifetime for log entries, per Section 6.2, this duration in milliseconds is stored in the maximum_lifetime field.

Finally, the root field contains the root value of the log tree with tree_size leaves. Hash.Nh is the output size of the cipher suite's hash function in bytes.

9.3. Auditor Tree Head Signature

In deployment scenarios where a Third-Party Auditor is present, the auditor's view of the Transparency Log is presented to users with an AuditorTreeHead structure:

```
struct {
  uint64 timestamp;
  uint64 tree_size;
  opaque signature<0..2^16-1>;
} AuditorTreeHead;
```

Users verify an AuditorTreeHead with the following steps:

1. If the user advertised a previously observed tree size in their request, verify that the advertised tree size is greater than or equal to Configuration.auditor_start_pos.

2. Verify that the timestamp of the rightmost log entry is greater than or equal to timestamp, and that the difference between the two is less than or equal to Configuration.max_auditor_lag.

3. Verify that tree_size is less than or equal to that of the TreeHead provided by the Transparency Log.

4. Verify signature as a signature over the AuditorTreeHeadTBS structure:

```
struct {
  Configuration config;
  uint64 timestamp;
  uint64 tree_size;
  opaque root[Hash.Nh];
} AuditorTreeHeadTBS;
```

The config field contains the long-term configuration for the Transparency Log. The timestamp and tree_size fields match that of AuditorTreeHead. The root field contains the value of the root node of the log tree when it had tree_size leaves.

9.4. Full Tree Head Verification

Tree heads are presented to users on the wire as follows:

```
enum {
  reserved(0),
  same(1),
  updated(2),
} FullTreeHeadType;

struct {
  FullTreeHeadType head_type;
  select (FullTreeHead.head_type) {
    case updated:
      TreeHead tree_head;
      select (Configuration.mode) {
        case thirdPartyAuditing:
          AuditorTreeHead auditor_tree_head;
      };
  };
} FullTreeHead;
```

The head_type field may be set to same if the user advertised a previously observed tree size in their request and the Transparency Log wishes to continue using this same tree head. Otherwise, head_type is set to updated and a new, more recent tree head is provided.

Users verify a FullTreeHead with the following steps:

1. If head_type is same, verify that the user advertised a previously observed tree size and that the rightmost log entry of this tree is still within the bounds set by max_ahead and max_behind.

2. If head_type is updated:

    1. If the user advertised a previously observed tree size, verify that TreeHead.tree_size is greater than the advertised tree size.

    2. Verify TreeHead.signature as a signature over the TreeHeadTBS structure.

    3. If there is a Third-Party Auditor, verify auditor_tree_head as described in Section 9.3.

9.5. Update Format

The leaves of the prefix tree contain commitments which open to the value of a label-version pair, potentially with some additional information depending on the deployment mode of the Transparency Log. The contents of these commitments is serialized as follows:

```
struct {
  select (Configuration.mode) {
    case thirdPartyManagement:
      opaque signature<0..2^16-1>;
  };
} UpdatePrefix;

struct {
  UpdatePrefix prefix;
  opaque value<0..2^32-1>;
} UpdateValue;
```

The value field contains the value associated with the label-version pair.

In the event that Third-Party Management is used, the prefix field contains a signature from the Service Operator, using the public key from Configuration.leaf_public_key, over the following structure:

```
struct {
  opaque label<0..2^8-1>;
  uint32 version;
  opaque value<0..2^32-1>;
} UpdateTBS;
```

The value field contains the same contents as UpdateValue.value. Users MUST successfully verify this signature before consuming UpdateValue.value.

9.6. Commitment

Commitments are computed with HMAC [RFC2104] using the hash function specified by the cipher suite. To produce a new commitment, the application generates a random Nc-byte value called opening and computes:

```
commitment = HMAC(Kc, CommitmentValue)
```

where Kc is a string of bytes defined by the cipher suite and CommitmentValue is specified as:

```
struct {
  opaque opening[Nc];
  opaque label<0..2^8-1>;
  UpdateValue update;
} CommitmentValue;
```

The output value commitment may be published, while opening should only be revealed to users that are authorized to receive the label's contents.

The Transparency Log MAY generate opening in a non-random way, such as deriving it from a secret key, as long as the result is indistinguishable from random to other participants. The Transparency Log SHOULD ensure that individual opening values can later be deleted in a way where they can not feasibly be recovered. This preserves the Transparency Log's ability to delete certain information in compliance with privacy laws.

9.7. Verifiable Random Function

Each label-version pair corresponds to a unique search key in the prefix tree. This search key is the output of executing the VRF, with the private key corresponding to Configuration.vrf_public_key, on the combined label and version:

```
struct {
  opaque label<0..2^8-1>;
  uint32 version;
} VrfInput;
```

9.8. Log Tree

The value of a leaf node in the log tree is computed as the hash, with the cipher suite hash function, of the following structure:

```
struct {
  uint64 timestamp;
  opaque prefix_tree[Hash.Nh];
} LogLeaf;
```

The timestamp field contains the timestamp that the leaf was created in milliseconds since the Unix epoch. The prefix_tree field contains the updated root hash of the prefix tree after making any desired modifications.

The value of a parent node in the log tree is computed by hashing together the values of its left and right children:

```
parent.value = Hash(hashContent(parent.leftChild) ||
                    hashContent(parent.rightChild))

hashContent(node):
  if node.type == leafNode:
    return 0x00 || node.value
  else if node.type == parentNode:
    return 0x01 || node.value
```

where Hash denotes the cipher suite hash function.

9.9. Prefix Tree

The value of a leaf node in the prefix tree is computed as the hash, with the cipher suite hash function, of the following structure:

```
struct {
    opaque vrf_output[VRF.Nh];
    opaque commitment[Hash.Nh];
} PrefixLeaf;
```

The vrf_output field contains the VRF output for the label-version pair. VRF.Nh denotes the output size of the cipher suite VRF in bytes. The commitment field contains the commitment to the corresponding UpdateValue structure.

The value of a parent node in the prefix tree is computed by hashing together the values of its left and right children:

```
parent.value = Hash(hashContent(parent.leftChild) ||
                    hashContent(parent.rightChild))

hashContent(node):
  if node.type == emptyNode:
    return 0 // all-zero vector of length Hash.Nh+1
  else if node.type == leafNode:
    return 0x01 || node.value
  else if node.type == parentNode:
    return 0x02 || node.value
```

10. Tree Proofs

10.1. Log Tree

In the interest of efficiency, KT combines multiple inclusion proofs and consistency proofs into a single batch proof. Recalling from the discussion in Section 3.2,

* Whenever the Transparency Log serves an inclusion proof for a leaf of the log tree, it provides the minimum set of head values from balanced subtrees that would allow the user to compute the root hash from the leaf's value.

* Whenever the Transparency Log serves a consistency proof, the user is expected to have retained the head values of the full subtrees of the previous version of the log. The Transparency Log provides the minimum set of head values from balanced subtrees that would allow the user to compute the root hash from their retained values.

These two proof types are composed together as such: considering the leaf values which will be proved included, and any node values the user is understood to have retained, the Transparency Log provides the minimum set of head values from balanced subtrees that would allow the user to compute the root hash from the leaf and retained values. This proof is encoded as follows:

```
opaque NodeValue[Hash.Nh];

struct {
  NodeValue elements<0..2^16-1>;
} InclusionProof;
```

The contents of the elements array is in left-to-right order: if a node is present in the root's left subtree then its value is listed before the values of any nodes in the root's right subtree, and so on recursively.

Batching together inclusion and consistency proofs creates an edge case that requires special care: when a user has requested a consistency proof, and also requested inclusion proofs for leaves located in one or more of the subtrees that the user has retained the head of. When this happens, the portion of the batch proof that shows inclusion for the leaves in these subtrees will itself be sufficient to recompute the retained head values. This makes the retained values redundant for the purpose of computing the new root hash, which could result in the retained values being disregarded in a naive implementation. To avoid accepting invalid proofs, users MUST verify that the computed value for the head of any such subtree matches the retained value.

10.2. Prefix Tree

A proof from a prefix tree authenticates that a search was done correctly for a given search key. Such a proof is encoded as:

```
enum {
  reserved(0),
  inclusion(1),
  nonInclusionLeaf(2),
  nonInclusionParent(3),
  (255)
} PrefixSearchResultType;

struct {
  PrefixSearchResultType result_type;
  select (PrefixSearchResult.result_type) {
    case nonInclusionLeaf:
      PrefixLeaf leaf;
  };
  uint8 depth;
} PrefixSearchResult;

struct {
  PrefixSearchResult results<0..2^8-1>;
  NodeValue elements<0..2^16-1>;
} PrefixProof;
```

The results field contains the search result for each individual value. Every index corresponds to the respectively indexed binary ladder step targeting the queried version. The result_type field of each PrefixSearchResult struct indicates what the terminal node of the search for that value was:

* inclusion for a leaf node matching the requested value.

* nonInclusionLeaf for a leaf node not matching the requested value. In this case, the terminal node's value is provided since it can not be inferred.

* nonInclusionParent for a parent node that lacks the desired child.

The depth field indicates the depth of the terminal node of the search, and is provided to assist proof verification. The root node of the prefix tree corresponds to a depth of 0, the root's children correspond to a depth of 1, and so on recursively.

The elements array consists of the fewest node values that can be hashed together with the provided leaves to produce the root. The contents of the elements array is kept in left-to-right order: if a node is present in the root's left subtree, its value must be listed before any values provided from nodes that are in the root's right subtree, and so on recursively. In the event that a node is not present, an all-zero byte string of length Hash.Nh is listed instead.

The proof is verified by hashing together the provided elements, in the left/right arrangement dictated by the bits of the search keys, and checking that the result equals the root value of the prefix tree.

10.3. Combined Tree

As users execute the algorithms for searching, monitoring, or updating their view of the tree, they inspect a series of log entries. For some of these, only the timestamp of the log entry is needed. For others, both the timestamp and a PrefixProof from the log entry's prefix tree are needed.

This subsection defines a general structure, called a CombinedTreeProof, that contains the minimum set of timestamps and PrefixProof structures that a user needs for their execution of these algorithms. For the purposes of this protocol, the user always executes the algorithm to update their view of the tree, described in Section 4, followed immediately by one of the algorithms to search or monitor the tree.

Proofs are encoded as follows:

```
struct {
  uint64 timestamps<0..2^8-1>;
  PrefixProof prefix_proofs<0..2^8-1>;
  NodeValue prefix_roots<0..2^8-1>;

  InclusionProof inclusion;
} CombinedTreeProof;
```

The timestamps field contains the timestamps of specific log entries and the prefix_proofs field contains search proofs from the prefix trees of specific log entries. There is no explicit indication as to which log entry the elements correspond to, as they are provided in the order that the algorithm the user is executing would request them. The elements of the prefix_roots field are, in left-to-right order, the prefix tree root hashes for any log entries whose timestamp was provided in timestamps but a search proof was not provided in prefix_proofs.

If a log entry's timestamp is referenced multiple times by algorithms in the same CombinedTreeProof, it is only added to the timestamps array the first time. Additionally, when a user advertises a previously observed tree size in their request, log entry timestamps that the user is expected to have retained are always omitted from timestamps. This may result in there being elements of prefix_proofs or prefix_roots that correspond to log entries whose timestamps are not included in timestamps

If different algorithms in the same CombinedTreeProof require a search proof from the same log entry, the prefix_proofs array will contain multiple PrefixProof structures for the same log entry. Users MUST verify that all PrefixProof structures corresponding to the same log entry compute the same prefix tree root hash.

Users processing a CombinedTreeProof MUST verify that the timestamps, prefix_proofs, and prefix_roots fields contain exactly the expected number of entries -- no more and no less.

Finally, the inclusion field contains the minimum set of intermediate node values from the log tree that would allow a user to compute:

* The root value of the log tree, and

* If an AuditorTreeHead was provided by the Transparency Log, the root value of the log tree when it had AuditorTreeHead.tree_size leaves,

from the following:

* The values of all leaf nodes where either a search proof was provided in prefix_proofs or the prefix tree root hash was provided directly in prefix_roots, and

* If the user advertised a previously observed tree size in their request, any intermediate node values the user is expected to have retained.

10.3.1. Updating View

For a user to update their view of the tree, the following is provided:

* If the user has not previously observed a tree head, the timestamp of each log entry along the frontier.

* If the user has previously observed a tree head, the timestamps of each log entry from the list computed in Section 4.2.

Users verify that the timestamps represent a monotonic series, and that the rightmost timestamp is within the bounds defined by max_ahead and max_behind.

10.3.2. Fixed-Version Search

For a user to search the combined tree for a specific version of a label, the following is provided:

* For each log entry touched by the algorithm in Section 6.3:

  - The log entry’s timestamp.

  - If the log entry has surpassed its maximum lifetime and is on the frontier, the right child’s timestamp.

  - If it is not the case that the log entry has surpassed its maximum lifetime, is on the frontier, and the log entry’s right child has also surpassed its maximum lifetime, then a PrefixProof corresponding to a binary ladder (Section 6.1) in the log entry’s prefix tree is provided.

* If the PrefixProof from the first log entry containing the target label-version pair didn’t include a lookup for the target version, provide a second PrefixProof from this log entry specifically looking up the target version.

Users verify the output as specified in Section 6.3.

10.3.3. Monitor

For a user to monitor a label in the combined tree, the following is provided:

* For each entry in the user's monitoring map:
  - The timestamps needed by the algorithm in Section 7.2 to determine where the monitoring algorithm would first reach a distinguished log entry. This may either be the log entry in the user's monitoring map, or some other log entry from the list computed in step 2 of Section 7.4.
  - Where necessary for the algorithm in Section 7.4, a binary ladder (Section 7.3) targeting the version in the user's monitoring map.
* If the user owns the label:
  - The timestamps needed by the algorithm in Section 7.2 to conduct a depth-first search for each subsequent distinguished log entry.
  - For each distinguished log entry, a binary ladder (Section 8.1) targeting the greatest version of the label that the log entry contains.

10.3.4. Greatest-Version Search

For a user to search the combined tree for the greatest version of a label, the following is provided:

* For each log entry along the frontier, starting from the log entry identified in Section 8: a binary ladder (Section 8.1) targeting the greatest version of the label that exists in the log overall.

Note that the log entry timestamps are already provided as part of updating the user's view of the tree and that no additional timestamps are necessary to identify the starting log entry. Users verify the proof as described in Section 8.

11. User Operations

The basic user operations are organized as a request-response protocol between a user and the Transparency Log.

Users MUST retain the most recent TreeHead they've successfully verified as part of any query response and populate the last field of any query request with the tree_size from this TreeHead. This ensures that all operations performed by the user return consistent results.

Modifications to a user's state MUST only be persisted once the query response has been fully verified. Queries that fail full verification MUST NOT modify the user's protocol state in any way.

11.1. Search

Users initiate a Search operation by submitting a SearchRequest to the Transparency Log containing the label that they're interested in. Users can optionally specify a version of the label that they'd like to receive, if not the greatest one.

```
struct {
  optional<uint64> last;

  opaque label<0..2^8-1>;
  optional<uint32> version;
} SearchRequest;
```

In turn, the Transparency Log responds with a SearchResponse structure:

```
struct {
  opaque proof[VRF.Np];
  opaque commitment[Hash.Nh];
} BinaryLadderStep;

struct {
  FullTreeHead full_tree_head;

  optional<uint32> version;
  BinaryLadderStep binary_ladder<0..2^8-1>;
  CombinedTreeProof search;

  opaque opening[Nc];
  UpdateValue value;
} SearchResponse;
```

Each BinaryLadderStep structure contains information related to one version of the label that's in the binary ladder. The proof field contains the VRF proof, and commitment contains the commitment to the label's value at that version. The binary_ladder field contains these structures in the same order that the versions are output by the algorithm in Section 5.

The search field contains the output of updating the user's view of the tree to match FullTreeHead.tree_head.size followed by either a fixed-version or greatest-version search for the requested label, depending on whether version was provided in SearchRequest or not. If searching for the greatest version of the label, this version is provided in SearchResponse.version; otherwise, the field is empty.

Users verify a search response by following these steps:

1. Compute the VRF output for each version of the label from the proofs in binary_ladder.

2. Verify the proof in search as described in Section 10.3.

3. Compute a candidate root value for the tree from the proof in search.inclusion and any previously retained full subtrees of the log tree.

4. With the candidate root value for the tree, verify FullTreeHead.

5. Verify that the commitment to the target version of the label opens to SearchResponse.value with opening SearchResponse.opening.

Depending on the deployment mode of the Transparency Log, the value field may or may not require additional verification, specified in Section 9.5, before its contents may be consumed.

11.2. Update

Users initiate an Update operation by submitting an UpdateRequest to the Transparency Log containing the new label and value to store.

```
struct {
  optional<uint64> last;

  opaque label<0..2^8-1>;
  opaque value<0..2^32-1>;
} UpdateRequest;
```

If the request passes application-layer policy checks, the Transparency Log adds a new label-version pair to the prefix tree, followed by adding a new entry to the log tree with an updated timestamp and prefix tree root. It returns an UpdateResponse structure:

```
struct {
  FullTreeHead full_tree_head;

  uint32 version;
  BinaryLadderStep binary_ladder<0..2^8-1>;
  CombinedTreeProof search;

  opaque opening[Nc];
  UpdatePrefix prefix;
} UpdateResponse;
```

Users verify the UpdateResponse as if it were a SearchResponse for the greatest version of label. To aid verification, the update response provides the UpdatePrefix structure necessary to reconstruct the UpdateValue.

11.3. Monitor

Users initiate a Monitor operation by submitting a MonitorRequest to the Transparency Log containing information about the labels they wish to monitor.

```
struct {
  uint64 position;
  uint32 version;
} MonitorMapEntry;

struct {
  opaque label<0..2^8-1>;
  MonitorMapEntry entries<0..2^8-1>;
  optional<uint64> rightmost;
} MonitorLabel;

struct {
  optional<uint64> last;
  MonitorLabel labels<0..2^8-1>;
} MonitorRequest;
```

Each MonitorLabel structure in labels contains the label to monitor in label, and a list in the entries field corresponding to the map described in Section 7.4. If the user owns the label, they additionally indicate in rightmost the position of the rightmost distinguished log entry where they have verified that the greatest version of the label is correctly represented.

The Transparency Log verifies the MonitorRequest by following these steps, for each MonitorLabel structure:

1. Verify that the label field of every MonitorLabel is unique. For all MonitorLabel structures with rightmost provided, verify that the user owns the label (according to application-layer policy). For all other MonitorLabel structures, verify that the user is currently, or was previously, allowed to lookup all versions of the label contained in a MonitorMapEntry.

2. Verify that each MonitorMapEntry in the same MonitorLabel structure is sorted in ascending order by position. Additionally, verify that each version field is unique and that position lies on the direct path of the first log entry to contain version version of the label.

3. Verify that rightmost is a distinguished log entry to the right of the first version of the label, or that it was the rightmost distinguished log entry immediately after the label was first inserted.

While access control decisions generally belong solely to the application, users must be able to monitor versions of a label they previously looked up, even if they would no longer be allowed to make the same query. One simple way for a user to prove that they were previously allowed to lookup a particular version of a label would be for them to provide the commitment opening for the version. However, there is no provision for this in the protocol; it would need to be done in the application layer.

If the request is valid and passes access control, the Transparency Log responds with a MonitorResponse structure:

```
struct {
  uint32 versions<0..2^8-1>;
} MonitorLabelVersions;

struct {
  FullTreeHead full_tree_head;
  MonitorLabelVersions label_versions<0..2^8-1>;
  CombinedTreeProof monitor;
} MonitorResponse;
```

The monitor field contains the output of updating the user's view of the tree to match FullTreeHead.tree_head.size followed by monitoring each label in labels, in the order provided. Each MonitorLabel structure where rightmost was present has a corresponding entry in label_versions containing the greatest version of the label present in a number of subsequent distinguished log entries.

Users verify a MonitorResponse by following these steps:

1. Verify that the number of entries in label_versions is equal to the number of MonitorLabel structures in labels with rightmost present. If a MonitorLabel has a rightmost field that is not the rightmost distinguished log entry, verify that the corresponding MonitorLabelVersion's versions field is not empty.

2. Verify the proof in monitor as described in Section 10.3.

3. Compute a candidate root value for the tree from the proof in monitor.inclusion and any previously retained full subtrees of the log tree.

4. With the candidate root value for the tree, verify FullTreeHead.

Some information is omitted from MonitorResponse in the interest of efficiency, because the user would have already seen and verified it as part of conducting other queries. In particular, VRF proofs for different versions of each label are not provided, given that these can be cached from the original Search or Update query.

12. Third Parties

Third-Party Management and Third-Party Auditing are two deployment modes that require the Transparency Log to delegate part of its operation to a third party. Users are able to run more efficiently as long as they can assume that the Transparency Log and the third party won't collude to trick them into accepting malicious results.

12.1. Management

With the Third-Party Management deployment mode, a third party is responsible for the majority of the work of storing and operating the Transparency Log. The Service Operator serves only to enforce access control, authenticate the addition of new entries, and prevent the creation of forks by the Third-Party Manager. Critically, the Service Operator is trusted to ensure that only one value for each version of a label is authorized.

All user queries specified in Section 11 are initially sent by users directly to the Service Operator to be forwarded to the Third-Party Manager if they pass access control. While other operations are forwarded by the Service Operator unchanged, UpdateRequest structures are forwarded to the Third-Party Manager with the Service Operator's signature attached:

```
struct {
  UpdateRequest request;
  opaque signature<0..2^16-1>;
} ManagerUpdateRequest;
```

The signature is computed as described in Section 9.5.

12.2. Auditing

With the Third-party Auditing deployment mode, the Service Operator obtains signatures from a Third-Party Auditor attesting to the fact that the Service Operator is constructing the tree correctly. These signatures are provided to users along with the responses to their queries.

For each new log entry the Service Operator adds to the log, it produces a corresponding AuditorUpdate structure and sends this to the Third-Party Auditor. The Third-Party Auditor MUST receive and successfully verify an AuditorUpdate structure for a log entry before providing the Service Operator with an AuditorTreeHead structure whose size field would include the log entry.

```
struct {
  uint64 timestamp;

  PrefixLeaf added<0..2^16-1>;
  PrefixLeaf removed<0..2^16-1>;

  PrefixProof proof;
} AuditorUpdate;
```

The timestamp field contains the timestamp of the corresponding log entry. The added field contains the list of PrefixLeaf structures that were added to the prefix tree in the corresponding log entry. The removed field contains the list of PrefixLeaf structures that were removed from the prefix tree.

The proof field contains a batch lookup proof in the previous log entry’s prefix tree for all search keys referenced by added or removed. Theproof.resultsfield contains the result of the search for each element ofaddedin the order provided, followed by the result of the search for each element ofremoved‘ in the order provided.

An auditor processes a single AuditorUpdate by following these steps:

1. Verify that timestamp is greater than or equal to the timestamp of the previous log entry.

2. Verify that the PrefixSearchResult provided in proof for each element of added has a result_type of nonInclusionParent or nonInclusionLeaf.

3. Verify that the PrefixSearchResult provided in proof for each element of removed has a result_type of inclusion.

4. For each element of removed, verify that, with the addition of the new log entry, the prefix tree leaf was published in at least one distinguished log entry before removal.

5. With proof and the PrefixLeaf structures in removed, compute the root value of the previous log entry’s prefix tree. Verify that this matches the auditor’s state.

6. With proof and the PrefixLeaf structures in added and removed, compute the new root value of the prefix tree. Compute the new root value of the log tree after adding a leaf with the specified timestamp and prefix tree root value.

7. Provide an AuditorTreeHead to the Service Operator where AuditorTreeHead.timestamp is set to timestamp and AuditorTreeHead.tree_size is set to the new size of the log tree after the addition of the new leaf. The signature is computed with the log tree root value computed in the previous step.

13. Security Considerations

The security properties provided by this protocol are discussed in detail in [ARCH]. Generally speaking, the Key Transparency protocol ensures that all users of a Transparency Log have a consistent view of the data stored in the log. Service Operators may still be able to make malicious modifications to stored data, such as by attaching new public keys to a user's account and encouraging other users to encrypt to these public keys when messaging the user. However, since the existence of these new public keys is equally visible to the user whose account they affect, the user can promptly act to have them removed from their account or inform contacts out-of-band that their communication may be compromised.

Key Transparency relies on users coming online regularly to monitor for unexpected or malicious modifications to their account. Users that go offline for longer than the log entry maximum lifetime may not detect if the Transparency Log made malicious modifications to their labels.

Similarly, Key Transparency relies on the ability of users to retain long-term state regarding their account and past views of the Transparency Log. Users which are unable to maintain long-term state, or may lose their state, have a correspondingly limited ability to detect misbehavior by the Service Operator. In particular, users which are completely stateless will generally gain nothing by participating in this protocol over simply verifying a signature from the Service Operator and, if there is one, the Third-Party Auditor or Manager.

Ultimately, ensuring that all users have a consistent view of the Transparency Log requires that the Service Operator is not able to create and maintain long-term network partitions between users. As such, users need access to at least one communication channel (even a very low-bandwidth one) that is resistant to partitions. The protocol directly provides for a Third-Party Auditor or Manager, which is trusted to prevent such partitions. Other options include allowing users to gossip with each other, or allowing users to contact the Transparency Log over an anonymous channel.

Key Transparency provides users with a limited assurance that query responses are authentic: a network attacker will not be able to forge false responses to queries but may provide responses which are up to max_behind milliseconds stale. Key Transparency provides no privacy from network observers and does not have the ability to authenticate specific users to the Transparency Log. To mitigate these limitations, users SHOULD contact the Transparency Log over a protocol that provides transport-layer encryption and an appropriate level of authentication for both parties.

14. IANA Considerations

This document requests the creation of the following new IANA registries:

* KT Cipher Suites (Section 14.1)

All of these registries should be under a heading of "Key Transparency", and assignments are made via the Specification Required policy [RFC8126]. See Section 14.2 for additional information about the KT Designated Experts (DEs).

RFC EDITOR: Please replace XXXX throughout with the RFC number assigned to this document

14.1. KT Cipher Suites

A cipher suite is a specific combination of cryptographic primitives and parameters to be used in an instantiation of the protocol. Cipher suite names follow the naming convention:

```
uint16 CipherSuite;
CipherSuite KT_LVL_HASH_SIG = VALUE;
```

The columns in the registry are as follows:

* Value: The numeric value of the cipher suite.

* Name: The name of the cipher suite.

* Recommended: Whether support for this cipher suite is RECOMMENDED. Valid values are "Y", "N", and "D", as described below. The default value of the "Recommended" column is "N". Setting the Recommended item to "Y" or "D", or changing an item whose current value is "Y" or "D", requires Standards Action [RFC8126].

- Y: Indicates that the item is RECOMMENDED. This only means that the associated mechanism is fit for the purpose for which it was defined. Careful reading of the documentation for the mechanism is necessary to understand the applicability of that mechanism. A cipher suite may, for example, be recommended that is only suitable for use in applications where the Transparency Log's contents are public. Mechanisms with limited applicability may be recommended, but in such cases applicability statements that describe any limitations of the mechanism or necessary constraints will be provided.

- N: Indicates that the item's associated mechanism has not been evaluated and is not RECOMMENDED (as opposed to being NOT RECOMMENDED). This does not mean that the mechanism is flawed. For example, an item may be marked as "N" because it has usage constraints or limited applicability.

- D: Indicates that the item is discouraged and SHOULD NOT be used. This marking could be used to identify mechanisms that might result in problems if they are used, such as a weak cryptographic algorithm or a mechanism that might cause interoperability problems in deployment.

* Reference: The document where this cipher suite is defined.

Initial contents:

Value	Name	R	Ref
0x0000	RESERVED	-	RFC XXXX
0x0001	KT_128_SHA256_P256	Y	RFC XXXX
0x0002	KT_128_SHA256_Ed25519	Y	RFC XXXX

Table 1

All cipher suites currently specified share the following primitives and parameters:

* The hash algorithm is SHA-256, as defined in [SHS].

* Nc: 16

* Kc: The byte sequence equal to the hex-encoded string d821f8790d97709796b4d7903357c3f5

The signature algorithm and VRF algorithm for each cipher suite is as follows:

Name	Signature	VRF Algorithm
KT_128_SHA256_P256	ecdsa_secp256r1_sha256	ECVRF-P256-SHA256-TAI
KT_128_SHA256_Ed25519	ed25519	ECVRF-EDWARDS25519-SHA512-TAI[32]

Table 2

The VRF algorithms are specified in [RFC9381]. For KT_128_SHA256_Ed25519, the final hash output of ECVRF-EDWARDS25519-SHA512-TAI is truncated to be 32 bytes.

14.2. KT Designated Expert Pool

Specification Required [RFC8126] registry requests are registered after a three-week review period on the KT Designated Expert (DE) mailing list kt-reg-review@ietf.org (mailto:kt-reg-review@ietf.org) on the advice of one or more of the KT DEs. However, to allow for the allocation of values prior to publication, the KT DEs may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the KT DEs' mailing list for review SHOULD use an appropriate subject (e.g., "Request to register value in KT registry").

Within the review period, the KT DEs will either approve or deny the registration request, communicating this decision to the KT DEs' mailing list and IANA. Denials SHOULD include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention for resolution using the iesg@ietf.org (mailto:iesg@ietf.org) mailing list.

Criteria that SHOULD be applied by the KT DEs includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or useful only for a single application, and whether the registration description is clear.

IANA MUST only accept registry updates from the KT DEs and SHOULD direct all requests for registration to the KT DEs' mailing list.

It is suggested that multiple KT DEs who are able to represent the perspectives of different applications using this specification be appointed, in order to enable a broadly informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular KT DE, that KT DE SHOULD defer to the judgment of the other KT DEs.

15. References

15.1. Normative References

[ARCH] McMillion, B., "Key Transparency Architecture", Work in Progress, Internet-Draft, draft-ietf-keytrans-architecture-03, 25 February 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-keytrans-architecture-03>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/rfc/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC9381] Goldberg, S., Reyzin, L., Papadopoulos, D., and J. Vト稿lテ。k, "Verifiable Random Functions (VRFs)", RFC 9381, DOI 10.17487/RFC9381, August 2023, <https://www.rfc-editor.org/rfc/rfc9381>.

15.2. Informative References

[CONIKS] Melara, M. S., Blankstein, A., Bonneau, J., Felten, E. W., and M. J. Freedman, "CONIKS: Bringing Key Transparency to End Users", 27 April 2014, <https://eprint.iacr.org/2014/1004>.

[Merkle2] Hu, Y., Hooshmand, K., Kalidhindi, H., Yang, S. J., and R. A. Popa, "Merkle^2: A Low-Latency Transparency Log System", 8 April 2021, <https://eprint.iacr.org/2021/453>.

[OPTIKS] Len, J., Chase, M., Ghosh, E., Laine, K., and R. C. Moreno, "OPTIKS: An Optimized Key Transparency System", 4 October 2023, <https://eprint.iacr.org/2023/1515>.

[SEEMLess] Chase, M., Deshpande, A., Ghosh, E., and H. Malvai, "SEEMless: Secure End-to-End Encrypted Messaging with less trust", 18 June 2018, <https://eprint.iacr.org/2018/607>.

[SHS] "Secure hash standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.180-4, 2015, <https://doi.org/10.6028/nist.fips.180-4>.

Appendix A. Implicit Binary Search Tree

The following Python code demonstrates efficient algorithms for navigating the implicit binary search tree:

```
# The exponent of the largest power of 2 less than x. Equivalent to:
#   int(math.floor(math.log(x, 2)))
def log2(x):
    if x == 0:
        return 0
    k = 0
    while (x >> k) > 0:
        k += 1
    return k-1

# The level of a node in the tree. Leaves are level 0, their parents
# are level 1, etc. If a node's children are at different levels,
# then its level is the max level of its children plus one.
def level(x):
    if x & 0x01 == 0:
        return 0
    k = 0
    while ((x >> k) & 0x01) == 1:
        k += 1
    return k

# The root index of a search if the log has 'n' entries.
def root(n):
    return (1 << log2(n)) - 1

# The left child of an intermediate node.
def left(x):
    k = level(x)
    if k == 0:
        raise Exception('leaf node has no children')
    return x ^ (0x01 << (k - 1))

# The right child of an intermediate node.
def right(x, n):
    k = level(x)
    if k == 0:
        raise Exception('leaf node has no children')
    x = x ^ (0x03 << (k - 1))
    while x >= n:
        x = left(x)
    return x
```

Appendix B. Binary Ladder

The following Python code demonstrates efficient algorithms for computing the versions of a label to include in a binary ladder:

```
   # Returns the set of versions that would be looked up to establish that n was
   # the greatest version of a label that existed.
   def base_binary_ladder(n):
       out = []

       # Output powers of two minus one until reaching a value greater than n.
       while True:
           value = (1 << len(out)) - 1
           out.append(value)
           if value > n:
               break

       # Binary search between the established lower and upper bounds.
       lower_bound = out[-2]
       upper_bound = out[-1]

       while lower_bound+1 < upper_bound:
           value = (lower_bound + upper_bound) // 2
           out.append(value)
           if value <= n:
               lower_bound = value
           else:
               upper_bound = value

       return out

   # Returns the set of versions that would be looked up in a binary ladder for a
   # fixed-version search where the target version is t and the greatest version of
   # the label that exists in a given version of the prefix tree is n.
   def fixed_version_binary_ladder(
       t, n,
       left_inclusion = [], right_non_inclusion = []
   ):
       def would_end(v):
           # (Proof of inclusion for a version greater than or equal to t) OR
           # (Proof of non-inclusion for a version less than t)
           return (v <= n and v >= t) or (v > n and v < t)

       def would_be_duplicate(v):
           return (v <= n and v in left_inclusion) or \
                  (v > n and v in right_non_inclusion)

       out = base_binary_ladder(n)
       end = next((i+1 for i,v in enumerate(out) if would_end(v)), len(out))
       filtered_out = [v for v in out[:end] if not would_be_duplicate(v)]

       return filtered_out
```

```
   # Returns the set of versions that would be looked up in a binary ladder for a
   # monitoring query where the monitored version of the label is t.
   def monitor_binary_ladder(t, left_inclusion = []):
       out = base_binary_ladder(t)
       filtered_out = [v for v in out if v <= t and v not in left_inclusion]

       return filtered_out

   # Returns the set of versions that would be looked up in a binary ladder for a
   # greatest-version search where the greatest version of a label that exists
   # globally is t but the greatest version of the label in a given version of the
   # prefix tree is n.
   def greatest_version_binary_ladder(
       t, n, distinguished,
       left_inclusion = [], right_non_inclusion = [], same_entry = []
   ):
       def would_end(v):
           # Proof of non-inclusion for a version less than t
           return (v > n and v < t)

       def would_be_duplicate(v):
           if distinguished:
               return (v <= n and v in left_inclusion) or \
                      (v > n and v in right_non_inclusion)
           else:
               return v in same_entry

       out = base_binary_ladder(t)
       end = next((i+1 for i,v in enumerate(out) if would_end(v)), len(out))
       filtered_out = [v for v in out[:end] if not would_be_duplicate(v)]

       return filtered_out
```

Authors' Addresses

Brendan McMillion
Email: brendanmcmillion@gmail.com

Felix Linker
Email: linkerfelix@gmail.com