Workgroup: KT Working Group
Internet-Draft:
draft-mcmillion-key-transparency-01
Published: 16 May 2023
Intended Status: Informational
Expires: 17 November 2023
Authors: B. McMillion

Key Transparency

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. In addition to distributing public keys, it can also be applied to ensure that a group of users agree on a shared value or to keep tamper-evident logs of security-critical events.

Discussion Venues

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

Source for this draft and an issue tracker can be found at https://github.com/Bren2010/draft-key-transparency.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 November 2023.

Table of Contents

1. Introduction

Before any information can be exchanged in an end-to-end encrypted system, two things must happen. First, participants in the system must provide to the service operator any public keys they wish to use to receive messages. Second, the service operator must distribute these public keys to any participants that wish to send messages to those users.

Typically this is done by having users upload their public keys to a simple directory where other users can download them as necessary. With this approach, the service operator is trusted to not manipulate the directory by inserting malicious public keys, which means that the underlying encryption protocol can only protect users against passive eavesdropping on their messages.

However most messaging systems are designed such that all messages exchanged between users flow through the service operator's servers, so it's extremely easy for an operator to launch an active attack. That is, the service operator can insert public keys into the directory that they know the private key for, attach those public keys to a user's account without the user's knowledge, and then inject these keys into active conversations with that user to receive plaintext data.

Key Transparency (KT) solves this problem by requiring the service operator to store user public keys in a cryptographically-protected append-only log. Any malicious entries added to such a log will generally be visible to all users, in which case a user can detect that they're being impersonated by viewing the public keys attached to their account. However, if the service operator attempts to conceal some entries of the log from some users but not others, this

creates a "forked view" which is permanent and easily detectable with out-of-band communication.

The critical improvement of KT over related protocols like Certificate Transparency [RFC6962] is that KT includes an efficient protocol to search the log for entries related to a specific participant. This means users don't need to download the entire log, which may be substantial, to find all entries that are relevant to them. It also means that KT can better preserve user privacy by only showing entries of the log to participants that genuinely need to see them.

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.

3. Protocol Overview

From a networking perspective, KT follows a client-server architecture with a central *Transparency Log*, acting as a server, which holds the authoritative copy of all information and exposes endpoints that allow clients to query or modify stored data. Clients coordinate with each other through the server by uploading their own public keys and downloading the public keys of other clients. Clients are expected to maintain relatively little state, limited only to what is required to interact with the log and ensure that it is behaving honestly.

From an application perspective, KT works as a versioned key-value database. Clients insert key-value pairs into the database where, for example, the key is their username and the value is their public key. Clients can update a key by inserting a new version with new data. They can also look up the most recent version of a key or any past version. From this point forward, "key" will refer to a lookup key in a key-value database and "public key" or "private key" will be specified if otherwise.

While this document uses the TLS presentation language [RFC8446] to describe the structure of protocol messages, it does not require the use of a specific transport protocol. This is intended to allow applications to layer KT on top of whatever transport protocol their application already uses. In particular, this allows applications to continue relying on their existing access control system.

Applications may enforce arbitrary access control rules on top of KT such as requiring a user to be logged in to make KT requests, only

allowing a user to lookup the keys of another user if they're "friends", or simply applying a rate limit. Applications **SHOULD** prevent users from modifying keys that they don't own. The exact mechanism for rejecting requests, and possibly explaining the reason for rejection, is left to the application.

Finally, this document does not assume that clients can reliably communicate with each other out-of-band (that is, away from any interference by the Transparency Log operator), or communicate with the Transparency Log anonymously. However, Section 12.1 gives guidance on how these channels can be utilized effectively when or if they're available.

3.1. Basic Operations

The operations that can be executed by a client are as follows:

1. **Search:** Performs a lookup on a specific key in the most recent version of the log. Clients may request either a specific version of the key, or the most recent version available. If the key-version pair exists, the server returns the corresponding value and a proof of inclusion.

2. **Update:** Adds a new key-value pair to the log, for which the server returns a proof of inclusion. Note that this means that new values are added to the log immediately in response to an Update operation, and are not queued for later insertion with a batch of other values.

3. **Monitor:** While Search and Update are run by the client as necessary, monitoring is done in the background on a recurring basis. It both checks that the log is continuing to behave honestly (all previously returned keys remain in the tree) and that no changes have been made to keys owned by the client without the client's knowledge.

3.2. Deployment Modes

In the interest of satisfying the widest range of use-cases possible, three different modes for deploying a Transparency Log are described in this document. Each mode has slightly different requirements and efficiency considerations for both the service operator and the end-user.

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

With both third-party modes, all requests from end-users are initially routed to the service operator and the service operator coordinates with the third party themself. End-users never contact the third party directly, however they will need a signature public key from the third party to verify its assertions.

With Third-party Management, the third party performs the majority of the work of actually storing and operating the log, and the service operator only needs to sign new entries as they're added. With Third-party Auditing, the service operator performs the majority of the work of storing and operating the log, and obtains signatures from a lightweight third-party auditor at regular intervals asserting that the service operator has been constructing the tree correctly.

Contact Monitoring, on the other hand, supports a single-party deployment with no third party. The tradeoff is that executing the background monitoring protocol requires an amount of work that's proportional to the number of keys a user has looked up in the past. As such, it's less suited to use-cases where users look up a large number of ephemeral keys, but would work ideally in a use-case where users look up a small number of keys repeatedly (for example, the keys of regular contacts).

The deployment mode of a Transparency Log is chosen when the log is first created and isn't able to be changed over the log's lifetime. This makes it important for operators to carefully consider the best long-term approach based on the specifics of their application, although migrating from a log operating in one deployment mode to another is possible if it becomes necessary (see Section 12.2).

3.3. Security Guarantees

A client that executes a Search or Update operation correctly (and does any required monitoring afterwards) receives a guarantee that the Transparency Log operator also executed the operation correctly and in a way that's globally consistent with what it has shown all other clients. That is, when a client searches for a key, they're guaranteed that the result they receive represents the same result that any other client searching for the same key would've seen. When a client updates a key, they're guaranteed that other clients will see the update the next time they search for the key.

If the Transparency Log operator does not execute an operation correctly, then either:

1. The client will detect the error immediately and reject the result of an operation, or

2. The client will permanently enter an invalid state.

Depending on the exact reason that the client enters an invalid state, it will either be detected by background monitoring or the next time that out-of-band communication is available. Importantly, this means that clients must stay online for some fixed amount of time after entering an invalid state for it to be successfully detected.

The exact caveats of the above guarantee depend naturally on the security of underlying cryptographic primitives, but also the deployment mode that the Transparency Log relies on:

* Third-Party Management and Third-Party Auditing require an assumption that the service operator and the third-party manager/auditor do not collude to trick clients into accepting malicious results.

* Contact Monitoring requires an assumption that the client that owns a key and all clients that look up the key do the necessary monitoring afterwards.

4. Tree Construction

KT relies on two combined hash tree structures: log trees and prefix trees. This section describes the operation of both at a high level and the way that they're combined. More precise algorithms for computing the intermediate and root values of the trees are given in Section 7.

Both types of 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 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.

4.1. Log Tree

Log trees are used for storing 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.

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.

While leaves contain arbitrary data, the value of a parent node is always the hash of the combined values of its left and right children.

Log trees are special 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 past version of the log.

An inclusion proof is given by providing the copath values of a leaf. The proof is verified by hashing together the leaf with the copath values and checking that the result equals the root value of the log. Consistency proofs are a more general version of the same idea. With a consistency proof, the prover provides the minimum set of intermediate node values from the current tree that allows the verifier to compute both the old root value and the current root value. An algorithm for this is given in section 2.1.2 of [RFC6962].

4.2. Prefix Tree

Prefix trees are used for storing key-value pairs while preserving the ability to efficiently look up a value by its corresponding key.

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

Every key stored in the tree is required to have the same length in bits, which allows every leaf node to exist at the same level of the tree (that is, every leaf has a direct path that's the same length). This effectively prevents users from being able to infer the total number of key-value pairs stored in the tree.

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 the leaf node for the desired key, or a parent node that lacks the desired child.

New key-value pairs are added to the tree by searching it according to this process. If the search terminates at a parent without a left or right child, the parent's missing child is replaced with a series of intermediate nodes for each remaining bit of the search key, followed by a new leaf. If the search terminates at the leaf corresponding to the search key (indicating that this search key already has a value in the tree), the old leaf value is simply replaced with a new one.

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).

4.3. Combined Tree

Log trees are desirable because they can provide efficient consistency proofs to assure 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 keys added.

In the combined tree structure, which is based on [Merkle2], a log tree maintains a record of updates to key-value pairs while a prefix tree maintains a map from each key to a pair of integers: a counter with the number of times the key has been updated, and the position in the log of the first instance of the key. Importantly, the root value of the prefix tree after adding the new key or updating the counter/position pair of an existing key, is stored in the log tree alongside the record of the update. With some caveats, this combined structure supports both efficient consistency proofs and can efficiently authenticate searches.

To search the combined structure, the server first provides the user with the position of the first instance of the key in the log. The user then follows a binary search for the log entry where looking up the search key in the prefix tree at that entry yields the desired version counter. As such, the entry that a user arrives at through binary search contains the update with the key-value pair that the user is looking for, even though the log itself is not sorted.

Providing the position of the first instance of the key in the log is necessary because the prefix tree structure used isn't able to provide proofs of non-inclusion (which would leak information about the number of keys stored in the prefix tree). Without proofs of non-inclusion, users aren't able to lookup the same key in any version of the prefix tree -- only versions of the prefix tree that were created after the key was initially added to the log. Because the server provides this position, users are able to restrict their binary search to only touching log entries where the search key can be successfully looked up in the prefix tree.

Following a binary search also ensures that all users will check the same or similar entries when searching for the same key, which is necessary for the efficient auditing of a Transparency Log. To maximize this effect, users rely on an implicit binary tree structure constructed over the leaves of the log tree (distinct from the structure of the log tree itself).

4.3.1. Implicit Binary Search Tree

Intuitively, the leaves of the log tree can be considered a flat array representation of a left-balanced binary tree. In this representation, "leaf" nodes are stored in even-numbered indices, while "intermediate" nodes are stored in odd-numbered indices:

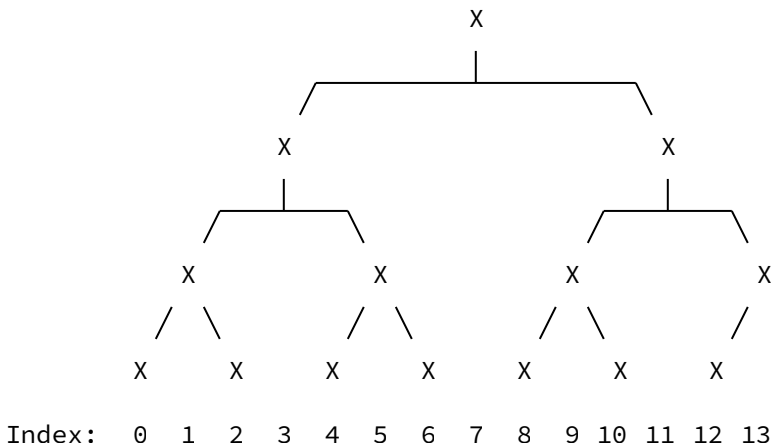

Figure 1: A binary tree constructed from 14 entries in a log

Following the structure of this binary tree when executing searches makes auditing the Transparency Log much more efficient because users can easily reason about which nodes will be accessed when conducting a search. As such, only nodes along a specific search path need to be checked for correctness.

The following Python code demonstrates the computations used for following this tree structure:

```
# 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

def left_step(x):
    k = level(x)
    if k == 0:
        raise Exception('leaf node has no children')
    return x ^ (0x01 << (k - 1))

def right_step(x):
    k = level(x)
    if k == 0:
        raise Exception('leaf node has no children')
    return x ^ (0x03 << (k - 1))

def move_within(x, start, n):
    while x < start or x >= n:
        if x < start: x = right_step(x)
        else: x = left_step(x)
    return x

# The root index of a search, if the first instance of a key is at
# `start` and the log has `n` entries.
def root(start, n):
    return move_within((1 << log2(n)) - 1, start, n)

# The left child of an intermediate node.
def left(x, start, n):
    return move_within(left_step(x), start, n)

# The right child of an intermediate node.
```

```
def right(x, start, n):
    return move_within(right_step(x), start, n)
```

The root function returns the index in the log at which a search should start. The left and right functions determine the subsequent index to be accessed, depending on whether the search moves left or right.

For example, in a search where the first instance of the key is at index 10 and the log has 60 entries, instead of starting the search at the typical "middle" entry of 10+60/2 = 35, users would start at entry root(10, 60) = 31. If the next step in the search is to move right, the next index to access would be right(31, 10, 60) = 47. As more entries are added to the log, users will consistently revisit entries 31 and 47, while they may never revisit entry 35 after even a single new entry is added to the log.

Additionally, while users searching for a specific version of a key can jump right into a binary search for the entry with that counter, other users may instead wish to search for the "most recent" version of a key. That is, the key with the highest counter possible. Users looking up the most recent version of a key start by fetching the **frontier**, which they use to determine what the highest counter for a key is.

The frontier consists of the root node of a search, followed by the entries produced by repeatedly calling right until reaching the last entry of the log. Using the same example of a search where the first instance of a key is at index 10 and the log has 60 entries, the frontier would be entries: 31, 47, 55, 59.

If we can assume that the log operator is behaving honestly, then checking only the last entry of the log would be sufficient to find the most recent version of any key. However, we can't assume this. Checking each entry along the frontier is functionally the same as checking only the last entry, but also allows the user to verify that the entire search path leading to the last entry is constructed correctly.

4.3.2. Monitoring

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

To monitor a given search key, users maintain a small amount of state: a map from a version counter, to an entry in the log where

looking up the search key in the prefix tree at that entry yields the given version. Users initially populate this map by setting a version of the search key that they've looked up, to map to the entry in the log where that version of the key is stored. A map may track several different versions of a search key simultaneously, if a user has been shown different versions of the same search key.

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

1. Compute the entry's direct path based on the current tree size.

2. If there are no entries in the direct path that are to the right of the current node, then skip updating this entry (there's no new information to update it with).

3. For each entry in the direct path that's to the right of the current node, from low to high:

    1. Obtain a proof from the server that the prefix tree at that entry maps the search key to a version counter that's greater than or equal to the current version.

    2. If the above check was successful, remove the current version-node pair from the map and replace it with a version-node pair corresponding to the entry in the log that was just checked.

This algorithm progressively moves up the tree as new intermediate/root nodes are established and verifies that they're constructed correctly. Note that users can often execute this process with the output of Search or Update operations for a key, without waiting to make explicit Monitor queries.

It is also worth noting that the work required to monitor several versions of the same key scales sublinearly, due to the fact that 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 key from then on.

Once a user has finished updating their monitoring map with the algorithm above, all nodes in the map should lie on the frontier of the log. For all the remaining nodes of the frontier, users request proofs from the server that the prefix trees at those entries are also constructed correctly. That is, that they map the search key to a version counter that's greater than or equal to what would be expected. Rather than checking the version counter, the primary purpose of these checks is to demonstrate that the position field in each prefix tree has been set correctly.

5. Preserving Privacy

In addition to being more convenient for many use-cases than similar transparency protocols, KT is also better at preserving the privacy of a Transparency Log's contents. This is important because in many practical applications of KT, service operators expect to be able to control when sensitive information is revealed. In particular, an operator can often only reveal that a user is a member of their service to that user's friends or contacts. Operators may also wish to conceal when individual users perform a given task like rotate their public key or add a new device to their account, or even conceal the exact number of users their application has overall.

Applications are primarily able to manage the privacy of their data in KT by enforcing access control policies on the basic operations performed by clients, as discussed in Section 3. However, the proofs given by a Transparency Log can indirectly leak information about other entries and lookup keys.

When users search for a key with the binary search algorithm described in Section 4.3, they necessarily see the values of several leaves while conducting their search that they may not be authorized to view the contents of. However, log entries generally don't need to be inspected except as specifically allowed by the service.

The privacy of log entries is maintained by storing only a cryptographic commitment to the serialized, updated key-value pair in the leaf of the log tree instead of the update itself. At the end of a successful search, the service operator provides the committed update along with the commitment opening, which allows the user to verify that the commitment in the log tree really does correspond to the provided update. By logging commitments instead of plaintext updates, users learn no information about an entry's contents unless the service operator explicitly provides the commitment opening.

Beyond the log tree, the second potential source of privacy leaks is the prefix tree. When receiving proofs of inclusion from the prefix tree, users also receive indirect information about what other valid lookup keys exist. To prevent this, all lookup keys are processed through a Verifiable Random Function, or VRF [I-D.irtf-cfrg-vrf].

A VRF deterministically maps each key to a fixed-length pseudorandom value. The VRF can only be executed by the service operator, who holds a private key. But critically, VRFs can still provide a proof that an input-output pair is valid, which users verify with a public key. When a user requests to search for or update a key, the service operator first executes its VRF on the input key to obtain the output key that will actually be looked up or stored in the prefix

tree. The service operator then provides the output key, along with a proof that the output key is correct, in its response to the user.

The pseudorandom output of VRFs means that even if a user indirectly observes that a search key exists in the prefix tree, they can't immediately learn which user the search key identifies. The inability of users to execute the VRF themselves also prevents offline "password cracking" approaches, where an attacker tries all possibilities in a low entropy space (like the set of phone numbers) to find the input that produces a given search key.

6. Ciphersuites

Each Transparency Log uses a single fixed ciphersuite, chosen when the log is initially created, that specifies the following primitives to be used for cryptographic computations:

- *A hash algorithm
- *A signature algorithm
- *A Verifiable Random Function (VRF) algorithm

The hash algorithm is used for computing the 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 lookup keys. One of the VRF algorithms from [I-D.irtf-cfrg-vrf] must be used.

Ciphersuites are represented with the CipherSuite type. The ciphersuites are defined in Section 14.1.

7. Cryptographic Computations

7.1. Commitment

As discussed in Section 5, commitments are stored in the leaves of the log tree and correspond to updated key-value pairs. Commitments are computed with HMAC [RFC2104], using the hash function specified by the ciphersuite. To produce a new commitment, the application generates a random 16 byte value called opening and computes:

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

where fixedKey is the 16 byte hex-decoded value:

```
d821f8790d97709796b4d7903357c3f5
```

and CommitmentValue is specified as:

```
struct {
  opaque opening<16>;
  opaque search_key<0..2^8-1>;
  UpdateValue update;
} CommitmentValue;
```

This fixed key allows the HMAC function, and thereby the commitment scheme, to be modeled as a random oracle. The search_key field of CommitmentValue contains the search key being updated (the search key provided by the user, not the VRF output) and the update field contains the value of the update.

The output value commitment may be published, while opening should be kept private until the commitment is meant to be revealed.

7.2. Prefix Tree

The leaf nodes of a prefix tree are serialized as:

```
struct {
    opaque key<VRF.Nh>;
    uint32 counter;
    uint64 position;
} PrefixLeaf;
```

where key is the VRF-output search key, counter is the counter of times that the key has been updated (starting at 0 for a key that was just created), position is the position in the log of the first occurrence of this key, and VRF.Nh is the output size of the ciphersuite VRF in bytes.

The parent nodes of a prefix tree are serialized as:

```
struct {
  opaque value<Hash.Nh>;
} PrefixParent;
```

where Hash.Nh is the output length of the ciphersuite hash function. The value of a parent node is computed by hashing together the values of its left and right children:

```
parent.value = Hash(0x01 ||
                    nodeValue(parent.leftChild) ||
                    nodeValue(parent.rightChild))

nodeValue(node):
  if node.type == emptyNode:
    return standIn(seed, counter)
  else if node.type == leafNode:
    return Hash(0x00 || node.key || node.counter || node.position)
  else if node.type == parentNode:
    return node.value
```

where Hash denotes the ciphersuite hash function. Whenever a parent's left or right child is missing, a stand-in value is computed from a random seed. The stand-in value is computed as:

```
standIn(seed, counter):
  return Hash(0x02 || seed || counter)
```

The seed value is a randomly sampled byte string of 16 bytes and the counter is an 8-bit integer. The counter starts at zero and increases by one for each subsequent stand-in value that's needed, counting from the root down.

7.3. Log Tree

The leaf and parent nodes of a log tree are serialized as:

```
struct {
  opaque commitment<Hash.Nh>;
  opaque prefix_tree<Hash.Nh>;
} LogLeaf;

struct {
  opaque value<Hash.Nh>;
} LogParent;
```

The value of a parent node 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 || nodeValue(node)
  else if node.type == parentNode:
    return 0x01 || nodeValue(node)

nodeValue(node):
  if node.type == leafNode:
    return Hash(node.commitment || node.prefix_tree)
  else if node.type == parentNode:
    return node.value
```

7.4. Tree Head Signature

The head of a Transparency Log, which represents the log's most recent state, is represented as:

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

where tree_size counts the number of entries in the log tree and timestamp is the time that the structure was generated in milliseconds since the Unix epoch. 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:
      opaque auditor_public_key<0..2^16-1>;
  };
} Configuration;

struct {
  Configuration config;
  uint64 tree_size;
  uint64 timestamp;
  opaque root_value<Hash.Nh>;
} TreeHeadTBS;
```

8. Tree Proofs

8.1. Log Tree

An inclusion proof for a single leaf in a log tree is given by providing the copath values of a leaf. Similarly, a bulk inclusion proof for any number of leaves is given by providing the fewest node values that can be hashed together with the specified leaves to produce the root value. Such a proof is encoded as:

```
opaque NodeValue<Hash.Nh>;

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

Each NodeValue is a uniform size, computed by passing the relevant LogLeaf or LogParent structures through the nodeValue function in Section 7.3. Finally, 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.

Consistency proofs are encoded similarly:

```
struct {
  NodeValue elements<0..2^8-1>;
} ConsistencyProof;
```

Again, each NodeValue is computed by passing the relevant LogLeaf or LogParent structure through the nodeValue function. The nodes chosen correspond to those output by the algorithm in Section 2.1.2 of [RFC6962].

8.2. 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:

```
struct {
  NodeValue elements<8*VRF.Nh>;
  uint32 counter;
} PrefixProof;
```

The elements array consists of the copath of the leaf node, in bottom-to-top order. That is, the leaf's sibling would be first, followed by the leaf's parent's sibling, and so on. In the event that a node is not present, then the random value generated when computing the parent's value is provided instead.

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

The position field of the PrefixLeaf structure isn't provided in PrefixProof to save space, as this value is expected to be the same across several proofs.

8.3. Combined Tree

A proof from a combined log and prefix tree follows the execution of a binary search through the leaves of the log tree, as described in Section 4.3. It is serialized as follows:

```
struct {
  PrefixProof prefix_proof;
  opaque commitment<Hash.Nh>;
} SearchStep;

struct {
  uint64 position;
  SearchStep steps<0..2^8-1>;
  InclusionProof inclusion;
} SearchProof;
```

Each SearchStep structure in steps is one leaf that was inspected as part of the binary search. The steps of the binary search are determined by starting with the "middle" leaf (according to the root function in Section 4.3.1), which represents the first node touched by the search. From there, the user moves incrementally left or right, based on the version counter found in each previous step.

The prefix_proof field of a SearchStep is the output of searching the prefix tree whose root is at that leaf for the search key, while the commitment field is the commitment to the update at that leaf. The inclusion field of SearchProof contains a batch inclusion proof for all of the leaves accessed by the binary search, relating them to the root of the log tree.

The proof can be verified by checking that:

1. The elements of steps represent a monotonic series over the leaves of the log, and

2. The steps array has the expected number of entries (no more or less than are necessary to execute the binary search).

Once the validity of the search steps has been established, the verifier can compute the root of each prefix tree represented by a prefix_proof and combine it with the corresponding commitment to obtain the value of each leaf. These leaf values can then be combined with the proof in inclusion to check that the output matches the root of the log tree.

9. Update Format

The updates committed to by a combined tree structure contain the new value of a search key, along with additional information depending on the deployment mode of the Transparency Log. They are 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 new value of the search key.

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

The search_key field contains the search key being updated (the search key provided by the user, not the VRF output), version contains the new key version, and value contains the same contents as UpdateValue.value. Clients **MUST** successfully verify this signature before consuming UpdateValue.value.

10. User Operations

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

10.1. Search

Users initiate a Search operation by submitting a SearchRequest to the Transparency Log containing the key that they're interested in. Users can optionally specify a version of the key that they'd like to receive, if not the most recent one. They can also include the tree_size of the last TreeHead that they successfully verified.

```
struct {
  opaque search_key<0..2^8-1>;
  optional<uint32> version;
  optional<uint64> last;
} SearchRequest;
```

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

```
struct {
  TreeHead tree_head;
  optional<ConsistencyProof> consistency;
  select (Configuration.mode) {
    case thirdPartyAuditing:
      AuditorTreeHead auditor_tree_head;
  };
} FullTreeHead;

struct {
  opaque index<VRF.Nh>;
  opaque proof<0..2^16-1>;
} VRFResult;

struct {
  FullTreeHead full_tree_head;
  VRFResult vrf_result;
  SearchProof search;

  opaque opening<16>;
  UpdateValue value;
} SearchResponse;
```

If last is present, then the Transparency Log **MUST** provide a consistency proof between the current tree and the tree when it was this size, in the consistency field of FullTreeHead.

Users verify a search response by following these steps:

1. Verify the VRF proof in VRFResult.proof against the requested search key SearchRequest.search_key and the claimed VRF output VRFResult.index.

2. Evaluate the search proof in search according to the steps in Section 8.3. This will produce a verdict as to whether the search was executed correctly, and also a candidate root value for the tree. If it's determined that the search was executed incorrectly, abort with an error.

3. If the user has monitoring information for this search key (because they own it or are performing Contact Monitoring),

verify that SearchProof.position is the same as in previous requests, and that the entry's version and position in the log are consistent with other known versions.

4. With the candidate root value for the tree:

    1. Verify the proof in FullTreeHead.consistency, if one is expected.

    2. Verify the signature in TreeHead.signature.

    3. Verify that the timestamp in TreeHead is sufficiently recent. Additionally, verify that the timestamp and tree_size fields of the TreeHead are greater than or equal to what they were before.

    4. If third-party auditing is used, verify auditor_tree_head with the steps described in Section 11.2.

5. Verify that the commitment in the terminal search step 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, before its contents may be consumed.

To be able to later perform monitoring, users retain the claimed position of the key's first occurrence in the log, SearchProof.position. They also retain, for each version of the key observed, the version number and its position in the log. Users **MUST** retain this information if the Transparency Log's deployment mode is Contact Monitoring, and they **SHOULD** retain the entire SearchResponse structure to assist with debugging or to provide non-repudiable proof if misbehavior is detected. If one of the third-party modes is being used, users **MAY** retain this information to perform Contact Monitoring even though it is not required.

10.2. Update

Users initiate an Update operation by submitting an UpdateRequest to the Transparency Log containing the new key and value to store. Users can also optionally include the tree_size of the last TreeHead that they successfully verified.

```
struct {
  opaque search_key<0..2^8-1>;
  opaque value<0..2^32-1>;
  optional<uint64> last;
} UpdateRequest;
```

If the request is acceptable by application-layer policies, the Transparency Log adds the new key-value pair to the log and returns an UpdateResponse structure:

```
struct {
  FullTreeHead full_tree_head;
  VRFResult vrf_result;
  SearchProof search;

  opaque opening<16>;
  UpdatePrefix prefix;
} UpdateResponse;
```

Users verify the UpdateResponse as if it were a SearchResponse for the most recent version of search_key, and they also check that their update is the last entry in the log. To aid verification, the update response provides the UpdatePrefix structure necessary to reconstruct the UpdateValue.

Users **MUST** retain the information required to perform monitoring as described in Section 10.1.

10.3. Monitor

Users initiate a Monitor operation by submitting a MonitorRequest to the Transparency Log containing information about the keys they wish to monitor. Similar to Search and Update operations, users can include the tree_size of the last TreeHead that they successfully verified.

```
struct {
  opaque search_key<0..2^8-1>;
  uint64 entries<0..2^8-1>;
} MonitorKey;

struct {
  MonitorKey owned_keys<0..2^8-1>;
  MonitorKey contact_keys<0..2^8-1>;
  optional<uint64> last;
} MonitorRequest;
```

Users include each of the keys that they own in owned_keys. If the Transparency Log is deployed with Contact Monitoring (or simply if the user wants a higher degree of confidence in the log), they also include any keys they've looked up in contact_keys.

Each MonitorKey structure contains the key being monitored in search_key, and a list of entries in the log tree corresponding to the values of the map described in Section 4.3.2.

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

1. Verify that the requested keys in owned_keys and contact_keys are all distinct.

2. Verify that the user owns every key in owned_keys, and is allowed to lookup every key in contact_keys, based on the application's policy.

3. Verify that each entries array is sorted in ascending order.

4. Verify that the entries in each entries array are all between the initial position of the requested key and the end of the log.

5. Verify each entry lies on the direct path of different versions of the key.

If the request is valid, the Transparency Log responds with a MonitorResponse structure:

```
struct {
  PrefixProof prefix_proof;
  opaque commitment<Hash.Nh>;
} MonitorProofStep;

struct {
  MonitorProofStep steps<0..2^8-1>;
  InclusionProof inclusion;
} MonitorProof;

struct {
  FullTreeHead full_tree_head;
  MonitorProof owned_proofs<0..2^8-1>;
  MonitorProof contact_proofs<0..2^8-1>;
} MonitorResponse;
```

The elements of owned_proofs and contact_proofs correspond one-to-one with the elements of owned_keys and contact_keys. Each MonitorProof is meant to convince the user that the key they looked up is still properly included in the log and has not been surreptitiously concealed.

The steps of a MonitorProof consist of the proofs required to update the user's monitoring data following the algorithm in Section 4.3.2, including proofs along the current frontier of the log. The steps are provided in the order that they're consumed by the monitoring algorithm. If same proof is consumed by the monitoring algorithm multiple times, it is provided in the MonitorProof structure only

the first time. Proofs along the frontier are provided from left to right, excluding any proofs that have already been provided, and excluding any entries of the frontier which are to the left of the leftmost entry being monitored.

Users verify a MonitorResponse by following these steps:

1. Verify that the lengths of owned_proofs and contact_proofs are the same as the lengths of owned_keys and contact_keys.

2. For each MonitorProof structure:

    1. Evalute the monitoring algorithm in Section 4.3.2. Abort with an error if the monitoring algorithm detects that the tree is constructed incorrectly, or if there are fewer or more steps provided than would be expected.

    2. Construct a candidate root value for the tree by combining the PrefixProof and commitment of each step, with the provided inclusion proof.

3. Verify that all of the candidate root values are the same. With the candidate root value:

    1. Verify the proof in FullTreeHead.consistency, if one is expected.

    2. Verify the signature in TreeHead.signature.

    3. Verify that the timestamp in TreeHead is sufficiently recent. Additionally, verify that the timestamp and tree_size fields of the TreeHead are greater than or equal to what they were before.

    4. If third-party auditing is used, verify auditor_tree_head with the steps described in Section 11.2.

Some information is omitted from MonitorResponse in the interest of efficiency, due to the fact that the user would have already seen and verified it as part of conducting other queries. In particular, the VRF output and proof for each search key is not provided, or each key's initial position in the log, given that both of these can be cached from the original Search or Update query for the key.

10.4. Distinguished

Users can request distinguished tree heads by submitting a DistinguishedRequest to the Transparency Log containing the approximate timestamp of the tree head they'd like to receive.

```
struct {
  uint64 timestamp;
  optional<uint64> last;
} DistinguishedRequest;
```

In turn, the Transparency Log responds with a DistinguishedResponse structure containing the FullTreeHead with the timestamp closest to what the user requested and the root hash of the tree at this point.

```
struct {
  FullTreeHead full_tree_head;
  opaque root<Hash.Nh>;
} DistinguishedResponse;
```

If last is present, then the Transparency Log **MUST** provide a consistency proof between the provided tree head and the tree when it had last entries, in the consistency field of FullTreeHead. Unlike the other operations described in this section, where last is always less than or equal to the tree_size in the provided FullTreeHead, a DistinguishedResponse may contain a FullTreeHead which comes either before or after last.

Users verify a response by following these steps:

1. Verify the proof in FullTreeHead.consistency, if one is expected.

2. Verify the signature in TreeHead.signature.

3. Verify that the timestamp and tree_size fields of the TreeHead are consistent with the previously held TreeHead.

4. If third-party auditing is used, verify auditor_tree_head with the steps described in Section 11.2.

11. Third Parties

11.1. Management

With the Third-party Management deployment mode, a third party is responsible for the majority of the work of storing and operating the log, while the service operator serves mainly to enforce access control and authenticate the addition of new entries to the log. All user queries specified in Section 10 are initially sent by users directly to the service operator, and the service operator proxies them to the third-party manager if they pass access control.

The service operator only maintains one private key that is kept secret from the third-party manager, which is the private key

corresponding to Configuration.leaf_public_key. This private key is used to sign new entries before they're added to the log.

As such, all requests and their corresponding responses from Section 10 are proxied between the user and the third-party manager unchanged with the exception of UpdateRequest, which needs to carry the service operator's signature over the update:

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

The signature is computed over the UpdateTBS structure from Section 9. The service operator **MUST** maintain its own records (independent of the third-party manager) for the most recent version of each key, for the purpose of producing this signature. The service operator **SHOULD** also attempt to proactively detect forks presented by the third-party manager.

11.2. Auditing

With the Third-party Auditing deployment mode, the service operator obtains signatures from a lightweight 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 for their queries.

The third-party auditor is expected to run asynchronously, downloading and authenticating a log's contents in the background, so as not to become a bottleneck for the service operator. This means that the signatures from the auditor will usually be somewhat delayed. Applications **MUST** specify a maximum amount of time after which an auditor signature will no longer be accepted. It **MUST** also specify a maximum number of entries that an auditor's signature may be behind the most recent TreeHead before it will no longer be accepted. Both of these parameters **SHOULD** be small relative to the log's normal operating scale so that misbehavior can be detected quickly.

Failing to verify an auditor's signature in a query **MUST** result in an error that prevents the query's response from being consumed or accepted by the application.

The service operator submits updates to the auditor in batches, in the order that they were added to the log tree:

```
enum {
  reserved(0),
  real(1),
  fake(2),
  (255)
} AuditorUpdateType;

struct {
  AuditorUpdateType update_type;
  opaque index<VRF.Nh>;
  opaque seed<16>;
  opaque commitment<Hash.Nh>;
} AuditorUpdate;

struct {
  AuditorUpdate updates<0..2^16-1>;
} AuditorRequest;
```

The update_type field of each AuditorUpdate specifies whether the update was real or fake (see Section 12.3). Real updates genuinely affect a leaf node of the prefix tree, while fake updates only change the random stand-in value for a non-existent child. The index field contains the VRF output of the search key that was updated, seed contains the seed used to compute new random stand-in values for non-existent children in the prefix tree, and commitment contains the service provider's commitment to the update. The auditor responds with:

```
struct {
  TreeHead tree_head;
} AuditorResponse;
```

The tree_head field contains a signature from the auditor's private key, corresponding to Configuration.auditor_public_key, over the serialized TreeHeadTBS structure. The tree_size field of the TreeHead is equal to the number of entries processed by the auditor and the timestamp field is set to the time the signature was produced (in milliseconds since the Unix epoch).

The auditor TreeHead from this response is provided to users wrapped in the following struct:

```
struct {
  TreeHead tree_head;
  opaque root_value<Hash.Nh>;
  ConsistencyProof consistency;
} AuditorTreeHead;
```

The root_value field contains the root hash of the tree at the point that the signature was produced and consistency contains a

consistency proof between the tree at this point and the most recent TreeHead provided by the service operator.

To check that an AuditorTreeHead structure is valid, users follow these steps:

1. Verify the signature in TreeHead.signature.

2. Verify that TreeHead.timestamp is sufficiently recent.

3. Verify that TreeHead.tree_size is sufficiently close to the most recent tree head from the service operator.

4. Verify the consistency proof consistency between this tree head and the most recent tree head from the service operator.

12. Operational Considerations

12.1. Detecting Forks

It is sometimes possible for a Transparency Log to present forked views of data to different users. This means that, from an individual user's perspective, a log may appear to be operating correctly in the sense that all of a user's Monitor operations succeed. However, the Transparency Log has presented a view to the user that's not globally consistent with what it has shown other users. As such, the log may be able to associate data with keys without the key owner's awareness.

The protocol is designed such that users always remember the last TreeHead that they observed when querying the log, and require subsequent queries to prove consistency against this tree head. As such, users always stay on an individually-consistent view of the log. If a user is ever presented with a forked view, they hold on to this forked view forever and reject the output of any subsequent queries that are inconsistent with it.

This provides ample opportunity for users to detect when a fork has been presented, but isn't in itself sufficient for detection. To detect forks, users must either use **out-of-band communication** with other users or **anonymous communication** with the Transparency Log.

With out-of-band communication, a user obtains a "distinguished" TreeHead that was issued closest to a given time, like the start of the day, by sending a Distinguished request to the Transparency Log. The user then sends the TreeHead along with the root hash that it verifies against to other users over some out-of-band communication channel (for example, an in-app screen with a QR code / scanner). The other users check that the TreeHead verifies successfully and matches their own view of the log. If the TreeHead verifies

successfully on its own but doesn't match a user's view of the log, this proves the existence of a fork.

With anonymous communication, a user first obtains a "distinguished" TreeHead by sending a Distinguished request to the Transparency Log over their normal communication channel. They then send the same Distinguished request, omitting any identifying information and leaving the last field empty, over an anonymous channel. If the log responds with a different TreeHead over the anonymous channel, this proves the existence of a fork.

In the event that a fork is successfully detected, the two signatures on the differing views of the log provide non-repudiable proof of log misbehavior which can be published.

12.2. Combining Multiple Logs

There are some cases where it may make sense to operate multiple cooperating log instances. For example, a service provider may decide that it's prudent to migrate to a new deployment mode. They can do this by creating a new log instance operating under the new deployment mode, and gradually migrating their data from the old log to the new log while users are able to query both. In another case, a service provider may choose to operate multiple logs to improve their ability to scale or to provide higher availability. Similarly, a federated system may allow each party in the federation to operate their own log for their own users.

When this happens, all users in the system **MUST** have a consistent policy for executing Search, Update, and Monitor queries against the multiple logs that maintains the high-level security guarantees of KT:

* If all logs behave honestly, then users observe a globally-consistent view of the data associated with each key.

* If any log behaves dishonestly such that the prior guarantee is not met (some users observe data associated with a key that others do not), this will be detected either immediately or in a timely manner by background monitoring.

In the specific case of migrating from an old log to a new one, this policy may look like:

1. Search queries should be executed against the old log first, and then against the new log only if the most recent version of a key in the old log is a tombstone.

2. Update queries should only be executed against the new log, adding a tombstone entry to the old log if one hasn't been already created.

3. Both logs should be monitored as they would be if they were run individually. Once the migration has completed and the old log has stopped accepting changes, the old log **SHOULD** stay operational long enough for all users to complete their monitoring of it (keeping in mind that some users may be offline for a significant amount of time).

Placing a tombstone entry for each key in the old log gives users a clear indication as to which log contains the most recent version of a key and prevents them from incorrectly accepting a stale version if the new log rejects a search query.

12.3. Obscuring Update Rate

While the protocol already prevents outside observers from determining the total number of key-value pairs stored by a server, some applications may also wish to obscure the frequency of updates. Revealing the frequency of updates may make it possible to deduce the total size of the tree, or it may expose sensitive information about an application's usage patterns. However, fully hiding the frequency of updates is impossible with any hash-based KT construction. Instead, an application may pad real updates with "fake" random updates, such that the update rate measured by observers is fixed to an arbitrary upper-bound value.

The service provider produces a fake update by first choosing three random values: one to represent the VRF output of the key being updated, one to represent the commitment to the update, and one which will be the seed for generating a new stand-in value in the prefix tree. It then traverses the prefix tree according to the random VRF output, and replaces the first stand-in value it reaches with the one generated from the chosen seed. Note that this means that fake updates don't affect a leaf of the prefix tree. Finally, the service provider adds a new entry to the log tree with the random commitment value and the updated prefix tree root.

The VRF output and commitment value can be chosen randomly, instead of being computed with the actual VRF or commitment scheme, because the server will never be required to actually open either of these values. No legitimate search for a key will ever terminate at this entry in the log.

13. Security Considerations

While providing a formal security proof is outside the scope of this document, this section attempts to explain the intuition behind the security of each deployment mode.

13.1. Contact Monitoring

Contact Monitoring works by splitting the monitoring burden between both the owner of a key and those that look it up. Stated as simply as possible, the monitoring obligations of each party are:

1. The key owner, on a regular basis, searches for the most recent version of the key in the log. They verify that this search results in the expected version of the key, at the expected position in the log.

2. The user that looks up a key, whenever a new parent is established on the key's direct path, searches for the key in the prefix tree stored in this new parent. They verify that the version counter returned is greater than or equal to the expected version.

To understand why this is secure, we look at what happens when the service operator tampers with the log in different ways.

First, say that the service operator attempts to cover up the latest version of a key, with the goal of causing a "most recent version" search for the key to resolve in a lower version. To do this, the service operator must add a parent over the latest version of the key with a prefix tree that contains an incorrect version counter. Left unchanged, the key owner will observe that the most recent version of their key is no longer available the next time they perform monitoring. Alternatively, the service operator could add the new version of the key back at a later position in the log. But even so, the key owner will observe that the key's position has changed the next time they perform monitoring. The service operator is unable to restore the latest version of the key without violating the log's append-only property or presenting a forked view of the log to different users.

Second, say that the service operator attempts to present a fake new version of a key, with the goal of causing a "most recent version" search for the key to resolve to the fake version. To do this, the service operator can simply add the new version of the key as the most recent entry to the log, with the next highest version counter. Left unchanged, or if the log continues to be constructed correctly, the key owner will observe that a new version of their key has been added without their permission the next time they perform monitoring. Alternatively, the service operator can add a parent

over the fake version with an incorrect version counter to attempt to conceal the existence of the fake entry. However, the user that previously consumed the fake version of the key will detect this attempt at concealment the next time they perform monitoring.

13.2. Third-party Management

Third-party Management works by separating the construction of the log from the ability to approve which new entries are added to the log, such that tricking users into accepting malicious data requires the collusion of both parties.

The service operator maintains a private key that signs new entries before they're added to the log, which means that it has the ability to sign malicious new entries and have them successfully published. However, without the collusion of the third-party manager to later conceal those entries by constructing the tree incorrectly, their existence will be apparent to the key owner the next time they perform monitoring.

Similarly, while the third-party manager has the ability to construct the tree incorrectly, it cannot add new entries on its own without the collusion of the service operator. Without access to the service operator's signing key, the third-party manager can only attempt to selectively conceal the latest version of a key from certain users. However, as discussed in Section 13.1, this is also apparent to the key owner through monitoring.

13.3. Third-party Auditing

Third-party Auditing works by requiring users to verify a signature from a third-party auditor attesting to the fact that the service operator has been constructing the tree correctly.

While the service operator can still construct the tree incorrectly and temporarily trick users into accepting malicious data, an honest auditor will no longer provide its signatures over the tree at this point. Once there are no longer any sufficiently recent auditor tree roots, the log will become non-functional as the service operator won't be able to produce any query responses that would be accepted by users.

14. IANA Considerations

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

*KT Ciphersuites (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 Ciphersuites

```
uint16 CipherSuite;
```

TODO

14.2. KT Designated Expert Pool

TODO

15. References

15.1. Normative References

[I-D.irtf-cfrg-vrf] Goldberg, S., Reyzin, L., Papadopoulos, D., and J. Včelák, "Verifiable Random Functions (VRFs)", Work in Progress, Internet-Draft, draft-irtf-cfrg-vrf-15, 9 August 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-15>.

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

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC Editor report, DOI 10.17487/rfc6962, June 2013, <https://doi.org/10.17487/rfc6962>.

[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>.

15.2. Informative References

[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>.

Acknowledgments

TODO acknowledge.

Author's Address

Brendan McMillion

Email: brendanmcmillion@gmail.com