Key Transparency
Internet-Draft
Intended status: Informational
Expires: 5 September 2024

B. McMillion
4 March 2024

Key Transparency Architecture
draft-ietf-keytrans-architecture-01

Abstract

This document defines the terminology and interaction patterns involved in the deployment of Key Transparency (KT) in a general secure group messaging infrastructure, and specifies the security properties that the protocol provides. It also gives more general, non-prescriptive guidance on how to securely apply KT to a number of common applications.

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-arch/draft-ietf-keytrans-architecture.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-keytrans-architecture/.

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-arch.

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/.

Copyright Notice

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 the service operator with any public keys they wish to use to receive messages. Second, the service operator must somehow distribute these public keys amongst the participants that wish to communicate with each other.

Typically this is done by having users upload their public keys to a simple directory where other users can download them as necessary, or by providing public keys in-band with the communication being secured. With this approach, the service operator needs to be trusted to provide the correct 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 provide fake public keys which it knows the private keys for, associate those public keys with a user's account without the user's knowledge, and then use them to impersonate or eavesdrop on conversations with that user.

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 equally visible to both the key's owner and the owner's contacts, in which case a user can detect that they are being impersonated by viewing the public keys attached to their account. 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.

End-to-end Encrypted Communication Service: A communications service that allows end-users to engage in text, voice, video, or other forms of communication over the Internet, that uses public key cryptography to ensure that communications are only accessible to their intended recipients.

End-user Device: The device at the final point in a digital communication, which may either send or receive encrypted data in an end-to-end encrypted communication service.

End-user Identity: A unique and user-visible identity associated with an account (and therefore one or more end-user devices) in an end-to-end encrypted communication service. In the case where an end-user explicitly requests to communicate with (or is informed they are communicating with) an end-user uniquely identified by the name "Alice", the end-user identity is the string "Alice".

User / Account: A single end-user of an end-to-end encrypted communication service, which may be represented by several end-user identities and end-user devices. For example, a user may be represented simultaneously by multiple identities (email, phone number, username) and interact with the service on multiple devices (phone, laptop).

Service Operator: The primary organization that provides the infrastructure and software resources necessary to operate an end-to-end encrypted communication service.

Transparency Log: A specialized service capable of securely attesting to the information (such as public keys) associated with a given end-user identity. The transparency log is usually run either entirely or partially by the service operator.

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 users to query or modify stored data. Users coordinate with each other through the server by uploading their own public keys and/or downloading the public keys of other users. Users 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. Users insert key-value pairs into the database where, for example, the key is their username and the value is their public key. Users 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. Users are considered to *own* a key if, in the normal operation of the application, they should be the only one making changes to it. 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.

KT 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.

4. User Interactions

As discussed in Section 3, KT follows a client-server architecture. This means users generally interact directly with the transparency log. The operations that can be executed by a user are as follows:

1. *Search:* Performs a lookup on a specific key in the most recent version of the log. Users 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 and no provisional inclusion proof, such as an SCT as defined in Section 3 of [RFC6962], is provided.

3. *Monitor:* While Search and Update are run by the user 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 user without the user's knowledge.

These operations are executed over an application-provided transport layer, where the transport layer enforces access control by blocking queries which are not allowed:

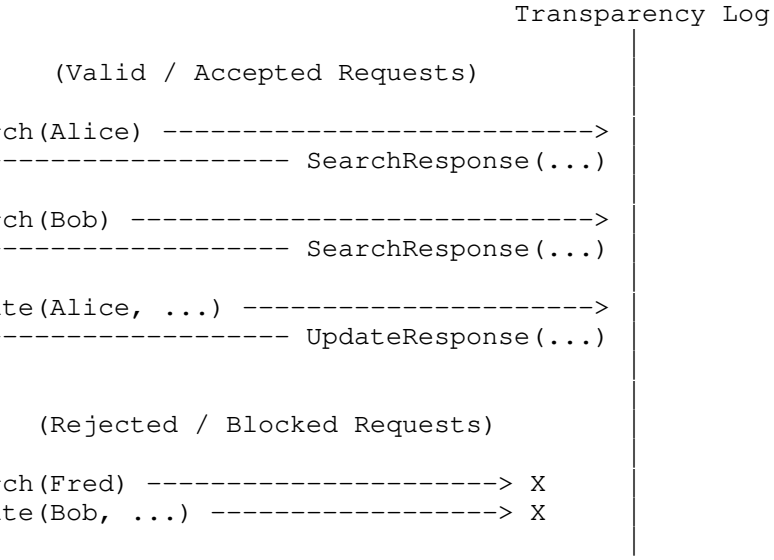

Figure 1: Example request/response flow. Valid requests receive a response while invalid requests are blocked by the transport layer.

An important caveat to the client-server architecture is that many end-to-end encrypted communication services require the ability to provide _credentials_ to their users. These credentials convey a binding between an end-user identity and potentially several encryption or signature public keys, and are meant to be verified with no/minimal network requests by the receiving users.

In particular, credentials that can be verified with minimal network access are often required by applications provide anonymous communication. These applications provide end-to-end encryption with a protocol like the Messaging Layer Security protocol [RFC9420] (with the encryption of handshake messages required), or Sealed Sender [sealed-sender]. When a user receives a message, these protocols have senders provide their own credential in an encrypted portion of the message. Encrypting the sender's credential prevents it from being visible to the service provider, while still assuring the recipient of the sender's identity. If users were to authenticate the sender's public key directly with the service provider, they would leak to the service provider who the they are communicating with.

Key Transparency credentials can be created by serializing one or more Search request-response pairs. These Search operations would correspond to the lookups a user needs to do to prove the relationship between their end-user identity and their cryptographic keys. Recipients can verify the request-response pairs themselves without contacting the Transparency Log.

Any future monitoring that may be required can be provided to recipients proactively by the sender. However if this fails, the recipient can still perform the monitoring themselves (including over an anonymous channel if necessary).

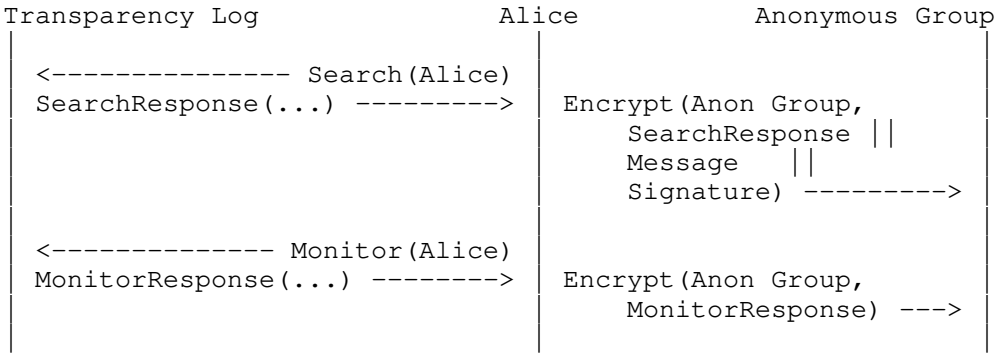

Figure 2: Example message flow in an anonymous deployment. Users request their own key from the Transparency Log and provide the serialized response, functioning as a credential, in encrypted messages to other users. Required monitoring is provided proactively.

4.1. Out-of-Band Communication

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 requests succeed and proofs verify correctly. 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 require subsequent queries to prove consistency with previous queries. As such, users always stay on a linearizable 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 *peer-to-peer communication* or *anonymous communication* with the Transparency Log.

With peer-to-peer communication, two users gossip with each other to establish that they both have the same view of the log's data. This gossip is able to happen over any supported out-of-band channel, even if it is heavily bandwidth-limited, such as scanning a QR code or talking over the phone.

With anonymous communication, a single user accesses the Transparency Log over an anonymous channel and tries to establish that the log is presenting the same view of data over the anonymous channel as it does over authenticated channels.

In the event that a fork is successfully detected, the user is able to produce non-repudiable proof of log misbehavior which can be published.

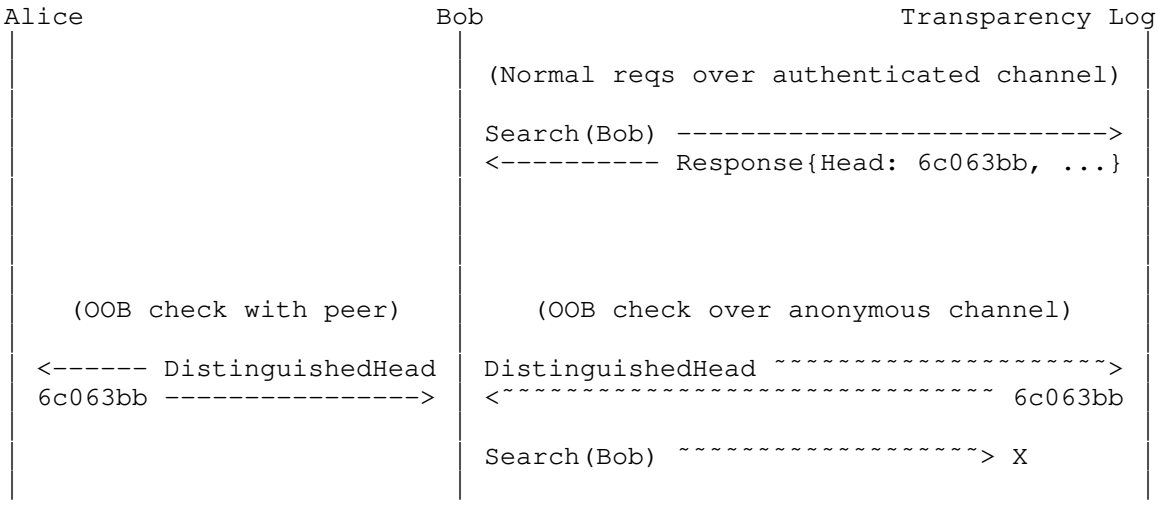

Figure 3: Users receive tree heads while making authenticated requests to a Transparency Log. Users ensure consistency of tree heads by either comparing amongst themselves, or by contacting the Transparency Log over an anonymous channel. Requests that require authorization are not available over the anonymous channel.

5. Deployment Modes

In the interest of satisfying the widest range of use-cases possible, three different modes for deploying a Transparency Log are supported. Each mode has slightly different requirements and efficiency considerations for both the transparency log and the end-user.

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.

With both third-party modes, all requests from end-users are initially routed to the transparency log and the log coordinates with the third party itself. 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 service, and the transparency log only signs new entries as they're added. With

Third-Party Auditing, the transparency log performs the majority of the work of storing and operating the service, and obtains signatures from a lightweight third-party auditor at regular intervals asserting that the tree has been constructed correctly.

Contact Monitoring, on the other hand, supports a single-party deployment with no third party. The cost of this 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 limited number of keys repeatedly (for example, the keys of regular contacts).

Deployment Mode	Supports ephemeral keys?	Single party?
Contact Monitoring	No	Yes
Third-Party Auditing	Yes	No
Third-Party Management	Yes	No

Table 1: Comparison of deployment modes

Applications that rely on a Transparency Log deployed in Contact Monitoring mode MUST regularly engage in out-of-band communication (Section 4.1) to ensure that they detect forks in a timely manner.

Applications that rely on a Transparency Log deployed in either of the third-party modes SHOULD allow users to enable a "Contact Monitoring Mode". This mode, which affects only the individual client's behavior, would cause the client to behave as if its Transparency Log was deployed in Contact Monitoring mode. As such, it would start retaining state about previously looked-up keys and regularly engaging in out-of-band communication. Enabling this higher-security mode allows users to double-check that the third-party is not colluding with the Transparency Log to serve malicious data.

5.1. Contact Monitoring

With the Contact Monitoring deployment mode, the monitoring burden is split between both the owner of a key and those that look up the key. 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 the key has not changed unexpectedly.

2. The users that looked up a key, at some point in the future, verify that the key-value pair they observed is still properly represented in the tree such that other users would find it if they searched for it.

This guarantees that if a malicious key-value pair is added to the log, then either it is detected by the key owner, or if it is removed/obscured from the log before the key owner can detect it, then any users that observed it will detect its removal.

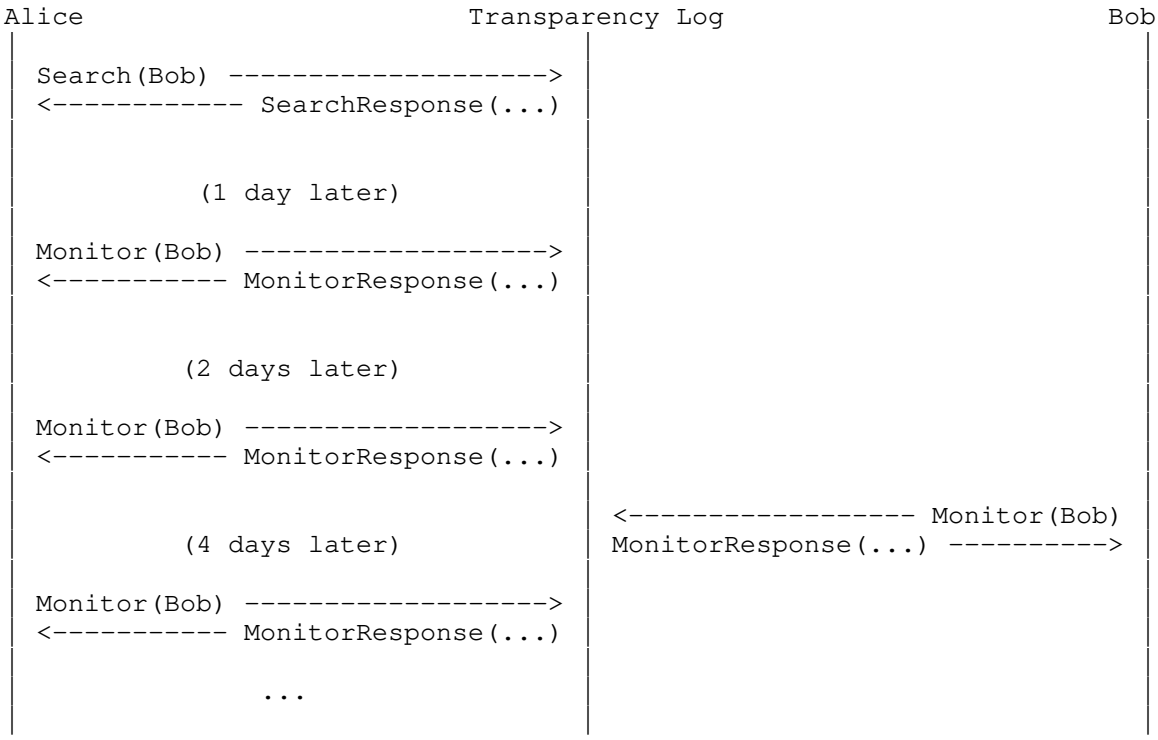

Figure 4: Contact Monitoring. When users make a Search request, they must check back in with the Transparency Log several times. These checks ensure that the data in the Search response wasn't later removed from the log. Overlap with the key owner's own monitoring guarantees a consistent view of data.

5.2. Third-Party Auditing

With the Third-Party Auditing deployment mode, the transparency log obtains signatures from a lightweight third-party auditor attesting to the fact that the tree has been constructed 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 transparency log.

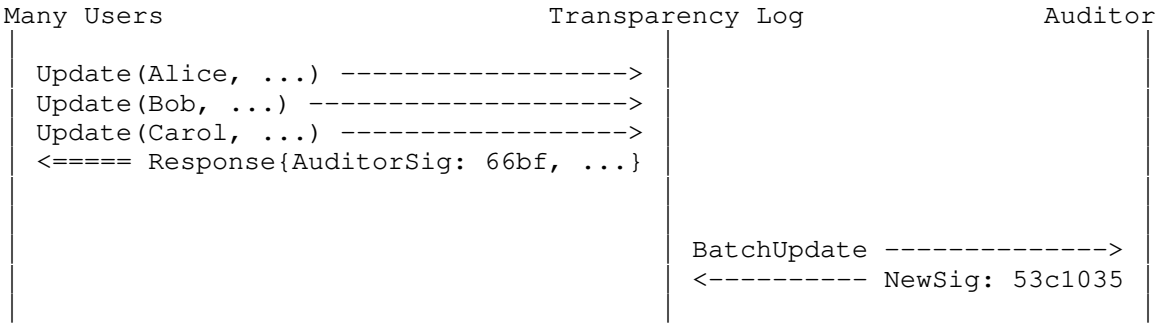

Figure 5: Third-Party Auditing. A recent signature from the auditor is provided to users. The auditor is updated on changes to the tree in the background.

5.3. Third-Party 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 transparency log serves mainly to enforce access control and authenticate the addition of new entries to the log. All user queries are initially sent by users directly to the transparency log, and the log operator proxies them to the third-party manager if they pass access control.

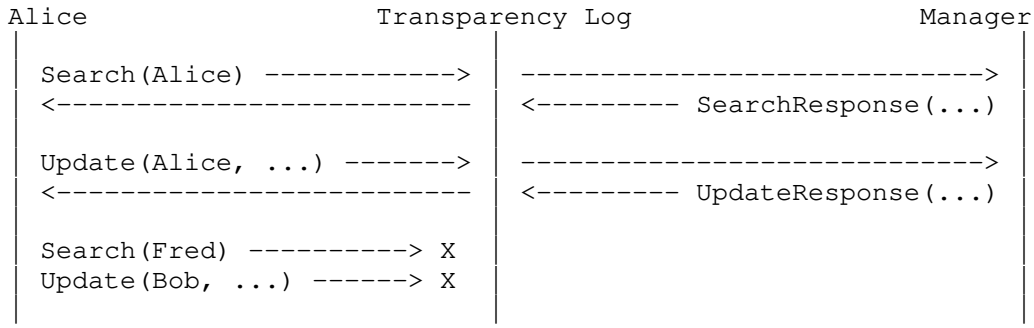

Figure 6: Third-Party Management. Valid requests are proxied by the Transparency Log to the Manager. Rejected requests are blocked.

6. Combining Logs

There are many cases where it makes sense to operate multiple cooperating log instances, for example:

* A service provider may decide that it's prudent to rotate its cryptographic keys, or migrate to a new deployment mode. They can do this by creating a new log instance with new cryptographic keys, operating under a new deployment mode if desired, and migrating their data from the old log to the new log while users are able to query both.

* A service provider may choose to operate multiple logs to improve their ability to scale or provide higher availability.

* A federated system may allow each participant in the federation to operate their own log for their own users.

Client implementations should generally be prepared to interact with multiple logs simultaneously. In particular, clients SHOULD namespace any configuration or state related to a particular log, such that information related to different logs do not conflict.

When multiple logs are used, all users in the system MUST have a consistent policy for executing Search, Update, and Monitor queries against the logs in a way 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.

6.1. Gradual Migration

In the case of gradually 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 special application-defined 'tombstone' entry.

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 MUST 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.

6.2. Immediate Migration

In the event of a key compromise, the service provider may instead choose to stop adding new entries to a log immediately and provide a new log that is pre-populated with the most recent versions of all keys. In this case, the policy may look like:

1. Search queries must be executed against the new log.

2. Update queries must be executed against the new log.

3. The final tree size and root hash of the old log should be provided to users over a trustworthy channel. Users will use this to do any final monitoring of the old log, and then ensure that the most recent versions of the keys they own are properly represented in the new log. From then on, users will monitor only the new log.

The final tree size and root hash of the prior log must be distributed to users in a way that guarantees all users have a globally-consistent view. This can be done either by storing them in a well-known key of the new log, or with the application's code distribution mechanism.

6.3. Federation

In a federated application, many servers that are owned and operated by different entities will cooperate to provide a single end-to-end encrypted communication service. Each entity in a federated system provides its own infrastructure (in particular, a transparency log) to serve the users that rely on it. Given this, there must be a consistent policy for directing KT requests to the correct transparency log. Typically in such a system, the end-user identity directly specifies which entity requests should be directed to. For example, with an email end-user identity like alice@example.com, the controlling entity is example.com.

A controlling entity like example.com may act as an anonymizing proxy for its users when querying transparency logs run by other entities (in the manner of [RFC9458]), but should not attempt to 'mirror' or combine other transparency logs with its own.

7. Pruning

As part of the core infrastructure of an end-to-end encrypted communication service, Transparency Logs are required to operate seamlessly for several years. This presents a problem for general append-only logs, as even moderate usage can cause the log to grow to an unmanageable size. This issue is further compounded by the fact that a substantial portion of the entries added to a log may be fake, having been added solely for the purpose of obscuring short-term update rates (as discussed in Section 8.1). Given this, Transparency Logs need to be able manage their footprint by pruning data which is no longer required by the communication service.

Broadly speaking, a Transparency Log's database will contain two types of data:

1. Serialized user data (the values corresponding to keys in the log), and

2. Cryptographic data, such as pre-computed portions of hash trees or commitment openings.

The first type, serialized user data, can be pruned by removing any entries that the service operator's access control policy would never permit access to. For example, a service operator may only permit clients to search for the most recent version (or n versions) of a key. Any entries that don't meet this criteria can be deleted without consideration to the rest of the protocol.

The second type, cryptographic data, can also be pruned, but only after considering which parts are no longer required by the protocol for producing proofs. For example, even though the key-value pair inserted at a particular entry in the append-only log may have been deleted, parts of the log entry may still be needed to produce proofs for Search / Update / Monitor queries on other keys. The exact mechanism for determining which data is safe to delete will depend on the implementation.

The distinction between user data and cryptographic data provides a valuable separation of concerns, given that the protocol document does not provide a mechanism for a service operator to convey its access control policy to a Transparency Log. That is: pruning user data can be done entirely by application-defined code, while pruning cryptographic data can be done entirely by KT-specific code as a subsequent operation.

8. Security Guarantees

A user that correctly verifies a proof from the Transparency Log (and does any required monitoring afterwards) receives a guarantee that the Transparency Log operator executed the key-value lookup correctly, and in a way that's globally consistent with what it has shown all other users. That is, when a user searches for a key, they're guaranteed that the result they receive represents the same result that any other user searching for the same key would've seen. When a user modifies a key, they're guaranteed that other users will see the modification the next time they search for the key.

If the Transparency Log operator does not execute a key-value lookup correctly, then either:

1. The user will detect the error immediately and reject the proof, or

2. The user will permanently enter an invalid state.

Depending on the exact reason that the user 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 users must stay online for some bounded amount of time after entering an invalid state for it to be successfully detected.

Alternatively, instead of executing a lookup incorrectly, the Transparency Log can attempt to prevent a user from learning about more recent states of the log. This would allow the log to continue executing queries correctly, but on outdated versions of data. To prevent this, applications configure an upper bound on how stale a query response can be without being rejected.

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

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

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

In short, assuming that the underlying cryptographic primitives used are secure, any deployment-specific assumptions hold (such as non-collusion), and that user devices don't go permanently offline, then malicious behavior by the Transparency Log is always detected within a bounded amount of time. The parameters that determine the maximum amount of time before malicious behavior is detected are as follows:

* How stale an application allows query responses to be (ie, how long an application is willing to go without seeing updates to the tree).

* How frequently users execute background monitoring.

* How frequently users exercise out-of-band communication.

* For third-party auditing: the maximum amount of lag that an auditor is allowed to have, with respect to the most recent tree head.

8.1. Privacy Guarantees

For applications deploying 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, and information about that user's account, to that user's friends or contacts.

KT only allows users to learn whether or not a lookup key exists in the Transparency Log if the user obtains a valid search proof for that key. Similarly, KT only allows users to learn about the contents of a log entry if the user obtains a valid search proof for the exact key and version stored at that log entry.

Applications are primarily able to manage the privacy of their data in KT by relying on these properties when they enforce access control policies on the queries issued by users, as discussed in Section 3. For example if two users aren't friends, an application can block these users from searching for each other's lookup keys. This prevents the two users from learning about each other's existence. If the users were previously friends but no longer are, the application can prevent the users from searching for each other's keys and learning the contents of any subsequent account updates.

Service operators also expect to be able to control sensitive population-level metrics about their users. These metrics include the size of their userbase, the frequency with which new users join, and the frequency with which existing users update their keys.

KT allows a service operator to obscure the size of its userbase by padding the tree with fake entries. Similarly, it also allows a service operator to obscure the rate at which changes are made by padding real changes with fake ones, causing outsiders to observe a baseline constant rate of changes.

8.1.1. Leakage to Third-Party

In the event that a third-party auditor or manager is used, there's additional information leaked to the third-party that's not visible to outsiders.

In the case of a third-party auditor, the auditor is able to learn the total number of distinct changes to the log. It is also able to learn the order and approximate timing with which each change was made. However, auditors are not able to learn the plaintext of any keys or values. This is because keys are masked with a VRF, and values are only provided to auditors as commitments. They are also not able to distinguish between whether a change represents a key

being created for the first time or being updated, or whether a change represents a "real" change from an end-user or a "fake" padding change.

In the case of a third-party manager, the manager generally learns everything that the service operator would know. This includes the total set of plaintext keys and values and their modification history. It also includes traffic patterns, such as how often a specific key is looked up.

9. Privacy Law Considerations

Consumer privacy laws often provide a 'right to erasure', meaning that when a consumer requests that a service provider delete their personal information, the service provider is legally obligated to do so. This may seem to be incompatible with the description of KT in Section 1 as an 'append-only log'. Once an entry is added to a transparency log, it indeed can not be removed.

The important caveat here is that user data is not directly stored in the append-only log. Instead, the log consists of privacy-preserving cryptographic commitments. By logging commitments instead of plaintext user data, users interacting with the log are unable to infer anything about an entry's contents until the service provider explicitly provides the commitment's opening. A service provider responding to an erasure request can delete the commitment opening and the associated data, effectively anonymizing the entry.

Other than the log, the second place where user information is stored is in the _prefix tree_. This is a cryptographic index provided to users to allow them to efficiently query the log, which contains information about which lookup keys exist and where. These lookup keys are usually serialized end-user identifiers, although it varies by application. To minimize leakage, all lookup keys are processed through a Verifiable Random Function, or VRF [RFC9381].

A VRF deterministically maps each lookup key to the 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 tries to search for or update a key, the service operator first executes its VRF on the input lookup 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.

A service provider responding to an erasure request can 'trim' the prefix tree, by no longer storing the full VRF output for any lookup keys corresponding to an end-user's identifiers. With only a small amount of the VRF output left in storage, even if the transparency log is later compromised, it would be unable to recover deleted identifiers. If the same lookup keys were reinserted into the log at a later time, it would appear as if they were being inserted for the first time.

As an example, consider the information stored in a transparency log after inserting a key K with value V. The value stored in the prefix tree would roughly correspond to VRF(key K) = pseudorandom bytes, and the value stored in the append-only log would roughly correspond to:

Commit(nonce: random bytes, body: version N of key K is V)

After receiving an erasure request, the transparency log deletes the key, value, and random commitment nonce. It also trims the VRF output to the minimum size necessary. The commitment scheme guarantees that, without the high-entropy random nonce, the remaining commitment reveals nothing about the key or value.

Assuming that the prefix tree is well-balanced (which is extremely likely due to VRFs being pseudorandom), the number of VRF output bits retained is approximately equal to the logarithm of the total number of keys logged. This means that while the VRF's full output may be 256 bits, in a log with one million keys, only 20 output bits would need to be retained. This would be insufficient for recovering even a very low-entropy identifier like a phone number.

10. Implementation Guidance

Fundamentally, KT can be thought of as guaranteeing that all the users of a service agree on the contents of a key-value database. Using this guarantee, that all users agree on a set of keys and values, to authenticate the relationship between end-user identities and the end-users of a communication service takes special care. Critically, in order to authenticate an end-user identity, it must be both _unique_ and _user-visible_. However, what exactly constitutes a unique and user-visible identifier varies greatly from application to

application.

Consider, for example, a communication service where users are uniquely identified by a fixed username, but KT has been deployed using an internal UUID as the lookup key. While the UUID might be unique, it is not user-visible. When a user attempts to lookup a contact by username, the service operator must translate the username into its UUID. Since this mapping (from username to UUID) is unauthenticated, the service operator can manipulate it to eavesdrop on conversations by returning the UUID for an account that it controls. From a security perspective, this is equivalent to not using KT at all. An example of this kind of application would be email.

However in other applications, the use of internal UUIDs in KT may be appropriate. For example, many applications don't have this type of fixed username and instead use their UI (underpinned internally by a UUID) to indicate to users whether a conversation is with a new person or someone they've previously contacted. The fact that the UI behaves in this way makes the UUID a user-visible identifer, even if a user may not be able to actually see it written out. An example of this kind of application would be Slack.

A *primary end-user identity* is one that is unique, user-visible, and unable to change. (Or equivalently, if it changes, it appears in the application UI as a new conversation with a new user.) A primary end-user identity should always be a lookup key in KT, with the end-user's public keys as the associated value.

A *secondary end-user identity* is one that is unique, user-visible, and able to change without being interpreted as a different account due to its association with a primary identity. Examples of this type of identity include phone numbers, or most usernames. These identities are used solely for initial user discovery, in which they're converted to a primary identity that's used by the application from then on. A secondary end-user identity should be a lookup key in KT, for the purpose of authenticating user discovery, with the primary end-user identity as the associated value.

While likely helpful to most common applications, the distinction between handling primary and secondary identities is not a hard-and-fast rule. Applications must be careful to ensure they fully capture the semantics of identity in their application with the key-value structure they put in KT.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. Normative References

[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>.

[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>.

12.2. Informative References

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/rfc/rfc6962>.

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

[RFC9420] Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", RFC 9420, DOI 10.17487/RFC9420, July 2023, <https://www.rfc-editor.org/rfc/rfc9420>.

[RFC9458] Thomson, M. and C. A. Wood, "Oblivious HTTP", RFC 9458, DOI 10.17487/RFC9458, January 2024, <https://www.rfc-editor.org/rfc/rfc9458>.

[sealed-sender] "Technology preview: Sealed sender for Signal", 29 October 2018, <https://signal.org/blog/sealed-sender/>.

Acknowledgments

TODO acknowledge.

Author’s Address

Brendan McMillion

Email: brendanmcmillion@gmail.com

KEYTRANS Working Group
Internet-Draft
Intended status: Informational
Expires: 12 December 2024

B. McMillion

F. Linker

10 June 2024

Key Transparency Protocol
draft-keytrans-mcmillion-protocol-00

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://Bren2010.github.io/draft-keytrans/draft-keytrans-mcmillion-protocol.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-keytrans-mcmillion-protocol/.

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

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 12 December 2024.

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 [I-D.ietf-keytrans-architecture].

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 chose 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

KT allows clients of a service to query the keys of other clients of the same service. To do so, KT maintains two structures: (i) a log of each change to any key's value, and (ii) a set containing all of the key-version pairs that have been logged. When clients query a KT service, they require a means to authenticate the responses of the KT service. To provide for this, the KT service maintains a _combined hash tree structure_, which commits to both these structures with a _root hash_. Two clients which have the same root hash are guaranteed to have the same view of the tree, and thus would always receive the same result for the same query.

The combined hash tree structure consists of two types of trees: log trees and prefix trees. The log tree commits to (i) and the prefix tree commits to (ii). 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 6.

Both types of trees consist of _nodes_ which have a byte string as their _hash 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.

3.1. 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.

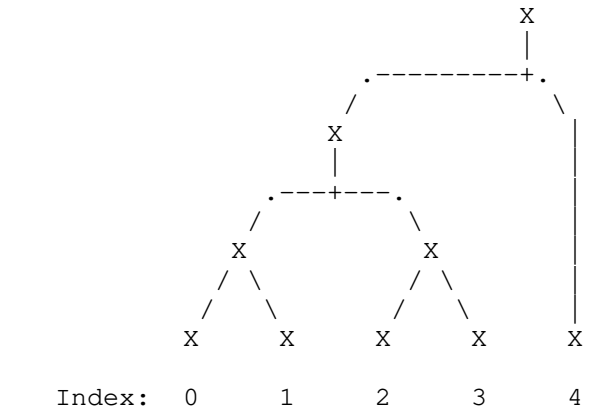

Figure 1: A log tree containing five leaves.

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.

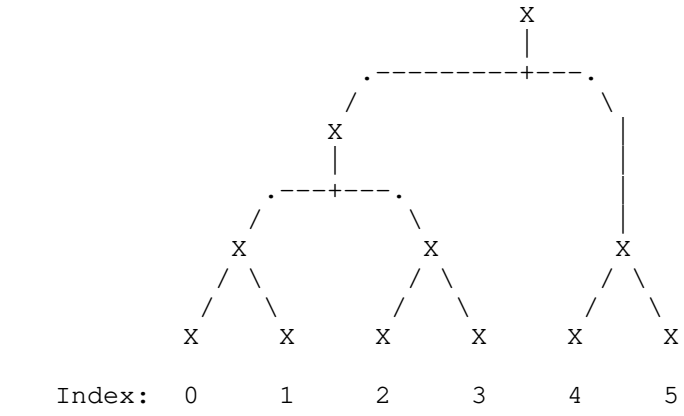

Figure 2: Example of inserting a new leaf with index 10 into the previously depicted log tree. Observe that only the nodes on the path from the new root to the new leaf change.

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 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 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 hash 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].

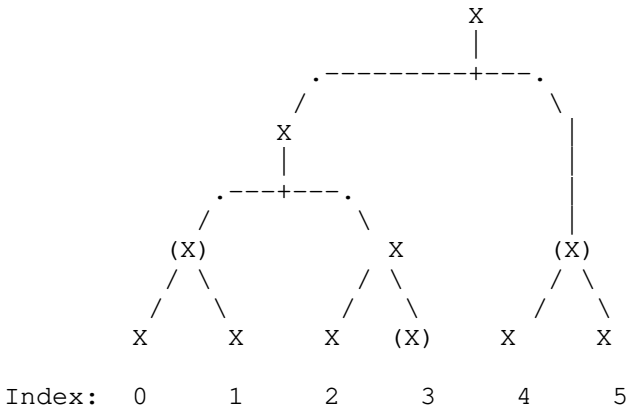

Figure 3: Illustration of a proof of inclusion. To verify that leaf 4 is included in the tree, the server provides the client with the hashes of the nodes on its copath, i.e., all hashes that are required for the client to compute the root hash itself. In the figure, the copath consists of the nodes marked by (X).

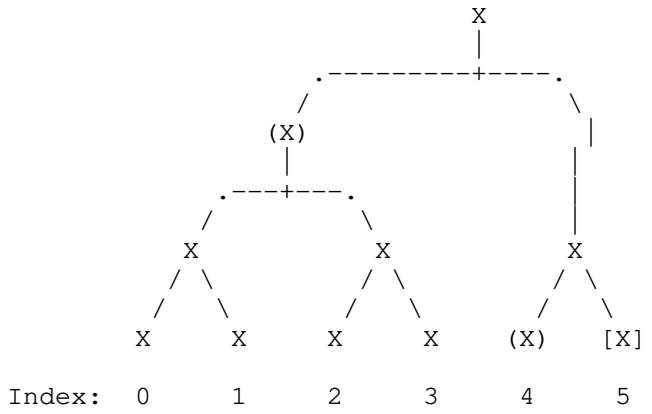

Figure 4: Illustration of a consistency proof. The server proves to the client that it correctly extended the tree by giving it the hashes of marked nodes ([X] / (X)). The client can verify that it can construct the old root hash form the hashes of nodes marked by (X), and that it can construct the new root hash when also considering the hash of the node [X].

3.2. Prefix Tree

Prefix trees are used for storing a set of values while preserving the ability to efficiently produce proofs of membership and non-membership in the set.

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

New values 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 value, 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 value and the existing value.

The value of a leaf node is the encoded set member, 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 hash value, along with the hash 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 search path for the intended value, but ends either at a stand-in value or a leaf for a different value. 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.

3.3. 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 values added.

In the combined tree structure, which is based on [Merkle2], a log tree maintains a record of each time any key's value is updated, while a prefix tree maintains the set of index-version pairs. Importantly, the root hash value of the prefix tree after adding a new index-version pair is stored in a leaf of the log tree alongside a privacy-preserving commitment to the update. 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 this tree results in a new root hash, which is then stored in the log tree. Therefore, when instructions refer to "looking up a key in the prefix tree at a given

log entry," this actually means searching in the specific version of the prefix tree that corresponds to the root hash stored at that log entry.

4. Searching the Tree

To search the combined tree structure described in Section 3.3, users do a binary search for the first log entry where the prefix tree at that entry contains the desired key-version pair. As such, the entry that a user arrives at through binary search contains the update that they're looking for, even though the log itself is not sorted.

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).

4.1. 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. In this representation, "leaf" nodes are stored in even-numbered indices, while "intermediate" nodes are stored in odd-numbered indices:

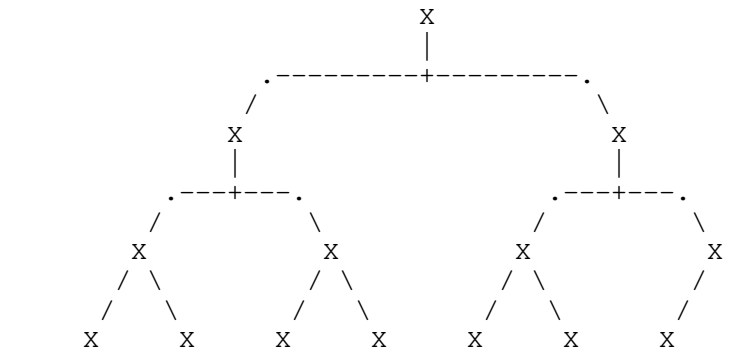

Figure 5: A binary tree constructed from 14 entries in a log

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

# 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
```

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 log has 50 entries, instead of starting the search at the typical "middle" entry of 50/2 = 25, users would start at entry root(50) = 31. If the next step in the search is to move right, the next index to access would be right(31, 50) = 47. As more entries are added to the log, users will consistently revisit entries 31 and 47, while they may never revisit entry 25 after even a single new entry is added to the log.

4.2. Binary Ladder

When executing searches on a Transparency Log, the implicit tree described in Section 4.1 is navigated according to a binary search. At each individual log entry, the binary search needs to determine whether it should move left or right. That is, it needs to determine, out of the set of key-version pairs stored in the prefix tree, whether the highest version present at a given log entry is greater than, equal to, or less than a target version.

A *binary ladder* is a series of lookups in a single log entry's prefix tree, which is used to establish whether the target version of a key is present or not. It consists of the following lookups, stopping after the first lookup that produces a proof of non-inclusion:

1. First, version x of the key is looked up, where x is consecutively higher powers of two minus one (0, 1, 3, 7, ...). This is repeated until x is the largest such value less than or equal to the target version.

2. Second, the largest x that was looked up is retained, and consecutively smaller powers of two are added to it until it equals the target version. Each time a power of two is added, this version of the key is looked up.

As an example, if the target version of a key to lookup is 20, a binary ladder would consist of the following versions: 0, 1, 3, 7, 15, 19, 20. If all of the lookups succeed (i.e., result in proofs of inclusion), this indicates that the target version of the key exists in the log. If the ladder stops early because a proof of non-inclusion was produced, this indicates that the target version of the key did not exist, as of the given log entry.

When executing a search in a Transparency Log for a specific version of a key, a binary ladder is provided for each node on the search path, verifiably guiding the search toward the log entry where the desired key-version pair was first inserted (and therefore, the log entry with the desired update).

Requiring proof that this series of versions are present in the prefix tree, instead of requesting proof of just version 20, ensures that all users are able to agree on which version of the key is _most recent_, which is discussed further in the next section.

4.3. Most Recent Version

Often, users wish to search for the "most recent" version of a key. That is, the key with the highest counter possible.

To determine this, users request a *full binary ladder* for each node on the *frontier* of the log. 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 log has 50 entries, the frontier would be entries: 31, 47, 49.

A full binary ladder is similar to the binary ladder discussed in the previous section, except that it identifies the exact highest version of a key that exists, as of a particular log entry, rather than stopping at a target version. It consists of the following lookups:

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

2. Once the first proof of non-inclusion is produced, a binary search is conducted between the highest 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 a proof of inclusion or non-inclusion, which guides the search left or right, until it terminates.

For the purpose of finding the highest version possible, requesting a full binary ladder for each entry along the frontier is functionally the same as doing so for only the last log entry. However, inspecting the entire frontier allows the user to verify that the search path leading to the last log entry represents a monotonic series of version increases, which minimizes opportunities for log misbehavior.

Once the user has verified that the frontier lookups are monotonic and determined the highest version, the user then continues a binary search for this specific version.

4.4. 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 position in the log to a version counter. The version counter is the highest version of the search key that's been proven to exist at that log position. Users initially populate this map by setting the position of an entry they've looked up, to map to the version of the key stored in that entry. 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 (in terms of the Implicit Binary Search Tree) based on the current tree size.

2. If there are no entries in the direct path that are to the right of the current entry, 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 entry, from low to high:

    1. Receive and verify a binary ladder from that log entry, for the version currently in the map. This proves that, at the indicated log entry, the highest version present is greater than or equal to the previously-observed version.

    2. If the above check was successful, remove the current position-version pair from the map and replace it with a position-version 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.

5. 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 [RFC9381] must be used.

Ciphersuites are represented with the CipherSuite type. The ciphersuites are defined in Section 11.1.

6. Cryptographic Computations

6.1. Verifiable Random Function

Each version of a search key that's inserted in a log will have a unique representation in the prefix tree. This is computed by providing the combined search key and version as inputs to the VRF:

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

6.2. Commitment

As discussed in Section 3.3, commitments are stored in the leaves of the log tree and correspond to updates of a key's value. 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.

6.3. Prefix Tree

The leaf nodes of a prefix tree are serialized as:

```
struct {
    opaque key_version<VRF.Nh>;
} PrefixLeaf;
```

where key_version is the VRF output for the key-version pair, 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 make([]byte, Hash.Nh)
  else if node.type == leafNode:
    return Hash(0x00 || node.key_version)
  else if node.type == parentNode:
    return node.value
```

where Hash denotes the ciphersuite hash function.

6.4. 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
```

6.5. Tree Head Signature

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

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

where tree_size counts the number of entries in the log tree. 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;
  opaque root_value<Hash.Nh>;
} TreeHeadTBS;
```

7. Tree Proofs

7.1. 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 6.4. 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].

7.2. Prefix Tree

A proof from a prefix tree authenticates that a set of values are either members of, or are not members of, the total set of values represented by the prefix tree. Such a proof is encoded as:

```
enum {
  reserved(0),
  inclusion(1),
  nonInclusionLeaf(2),
  nonInclusionParent(3),
} 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. It is sorted lexicographically by corresponding value. 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 given that 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 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 tree structure, and checking that the result equals the root value of the prefix tree.

7.3. 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 3.3. It is serialized as follows:

```
struct {
  opaque proof<VRF.Np>;
} VRFProof;

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

struct {
  optional<uint32> version;
  VRFProof vrf_proofs<0..2^8-1>;
  ProofStep steps<0..2^8-1>;
  InclusionProof inclusion;
} SearchProof;
```

If searching for the most recent version of a key, the most recent version is provided in version. If searching for a specific version, this field is omitted.

Each element of vrf_proofs contains the output of evaluating the VRF on a different version of the search key. The versions chosen correspond either to the binary ladder described in Section 4.2 (when searching for a specific version of a key), or to the full binary ladder described in Section 4.3 (when searching for the most recent version of a key). The proofs are sorted from lowest version to highest version.

Each ProofStep structure in steps is one log entry that was inspected as part of the binary search. The first step corresponds to the "middle" leaf of the log tree (calculated with the root function in Section 4.1). From there, each subsequent step moves left or right in the tree, according to the procedure discussed in Section 4.2 and Section 4.3.

The prefix_proof field of a ProofStep is the output of executing a binary ladder, excluding any ladder steps for which a proof of inclusion is expected, and a proof of inclusion was already provided in a previous ProofStep for a log entry to the left of the current one.

The commitment field of a ProofStep contains 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.

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.

8. 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.

9. 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.

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

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.

9.1. 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.

```
struct {
  optional<uint32> last;

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

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

```
struct {
  FullTreeHead full_tree_head;

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

Users verify a search response by following these steps:

1. Evaluate the search proof in search according to the steps in Section 7.3. 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.

2. With the candidate root value for the tree, verify the given FullTreeHead.

3. 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 8, before its contents may be consumed.

9.2. Update

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

```
struct {
  optional<uint32> last;

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

If the request passes application-layer policy checks, the Transparency Log adds the new key-value pair to the log and returns an UpdateResponse structure:

```
struct {
  FullTreeHead full_tree_head;

  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. To aid verification, the update response provides the UpdatePrefix structure necessary to reconstruct the UpdateValue.

9.3. Monitor

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

```
struct {
  opaque search_key<0..2^8-1>;
  uint32 highest_version;
  uint64 entries<0..2^8-1>;
} MonitorKey;

struct {
  optional<uint32> last;

  MonitorKey owned_keys<0..2^8-1>;
  MonitorKey contact_keys<0..2^8-1>;
} 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, the highest version of the key that the user has observed in highest_version, and a list of entries in the log tree corresponding to the keys of the map described in Section 4.4.

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 (or was previously allowed) to lookup every key in contact_keys, based on the application's policy.

3. Verify that the highest_version for each key is less than or equal to the most recent version of each key.

4. Verify that each entries array is sorted in ascending order, and that all entries are within the bounds 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 {
  uint32 version;
  VRFProof vrf_proofs<0..2^8-1>;
  ProofStep steps<0..2^8-1>;
} MonitorProof;

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

The elements of owned_proofs and contact_proofs correspond one-to-one with the elements of owned_keys and contact_keys. Each MonitorProof in contact_proofs 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. Each MonitorProof in owned_proofs conveys the same guarantee that no past lookups have been concealed, and also proves that MonitorProof.version is the most recent version of the key.

The version field of a MonitorProof contains the version that was used for computing the binary ladder, and therefore the highest version of the key that will be proven to exist. The vrf_proofs field contains VRF proofs for different versions of the search key, starting at the first version that's different between the binary ladders for MonitorKey.highest_version and MonitorProof.version.

The steps field of a MonitorProof contains the proofs required to update the user's monitoring data following the algorithm in Section 4.4. That is, each ProofStep of a MonitorProof contains a binary ladder for the version MonitorProof.version. 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.

For MonitorProof structures in owned_keys, it is also important to prove that MonitorProof.version is the highest version of the key available. This means that such a MonitorProof must contains full binary ladders for MonitorProof.version along the frontier of the log. As such, any ProofStep under the owned_keys field that's along the frontier of the log includes a full binary ladder for MonitorProof.version instead of a regular binary ladder. For additional entries on the frontier of the log that are to the right of the leftmost frontier entry already provided, an additional ProofStep is added to MonitorProof. This additional ProofStep contains only the proofs of non-inclusion from a full binary ladder.

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, verify that MonitorProof.version is greater than or equal to the highest version of the key that's been previously observed.

3. For each MonitorProof structure, evalute the monitoring algorithm in Section 4.4. 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 (keeping in mind that extra steps may be provided along the frontier of the log, if a MonitorProof is a member of owned_keys).

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

5. With the candidate root value, verify the provided FullTreeHead.

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, VRF proofs for different versions of each search key are not provided, given that these can be cached from the original Search or Update query.

10. Security Considerations

11. IANA Considerations

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

* KT Ciphersuites (Section 11.1)

All of these registries should be under a heading of "Key Transparency", and assignments are made via the Specification Required policy [RFC8126]. See Section 11.2 for additional information about the KT Designated Experts (DEs).

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

11.1. KT Ciphersuites

```
uint16 CipherSuite;
```

11.2. KT Designated Expert Pool

12. References

12.1. Normative References

[I-D.ietf-keytrans-architecture]
McMillion, B., "Key Transparency Architecture", Work in Progress, Internet-Draft, draft-ietf-keytrans-architecture-01, 4 March 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-keytrans-architecture-01>.

[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>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/rfc/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>.

[RFC9381] Goldberg, S., Reyzin, L., Papadopoulos, D., and J. VÄ\215elÃ¡k, "Verifiable Random Functions (VRFs)", RFC 9381, DOI 10.17487/RFC9381, August 2023, <https://www.rfc-editor.org/rfc/rfc9381>.

12.2. 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

Authors' Addresses

Brendan McMillion
Email: brendanmcmillion@gmail.com

Felix Linker
Email: linkerfelix@gmail.com