Key Transparency
Internet-Draft
Intended status: Informational
Expires: 22 August 2026

B. McMillion
18 February 2026

Key Transparency Architecture
draft-ietf-keytrans-architecture-07

Abstract

This document defines the terminology and interaction patterns involved in the deployment of Key Transparency 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 Key Transparency 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

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. If the service operator is simply trusted to correctly forward public keys between users, this 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 are exchanged through the service operator's servers, which makes it extremely easy for an operator to launch an active attack. That is, the service operator can take public keys for which it knows the corresponding private keys, and associate those public keys with a user's account without the user's knowledge 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 affected user and the user's contacts. This allows users to detect whether 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.

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 communication service that allows end-users to engage in text, voice, video, or other forms of communication over the internet, and 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 for an end-to-end encrypted communication service and the software to participate in it.

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

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 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 can be thought of 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 previous version. From this point forward, the term *label* will be used to refer to lookup keys in the key-value database that a transparency log represents to avoid confusion with cryptographic public or private keys.

Users are considered to *own* a label if they are understood to either initiate all changes to the label’s value, or if they must be informed of all changes to the label’s value. The latter situation may occur if, for example, KT is deployed in a way where the service operator makes automated modifications to stored data. The owning user would then be informed, after the fact, of modifications to verify that they were legal.

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.

With some small exceptions, applications may enforce arbitrary access control rules on top of KT. This may include requiring a user to be logged in to make KT requests, only allowing a user to lookup the labels of another user if they’re "friends", or applying a rate limit. Applications SHOULD prevent users from modifying labels they do not own. The exact mechanism for rejecting requests, and possibly explaining the reason for rejection, is left to the application.

3.1. User Operations

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

1. *Search:* Looks up the value of a specific label in the most recent version of the log. Users may request either a specific version of the label or the most recent version available. If the label-version pair exists, the server returns the corresponding value and an inclusion proof.

2. *Update:* Adds a new label-value pair to the log, for which the server returns an inclusion proof. 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 can both check that the log is continuing to behave honestly (all previously returned labels remain in the tree) and that no changes have been made to labels owned by the user without the user's knowledge.

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

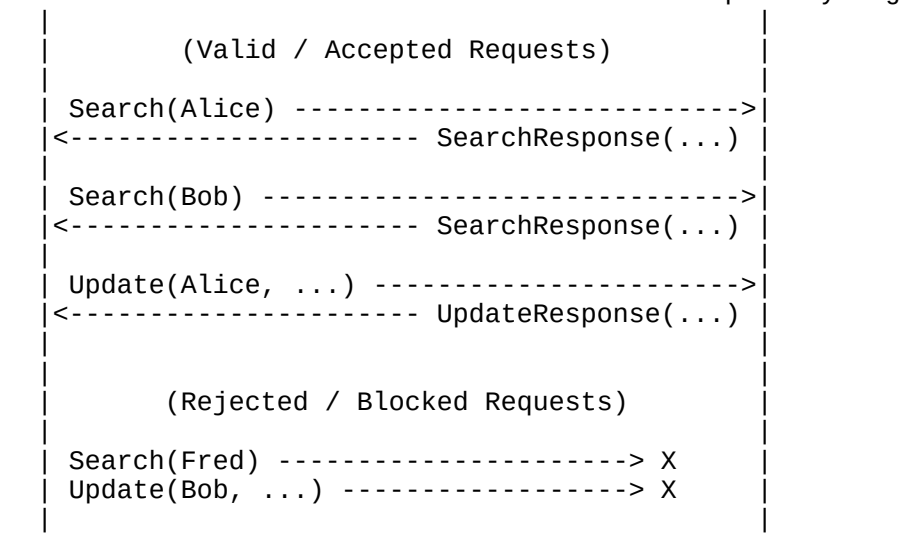

Figure 1: Example request and response flow. Valid requests receive a response while invalid requests are blocked by the transport layer.

3.2. Credentials

While users are generally understood to interact directly with the transparency log, many end-to-end encrypted communication services require the ability to provide _credentials_ to their users. Credentials convey a binding between an end-user identity and public keys or other information, and can be verified with minimal network access.

In particular, credentials that can be verified with minimal network access are often desired by applications that support 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 sends a message, these protocols have the sender provide their own credential in an encrypted portion of the message.

Encrypting the sender’s credential allows the sender to submit messages over an anonymous channel by specifying only the recipient’s identity. The service operator can deliver the message to the intended recipient, who can decrypt it and validate the credential inside to be assured of the sender’s identity. Note that the recipient does not need access to an anonymous channel to preserve the sender’s anonymity.

At a high level, KT credentials are created by serializing one or more Search request-response pairs. These Search operations correspond to the lookups the recipient would do to authenticate the relationship between the presented end-user identity and their public keys. Recipients can verify the request-response pairs themselves without contacting the transparency log.

Any future monitoring that may be required SHOULD be provided to recipients proactively by the sender. However, if this fails, the recipient will need to perform the monitoring themself over an anonymous channel.

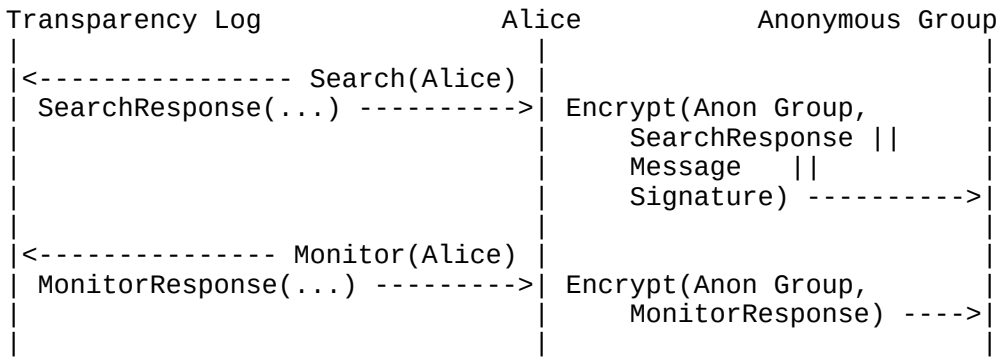

Figure 2: Example message flow in an anonymous deployment. Users request their own label from the transparency log and provide the serialized response, functioning as a credential, in encrypted messages to other users. Required monitoring is provided proactively.

3.3. 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 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 change a label's value without the label's owner becoming aware.

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 require either a *trusted third party*, *anonymous communication* with the transparency log, or *peer-to-peer communication*.

With a trusted third party, such as a Third-Party Auditor or Manager as described in Section 4.2 and Section 4.3, an outside organization monitors the operation of the transparency log. This third party verifies, among other things, that the transparency log is growing in an append-only manner. If verification is successful, the third

party produces a signature on the most recent tree head. The transparency log provides this signature to users inline with their query responses as proof that they are not being shown a fork. This approach relies on an assumption that the third party is trusted not to collude with the transparency log to sign a fork.

With anonymous communication, a single user accesses the transparency log over an anonymous channel and checks that the transparency log is presenting the same tree head over the anonymous channel as it does over an authenticated channel. The security of this approach relies on the fact that, if the transparency log doesn't know which user is making the request, it will show the user the wrong fork with high probability. Repeating this check over time makes it overwhelmingly likely that any fork presented to any user will be detected.

With peer-to-peer communication, two users gossip with each other to establish that they both have the same view of the transparency log. This gossip is able to happen over any supported out-of-band channel even if it is heavily bandwidth-constrained, such as scanning a QR code. However, this approach is only secure if gossip can be implemented such that gossipping users are reasonably expected to form a connected graph of all users. If not, then the transparency log can attempt to partition users into subsets that do not gossip and can present each subset of users with different forks.

Regardless of approach, 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.

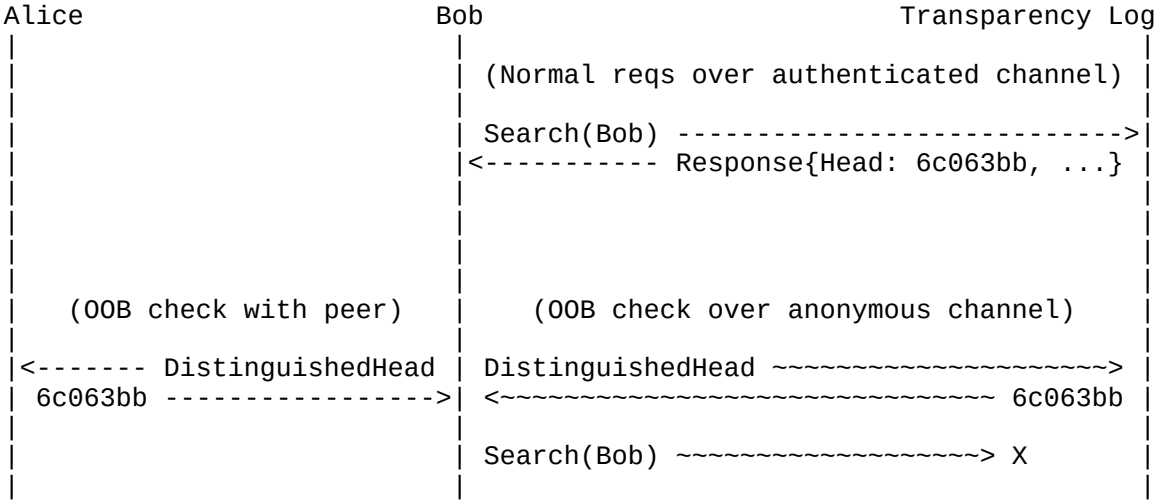

Figure 3: Users receive tree heads while making authenticated requests to a transparency log. Users ensure consistency of tree heads by either comparing amongst themselves, or by contacting the transparency log over an anonymous channel. Requests that require authentication do not need to be available over the anonymous channel.

4. 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, although 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, only obtaining signatures from a third-party auditor at regular intervals asserting that the tree has been constructed correctly.

To reduce the probability of collusion between the transparency log and the third party, a transparency log can have two or more independent third parties coordinate as one and produce threshold signatures. In this scenario, the threshold for a valid signature MUST be at least a majority of the third parties, to prevent different subsets from authenticating forked views.

Contact Monitoring, on the other hand, supports a single-party deployment with no third party. The cost of this is that, when a user looks up a version of a label that was inserted very recently, the user may need to retain some additional state and monitor the label until it is included in a _distinguished log entry_ (defined in [PROTO]). If a user looks up many label-version pairs that were inserted very recently, monitoring may become relatively expensive.

Additionally, applications that rely on a transparency log deployed in Contact Monitoring mode MUST regularly attempt to detect forks through anonymous communication with the transparency log or peer-to-peer communication, as described in Section 3.3.

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 labels 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.

4.1. Contact Monitoring

With the Contact Monitoring deployment mode, the monitoring burden is split between both the owner of a label and those that look up the label. Stated as simply as possible, the monitoring obligations of each party are:

1. On a regular basis, the label owner verifies that the most recent version of their label has not changed unexpectedly.

2. When a user that looked up a label sees that it was inserted very recently, they check back later to see that the label-version pair they observed was not removed before it could be detected by the label owner.

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

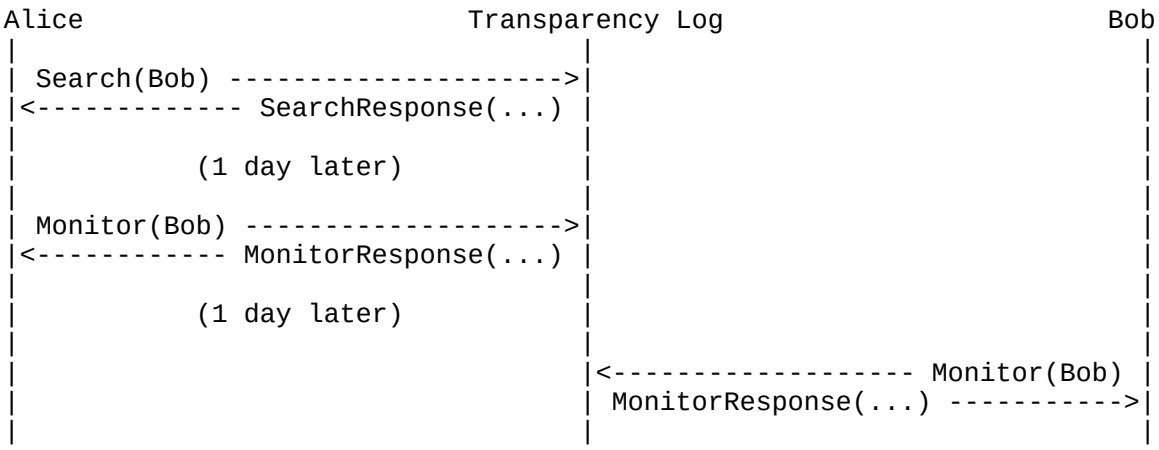

Figure 4: Contact Monitoring. Alice searches for Bob's label. One day later, Alice verifies the label-version pair she observed remained in transparency log. Another day later, Bob comes online and monitors his own label. Note that Alice does not need to wait on Bob to make his Monitor request before making hers.

Importantly, Contact Monitoring impacts how the server is able to enforce access control on Monitor queries. While Search and Update queries can enforce access control on a "point in time" basis, where a user is allowed to execute the query at one point in time but maybe not the next, Monitor queries MUST have _accretive_ access control. This is because, if a user is allowed to execute a Search or Update query for a label, the user may then need to issue at least one Monitor query for the label some time in the future. These Monitor queries MUST be permitted, regardless of whether or not the user is still permitted to execute such Search or Update queries.

4.2. Third-Party Auditing

With the Third-Party Auditing deployment mode, the transparency log obtains signatures from a third-party auditor attesting (at minimum) to the fact that the tree has been constructed correctly. These signatures are provided to users as part of the responses for their queries.

When running synchronously, the auditor can easily become a bottleneck for the transparency log. It's generally expected that third-party auditors run asynchronously, downloading and authenticating a log's contents in the background. As a result, signatures from the auditor may lag behind the view presented by the transparency log. The maximum amount of time that the auditor may lag behind the transparency log without its signature being rejected by users is set in the transparency log's configuration.

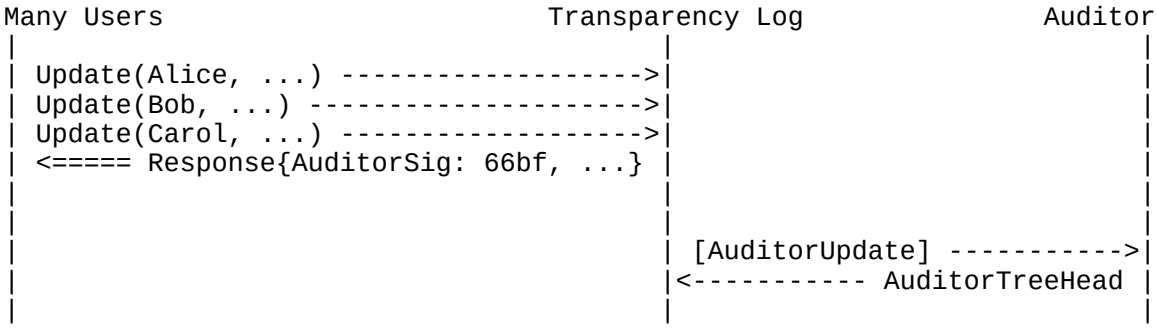

Figure 5: Third-Party Auditing. A recent signature from the auditor is provided to users. The auditor is updated on changes to the tree in the background.

Given the long-lived nature of transparency logs and the potentially short-lived nature of third-party auditing arrangements, KT allows third-party auditors to start auditing a log at any arbitrary point. This allows a new third-party auditor to start up without ingesting the transparency log's entire history. The point at which an auditor started auditing is provided to users in the transparency log's configuration. When verifying query responses, users verify that the auditor started auditing at or before the point necessary for the query to be secure.

4.3. 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. 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 transparency log proxies them to the third-party manager if they pass access control.

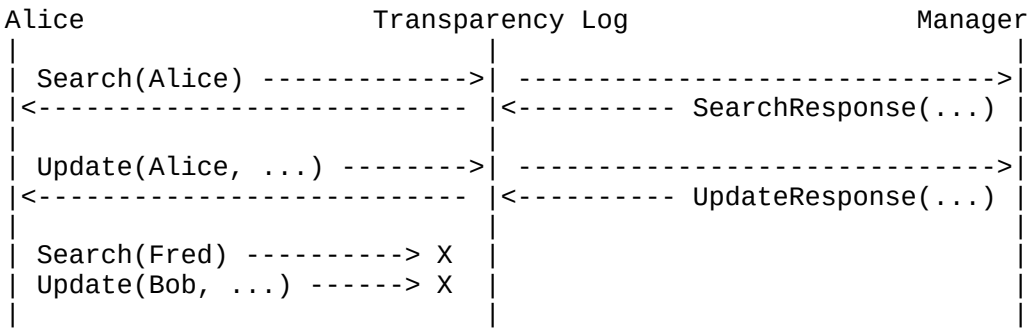

Figure 6: Third-Party Management. Valid requests are proxied by the transparency log to the manager. Invalid requests are blocked.

The security of the Third-Party Management deployment mode comes from an assumption that the transparency log and the third-party manager do not collude to behave maliciously. If the third-party manager behaves honestly, then any improper modifications to a label's value that were requested by the transparency log will be properly published such that the label owner will detect them when monitoring. If the transparency log behaves honestly, the third-party manager will be unable to add any new unauthorized versions of a label such that a user will accept them, or remove any authorized version of a label without the label owner detecting it.

The transparency log MUST implement some mechanism to detect when forks are presented by the third-party manager. Additionally, the transparency log MUST implement some mechanism to prevent the same version of a label from being submitted to the third-party manager multiple times with different associated values.

5. Combining Logs

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

* A service provider may wish to gradually migrate to a transparency log that uses different cryptographic keys, a different cipher suite, or different deployment mode.

* A service provider may 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 transparency log for their own users.

Client implementations SHOULD generally be prepared to interact with multiple independent transparency logs. When multiple transparency logs are used as part of one application, all users 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 transparency logs behave honestly, then users observe a globally consistent view of the data associated with each label.

* If any transparency log behaves dishonestly such that the prior guarantee is not met, this will be detected in a timely manner by background monitoring or out-of-band communication.

5.1. Gradual Migration

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

1. Search queries should be executed against the old transparency log first, and then against the new transparency log only if the most recent version of a label in the old transparency log is a special application-defined ’tombstone’ entry.

2. Update queries should only be executed against the new transparency log, with the exception of adding a tombstone entry for the label to the old transparency log if one hasn’t been added already.

3. Both transparency logs should be monitored as they would be if they were run individually. Once the migration has completed and the old transparency log has stopped accepting modifications, the old transparency 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 label in the old transparency log gives users a clear indication as to which transparency log contains the most recent version of a label. Importantly, it prevents users from incorrectly accepting a stale version of a label if the new transparency log is unreachable.

5.2. Immediate Migration

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

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

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

3. The final tree size and root hash of the old transparency log is provided to users over a trustworthy channel. Users issue their final Monitor queries to complete monitoring up to this point. Label owners initiate monitoring state for the new transparency log by processing an Update for the migrated versions of their labels and verifying that the migration was done correctly. From then on, users will monitor only the new transparency log.

The final tree size and root hash of the prior transparency log need to be distributed to users in a way that guarantees all users have a globally consistent view. This can be done by storing them in a well-known label of the new transparency log. Users MUST process this well-known label as if they own it, so that they continue to monitor it for unexpected changes for the duration of the migration period. Alternatively, the final tree size and root hash may be distributed with the application's code distribution mechanism.

5.3. 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 they query 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.

6. 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 in that time frame. 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 (discussed in Section 7.2). 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 labels 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 have either expired, or to which the service operator's access control policy would never permit access. A version of a label expires when it is no longer possible to produce a valid search proof for the label-version pair, which happens when all of the necessary log entries have passed their *maximum lifetime* (as defined in [PROTO]).

Notably, the greatest version of a label is the only version that never expires through the maximum lifetime mechanism. However, service operators may define arbitrary access control policies that permanently block access to the greatest (or any other versions) of a label. The values corresponding to these label-version pairs may also be deleted without consideration to the rest of the protocol.

The second type of data, 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 a particular version of a label may have been deleted, the corresponding VRF output and commitment may still need to exist in the latest version of the transparency log's prefix tree to produce valid search proofs for other versions of the label. The exact mechanism for determining which data is safe to delete will depend on the protocol and implementation.

The distinction between user data and cryptographic data provides a valuable separation of concerns since [PROTO] does not provide a way for a service operator to convey its access control policy to a transparency log. That is, it allows the pruning of user data to be done entirely by application-defined code, while the pruning of cryptographic data can be done entirely by KT-specific code as a subsequent operation.

7. 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 operation 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 label, they're guaranteed that the result they receive represents the same result that any other user searching for the same label at roughly the same time would've seen. When a user modifies a label, they're guaranteed that other users will see the modification within a bounded amount of time, or will themselves permanently enter an invalid state as discussed below.

If the transparency log does not execute an operation 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 by the mechanisms described in Section 3.3. Importantly, this means that users must stay online for a 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 stale 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 the 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 or auditor do not collude to trick users into accepting malicious results.

* Contact Monitoring requires an assumption that the user that owns a label and all users that look up the label 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:

* The configured maximum amount of time by which a query response can be stale.

* The configured Reasonable Monitoring Window (described in Section 7.1 of [PROTO]), weighed against how frequently users execute background monitoring in practice.

* For logs that use the Contact Monitoring deployment mode: how frequently users engage in anonymous communication with the transparency log, or peer-to-peer communication with other users.

* For logs that use the Third-Party Auditing deployment mode: the configured maximum acceptable lag for an auditor.

* For logs that use the Third-Party Management deployment mode: the amount of lag, or potential inefficacy, of the service operator's approach to detecting forks.

7.1. State Loss

The security of KT often depends on the ability of users to maintain robust local state. Users that lose their state in a Contact Monitoring or Third-Party Auditing deployment will have a correspondingly reduced ability to detect if they were shown a fork, or if the transparency log later obscured data that they consumed.

In a Contact Monitoring deployment mode, this can happen when a user loses their state after consuming a version of a label that was created either within the Reasonable Monitoring Window, or within a portion of the log that was insufficiently gossipped. In a Third-Party Auditing deployment mode, this can happen when a user loses their state after consuming a version of a label that was created within the auditor's maximum acceptable lag.

Applications should consider the nature of possible state loss in their clients when configuring a transparency log and MUST ensure that the relevant protocol parameters are set in a way that appropriately mitigates this risk. For example, in a Contact Monitoring deployment, the Reasonable Monitoring Window and the duration between out-of-band communication attempts SHOULD be much less than the typical time between state loss events. Similarly, in

a Third-Party Auditing deployment, the maximum acceptable lag for an auditor SHOULD be much less than the typical time between state loss events.

In applications where client state is typically ephemeral (like a web page), or where state loss could possibly be triggered adversarially, a Third-Party Management deployment mode is RECOMMENDED. Alternatively, applications could also consider implementing a policy of not consuming label-version pairs that were inserted too recently. Once a label-version pair is outside of the Reasonable Monitoring Window in a Contact Monitoring deployment, or beyond the maximum acceptable auditor lag in a Third-Party Auditing deployment, the risks associated with state loss are often already sufficiently mitigated.

7.2. Privacy Guarantees

For applications deploying KT, service operators expect to be able to control when sensitive information is revealed. In particular, a service 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 label exists in the transparency log if the user obtains a valid search proof for that label. Similarly, KT only allows users to learn about the value of a label if the user obtains a valid search proof for that exact version of the label.

When a user was previously allowed to lookup or change a label's value but no longer is, KT prevents the user from learning whether or not the label's value has changed since the user's access was revoked. This is true even in Contact Monitoring mode, where users are still permitted to perform monitoring after their access to perform other queries has been revoked.

Applications determine the privacy of 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 labels. This prevents both 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 labels 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 user base, the frequency with which new users join, and the frequency with which existing users update their labels.

KT allows a service operator to obscure the size of its user base by batch inserting a large number of fake label-version pairs when a transparency log is first initialized. Similarly, KT also allows a service operator to obscure the rate at which "real" changes are made to the transparency log by padding "real" changes with the insertion of other fake label-version pairs, such that it creates the outside appearance of a constant baseline rate of insertions.

7.2.1. 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 labels or values. This is because labels 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 label 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 labels and values and their modification history. It also includes traffic patterns, such as how often a specific label is looked up.

8. Privacy Law Considerations

Consumer privacy laws often provide a _right to erasure_. This means 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, it primarily contains privacy-preserving VRF outputs and cryptographic commitments.

The value corresponding to a label is typically some serialized user account data, like a public key or internal identifier. KT uses cryptographic commitments to ensure that users interacting with a transparency log are unable to learn anything about a label's value until the transparency log explicitly provides the commitment's opening. A transparency log responding to an erasure request can delete the commitment opening and the associated data. This can be done immediately and permanently prevents recovery of the associated value.

Labels themselves are typically serialized end-user identifiers, like a username or email address. All labels are processed through a Verifiable Random Function, or VRF [RFC9381], which deterministically maps each label to a fixed-length pseudorandom value. The set of all labels stored in a transparency log is committed to by a prefix tree, and each version of the prefix tree is committed to by a log tree.

While all the VRF outputs corresponding to labels affected by an erasure request can be deleted from the most recent version of the prefix tree immediately, previous versions of the prefix tree will still contain the VRF outputs and will still be needed by the protocol for a bounded amount of time. This bound is defined by the log entry maximum lifetime discussed in [PROTO]. As such, the VRF outputs can only be fully purged from the transparency log once all log entries that contain them have passed their maximum lifetime. After this point, they are no longer necessary for the protocol to operate.

9. 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 (noting that this document refers to these keys as "labels"). It takes special care to turn the guarantee that all users agree on a set of labels and values into a guarantee that the mapping between end-users and their public keys is authentic. Critically, 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 each user's internal UUID as their label. While the UUID might be unique, it is not necessarily user-visible. When a user attempts to lookup a contact by username, the service operator translates the username into a user UUID under the hood. If this mapping (from username to UUID) is unauthenticated, the service operator could manipulate it to eavesdrop on conversations by returning the UUID for an account that it controls. From a security perspective, this would be equivalent to not using KT at all.

However, that's not to say that the use of internal UUIDs in KT is never appropriate. Many applications don't have a concept of a fixed explicit identifier, like a username, and instead rely on their UI (underpinned internally by a user's ID) to indicate to users whether a conversation is with a new person or someone they've previously contacted. The fact that the UI behaves this way makes the user's ID a user-visible identifier 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.) An example of this type of identifier would be an email address on an email service. A primary end-user identity SHOULD always be a label in KT, with the end-user's public keys and other account information 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 end-user identity. These identities are used solely for initial user discovery, during which they're converted into a primary end-user identity (like a UUID) that's used by the application to identify the end-user from then on. An example of this type of identity would be a username, since users can often change their username without disrupting existing communications. A secondary end-user identity SHOULD be a label in KT with the primary end-user identity as the associated value, such that it can be used to authenticate the user discovery process.

While likely helpful to most common applications, the distinction between handling primary and secondary end-user identities is not a guaranteed rule. Applications must be careful to ensure they fully capture the semantics of identity in their application with the labels and values they store in KT.

10. IANA Considerations

This document has no IANA actions.

11. References

11.1. Normative References

[PROTO] McMillion, B. and F. Linker, "Key Transparency Protocol", Work in Progress, Internet-Draft, draft-ietf-keytrans-protocol-03, 19 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-keytrans-protocol-03>.

[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>.

11.2. 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. Vト稿lテ。k, "Verifiable Random Functions (VRFs)", RFC 9381, DOI 10.17487/RFC9381, August 2023, <https://www.rfc-editor.org/rfc/rfc9381>.

[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/>.

Author’s Address

Brendan McMillion
Email: brendanmcmillion@gmail.com