Design Specification: Transparency Server

Key Transparency (KT) is a safe, publicly-auditable way to distribute public keys for end-to-end encryption. More generally, the same technology can also be used for tamper-evident logging, key usage detection, ensuring that a group of users agree on a shared value, and many other applications.

This document describes the design constraints, high-level construction, and corresponding implementation of a Transparency Server, based on the Merkle^2 construction described in academic literature.

Design Constraints

Efficient verification processes and small state. We want to put the least burden possible on the users of a KT server, to ensure that we cover high-performance or constrained applications. In particular, we want clients to keep only a constant-sized amount of state that represents the most recent tree root a user has seen.

No need for third-party auditing. Third-party auditing of the integrity of the tree structure, like is done in some versions of KT, has the potential to be incredibly expensive because the entire contents of the tree needs to be available for anyone on the Internet to download. It also just doesn't convey a huge amount of actual trust to end users, who may not know of or trust any of a server's auditors. However, clients may still gossip tree roots to ensure global consistency.

New entries should be added to the log immediately. Many use-cases of KT are significantly simplified by having a KT server that can add entries and provide a proof of inclusion to a signed tree root with a single network request. For example, in the use-case of distributing public keys for end-to-end encryption, users benefit from being able to immediately use new keys without resorting to a second less-efficient auditing mechanism.

Avoid cryptographic algorithms that don't have a straightforward path to being post-quantum secure. Some KT constructions achieve security in stronger threat models by relying on special properties of cryptographic algorithms like RSA. In the event that a powerful quantum computer is ever built, there's not guaranteed to be a drop-in replacement for these algorithms that would allow the service to continue operating securely.

High-Level Design

The proposal called Merkle^2 from the Literature Review is the closest to what we're looking for, as it:

- Doesn't require external monitoring of the tree structures,
- Relies on only hash functions and signatures, and
- Uses a Merkle log as the top-level data structure, which allows us to provide efficient consistency proofs between new and old versions of the state.

As the note in the literature review says, instead of storing prefix tree roots in every intermediate node of the log we can instead store them only in the leaf nodes. This way the implementation needs to maintain only one prefix tree which supports versioned lookups, rather than a distinct prefix tree for each interior node of the log tree.

Each leaf/entry of the log contains a commitment to the new value of a single lookup key, and the root hash of a prefix tree. The prefix tree maps each lookup key to the number of occurrences of that key in the log so far. Note that this means each prefix tree is different from the subsequent tree in the log, only by having one new value inserted or an existing value incremented by one.

In all interactions with the KT log (whether searching, inserting, or monitoring), clients have the option to request a consistency proof from the last epoch they're aware of to whatever the most recent epoch is. This is an optimization that generally prevents clients from needing to request consistency proofs specifically.

Authenticating Updates

Both in this construction and in Merkle^2, the tree structure itself doesn't fully prevent a log operator from adding malicious entries and then later covering them up so that they're unlikely to be found. This can be solved with one of a few approaches.

Owner Signing. When a user first claims a lookup key in the log, they store a signing key in the body of the version zero which is used to sign all subsequent versions. Searches for a user's lookup key return both version zero and the most recent version – version zero provides the signing key for authentication, and the most recent version provides the up-to-date account data.

Contact Monitoring. Instead of the user that owns a lookup key signing it, the users that lookup a key can keep a record of the most recent version of a key they saw and what position it was at in the tree. When they lookup the same key again later, they'll additionally check that a binary search for the key at the last version observed yields the same position in the tree.

Third-party Management. With this approach, the transparency log is operated by a third-party but all updates to the log are signed by the service provider with a Service Provider Key. This approach requires both the service provider and the log operator to collude to impersonate users. The log operator is unable to add malicious entries because they would not be signed by the Service Provider Key, and any malicious entries added by the service provider would be properly detected by users unless the log operator colludes to suppress them.

Owner Signing is more storage-efficient but requires a signature key that, if lost, means a lookup key is now unusable. Contact Monitoring requires more client-side storage and can potentially miss attacks when a key is never looked up again, but doesn't have a key that can be lost. Third-party management is the most efficient and flexible, but creates a third-party dependency.

Protocols

In simplest terms, looking up a specific key at a specific version in the log corresponds to executing a binary search in the log for the first entry that has that key at that version. Discovering which version of a key is most recent requires sampling the frontier of the log.

The frontier of a log with N entries consists of the entries at the following positions:

```
    X = The largest power of 2 less than N
    Entries = [X]
    while X < N:
        X += The largest power of 2 such that X is less than or
equal to N
        If X < N, then append X to Entries
```

Formally, the **Search Protocol** starts by looking up a requested key in the prefix tree at each entry in the frontier of the log, ensuring that the returned counters in the entries are monotonic, and determining the maximum counter. As mentioned, it then executes a binary search (possibly re-using lookups already done on the frontier, and still ensuring monotonicity) for the key at that version until it finds the correct entry. The commitment is taken from that entry and opened by the server. If the Owner Signing approach previously described is being used and the most recent version is not version zero, a binary search for version zero may also occur.

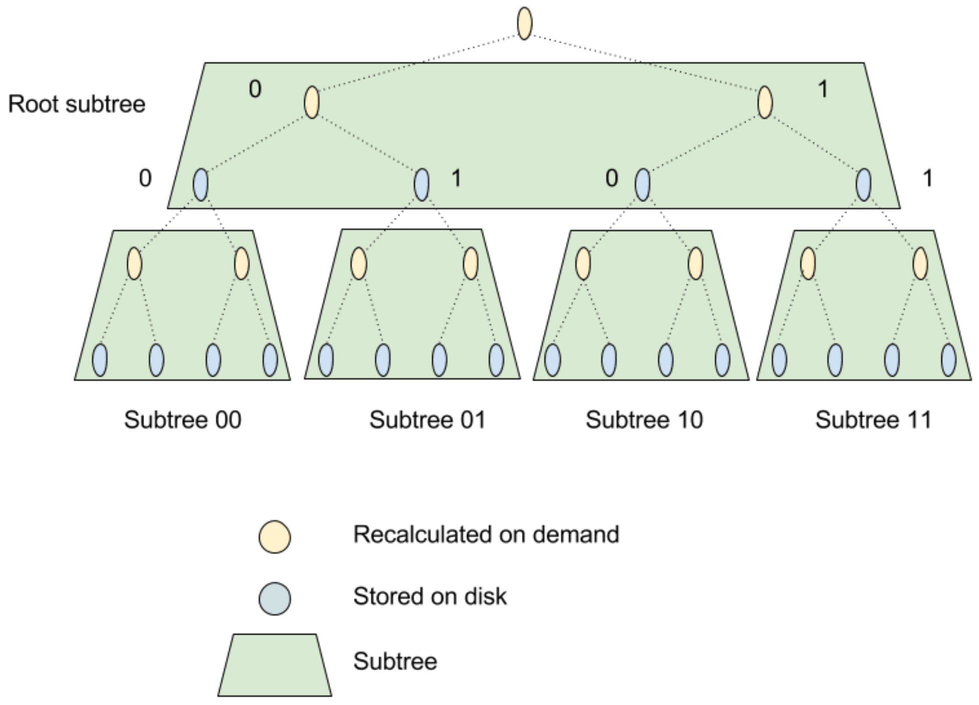
The search protocol has a preference to move around by powers of two, and this ensures that different users look at the same entries when searching the same parts of the tree. For example, if the log has N entries, rather than the "middle" being the entry at position N/2, it's the entry whose position is the largest power of 2 less than N. This prevents the log from easily adding dummy entries to hide misbehavior.

The **Update Protocol** adds a new version of a key to the log. The user generates the new value for their key, signing it or doing any sort of chaining, and then submits it to the KT server. The server applies the update and returns an inclusion proof (the output of the Search Protocol) for the new entry and a signature over the new root.

The **Monitoring Protocol** is executed in the background by the user that "owns" a lookup key in the log to check that no unauthorized changes have been made to their account, and also to check for potential log misbehavior. It consists of repeatedly searching for the same key and validating that either a new (authorized) version is returned, or the same version is returned in the same position in the log as before. The most recent valid search result, which is initially the output of the Update Protocol, is always saved to provide non-repudiable evidence of log misbehavior in the event that misbehavior is detected.

Implementation

Log Tree. The log tree implementation is designed to work with a standard key-value database. The tree is stored in the database in "chunks", which are 8-node-wide (or 4-node-deep) subtrees. Chunks are addressed by the id of the root node of the chunk. Only the leaf values of each chunk are stored, which in the context of the full log tree is either a leaf value or a cached intermediate hash. The intermediate hashes of each chunk are re-computed on the fly as necessary.

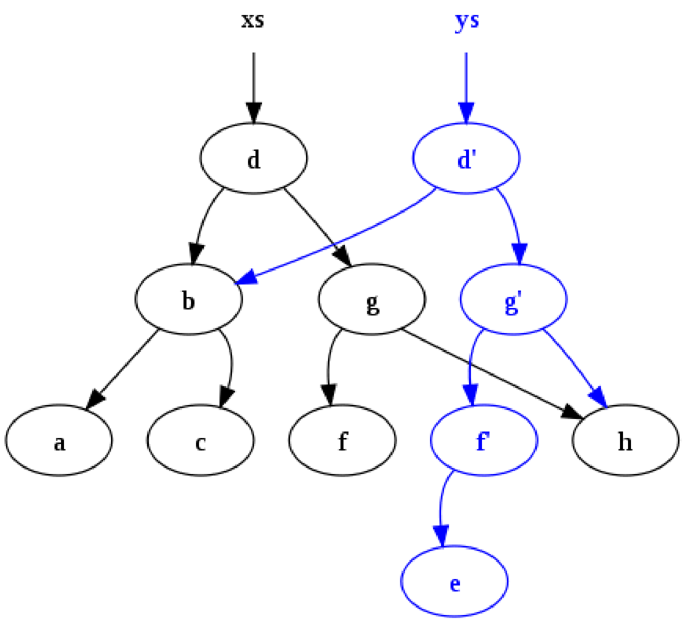


Chunking reduces the number of lookups required to answer a query about the log, and not storing the intermediate hashes of each chunk reduces the amount of storage space necessary by roughly 40%. Recomputing intermediate hashes on-the-fly doesn’t seem to create a significant computational burden.

Versioned lookups in the log tree are possible by simply ignoring any entries which may have been added or modified after the intended epoch.

Prefix Tree. The prefix tree implementation is also designed to work with a standard key-value database. Each modification to the tree produces a new version of the tree, and each version corresponds to a new entry in the database which is addressed by the version counter. Database entries contain the leaf node which was modified in this version, and the calculated copath of the leaf node. The steps of the copath may be either an unmodified leaf node, an empty node, or a pointer to a parent node in a previous version of the tree.

Searches in the tree correspond to a standard tree search, starting at the copath in the most recent version, and following pointers to other parent nodes as necessary. Note that this means searches can be executed in any version of the tree simply by starting the tree at a different database entry.


xs
ys
d
d'
b
g
g'
a
c
f
f'
h
e