Proton

Proton Key Transparency Whitepaper

Thore Göbel (ETH Zurich),
Daniel Huigens (Proton)

2024-01-10

Contents

1 Introduction

At Proton, we are always working to increase the security and trustworthiness of our applications. We use end-to-end encryption to protect user data, and our applications and cryptography libraries are open source[1], such that anyone can review them, and verify the security of our applications.

In order to work, our end-to-end encryption uses key pairs: a private key only known to its owner, and a corresponding public key known to the general public. When you compose an email, when you share a calendar, or when you share a password vault, you need the public key of the recipient. With the public key you can encrypt the data you want to send. Once encrypted, only the owner can decrypt it again, using the private key.

The Achilles heel of end-to-end encryption is getting the public key. To get it, the Proton client app looks up the recipient's username on a key server run by Proton, and the key server returns the public key. The key server stores a database that maps usernames to public keys.

A sophisticated attacker could try to compromise the key server, or try to otherwise intercept the key lookup. The attacker could then launch a Man-in-the-Middle (MITM) attack: it could modify the lookup to return a wrong public key, and use that to intercept and read data before re-encrypting it with the correct public key. To prevent this MITM attack, the client app needs to make sure that it gets the *correct* public key of the recipient (instead of simply trusting the key server). But how can the client app verify that the public key is correct?

So far, our solution to this problem was *Address Verification*[2]. It allows users to manually verify each others' public address keys out-of-band and pin them. Once a contact's key is pinned, Proton clients will warn the user when another, unverified key is used by that contact. However, Address Verification is manual and only protects users who enable it. It also requires users to have an out-of-band channel, which may not be the case if you are an anonymous source emailing a journalist. Therefore, we set out to find a solution that is automatic and protects all users.

Key Transparency (KT)[3] offers a new approach to the problem of getting correct public keys. It improves over Address Verification by automatically running in the background and not requiring any user interaction. With KT, whenever a Proton client looks up a key, e.g., when sending an email or when sharing a password vault, it checks that the key it received from the key server is logged in the KT system. Additionally, Proton clients monitor their own keys and check that only their real keys are logged, i.e., no other, possibly malicious keys were logged in KT. If an unexpected key is found, the client warns the user.

[1] https://proton.me/community/open-source
[2] https://proton.me/support/address-verification
[3] https://proton.me/support/key-transparency

KT ensures that MITM attacks at the address key level will always result in a warning. In other words, Proton users no longer need to trust Proton's key server to serve correct public keys. With KT, users enjoy a similar level of security as with Address Verification, but without having to manually compare key fingerprints.

It is important to note that the problem that KT solves (malicious or hacked key servers returning wrong public keys) is not unique to Proton. It affects every end-to-end encrypted service, no matter whether it is email, instant messaging, or file sharing. For a long time, the only solution was manual out-of-band verification. In 2014, researchers proposed the first KT design called CONIKS [1]. This was followed up by the later academic works of SEEMless [2], Parakeet [3], and others. Our Key Transparency design (dubbed ProtonKT) is inspired by CONIKS and SEEMless. We made some additions to support catch-all addresses, as well as disabling and deleting accounts, among other things.

As the name suggests, Key Transparency (as a general concept) is inspired by Certificate Transparency (CT)[4]. Both KT and CT make use of a Merkle hash tree to log certificates and public keys respectively. In addition to this design similarity, ProtonKT makes use of the existing CT ecosystem to bootstrap trust.

This whitepaper describes how the Proton Key Transparency (ProtonKT) protocol works, including its architecture and its subprotocols. The whitepaper also provides a privacy and a security analysis. We assume that the reader is familiar with basic concepts such as hash functions, Merkle hash trees, and public key cryptography.

[4] https://certificate.transparency.dev

2 Background

2.1 Keys at Proton

Proton uses a variety of asymmetric PGP keys across its products. Proton clients automatically manage these keys on behalf of Proton users. The main types of keys are:

- *User keys* (sometimes called *account keys*): User keys encrypt account-specific data. For example, contacts are encrypted and signed with a user key. User keys are local to a user, i.e., not known to or used by other Proton users.
- *Address keys* (sometimes called *email encryption keys*): Address keys are used for sending end-to-end encrypted data to other users: sending email, sharing calendars, sharing files on Drive, or sharing password vaults.
- Other keys: e.g., *calendar keys* to encrypt calendars or *share keys* and *node keys* to encrypt files on Drive. These keys are part of key hierarchies that lead back to an address key. For more information, see the Proton Calendar Security Model[1] and the Proton Drive Security Model[2].

Users can manage their user keys and their address keys in the web client (see Figure 2.1). For example, users with old Proton accounts that were created with RSA-2048 keys can generate new Curve25519 keys. Old keys are retained to allow decrypting old data.

Address keys are central when sharing data in an end-to-end encrypted manner. Address keys are used to send end-to-end encrypted data (preventing MITM), and to verify signatures when receiving data (preventing impersonation). Thus the address keys are the keys that are protected by KT. All other key types are not included in KT, and do not need to.

Keys have flags. For example, when the user marks a key as "obsolete" or as "compromised" in the web client, this is recorded in the key flags. These flags are also included in and protected by KT.

2.2 Verifiable Random Functions

Since VRFs are not a common primitive, this section briefly introduces them. In KT, VRFs will be important for privacy.

The output of a *pseudorandom function* f_s should be indistinguishable from the output of a truly random function. This means that – by design – without knowing the PRF seed s one cannot distinguish whether f_s was correctly computed.

[1] https://proton.me/blog/protoncalendar-security-model
[2] https://proton.me/blog/protondrive-security

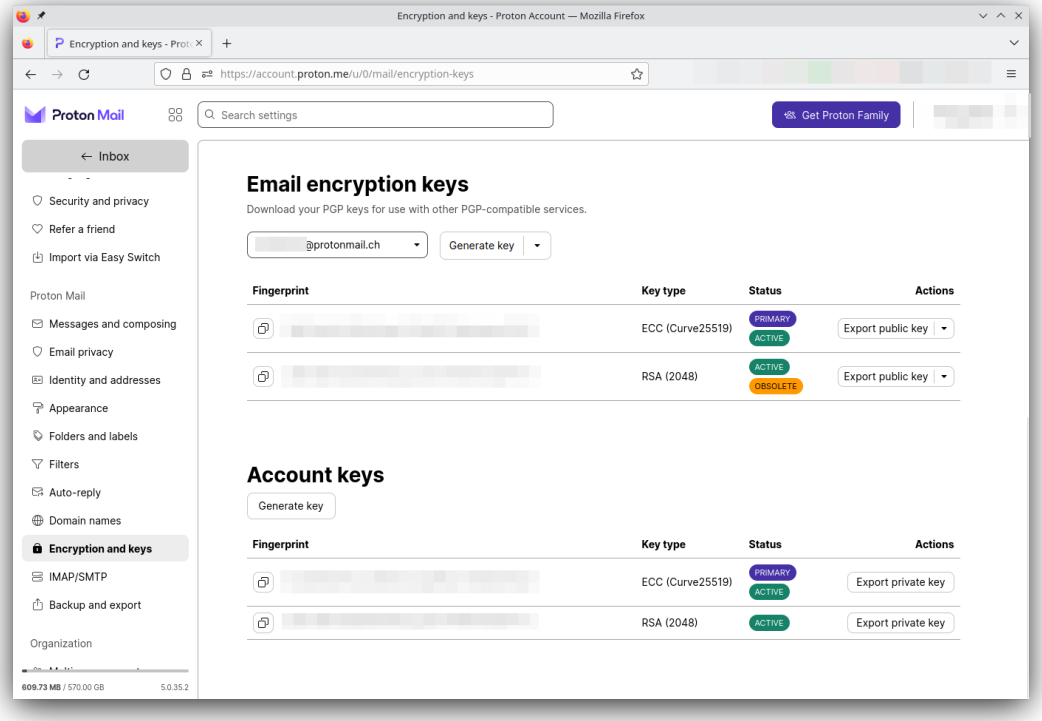

Figure 2.1: Managing Address Keys (top) and User Keys (bottom)

A *verifiable random function (VRF)*, introduced in 1999 [4], provides such a correctness proof alongside the output value.

A more informal way to think about VRFs is in terms of hash functions: A plain hash function is publicly computable and verifiable. A keyed hash function is privately computable and privately verifiable. A VRF is privately computable and publicly verifiable.

VRFs achieve this by having an asymmetric key pair $(\mathsf{sk}, \mathsf{pk})$. To evaluate the VRF, one needs the secret key sk. (In practice this is useful if we want to force an adversary to go through an oracle to evaluate the "hash" function, e.g. to apply access control or rate limiting. A common example is preventing offline enumeration attacks of hash-based data stores.) Given a VRF output and a proof, one can use the public key pk to check that the output is correctly computed. (This is useful if we don't trust the party evaluating the VRF.)

Algorithms We follow the notation of RFC 9381 (an informational RFC from the IRTF's CFRG) [5], which defines a VRF as a set of algorithms:

$$\beta \leftarrow hash(\mathsf{sk}, \alpha)$$

$$\pi \leftarrow prove(\mathsf{sk}, \alpha)$$

$$\beta / \perp \leftarrow verify(\mathsf{pk}, \alpha, \pi)$$

where α is the input value, β is the output value, and π the proof. RFC 9381 also defines another function $proofToHash$:

$$\beta \leftarrow proofToHash(\pi) \quad \text{such that} \quad hash(\mathsf{sk}, \alpha) = proofToHash(prove(\mathsf{sk}, \alpha))$$

which allows it to specify $proofToHash$ instead of $hash$. This also allows $verify$ to compute the hash β itself, which means that in practice the prover/evaluator (who holds sk) only needs to output π.

Properties VRFs have the following security properties:

- *Pseudorandomness*: Knowing pk but not sk, an adversary can choose an input α and gets the output β, but not its corresponding π. The adversary can only distinguish β from random with a negligible advantage.

  Note that β *is* distinguishable from random if you know π (verify the proof) or sk (recompute β).

- *Uniqueness*: For a fixed pk, for all inputs α it is hard to find two different proofs $\pi_1 \neq \pi_2$ that correspond to different outputs $\beta_1 \neq \beta_2$ In other words, each input value α has a unique output value β. [3]

- *Collision Resistance*: It is hard to find a pk, two different inputs $\alpha_1 \neq \alpha_2$, and proofs π_1, π_2, such that they correspond to the same outputs $\beta_1 = \beta_2$. In other words, no two input values should collide on the output value.

Depending on the VRF construction, some security properties don't hold if the key pair is maliciously generated. In EC-based VRFs, clients can validate the key to detect this. See section 7 of RFC 9381 [5].

[3] But it is admissable to have two different proofs $\pi_1 \neq \pi_2$ for α that correspond to the same β.

3 Specification

This chapter specifies the ProtonKT security protocol. We begin by giving a high-level overview. Next, we describe the protocol in more detail (but still informal): how labels (email addresses) and values (public keys) in the key directory are defined, how the Merkle tree is built, and how Self Audits and External Audits work. Finally, we give a more formal specification of ProtonKT, stating the subprotocols and giving their detailed steps as message sequence diagrams.

3.1 Overview

This section gives a high-level overview of how ProtonKT works. We begin by describing the different parts, before we finally look at the subprotocols in detail.

Label–Value Store/Dictionary At its core, a key server maintains a dictionary that maps email addresses to public keys. We call this a map from *labels* (email addresses) to *values* (public keys). We don't use the term key–value store to avoid confusion between dictionary keys and cryptographic keys.

Roles and Basic Functionality ProtonKT has the same roles as other KT protocols: a server, some clients, and some auditors.

The server hosts the key directory. Clients can upload their own public key to the server, and clients can query (look up) other clients' public keys. Clients also regularly audit their own keys and check that they are correct. The server responds to queries with the requested public key, as well as an inclusion proof leading to a tree root and a commitment to that tree root. External auditors check that the server correctly constructs the tree and does not equivocate (present split-views). Clients have a trust relationship with the auditors: they trust that at least one honest auditor is auditing the tree.

Merkle Tree The key directory is a Merkle Hash Tree. Each leaf corresponds to a (username, revision) pair. That is, whenever a user updates its key, a new leaf is inserted. In ProtonKT, each user has a dedicated subtree: all its revisions belong to the same subtree. Revision 0 is the left-most leaf of a user's subtree, revision n is the right-most leaf.

The ProtonKT server can also delete old values. External Auditors check that the server only deletes revisions that have been superseded by a new revision more than 90 days ago. That is, the latest leaf is always retained. This allows the server to clean up old values.

Root Hash Consistency via Commitments ProtonKT introduces a new approach for committing to the root hash: it requests a web-PKI certificate that contains the

root hash (or more specifically, the hash chain of root hashes) from a CA (e.g. Let's Encrypt). The CA issues the certificate, and it will be logged in CT logs. Assuming that CT logs are append-only and trusted, this allows ProtonKT to publicly commit to the tree root.

Clients then verify the commitment to the root by verifying the certificate and the SCTs inside it. Clients trust the SCTs as a promise of CT log inclusion, and they trust that some External Auditor somewhere is scanning CT logs for equivocation.

Epochs The server publishes an updated tree at regular intervals, called epochs. It collects all insertion requests, inserts them as a batch into the tree, requests the web-PKI certificate for this new tree, and then announces this as a new epoch.

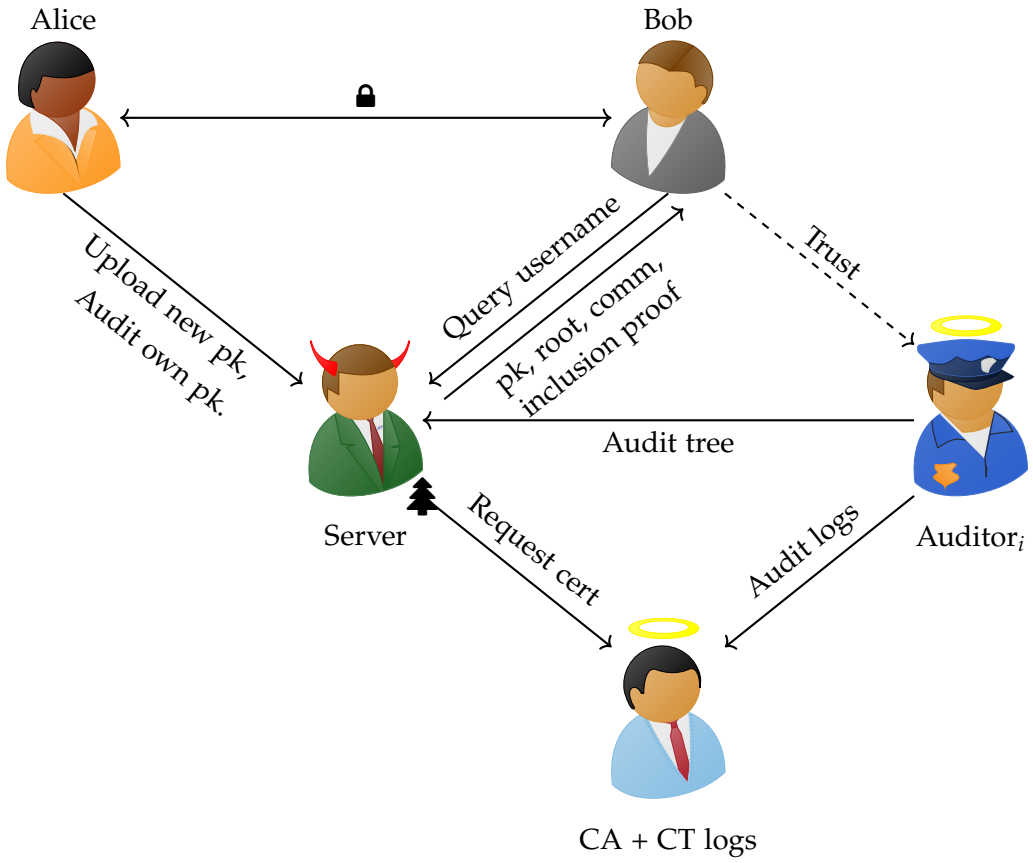

Figure 3.1: ProtonKT Overview

3.2 Labels: Email Addresses

Informally, the labels in the key directory are email addresses. However, there are some subtleties. Consider the following use cases that make email addresses complicated:

- *Address normalization:* Email addresses can have different formats that all map to the same normalized address. For example, `john.doe@proton.me`, `johndoe@proton.me`, and `john.doe+alias@proton.me` are all equivalent. Mail sent to any of these addresses will be delivered to the same inbox.

To solve this, we need to apply a set of normalization rules that are applied to a label before it interacts with ProtonKT. The normalization rules are a *public system parameter* that is defined in advance. This is important: otherwise the server could give arbitrary and different rules to different clients. For example consider the case where Bob wants to query for `alice@proton.me`. The server could give Bob a rule to strip all vowels from the local part. Bob would then look up the public key for `lc@proton.me` and send his email to `alice@proton.me` encrypted with the key for `lc@proton.me`. But when Alice verifies her key history, the server returns only non-malicious rules.

Proton has two normalization rules for the local part (the domain part is not modified): (1) First, if a plus (+) character is present, strip it and everything after it. (2) Then strip all dashes (-), underscores (_), and dots (.).

- *Multiple addresses:* One user account can have multiple email addresses (each with their own PGP key pair), with one being the primary address. This does not affect Bob's lookups of an address (since from Bob's perspective it does not matter which account controls an address).

  Alice, however, needs to call run Self Audits for *all* of the addresses that are associated with her account. If she doesn't monitor an address, a malicious server could insert a fake address into ProtonKT and re-encrypt all emails for Alice towards her real key on-the-fly.

- *Internal vs External vs Non-Proton Addresses:* When a Proton client looks up an email address, this can either either be a Proton or a Non-Proton Address, depending on whether there exists a Proton account with this address. Proton addresses are again split into Internal and External Addresses.

  A *Non-Proton Address* is an email address belonging to a user who is not a Proton customer (e.g. alice@example.com). An *Internal Address* is a Proton-managed email address (bob@proton.me or custom domains). An *External Address* (e.g. charlie@example.com) is an email address that is linked to a Proton account, but whose email is not handled by Proton Mail.[1]

  For Internal Addresses, the server should either return keys and an inclusion proof, or no keys and an absence or obsolescence proof.

  For External Addresses, the server may return email encryption keys that it found in the Web Key Directory (WKD) [6] (since email is hosted elsewhere). The server may also return data encryption keys, used e.g. for Proton Drive. The former should have an absence proof in KT, and the latter should have an inclusion proof.

  For Non-Proton Addresses, the server may also return keys that it found in the WKD. This way clients can automatically encrypt emails to it. These keys won't be in ProtonKT, thus KT should return an absence proof.

  Clients should always verify the claims of the server against KT, independent of the address type. KT proofs should not be skipped for Non-Proton Addresses.

  Overall, this means that labels are typed.

- *Custom domains:* In Proton, users can bring their own domain and delegate email handling to Proton. These are called *custom domains*. This means that Internal Addresses can have arbitrary domains, and not just @proton.me. The only

[1] This allows users to sign up for a Proton account with a non-Proton email address.

exception are domains from well-known email providers such as gmail.com or outlook.com. These exceptional domains can be assumed to never be Internal Addresses – but they can be External Addresses.

- *Catch-all:* Owners of custom domains can configure a so-called *catch-all address*. The catch-all address of a domain receives all email that is sent to non-existent addresses for this domain. For example, any email sent to `nonexistent@example.com` will be delivered to the catch-all address `admin@example.com`. (This assumes that the administrator of `example.com` has configured this redirection. If they haven't, the email will bounce.)

  When a Proton client looks up `nonexistent@example.com`, it should get a key back, so that it can still encrypt the email. This should be the same key as the key for `admin@example.com`, so that the catch-all address can decrypt it.

  Thus whenever a client looks up an address, the KT server returns two proofs: (1) a proof for the queried address, and (2) a proof for the catch-all address. If the first proof is an absence proof (or an obsolescence proof, see below) and the second is an inclusion proof, then the client can deduce that it should use the key for the catch-all address. If the domain owner has not configured a catch-all address, the second proof is an absence proof.

  ProtonKT uses `@example.com` as the label for the catch-all address, i.e. an @ followed by the domain name.

3.3 Values

ProtonKT maps labels (email addresses) to values (public keys). In this section, we discuss how these values are structured in ProtonKT, and how these values are returned upon querying.

When a client queries for a label, there are three possible outcomes: *absence*, *inclusion*, and *obsolescence*. Depending on the outcome, the returned value differs: For absence, the value is empty. For inclusion, the value are public keys, represented by a data structure called *Signed Key List (SKL)*. For obsolescence, the value is a special *ObsolescenceToken*. Let us now look at these values in detail.

3.3.1 Signed Key Lists

A *Signed Key List (SKL)* is a list of public key fingerprints signed by the primary key. The SKL data structure looks as follows:

```
struct {
    boolean primary;
    uint64 flags;
    string fingerprint; // hex-encoded SHA-1/SHA-256 hash of the public key
    vector<string> SHA256Fingerprints; // fingerprint of the key
                                          and of its subkeys, if any
} KeyListItem;

struct {
    vector<KeyListItem> items;
} KeyList;
```

```
struct {
    string data;          // KeyList.to_json().to_string()
    string signature;     // Armored PGP signature over data
} SignedKeyList;
```

Each *KeyListItem* corresponds to one PGP key. It has a fingerprint, and possibly a list of fingerprints of its subkeys. [2] It also has a primary bit and some flags. See section 2.1 for an explanation of the flags. Exactly one *KeyListItem* should have the primary bit set.

The *SignedKeyList* contains a PGP signature over a JSON-encoded list of *KeyListItem*s. This signature *should* be produced by the primary *KeyListItem* (but *may* be produced by any of the *KeyListItem*s). The signature ensures that the server cannot tamper with the *data* field. (KT would make such tampering transparent. Historically the SKL data structure is also used elsewhere by Proton, hence the signature.)

Note that while the *SignedKeyList* uses PGP features (fingerprints, signatures), this data structure is specified by Proton, not by OpenPGP.

Finally note that the public keys themselves are *not* included in the *SKL.data*, only their fingerprints. Clients need to query the public keys from the key server (like they would without KT). Clients then use the *SKL.data* to verify that the returned public keys are consistent with KT. In an abuse of notation, and for simplicity, we nevertheless say that KT queries return "keys", instead of "fingerprints".

3.3.2 Disabled Addresses and Obsolescence Tokens

Email addresses can become *disabled* in different situations: If users delete their account, their email address is disabled so that it cannot be reused. To handle abuse, Proton can disable user accounts and the associated email addresses. Businesses who manage their organization on Proton can disable an email address, e.g. when an employee leaves.

In a first KT design, one would start by considering two cases either a label is present and has a value (inclusion proof), or it is absent and has no value (absence proof). Disabled addresses create a third use case: a label is present and had values in the past, but now it should be "gone".

Because of the append-only-ness property of KT, we cannot simply delete the label to make it absent. [3] In addition, we want all server actions to be transparent, so the fact that the server disabled an address should be logged. We also want to be able to disable an address immediately in the next epoch.

Therefore we need to insert a new value for the same label (email address). For example, we could insert an SKL without any keys, i.e. with an empty data field. With this empty SKL in the tree, we can create an inclusion proof committing to this empty set of keys.

However, an auditor with access to the tree can then easily recognize this empty *SKL.data* value. For privacy, we want that an auditor cannot distinguish between

[2] In PGP, the fingerprint is the hash of the public keying material, either using SHA-256 or SHA-1, depending on the version. Proton additionally adds SHA-256 hashes to mitigate against attacks on SHA-1.

[3] A malicious server can of course reset a value to being absent. However, this should be flagged as a protocol violation by an honest auditor or by a client doing a self-audit.

active and disabled addresses. Thus instead of an empty data field, we insert an *ObsolescenceToken*. It is defined as:

$$ObsolescenceToken = UnixTime.now().asU64().toHex() \;||\; SecureRandom().getBytes(20).toHex()$$

We concatenate a timestamp of 64 bits / 16 hex characters with some randomness of 160 bits / 20 bytes / 40 hex characters to form a 56 hex character string.

The randomness makes it harder for an auditor to distinguish a present leaf (which contains $h(SKL.data)$) from an obsolete leaf (which contains $h(ObsolescenceToken)$, see section 3.6). A malicious auditor can guess the timestamp based on the epoch id when the obsolete leaf was inserted, but it would still need to brute-force the 160 random bits.

The timestamp is included to commit the time at which the address was disabled. While the server can set any time, it does immutably commit to it, and KT will ensure that everyone sees a consistent view of the timestamp. The SKL also contains a (hidden) timestamp: the signature field has a timestamp, because the OpenPGP standard includes timestamps in PGP signatures. Overall, the timestamp is an informational feature (e.g. to show a time in the UI).

3.3.3 Query Outputs: Values for Absence, Inclusion, Obsolescence

In reality, when a client queries the server for a label, the server returns more than just an SKL or an Obsolescence Token. It also returns additional metadata, for example the revision and the epoch at which this value was inserted. In the REST-API, all the metadata fields are included in the SKL struct for implementation convenience. That is, a query will always return a `SignedKeyListWithMeta` struct. All its fields are nullable, as indicated by the question mark "?". Which fields must be present and which are null depends on the outcome type (absence/inclusion/obsolescence).

```
struct {
      string? data;
      string? signature;

      /* Metadata */
      uint64? revision;
      uint64? min_epoch_id;
      uint64? max_epoch_id;
      uint64? expected_min_epoch_id;
      string? obsolescence_token;
} SignedKeyListWithMeta;
```

We can formalize this as follows: Querying a label (an email address) results in a *query outcome* $O = (\tau, \mathsf{rev}, \mathsf{val})$. The query outcome is the *outcome type* τ (absence/inclusion/obsolescence), the latest revision rev of the value, and the *outcome values* val (e.g. pk_A). For each type, the following values must be present in the SKL, thus forming the val:

- $\mathsf{val}_{abs} = \emptyset$
- $\mathsf{val}_{incl} = \{data, minEpochId\}$
- $\mathsf{val}_{obs} = \{ObsolescenceToken, minEpochId\}$

For absence, the entire SKL is null (there is no value, so there is no latest revision). For inclusion and obsolescence, we have for the other values:

- *revision* should be set.
- *signature* we could technically agree on, but it is not very useful: we don't have any keys, so we use KT to at least agree on some untrusted keys in *data*, and then we would verify the signature using those untrusted keys. (The signature is useful to the holder of the private keys. The fact that the signature exists proves to them that they indeed requested this key to be inserted earlier.)
- *maxEpochId* for revision t is implied by the *minEpochId* of revision $t+1$ (see section 3.4). It is not used in the current protocol, but is still present as a helper field from an earlier ProtonKT version.
- *expectedMinEpochId* is used for values that are not yet included in the tree; so we cannot agree on it. If the server answers a query with a value that will only be included in the next epoch, *minEpochId* is null and instead *expectedMinEpochId* must be set.

In our security properties later, we will require that clients who query a label always agree on the outcome $O = (\tau, \mathsf{rev}, \mathsf{val})$.

3.4 Epochs

We divide time into *epochs*. Epochs are identified by their *epoch id*, a strictly increasing, positive integer.

All key insertion and update requests are collected into batches, which are then collectively inserted into the KT tree. When a batch is fully processed a new tree is published, marking a new epoch.

This means that there is a gap between a client's insertion request and the publishing of a new epoch. Clients inserting/updating their keys use self-auditing to verify that their insertion request is fulfilled in the next epoch.

To allow clients to immediately use new keys, the KT server may respond to queries with the new key immediately, even if the next epoch is not yet published. In this case, the server can not yet provide a tree inclusion proof. It returns the key anyway, together with the *expectedMinEpochId* when the key will be included in the tree. The querying client must store the received value locally and later audit that this value was correctly inserted before or during *expectedMinEpochId*.

This still fulfills the transparency requirement: if the server fails to include the key, the client has evidence of server misbehavior. (This is local evidence. That is, the client cannot convince other clients of the server's misbehavior. To fix that, the server would need to sign the query response.)

Intervals Normally a ProtonKT epoch should be published every 4 hours. The maximum publishing interval is 72 hours; after that the server is considered to be misbehaving.

Epoch IDs in the SKL The epoch IDs in the SKL-with-Metadata have the following meaning:

- $minEpochId$: is the epoch in which the SKL was inserted into the tree. It can be null if the SKL is not yet inserted.

  $minEpochId$ is committed into the tree via the leaf hash.

- $maxEpochId$: is the last epoch during which this SKL was the latest SKL in the tree. If the SKL is not yet inserted, $maxEpochId$ is null. If the SKL is inserted and the latest one, $maxEpochId$ is equal to the latest epoch id. (I.e. $maxEpochId$ changes every time a new epoch is published.) If the SKL is inserted and is superseded by a newer SKL, $maxEpochId$ is set to the id of the epoch just before the insertion of the superseding SKL.

  In other words, $[minEpochId, maxEpochId]$ is the inclusive interval of the time where an SKL was the latest one in the tree. [4]

  $maxEpochId$ is not explicitly committed into the tree, but it is implicit from the next higher revisions $minEpochId$ (or, if the next higher revision is absent, it must be equal to the latest epoch).

- $expectedMinEpochId$: is used by clients to locally store the epoch in which they expect a fresh SKL to be included in KT. Once an SKL is included $expectedMinEpochId$ is always null.

  This happens (1) when a client generates a new SKL for itself and uploads it for insertion, and (2) if the server answers another client's query with a new key that is not yet included in KT (because the next epoch has not yet been issued).

  In both cases, the server chooses a $expectedMinEpochId$. The client then locally stores the SKL and uses $expectedMinEpochId$ as a non-binding hint for when to audit that the SKL was included.

  $expectedMinEpochId$ trivially cannot be committed into the tree.

3.5 The Merkle Hash Tree

In this section we describe the Merkle Hash Tree (MHT) used in ProtonKT. First we define how ProtonKT constructs the tree, and what is hashed into it. Later we define the *inclusion* and *absence proofs* (for querying values), as well as the *update proofs* (proving append-only-ness between trees). [5]

3.5.1 Leaf indices

In ProtonKT, every (email address, revision) pair maps to a unique leaf in the tree. Proton clients look at the value that is stored at this leaf to look up the public keys of an email address at a given revision.

The index of a leaf is calculated by taking the VRF hash of the label and concatenating it with the revision.

$$idx = VRF.hash(\mathsf{sk}, \mathsf{label}) \;||\; \mathsf{rev}$$

[4] Recall that there can be multiple SKLs in the tree, because new SKLs are always appended.

[5] *Update proofs* are called *consistency proofs* in CT, because they prove that the set of elements in two trees are consistent. In KT we also have "consistency" in a second context: the consistency of the tree root between different clients and/or auditors. To avoid confusion about which consistency we mean, we use the term *update proof* instead.

ProtonKT instantiates VRF with ECVRF-EDWARDS25519-SHA512-TAI [5], which has output size 512 bits. A leaf index in ProtonKT has 256 bits. The first 224 bits are from the VRF hash (ignoring the remaining $512 - 224 = 288$ bits), the other 32 bits are the revision. Hence the Merkle tree has a fixed depth of 256 levels. Each bit of the leaf index (0/1) defines whether a left or a right branch is taken. Being a bit-string of 32 bits, revisions as integers go from 0 up to $2^{32} - 1$ (inclusive).

Note that each label (each email address) has a unique subtree. This subtree is defined by the trailing index bits of the revision.

Also note that every label has a unique, non-colliding subtree. Uniqueness of VRFs ensures that every label has exactly one VRF hash (and not two or more). Collision resistance of VRFs ensures that no two labels map to the same VRF hash. More precisely, it is hard for a poly-time adversary to find cases breaking these two properties.

For more details on VRFs, see RFC 9381 [5].

3.5.2 Tree Construction

The tree is at the core of KT: it allows us to efficiently achieve consistency by simply comparing tree root hashes. In this section we describe how the tree is constructed in ProtonKT.

Nodes The MHT consists of leaf nodes and inner nodes. The topmost inner node is called the *root*. Each node has exactly one parent, except the root which has no parent. All inner nodes have exactly two children: a left child and a right child.

Hashing Each node stores a hash. The leaf hash is described below. The inner node hash is computed as:

$$h(hash_{left} \mid\mid hash_{right})$$

By hashing from the leaves up towards the root, we build the hash tree. The hash of the root node is called the *tree root hash*, or *root hash*, or *tree hash*. We can identify the node with its hash, i.e. sometimes we will say "tree root" when we mean the tree root hash.

Proton instantiates the hash function h with SHA-256.

Leaf Hashes Recall that each leaf has a leaf index which corresponds to a label and a revision. Each leaf stores such a value. If the value is present, the leaf hash is computed as:

$$h(h(SignedKeyList.data) \mid\mid minEpochId)$$

If the value is obsolete, the leaf hash is computed as:

$$h(h(ObsolescenceToken) \mid\mid minEpochId)$$

If there is no value at the leaf, the leaf hash is set to 0^n, i.e. a string of n zeros, where n is the output size of the hash function (in ProtonKT $n = 256$).

Note that *SKL.data* is a JSON-encoded string while *ObsolescenceToken* is a 56-hex-character string. Clients *must* check that the encoding is the one they expect for the respective token type (else we can have type confusion).

Overall, this leaf hashing commits the values *SKL.data*, *ObsolescenceToken*, and *minEpochId* into the tree. The *revision* is implicitly committed through the leaf indices. This allows all parties to agree on $(\tau, \mathsf{rev}, \mathsf{val})$, given that they agree on the root hash.

Label Subtree and Revision/User Subtrees The Merkle Tree in ProtonKT has a fixed depth of 256 levels. We logically split the tree into an upper part (from the root at depth 1 up to and including the inner nodes at depth 224), and a lower part (from depth 225 until depth 256, i.e. 32 levels).

We call the upper part the *label subtree*. The leaves of the label subtree (i.e. the inner nodes of the overall tree at level 224) correspond to the labels, i.e. to email addresses. By taking the first 224 bits (28 bytes) of the VRF hash of an email address, we thus obtain a unique (up to hash collisions) label-subtree-leaf: starting from the root, go left if the VRF bit is 0, or right if it is 1. Repeat with the next VRF bit for the next level of the tree.

Each label-subtree-leaf is the root of a *revision subtree*. These revision subtrees are formed by the remaining 32 levels in the overall tree. The lower part of the overall tree is therefore made up of many parallel revision subtrees. Because each user has their own distinct revision subtree, we also call it the *user subtree*. The leaves of a revision subtree contain the increasing revisions of the values for this label from left to right. The leaves of all revision subtrees together are exactly the leaves of the overall tree.

Sparseness The label subtree is sparse, while the individual revision subtrees are dense. That is, within a revision subtrees all values are in the lower left corner while all leaves on the right are empty. In the label subtree the non-empty leaves are evenly distributed due to the VRF hash.

Overall the Merkle Tree in ProtonKT is sparse. This is in contrast to the Merkle Tree in CT which is dense, because values are inserted from left to right (like in the revision subtree).

Size There are $2^{224} \approx 10^{67}$ label-subtree-leaves. This is how many labels (email addresses) can be inserted into the tree. Each revision subtree has $2^{32} = 4'294'967'296$ leaves. Thus every email address can have roughly up to 4 billion revisions.

With proper rate limiting and abuse prevention, none of these bounds should be an issue in practice (also recall that in Proton's setting keys rarely change). If the bounds are reached, one could start a second tree.

Revisions and Inserting Values Initially the revision subtree of a label starts completely empty. The first value is inserted with revision 1. After that every update increments the revision by 1. This means that revision 0 is always absent!

3.5.3 Proofs of Inclusion and Absence

The ProtonKT proofs are of the form

$$(\tau, \pi_{vrf}, \pi_{copath})$$

τ is the outcome type. π_{vrf} is the VRF proof that can be verified with the server's VRF public key, and that can be used to calculate the VRF hash to obtain the leaf index.

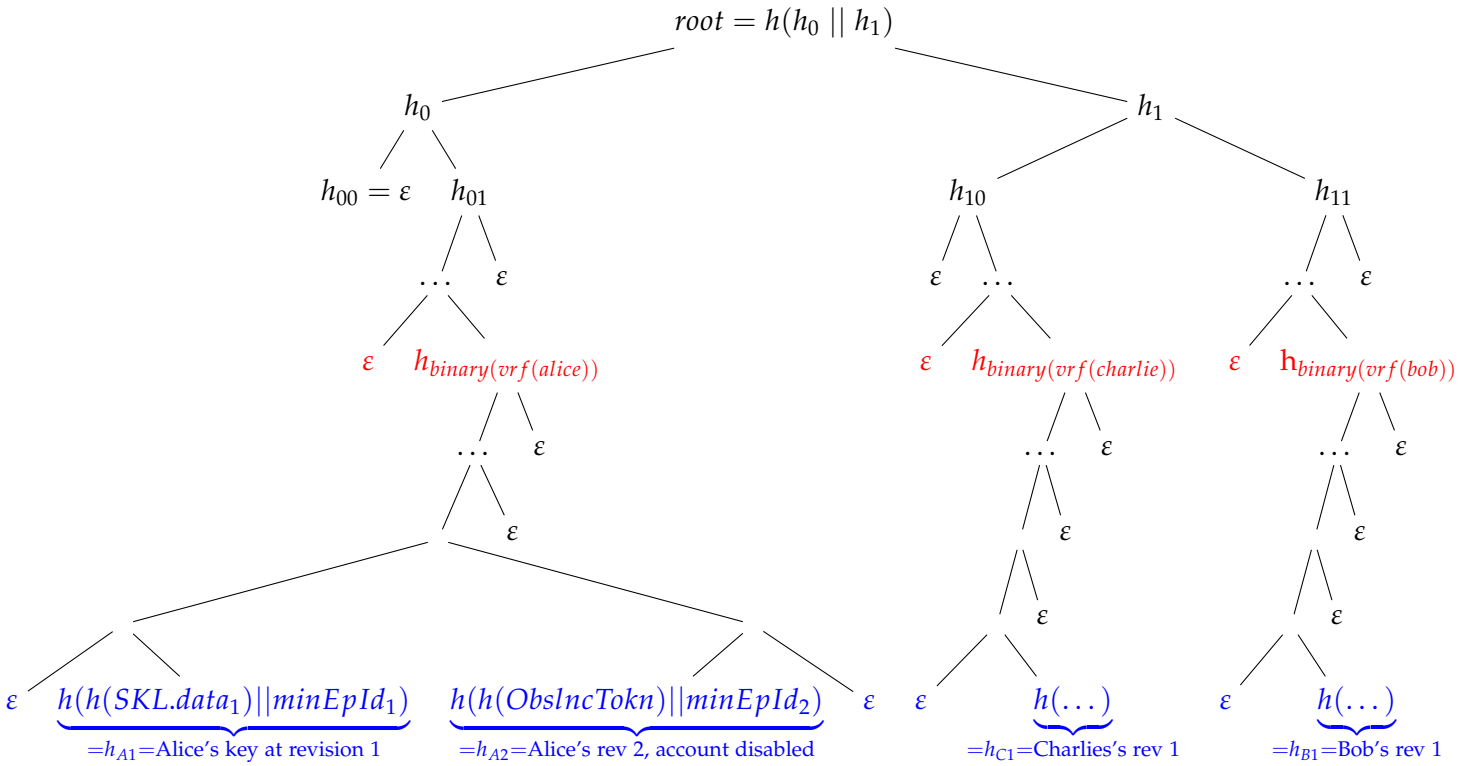

Figure 3.2: Tree structure example. Leaves of the label subtree (at depth 224) are shown in red and leaves of the three revision subtrees (at depth 256) are in blue.

π_{copath} is the inclusion/absence proof. [6] It is a list of all the neighbors on the path from the leaf to the root, excluding the root hash.

Verifying the proof

1. First, we verify the VRF proof for the label:

   $$\beta/\perp = VRF.verify(\mathsf{pk}, \mathsf{label}, \pi_{vrf})$$

   If it passes, we use the VRF hash and the revision to calculate the VRF index:

   $$idx = \beta \,||\, \mathsf{rev}$$

2. Second, from the outcome type τ and the query outcome values val we can construct the leaf hash (as defined above). Inclusion and obsolescence proofs *must* check that the *SKL.data* is JSON-encoded and the *ObsolescenceToken* is hex-encoded.

3. Third, we (re-)compute the hash chain from the leaf to the root using the neighbors in π_{copath}. The bits in the leaf index determine whether a neighbor should be $hash_{left}$ (bit 1, path goes right) or $hash_{right}$ (bit 0, path goes left).

[6] Obsolescence proofs are a variant of inclusion proofs. They prove that an ObsolescenceToken is included in the tree.

4. Finally, we compare the computed *roothash* against the provided one. If they match, the proof verifies.

In practice, we also need sanity checks such as checking that the input values are non-null.

Pruning empty subtrees (absence proofs only) Recall that the tree is sparse, i.e. it will have many empty subtrees. We want to avoid hashing $h(0^n \mid\mid 0^n)$, and $h\left(h\left(0^n \mid\mid 0^n\right) \mid\mid h\left(0^n \mid\mid 0^n\right)\right)$, etc. many times. To prune these empty subtrees, we ignore all the initial empty neighbors on the co-path, and we start hashing only when we hit the first non-null neighbor. (If we have a null/empty neighbor after that, we treat it as 0^n.) Proton calls this *incomplete hashing*.

Note that this is only allowed for absence proofs! For inclusion and obsolescence proofs, we always immediately start hashing at the second last level:

$$h\Big(\underbrace{h(h(SKL.data) \mid\mid minEpochId)}_{\text{leaf hash } h_{left}} \mid\mid h_{right}\Big)$$

In the example in Figure 3.3, the co-path proofs are as follows (neighbors listed from bottom-to-top):

- $\pi_{rev0} = (h_{rev1}, h(h_{rev2} \mid\mid \varepsilon), \varepsilon, \dots)$
- $\pi_{rev1} = (\varepsilon, h(h_{rev2} \mid\mid \varepsilon), \varepsilon, \dots)$
- $\pi_{rev2} = (\varepsilon, h(\varepsilon \mid\mid h_{rev1}), \varepsilon, \dots)$
- $\pi_{rev3} = (h_{rev2}, h(\varepsilon \mid\mid h_{rev1}), \varepsilon, \dots)$
- $\pi_{rev4} = \pi_{rev5} = \pi_{rev6} = \pi_{rev7} = (\varepsilon, \varepsilon, h_{03}, \dots)$

In particular, we set $h_{47} = \varepsilon$ and not $h_{47} = h\left(h\left(0^n \mid\mid 0^n\right) \mid\mid h\left(0^n \mid\mid 0^n\right)\right)$.

Note that because for inclusion and absence proofs we start hashing immediately from the leaf onward (even if the first few neighbors are empty, e.g. in π_{rev2}), we ensure that the nodes on the path from non-empty leaves to the root always have hash values. Not ignoring empty neighbors in this case makes sense, because if the leaf has a value, the subtree induced by the leaf's parent is no longer empty, so we cannot prune it.

With this incomplete hashing/subtree pruning, the hash chain computation for *absence* proofs will often be non-constant time, because the number of hashes is no longer fixed at 256. This opens up an (acceptable) side-channel, where an attacker can try and learn how many hashes a client did, and where in the tree the queried label may be. This can leak which labels a client has looked up.

3.6 Committing to the Tree Root

Recall that the goal of KT is to give clients a consistent view of all keys in the system. To achieve this, a consistent view of the tree is required, which in turn is done by ensuring a consistent view of the tree root. The tree root represents a commitment to the entire tree.

If clients don't check for inconsistent views, equivocation attacks are possible: the server could present one view of the tree to client A and a different view of the tree to client B.

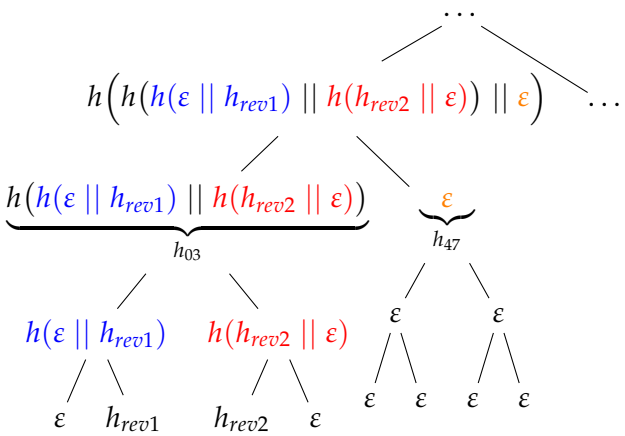

Figure 3.3: Tree structure example (incomplete). Only the lower left part of a single revision subtree is shown. $h_{rev1,2}$ are non-empty leaf hashes of the form $h_{revi} = h(h(SKL.data \mid\mid minEpochId))$

Proton's approach ProtonKT piggybacks on Certificate Transparency (CT). The server publishes the tree root in a web PKI TLS certificate that is logged in CT. This bootstraps ProtonKT on top of CT, with the assumption that there are existing auditors that ensure that CT provides an append-only log.

To publish a new epoch with a new tree root the server requests a new TLS certificate with the Subject Alternative Name for the following DNS domain (called *full KT domain*):

`{chainhash[0:32]}.{chainhash[32:64]}.{issuanceTime}.{epochid}.1.keytransparency.ch.`

`chainhash` is the hex-encoded SHA-256 chainhash (not the root hash – see below).[7] `issuanceTime` is the epoch issuance Unix timestamp claimed by the server.[8] `epochid` is the integer epoch id. `1` is the integer version number of the ProtonKT protocol.

The Certificate Authority will log the issued certificate in a CT log. Later anyone can query the CT log and search for all such KT domains, thus obtaining all logged chain hashes.

Chain Hashes and Root Hashes Notice how we commit the chain hash and not the root hash to the CT logs. The *chainhash* links different epochs together, and for epoch i it is computed as:

$$chainhash_i = h(chainhash_{i-1} \mid\mid roothash_i)$$

For the first epoch (which has $i = 1$) we set:

$$chainhash_0 = 0^n$$

where n is the output size of the hash function being used (SHA-256, i.e. $n = 256$).

[7] The hash is split into two subdomains because RFC 1035 specifies that each label must be "63 octets or less".

[8] Clients cannot verify that this is the correct timestamp when the epoch was issued. But at least it commits the server to a single timestamp that everyone then agrees on.

This chaining commits the entire history of all trees across all epochs as a hash chain to CT. It does *not* guarantee that the values in the tree are append-only. This needs to be verified separately.

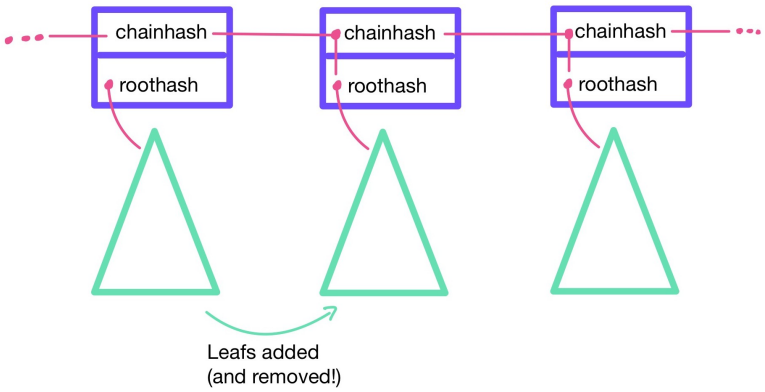

Figure 3.4: Chain Hashes and Root Hashes

Verifying the commitment The certificate contains some *Signed Certificate Timestamps (SCTs)*. An SCT is a signed promise by a CT log that the certificate will be included within a maximum merge delay in the log. Clients verify that the commitment is valid by verifying (1) that the certificate is signed by one of the hardcoded CAs, and (2) that there are at least two SCTs signed by two CT logs, run by different operators, and on a hardcoded list. Once the certificate is verified, the client can extract the full KT domain from the Subject Alternative Name field, and thus learn the epoch ID and the chain hash.

Short KT Domain A short domain format is used for a given epoch id:

`epoch.{epochid}.1.keytransparency.ch`.

This was introduced because the CommonName in the Subject can be at most 64 bytes (as per RFC 5280, page 124), hence a second shorter domain is needed. Both the *full KT domain* and this *short KT domain* are included as Subject Alternative Names in the web-PKI certificate.

It is tempting to use this short domain to search CT logs. However, clients currently do *not* check whether the short KT domain is present and correctly formatted. Hence a malicious server can put any short domain in the CommonName, thus hiding the certificate in the CT logs. CT log searches should always search for `*.{epochid}.1.keytransparency.ch` (note the leading dot between the wildcard and the epoch id!).

DoS Risk There is the risk of CAs or CT logs DoS-ing the KT system by refusing to issue certificates or SCTs. This is mitigated by ProtonKT relying on multiple CAs and CT logs and hardcoding all of them in the client. If one CA fails, the server can request a certificate from another CA.

3.7 Timestamps and Recentness

There are several timestamps associated with every epoch:

- *issuanceTime*: The Unix timestamp at which the new tree has been constructed and the epoch is except for the certificate fully issued. This time is chosen by the server. Proton's API calls this `CertificateTime`.
- *Certificate.notBefore*: Timestamp field in the X.509 certificate. Chosen by the CA.

  Let's Encrypt sets it to the actual certificate issuance time. ZeroSSL, Cloudflare, or Amazon backdate it slightly to 00:00 (hour:min) on the day of the issuance.
- *SCT.timestamp*: The time at which the CT log accepted the (pre-)certificate. Chosen by the CT log.
- *claimedTime*: The time at which the epoch is fully issued, including the certificate. This timestamp is exposed by Proton's REST API for engineering and monitoring purposes. It is not used in the protocol, hence we ignore it in our specification. We mention it here to avoid confusion when reviewing the implementation and API.

It is tempting to think that there exists an ordering like:

$$issuanceTime \leq Certificate.notBefore \leq SCT.timestamp \leq claimedTime$$

However, as mentioned above the *notBefore* is chosen by the CA, and when ZeroSSL sets the hour:minute to 00:00 we have $issuanceTime \geq Certificate.notBefore$.

Similarly, we cannot assume that *notBefore* is ordered across epochs like:

$$notBefore_t \leq noteBefore_{t+1}$$

If epoch t has a certificate from Let's Encrypt and epoch $t+1$ has a certificate from ZeroSSL, then $notBefore_t \geq notBefore_{t+1}$.

Because of that, certificate timestamps can only be used as a rough indication of liveness. For monotonicity, we rely on the epoch ids to be strictly increasing. We require that the server sets the issuanceTimes to be strictly increasing as well:

$$issuanceTime_t < issuanceTime_{t+1}$$

Recentness When querying keys, the server needs to convince the client that an epoch is recent. Otherwise, if Bob is compromised and rotates his keys, the server could keep showing Alice an old epoch and pretend it is still the latest one. Then Alice would continue to use the compromised keys.

A first idea to ensure recentness is to require that an epoch must be accompanied by a web certificate that has a *notBefore* timestamp within the past, say, 24 hours. However, this can be easily attacked: the server can just request a new certificate every 24 hours for the same old KT domain.

A better idea is to also require that the *notBefore* is within 24 hours of the *issuanceTime* (in addition to being within 24 hours of the current time). Because the *issuanceTime* is committed in the domain name and thus cannot be changed without equivocating, the previous attack does not work *if* we assume that CAs always sets the *notBefore* close to the time of the certificate request. This is a reasonable assumption. For example, Let's Encrypt does not even allow requesters to choose a *notBefore*. Generally CAs have some flexibility of setting the *notBefore*, with the caveat that they should not backdate it too far (which is exactly what we require). [9]

[9] https://wiki.mozilla.org/CA/Forbidden_or_Problematic_Practices#Backdating_the_notBefore_Date

Therefore, to verify that an epoch is recent, clients should to the following two checks:

$$|notbefore - isstime_t| \leq 24h$$
$$|currentTime - isstime_t| \leq 24h$$

3.8 Deletions

So far, we have only considered a label–value dictionary that is append-only. However, in practice we also want to delete old, unused entries. For example for legal reasons when a user requests their account deletion, or simply for business reasons as Proton does not want to operate infinitely growing storage.

Recall that in ProtonKT leaf indices are of the form $VRF.hash_{\mathsf{sk}}(\mathsf{label}) \mid\mid \mathsf{rev}$. This comes at the cost of having additional leakage towards the auditors. However, this also enables auditors to monitor correctness of deletion, because they can "see" into the revision subtrees.

The server is allowed to delete values under the following conditions, verified by the External Auditors:

- The latest revision is never deleted. At least one revision must always be retained.
- Only values that are old enough are deleted. Auditors see when a leaf was inserted ($minEpochId$). A leaf of revision t may only be deleted if it has been superseded by revision $t+1$ more than DeletionParam ago. That is, revision $t+1$ was inserted more than DeletionParam ago, and the $maxEpochId$ of t is more than DeletionParam ago.

  In ProtonKT, the system parameter DeletionParam is set to 90 days. This is motivated by the web-PKI certificates for the tree root commitments which expire after 90 days. [10]
- are deleted starting from the lowest revision, and they are deleted continuously without any gaps. E.g. it is not allowed to delete revisions 1 and 3 but not revision 2.

Adding deletion to ProtonKT only changes the epoch generation by the server and adds some checks to the auditors. Client algorithms (in particular Query and Self Audit) do not change.

3.9 Self Audit

Recall that the goal of Key Transparency is to achieve consistency of username-to-public-key bindings. Given this consistency, each user can then locally verify that their binding is correct (i.e. it contains their real keys, and not malicious keys).

Checking correctness done by the *Self Audit* in ProtonKT: a user's client looks up its own label and compares the result against its own keys that it locally knows.[11] It checks that the latest revision in the tree is correct, and that its key history is correct as well. If there are any unexpected keys, it raises a warning.

[10] Renewing the certificates does not help because the *issuanceTime* in the full KT domain must be within 24 hours of the certificate's notBefore timestamp (see section 3.11). The other option is to set the notAfter to a value higher than 90 days, but for that ProtonKT would need a CA other than Let's Encrypt.

[11] How does "locally know" work in the Proton web client, where a user can log in on any new device with an empty state? The Proton server stores the user's private keys in a keyring encrypted under the

Algorithm The client stores a $verifiedRev$, initialized to 0.[12] This is the last, lowest revision that the client has audited. It also stores $verifiedCreationTimestamp$, the creation timestamp of the last audited revision. The Self Audit then does the following:

1. Get the latest epoch and verify its commitment.
2. Request all values in the interval $[verifiedRev + 1, latestRev]$, together with their proofs.

   $latestRev$ is only known to the server, since the client does not know how many values were inserted since the last Self Audit. Note that the interval may be empty when $verifiedRev = latestRev$.
3. Also request an absence proof for the $latestRev + 1$.
4. Check all proofs against the epoch's root hash.
5. If $latestRev = 0 (= verifiedRev)$: Raise a warning and end Self Audit. [13]
6. Check all SKL signatures
7. Check that the timestamps in the SKL signatures of the included values are strictly monotonically increasing, starting from $verifiedCreationTimestamp$.
8. For each non-absent val:
   - If val is obsolete: Raise a warning. [14]
   - If val is included: Check that all keys in SKL.data are locally known. If not, raise a warning.
9. If there are no errors, store $verifiedRev \leftarrow latestRev$
   and $verifiedCreationTimestamp \leftarrow latestRev.SKL.signature.creationTimestamp$.

Not Hiding Newer Revisions Client must be sure that $latestRev$ really is the latest revision in the tree, otherwise the server could hide malicious key updates by not showing higher revisions. Currently this is done by checking the absence of $latestRev + 1$. This works because if the tree is correctly constructed, revisions are never modified, and new revisions are created incrementally and continuously without any gaps. Correct tree construction is checked by the External Audit, see section 3.11.

user's password. When the user logs in, the user can simply decrypt these keys with their password, thus now "locally knowing" their keyring.

We assume that the Proton key server cannot tamper with the encrypted keyring. In particular, it cannot add a malicious key or remove a key because it doesn't know the password; hence the server cannot produce a ciphertext that decrypts successfully with the user's password.

Of course, all of this assumes that the web server honestly delivers the web client and does not modify the client code.

[12] Recall that revision 0 is always absent. The first non-empty revision is 1.

[13] Warn the user that their keys are not yet included in the tree and that they should check back later. (We assume a user always has keys.)

[14] Recall that if an address is obsolete, querying clients fall back to the catch-all address of the domain. Thus emails intended for the disabled user will have been encrypted for the catch-all address. Warn the user that emails for them may have been rerouted.

Rollback Prevention The Self Audit checks that signature timestamps are strictly monotonically increasing. This is to prevent the server from "rolling back" values to previous SKLs. Consider Alice who has the following keys: revision 1 contains an RSA-1024 key. Revision 2 contains both the old RSA-1024 key and a new Curve25519 key. The new ECC key is marked as primary, thus all emails will be encrypted with it. The RSA key is still kept in the keylist of revision 2 to allow Alice to decrypt old emails. If the server re-inserted the SKL from revision 1 again as revision 3 (thus rolling back), then other querying clients would use the old RSA key.

If Alice's Self Audit checks that SKL signatures are temporally increasing, then this attack does not work, because the server cannot produce an SKL signature with the required newer timestamp. [15]

Note that even though the ObsolescenceToken contains a timestamp, we need not it during Self Audit. Unlike the signature timestamp, the ObsolescenceToken's timestamp is chosen by the server so we cannot trust it anyway.

Run Self Audits Regularly Clients are assumed to be online regularly to run Self Audits, at least every 90 days (due to DeletionParam). If they are offline for longer, a server may insert a malicious key, later insert the original key back, and delete the malicious key once it is old enough. Because this increments the revision twice, the Self Audit will notice that something happened thanks to $verifiedRev$, but it won't see the malicious key.

Which labels to audit Users can have multiple email addresses associated with their accounts. Most of them will be active, but some may have been disabled. Clients need to run Self Audit for all active email addresses (= labels). Disabled addresses should not be audited, not even for correct obsoletion. This is because in organization non-personal addresses such as `finance@example.com` might get reassigned from Alice's personal account to Bob's personal account, e.g. when Alice changes jobs from Head of Finance to Head of Legal. Then the server tells Alice that `finance@example.com` was disabled for her, but the tree will now contain Bob's keys, which Alice would not recognize.

For catch-all addresses, the domain owner should run the Self Audit for the label `@example.com` (i.e., just the domain prefixed by an @). If there is no Self Audit, the catch-all address is similarly vulnerable as normal addresses.

3.10 Promise Audit

Sometimes the KT server makes a promise to include a value. With a Promise Audit clients verify that the server fulfills these promises within the *Maximum Merge Delay (MMD)* set to 72 hours.

The server makes promises in two cases: First, when a client requests a new value to be inserted. Second, when a query returns a new, not-yet-included value. This happens because epochs are only issued every four hours on average.

In both cases, the client locally stores the promised values:[16]

$$promises = \{(\mathsf{label}_i, \tau_i, SKL.data_i, ObsolescenceToken_i, expectedRev_i, creationTime_i)\}_i$$

[15] Side note: Recall that SKL signatures are not committed in the leaf hash, only the SKL data. However, the server still stores the signature and has to be able to give it to Alice in order for Alice's Self Audit to pass.

[16] Depending on τ_i, either SKL_data_i or $ObsolescenceToken_i$ must be set, but not both.

expectedRev is used to later query this specific revision and check that it matches *SKL.data*. The client calculates the duration between *creationTime* and the then-current time and checks that it is less than the MMD. If the server does not include the SKL within the MMD, this is considered misbehavior and a warning is raised. If a different value is included at *expectedRev*, this is also misbehavior.

3.11 External Audit

So far we have seen how the tree *should* be constructed. However, the KT server is untrusted and may not adhere to the specification. Therefore, we need at least one honest auditor that checks that the server behaves correctly. If this external auditor finds an error, this should result in non-repudiable proof. The auditor can use this proof to convince others that the server behaved maliciously.

These *External Audits* complement the Self Audits that clients run. Clients are assumed to be thin end-user devices like phones and laptops. They are energy constrained and are offline for longer periods of time (multiple weeks, up to 90 days). External Auditors on the other hand are assumed to be more powerful, have unconstrained storage, and are online regularly (always online, or at least every few hours). External individuals or organizations can easily deploy the Auditor code on their servers and have it run in the background. As of November 2023, an audit of the tree containing roughly 50 million leaves takes about 15 minutes on a 4-core CPU and takes up to 8 GB of disk space. The download of a single epoch is roughly 4 GB, subsequent epochs can be delta updates.

We move certain checks into the External Auditor, which allows us to reduce the work that clients have to do. For example, clients only have to Self Audit the latest epoch because the external auditors guarantee that the tree is append-only. Without this guarantee, clients would need to look up their own keys with respect to every single epoch to do a Self Audit.

Properties to check Auditors verify the following properties:

1. *Epoch IDs:* are incrementally increasing and continuous (no gaps, no missing epoch).

2. *Tree commitments:* Each *chainhash* claimed by the server appears in at least one CT log. The auditor must check the log, i.e. not simply trust the SCT.

3. *Non-equivocation:* There is exactly one $(epochid, chainhash, issuanceTime)$ tuple logged for each epoch with the expected full KT domain. There may be multiple CT log entries (e.g. pre-certificate and certificate), or even multiple different certificates, as long as they all contain the same full KT domain.

4. *IssuanceTime consistency:* The *issuanceTime*s are strictly monotonically increasing from one epoch to the next.

5. *IssuanceTime-to-Certificate consistency:* For all certificates logged for an epoch the following holds:

$$|IssuanceTime - certificate.NotBefore| \leq 24h$$

6. *Insertions:* new values are inserted with revision 1.

7. *Updates:* updates to existing values increment the revision by 1 (and don't skip revisions). Old revisions are never overwritten (only deleted, see below).

8. *Update consistency:* Consider the trees at epoch t and $t+1$, committed as $roothash_t$, $roothash_{t+1}$. Check that $roothash_{t+1}$ is reachable from $roothash_t$ by only inserting new values and deleting old enough values.

   A naive approach is for the auditor to simply re-construct the tree from scratch, and compare the computed root hash against the claimed one. Starting from an empty tree at epoch 0, the KT server simply gives the external auditor all the elements that were inserted and all the elements that were deleted. All other values are not modified.

9. *Deletions:* Only $(\mathsf{val}, \mathsf{rev})$ entries are deleted have been superseded by a newer revision, and this newer revision was inserted at an epoch that was issued more than 90 days ago (according to the epoch's *issuanceTime*).

   Values are deleted incrementally and continuously, starting from revision 1 to high revisions.

The ordering of checks is loosely relevant. For example, the auditor should check for equivocation before it recomputes the root hash (to save computation if an equivocation is found). It should also check that the *issuanceTime* is unique and consistent with the certificates before using it to verify correctness of deletions.

Practical implementations In practice, an auditor implementation may want to stream epoch information from the KT server as new epochs are issued, and assemble its own local copy of the KT state (i.e. it ingests every epoch state once). Then it also listens on all CT logs for new certificates for *all* epochs. As new certificates come in (even for old epochs), the auditor verifies that the certificate is consistent with its local state. This is important because CAs may backdate the notBefore, i.e. issue a certificate for an old epoch (even though Mozilla finds significant backdating problematic [17]).

3.12 ProtonKT Subprotocols

In this section we describe the subprotocols that make up ProtonKT.

Definition 3.1 (ProtonKT)**.** *The ProtonKT protocol consists of the following subprotocols:* ProtonKT.RequestInsertion, ProtonKT.Publish, ProtonKT.QueryEpoch, ProtonKT.QueryValue, ProtonKT.SelfAudit, ProtonKT.PromiseAudit, *and* ProtonKT.ExtAudit.

Below, we define these subprotocols in more detail.

- $0/1 \leftarrow \mathsf{ProtonKT.RequestInsertion}(\mathsf{label}, SKL.data)$

  Requests the insertion of a new value for label.

  If successful, the client stores $(\mathsf{label}, SKL.data, expectedRev, currentTime)$ and checks that it is included within the Maximum Merge Delay (MMD) of 72 hours.

  The server collects all the requests in its local state in a set of pending updates S.

[17] https://wiki.mozilla.org/CA/Forbidden_or_Problematic_Practices#Backdating_the_notBefore_Date

- $\perp / (\mathsf{Dir}_t, roothash_t, chainhash_t, cert_t) \leftarrow \mathsf{ProtonKT.Publish}(\mathsf{Dir}_{t-1}, \{\mathsf{label}_i, \mathsf{val}_i\}_i)$

  Publishes a new epoch. Inserts all the $(\mathsf{label}, \mathsf{val})$ pairs into the key directory Dir (i.e. the tree) as new revisions and requests $cert_t$ as a commitment to the updated tree.

  Returns $\perp$ if the epoch could not be generated (e.g. the CA refused to issue $cert_t$).

- $\perp / (roothash_t, issuanceTime_t, chainhash_t) \leftarrow \mathsf{ProtonKT.QueryEpoch}(t, chainhash'_{t-1})$

  Gets for epoch t the tree $roothash_t$ and verifies it against the commitment.

  Internally, it requests all of $(roothash_t, chainhash_t, cert_t, chainhash_{t-1})$. The web-PKI certificate $cert_t$ serves as the commitment. $cert_t$ should contain two or more SCTs. This algorithm trusts the SCTs as a promise of CT log inclusion; it does *not* scan CT logs. $chainhash_{t-1}$ is included for convenience (it is needed to compute $chainhash_t$). If $chainhash'_{t-1}$ is non-null, checks it against the server-claimed $chainhash_{t-1}$.

  Returns $\perp$ if the epoch id t does not exist.

- $(\tau, \mathsf{rev}, \mathsf{val}) \leftarrow \mathsf{ProtonKT.QueryValue}(roothash_t, \mathsf{label})$

  Gets the latest value of a given label as well as its revision and type. Also verifies that the value is consistent with the key directory state at epoch t.

  τ is the type of proof (absence, inclusion, obsolescence); in the REST API it is called `Type`. val (called `SKL` in the API) contains other necessary fields (depending on τ) such as: $SKL.data$, $ObsolescenceToken$, $minEpochId$ (see section 3.3).

  If $\tau = abs$ then rev and val are null. Since we are querying the latest value, absence is only allowed if $\mathsf{rev} = 0$.

  If $\tau = incl$ or $\tau = obs$ and $minEpochId$ is null, then the value is not yet included in the tree (but the server promises to include it). In this case, the client can skip getting and verifying the proof. It should store $(\tau, \mathsf{rev}, \mathsf{val})$ and verify it later using ProtonKT.PromiseAudit.

  Internally, this function gets the VRF proof π_{vrf} and the Merkle tree proof π_{copath} and verifies them against $roothash_t$.

  Returns $\perp$ if the epoch id t does not exist.

- $0/1 \leftarrow \mathsf{ProtonKT.SelfAudit}(roothash_t, \mathsf{label}, keylist, verifiedRev, verifiedCreationTimestamp)$

  Checks the key history of the user for correctness. Requests all values for label that are equal or larger than $verifiedRev$. Then checks that there are no unexpected values. See section 3.9 for full details.

- $0/1 \leftarrow \mathsf{ProtonKT.PromiseAudit}(roothash_t, promises)$

  Checks that the promises that the server made to include a value were fulfilled.

- $0/1 \leftarrow \mathsf{ProtonKT.ExtAudit}(r, s, \mathsf{Dir}_{r-1}, chainhash_{r-1}, issTime_{r-1})$

  Runs an External Audit.

Verifies that the server adhered to the protocol and evolved the key directory correctly from epoch r until epoch s. Needs $\mathsf{Dir}_{r-1}, chainhash_{r-1}, issTime_{r-1}$ as a trusted basis. See also section 3.11.

3.12.1 Message Sequence Diagrams

In this section we state the subprotocols in detail. We use *Alice & Bob notation*, also known as *security protocol notation*. These are simply message sequence diagrams showing how the different protocol roles exchange messages and which steps and checks they execute locally. For each role, the diagrams also show the knowledge that a role must have before the protocol can execute. For brevity, errors are generally not shown. The protocol is expected to abort upon errors, e.g. if it does not receive a message or if a check fails.

ProtonKT has the following roles: Server, Client, Auditor, CA, CT Log. We write CT Log_i to indicate that multiple instances of the CT Log role are involved in a protocol. For simplicity, we only show the roles and the knowledge that are relevant for each algorithm.

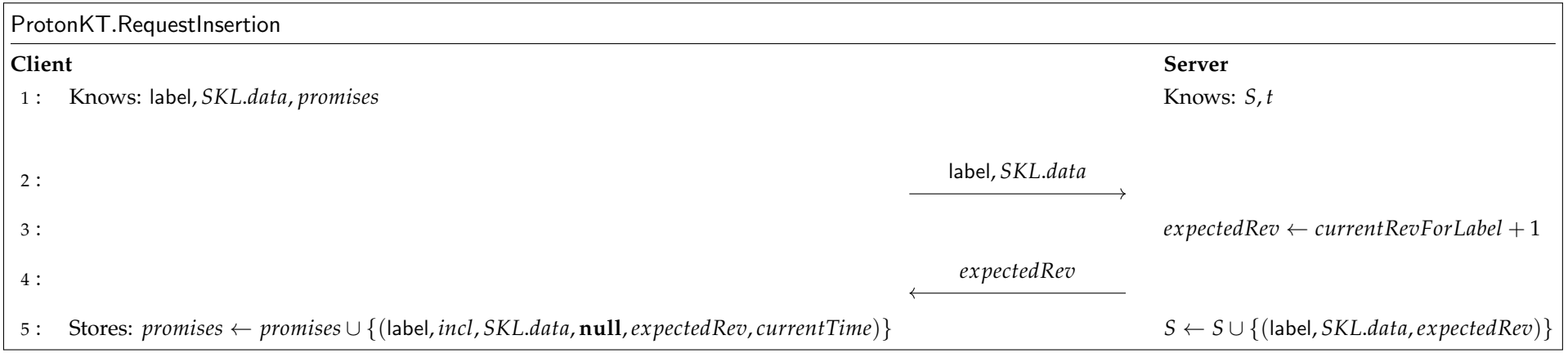

Figure 3.5: ProtonKT.RequestInsertion Sequence Diagram

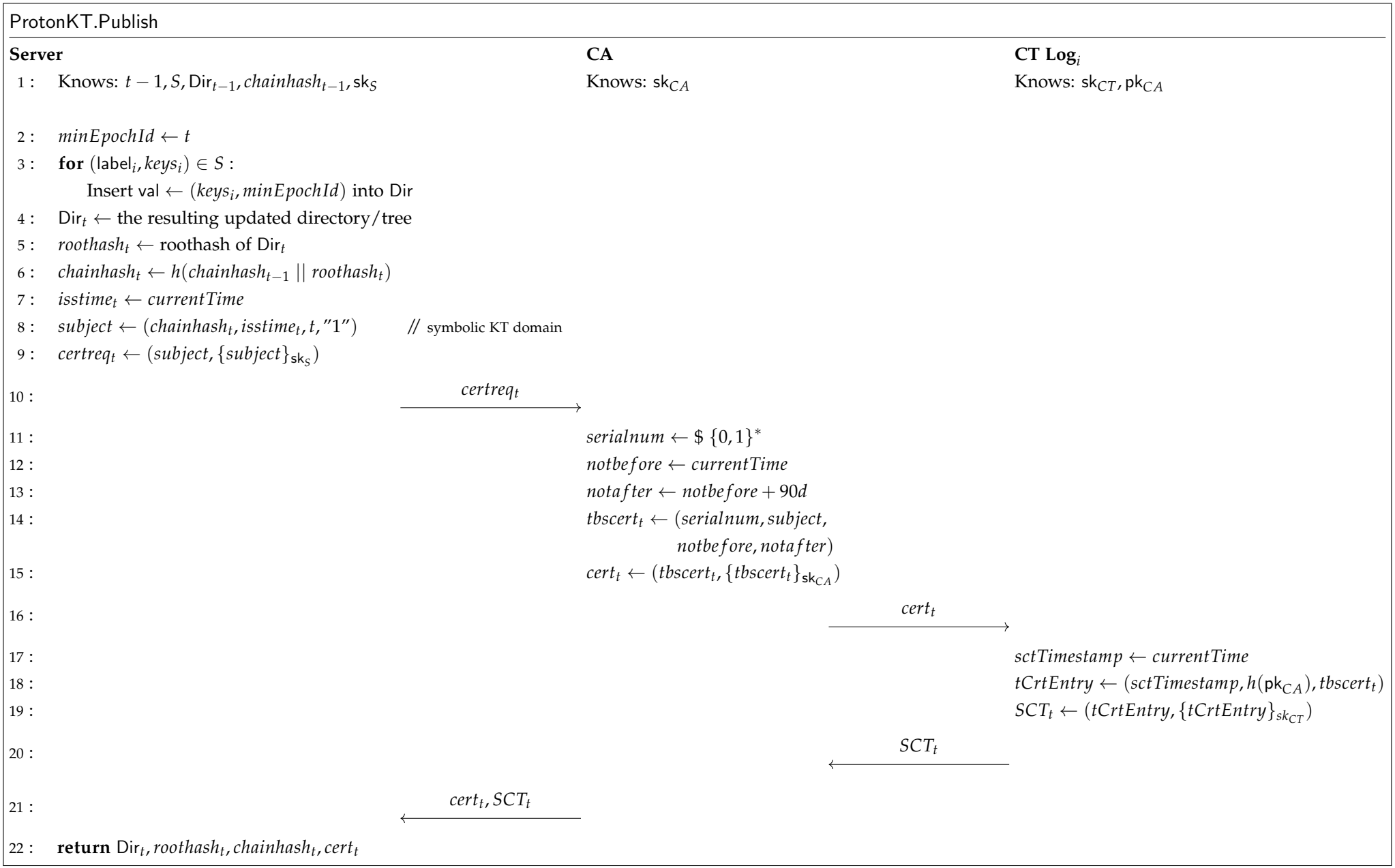

Figure 3.6: ProtonKT.Publish Sequence Diagram

ProtonKT.QueryEpoch

Client		**Server**
Knows: $t, \mathsf{pk}_{CA}, \mathsf{pk}_{CT}$, optionally: $chainhash'_{t-1}$		Knows: $cert_t, SCT_t, roothash_t, chainhash_t, chainhash_{t-1}$
	$\xrightarrow{t}$	
	$\xleftarrow{\overbrace{(tbscert_t, tbscertsig_t)}^{cert_t}, \overbrace{(tCrtEntry, tCrtEntrysig)}^{SCT_t}}$	
	$\xleftarrow{roothash_t, chainhash_t, chainhash_{t-1}}$	
Checks: $\{tCrtEntrysig\}_{\mathsf{pk}_{CT}} = tCrtEntry$	// Check SCT	
Checks: $snd(tCrtEntry) = h(\mathsf{pk}_{CA})$		
Checks: $trd(tCrtEntry) = tbscert_t$		
Repeat for SCTs from at least two different log operators		
Checks: $\{tbscertsig_t\}_{\mathsf{pk}_{CA}} = tbscert_t$	// Check certificate	
$subject \leftarrow snd(tbscert_t)$	// Extract fields	
$notbefore \leftarrow trd(tbscert_t)$		
$chainhash_t \leftarrow fst(subject)$		
$isstime_t \leftarrow snd(subject)$		
Checks: $subject = (chainhash_t, isstime_t, t, "1")$	// Check subject	
Checks: $chainhash'_{t-1} = chainhash_{t-1}$	// If $chainhash'_{t-1}$ is known (see External Audit)	
Checks: $chainhash_t = h(chainhash_{t-1} \mid\mid roothash_t)$	// Check epoch chaining	
Checks: $\lvert notbefore - isstime_t \rvert \leq 24h$	// Check claimed epoch issuanceTime aligns with CA's notBefore	
Checks: $\lvert currentTime - isstime_t \rvert \leq 24h$	// Check epoch is recent (may be skipped by External Audit querying an old epoch)	
return $roothash_t, isstime_t, chainhash_t$		

Figure 3.7: ProtonKT.QueryEpoch Sequence Diagram

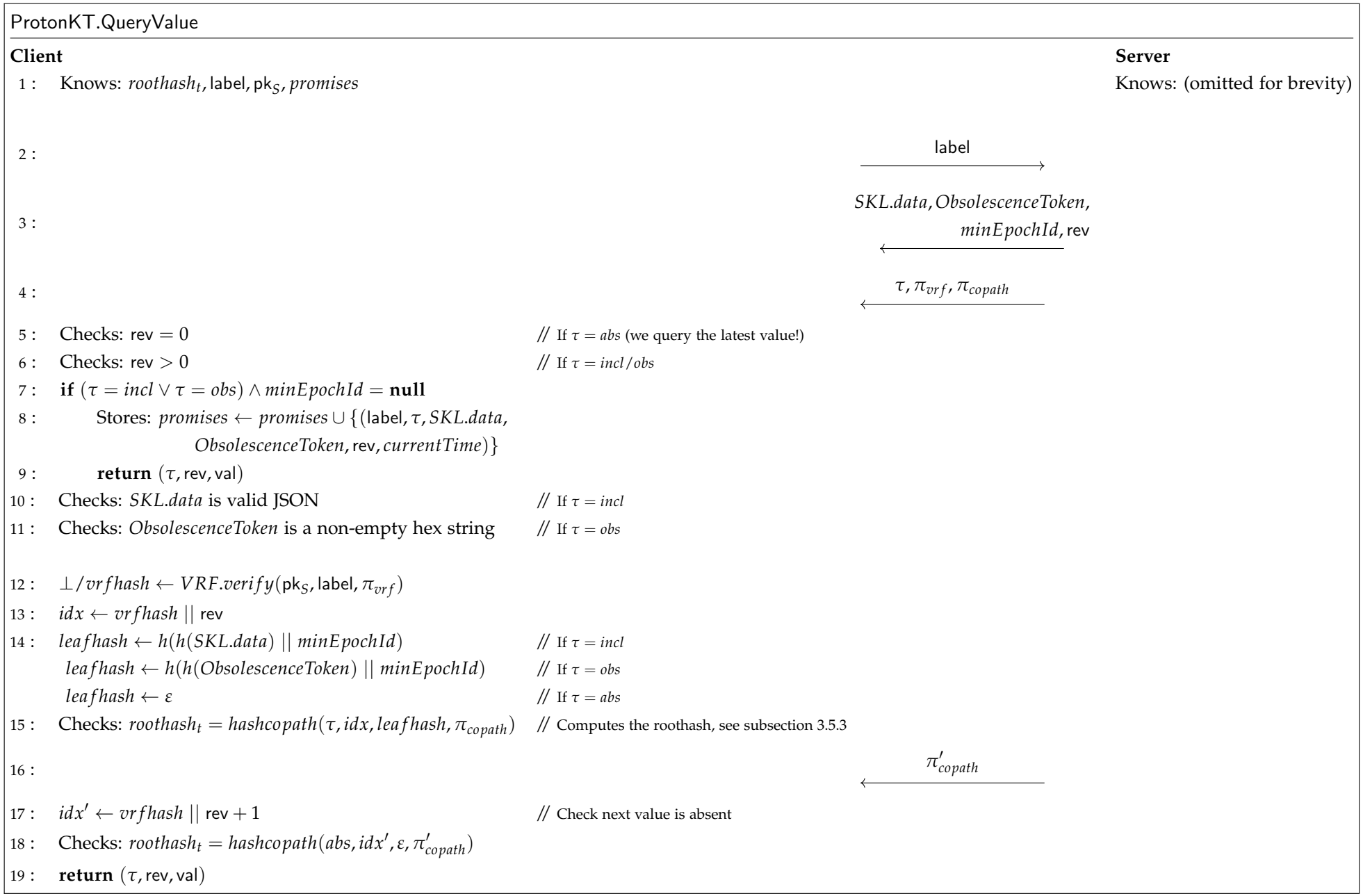

Figure 3.8: ProtonKT.QueryValue Sequence Diagram

ProtonKT.SelfAudit

Client		**Server**
Knows: $roothash_t$, label, $keylist$, $verifiedRev$, $verifiedCreationTimestamp$		Knows: (omitted)
	label, $verifiedRev$ $\longrightarrow$	
	$\{SKL.data_i, SKL.sig_i, ObsolescenceToken_i, minEpochId_i\}_{i=verifiedRev+1}^{latestRev}$ $\longleftarrow$	
	$\pi_{vrf}, \{\tau_i, \pi_{copath,i}\}_{i=verifiedRev+1}^{latestRev+1}$ $\longleftarrow$	
$\perp / vrfhash \leftarrow VRF.verify(\mathsf{pk}_S, \mathsf{label}, \pi_{vrf})$		
for $i \in [verifiedRev+1, latestRev]$:	// Check incl/obs proofs (note: interval may be empty)	
Checks: $SKL.data_i$ is valid JSON	// If $\tau_i = incl$	
Checks: $ObsolescenceToken_i$ is a non-empty hex string	// If $\tau_i = obs$	
$idx \leftarrow vrfhash \mid\mid i$		
$leafhash \leftarrow h(h(SKL.data_i) \mid\mid minEpochId_i)$	// If $\tau_i = incl$	
$leafhash \leftarrow h(h(ObsolescenceToken_i) \mid\mid minEpochId_i)$	// If $\tau_i = obs$	
$leafhash \leftarrow \varepsilon$	// If $\tau_i = abs$	
Checks: $roothash_t = hashcopath(\tau_i, idx, leafhash, \pi_{copath,i})$		
$idx \leftarrow vrfhash \mid\mid latestRev + 1$	// Check absence proof	
Checks: $roothash_t = hashcopath(abs, idx, \varepsilon, \pi_{copath,latestRev+1})$		
if $latestRev = 0$: Raise warning and **return**	// Keys not yet included	
$tempCreationTime \leftarrow verifiedCreationTimestamp$		
for $i \in [verifiedRev+1, latestRev]$:		
if $\tau_i = incl$:		
Checks: $SKL.sig_i$ verifies with a key in $keylist$	// Check signatures	
Checks: $tempCreationTime < SKL.sig_i.creationTimestamp$	// Check strict monotonicity	
$tempCreationTime \leftarrow SKL.sig_i.creationTimestamp$		
for $i \in [verifiedRev+1, latestRev]$:	// Check new revisions for correctness	
if $\tau_i = abs$: Raise warning	// Unexpected gap, either legal deletion or misconstructed tree	
if $\tau_i = obs$: Raise warning	// Emails may have gone to catchall while label was obsolete	
if $\tau_i = incl$ and any of the keys in $SKL.data_i$ not in $keylist$: Raise warning	// Unexpected key	
Stores: $verifiedRev \leftarrow latestRev$		
Stores: $verifiedCreationTimestamp \leftarrow tempCreationTime$		

Figure 3.9: ProtonKT.SelfAudit Sequence Diagram

ProtonKT.PromiseAudit

	Client		**Server**
1 :	Knows: $roothash_t, promises$		Knows: (omitted for brevity)
2 :	**for** $(\mathsf{label}, \tau, SKL.data, ObsolescenceToken, expectedRev, creationTime) \in promises$		
3 :		$\mathsf{label}, expectedRev \longrightarrow$	
4 :		$\longleftarrow minEpochId$	
5 :		$\longleftarrow \tau', \pi_{vrf}, \pi_{copath}$	
6 :	**if** $\tau' = abs$:		
7 :	**if** $\lvert currentTime - creationTime \rvert > \mathsf{MMD}$:	// Check inclusion within MMD	
8 :	Raise warning		
9 :	**else** :		
10 :	**continue**	// Try again in a later Promise Audit	
11 :	Checks: $SKL.data$ is valid JSON	// If $\tau = incl$	
12 :	Checks: $ObsolescenceToken$ is a non-empty hex string	// If $\tau = obs$	
13 :	$\perp / vrfhash \leftarrow VRF.verify(\mathsf{pk}_S, \mathsf{label}, \pi_{vrf})$		
14 :	$idx \leftarrow vrfhash \mid\mid expectedRev$		
15 :	$leafhash \leftarrow h(h(SKL.data) \mid\mid minEpochId)$	// If $\tau = incl$	
	$leafhash \leftarrow h(h(ObsolescenceToken) \mid\mid minEpochId)$	// If $\tau = obs$	
16 :	Checks: $roothash_t = hashcopath(\tau, idx, leafhash, \pi_{copath})$	// If proof fails, raise warning for this item	
17 :	Stores: $promises \leftarrow promises \backslash \{$all successful item audits$\}$		

Figure 3.10: ProtonKT.PromiseAudit Sequence Diagram

ProtonKT.ExtAudit

Auditor		Server	CT Log_i
Knows: $r, s, \mathsf{Dir}_{r-1}, chainhash_{r-1}, issTime_{r-1}$		Knows: (omitted)	Knows: $\{\{cert_i\}_{i,t}\}_t$
Precondition: Dir_{r-1} correctly constructed			
for $t \in [r, s]$:			
$roothash_t, issTime_t, chainhash_t \leftarrow$ ProtonKT.QueryEpoch$(t, chainhash_{t-1})$	$\leftarrow\text{-}\text{-}\text{-}\rightarrow$		
Checks: $issTime_{t-1} < issTime_t$			
	$*.\{\mathtt{t}\}.\mathtt{1.keytransparency.ch}$ $\longrightarrow$		
	$\longleftarrow$ $\{cert_i\}_{i,t}$		
Checks: only certs with the expected full KT domain and no others			
	t $\longrightarrow$		
	$\longleftarrow$ $\underbrace{\{(\overbrace{vrfhash_i \mid\mid \mathsf{rev}_i}^{idx_i}, valhash_i, minEpochId_i)\}_i}_{insertions}$		
for $ele_i \in insertions$:			
Checks: $minEpochId_i = t$			
Checks: $\mathsf{rev}_i = 0$			
Checks: The entire revision subtree for $vrfhash_i$ was empty in Dir_{t-1}			
	$\longleftarrow$ $\underbrace{\{(vrfhash_i \mid\mid \mathsf{rev}_i, valhash_i, minEpochId_i)\}_i}_{updates}$		
for $ele_i \in updates$:			
Checks: $minEpochId_i = t$			
Checks: $vrfhash_i \mid\mid \mathsf{rev}_i$ was empty in Dir_{t-1}			
Checks: $vrfhash_i \mid\mid \mathsf{rev}_i - 1$ was non-empty in Dir_{t-1}			

Figure 3.11: ProtonKT.ExtAudit Sequence Diagram (Part 1)

ProtonKT.ExtAudit (continued)

Auditor **Server** **CT Log**$_i$

20 : $\underbrace{\{vrfhash_i \,||\, \mathsf{rev}_i\}_i}_{deletions}$ $\longleftarrow$

21 : **for** $ele_i \in deletions$:

22 : Checks: $vrfhash_i \,||\, \mathsf{rev}_i - 1$ was empty in Dir_{t-1} or $\mathsf{rev}_i = 0$

23 : Checks: $vrfhash_i \,||\, \mathsf{rev}_i$ was non-empty in Dir_{t-1}

24 : Checks: $vrfhash_i \,||\, \mathsf{rev}_i + 1$ was non-empty in Dir_{t-1} // $\mathsf{rev}_i + 1$ cannot have been upserted in t

25 : $minEpochId' \leftarrow minEpochId$ of leaf $vrfhash_i \,||\, \mathsf{rev}_i + 1$

26 : Checks: $|minEpochId'.issTime - t.issTime| > \mathsf{DeletionParam}$

27 : $\mathsf{Dir}_t \leftarrow$ apply $insertions, updates, deletions$ to Dir_{t-1}

28 : $roothash_{comp} \leftarrow$ roothash of Dir_t

29 : Checks: $roothash_t = roothash_{comp}$

30 : Postcondition: Dir_s correctly constructed

Figure 3.12: ProtonKT.ExtAudit Sequence Diagram (Part 2)

4 Analysis

In this section, we analyze ProtonKT. We first look at the privacy goals. Next, we define the security properties that we want ProtonKT to achieve. We also describe the adversary model under which these properties should hold. After that we do a security analysis, and we argue why the ProtonKT scheme achieves the stated properties.

4.1 Privacy Analysis

In this section we describe the privacy goals that ProtonKT has. To limit the scope we do not give a detailed analysis for the privacy properties. We do, however, give a rough sketch of the design decisions that are intended to support the privacy goals.

The goal of privacy is to leak as little information about the contents of the key directory as possible. The main threat actors are the External Auditors, because they have access to the entire tree (which they need to recompute the tree to check its properties).

ProtonKT accepts that auditors can learn the following from the tree:

- An upper bound on the number of accounts in the tree. It is obvious from the tree whether a revision subtree is empty or not. However, auditors do not learn how many of these accounts are active and how many are disabled.
- How many revisions any given user has. This is obvious from the revision subtree leaves.
- In which epoch any given revision was inserted. This is necessary for auditors to check that only old-enough values are deleted.

However, ProtonKT's goal is to prevent auditors from learning the following:

- Which email addresses are in the tree. The VRF used to compute leaf indices enforces that one must actively query the server to learn an email address' index.

  This allows the server to prevent *enumeration attacks*, e.g., via rate-limiting. Similarly, Proton could hide the email addresses of business customers by not responding to queries from users outside the business' organization.
- Whether a revision that is present in the tree is active or disabled. Recall that both the $SKL.data$ and the $ObsolescenceToken$ contain randomness (key fingerprints are hashes,[1] and the random part of the $ObsolescenceToken$). The auditor would need to compute the preimage of $h(SKL.data)$ and $h(ObsolescenceToken)$ to distinguish them, which is infeasible.

[1] We work in the random oracle model.

Another threat actor are clients. Their queries and Self Audits necessarily reveal the co-path to them. By repeatedly querying a label, the difference between the co-paths leaks which neighboring parts of the tree have changed. This allows for the tracing attack described by SEEMless [2].

Privacy is also a core design goal of CONIKS, SEEMless, and Parakeet. For example, Parakeet defines leakage functions to specify how much information the protocol leaks. We refer to these works for an in-depth analysis of privacy in KT protocols.

4.2 Security Properties

We first define the security property that the ProtonKT protocol should provide: Query-to-SelfAudit Consistency. After that we will analyze how ProtonKT achieves it.

Informally we want that everything that can be queried must also be seen by a Self Audit. The queries and the Self Audit should agree on the $(\mathsf{label}, \tau, \mathsf{rev}, \mathsf{val})$ tuples. Unless there is already some detection of server misbehavior. Slightly more formally this means:

Definition 4.1 (Query-to-SelfAudit Consistency)**.** *We say that ProtonKT provides* Query-to-SelfAudit Consistency*, if*

- *whenever there was a successful External Audit of epoch t*
- *and client A runs a successful Self Audit SA for its* label *at epoch $s \leq t$ and SA passes with $latestRev \geq$* rev*,*
- *and prior to epoch t A has run a successful Self Audit at least once every* DeletionParam *(e.g. every 90 days),*
- *and a query Q for* label *in epoch $r \leq t$ returned outcome $O = (\tau, \mathsf{rev}, \mathsf{val})$,*
- *and – if Q returned O as a promise P – there was a successful Promise Audit that sees P at an epoch p with $r < p \leq t$,*
- *then client A agrees that $(\tau, \mathsf{rev}, \mathsf{val})$ is the expected outcome for* rev*.*

Note that there is no relationship between r, s, and p, other than all being bounded by t. Figure 4.1 visualizes the components that are mentioned in the property, as well as highlighting the time points at which they can occur.

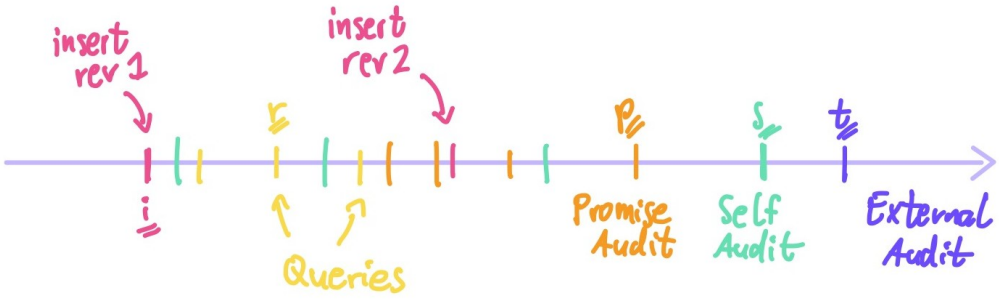

Figure 4.1: Visualization for Query-to-SelfAudit Consistency
(showing also other possible locations of queries, Self Audits, and Promise Audits)

Query-to-SelfAudit Consistency relates Self Audits to queries to make sure they are consistent. Importantly, it also relates queries to other queries (in the presence of a Self Audit!): if Query-to-SelfAudit consistency holds, then two clients B, C who query label A agree with client A's Self Audit; therefore B and C also agree with each other.

Deletions DeletionParam is the minimum age a value need to have to be eligible for deletion if it has been superseded (90 days in ProtonKT). We can only have security for queries that are not too far before any Self Audit: Consider a client who is offline for a long time (e.g. a user traveling for one year on a sabbatical and not checking their business email). Also consider that some values are inserted and later deleted during that time, and the client runs a Self Audit only after they were deleted. Then the deleted value may have been queried by other users, but the traveler's Self Audit does not see them.

Need for Audits This security property only holds in the presence of a successful External Audit. If there was no such audit, then the server could have behaved arbitrarily, e.g. it could have equivocated, or modified values in the tree, or illegally deleted values.

In addition, if the query returned a promise, we also require that this promise is verified. Otherwise, the server could trivially reply with a promise for any value.

Furthermore, client A who owns the label must do regular Self Audits. For example, consider that A runs Self Audit $SA1$, then is offline for 120 days ($>$ DeletionParam), and then runs Self Audit $SA2$. Then there may be revisions that were inserted in the tree just after the $SA1$ and deleted before $SA2$. $SA2$ would only see the unexpected jump in the revision, but not the values. However, these values could have been successfully queried in the meantime.

Comparison Query-to-SelfAudit Consistency is roughly comparable to SEEMless' VKD soundness definition [2, Appendix B], with the difference that we also need to account for the DeletionParam and the Promise Audits.

Compared with the cVKD soundness definition given by Parakeet [3, Definition 4, Appendix B], we have added the Promise Audits but do not have tombstones.

4.3 Adversary Model

In this section we discuss what kind of adversary can try and attack ProtonKT. We define what power the adversary is allowed to have, and what we assume it cannot do.

We give the adversary the following power:

- The adversary controls the network (Dolev-Yao style adversary). It can reorder, replay, drop, insert, and modify messages.
- The adversary can corrupt the KT server. The server is not trusted and can deviate from the protocol. In particular, it can insert, modify, delete leaves in the Merkle tree.

However, we assume that the cryptographic primitives hold. In particular, we assume that SHA-256 is collision resistant and preimage resistant. We also assume

that the VRF satisfies uniqueness (even under malicious key generation).[2] We also assume that at most one CT log operator is malicious and that at least one CA and at least one CT log are online and handling requests. Finally, we assume that at least one honest auditor has a consistent view of the global CT state.

4.4 Security Analysis

In this section we provide an analysis of the security of ProtonKT. Our analysis will proceed as follows: we begin by assuming that the security property is broken. Next, we consider possible broken states. For each of them we will reason about how this state was reached. Finally, if all goes well, we will see that the starting point from which we could reach this broken state cannot happen. In other words, we reason backwards. We start from an attack state, look at how it could have been reached, and then end in a contradiction.

By nature, such an analysis can only consider the high-level protocol ideas. There may still be subtle bugs, or cases we forgot to consider.

4.4.1 Classic Attacks are Detectable by External Audits

There are two classical threats to transparency protocols: equivocation and non-append-only-ness of the tree (up to allowed deletions). These affect all KT protocols, and also ProtonKT. Both of them can break Query-to-SelfAudit consistency.

In *(root hash) equivocation*, also known as split-world-view attack, the server forks the tree into two different histories. This allows it to present one view of the tree to some clients, and a different view to other clients, thus breaking Query-to-SelfAudit Consistency. However, this split-view has to be maintained indefinitely.

In *non-append-only-ness*, the server deletes elements from the tree that it shouldn't. For example, it inserts a fake entry, waits for the victim to query the malicious entry, and then deletes the entry again for the owner runs a Self Audit. This breaks Query-to-SelfAudit consistency.

Detection not Prevention ProtonKT does *not* aim to prevent a malicious server from executing these attacks. Instead, ProtonKT aims to detect them. This is again the idea of transparency: if we cannot prevent, then at least detect. This is why our security properties require that an External Audit has passed.

Next, we argue that these classic transparency attacks are detected by External Audits. In other words, External Audits only pass if there is no such attack.

Root Hash Equivocation

Assume that the server has equivocated at epoch t, i.e. it was possible for the server to reply to two epoch queries Q, U for the same epoch t with different roothashes $roothash_t^Q \neq roothash_t^U$, such that both queries accept their respective root hashes, and – importantly – such that this equivocation cannot be detected by external auditors. [3]

[2] For ECVRF this should hold if the curve and group generator are from a trusted source [5, Section 7]. Since ProtonKT hardcodes the parameters of ECVRF-EDWARDS25519-SHA512-TAI, the parameters can easily be checked.

[3] By *epoch query* we mean a run of ProtonKT.QueryEpoch. Recall that the other algorithms such as ProtonKT.QueryValue, ProtonKT.SelfAudit, ProtonKT.PromiseAudit, ProtonKT.ExtAudit all need to be preceded by a run of ProtonKT.QueryEpoch to get the root hash.

Since Q accepted $roothash_t^Q$, there must exist a $chainhash_t^Q$ and a $chainhash_{t-1}^Q$ such that $chainhash_t^Q = h(chainhash_{t-1}^Q || roothash_t^Q)$. There must also exist a $cert_t^Q$ that contains $chainhash_t^Q$ and two or more $SCT_{t,i}^Q$. The same reasoning also applies to U, $chainhash_t^U$, $chainhash_{t-1}^U$, $cert_t^U$.

Case 1 ($chainhash_t^Q \neq chainhash_t^U$). If auditors cannot detect the equivocation, this means that they don't see either $cert_t^Q$ or $cert_t^U$, or neither of them, in any CT log.
This can happen if the auditor has an inconsistent view of the global CT logs. However, we assumed that this does not happen.
This can also happen when at least two different CT logs by different operators (since each certificate contains two or more SCTs from different log operators,[4] and ProtonKT clients check this) either did not include the certificate (thus breaking their promise), or when both of them are down and not responding to queries. In any case, this is a contradiction to our assumption that at least one CT log operator is a trusted third party and that the auditor can see the global CT state.

Case 2 ($chainhash_t^Q = chainhash_t^U$). In this case auditors cannot detect equivocation because the server can set $cert_t^Q = cert_t^U$ and only log a single certificate to CT. But since $roothash_t^Q \neq roothash_t^U$ this means that the server has found a hash collision, which contradicts our assumptions. In other words the server has found two different $(rh||ch) \neq (rh'||ch)$ such that $h(rh||ch) = h(rh'||ch)$.

In both cases we have a contradiction. Thus equivocation is detectable by External Auditors.

Non-append-only-ness

Assume that the server has violated append-only-ness-with-deletion. That is, either (case 1) it has overwritten an existing non-absent value with a different non-absent value, or (case 2) it has deleted a value that it should not have (e.g. the value was the latest revision, or the next-higher revision was not yet old enough). Also assume that this is not detectable by auditors.

However, the external audit iterates over all leaves and compares for each index the old and the new leaf: a leaf in the new tree is either unchanged, or changed from absent to present (insertion), or changed from present to absent while satisfying the deletion conditions. Hence the auditors must see the wrongly changed value, which is a contradiction to the assumption that they don't detect it.

Thus if an External Audit has passed for epoch t, neither equivocation nor non-append-only-ness (up to legal deletion) can have happened for any epoch $s \leq t$.

4.4.2 Analysis of Query-to-SelfAudit Consistency

Assume there was a successful External Audit at epoch t. Additionally assume there was a Self Audit SA by a client A for its label at epoch $s \leq t$ and SA passes with $latestRev \geq$ rev. Also assume that A has run Self Audits at least every DeletionParam time. Next, assume we have a value query Q for label at epoch $r \leq t$. Let Q return

[4] This is required by Apple Safari (https://support.apple.com/en-ca/HT205280) and Google Chrome (https://github.com/GoogleChrome/CertificateTransparency/blob/master/ct_policy.md). Thus CAs such as Let's Encrypt automatically submit certificates to two or more logs when issuing them.

outcome $O^Q = (\tau^Q, \mathsf{rev}, \mathsf{val}^Q)$. If Q returns O^Q as a promise P, assume that there exists a Promise Audit PA of P at epoch p with $p \leq t$. Finally, for the contradiction assume that client A sees outcome $O^A = (\tau^A, \mathsf{rev}, \mathsf{val}^A)$ with $O^Q \neq O^A$. When can this situation happen?

Case i ($\mathsf{rev} = 0$). In this case we must have $\tau^Q = abs$ because for inclusion and obsolescence ProtonKT.QueryValue enforces that $\mathsf{rev} > 0$. We must also have $\tau^A = abs$ because the Self Audit is initialized with $verifiedRev = 0$ under the assumption that revision 0 is absent. Thus the Self Audit cannot be convinced that revision 0 is not absent. Recall that $\mathsf{val}_{abs} = \emptyset$ by definition. Thus we must have $O^Q = O^A$.

Case ii ($\mathsf{rev} > 0$). W.l.o.g. assume that SA is the first Self Audit of A that has $latestRev \geq \mathsf{rev}$. That is, SA sees O^A and upon completion it sets $verifiedRev' \leftarrow latestRev$. All later Self Audits of A don't see O^A again because they only look at values $\geq verifiedRev' + 1$. Thus this single SA which sees O^A fully defines client A's view of what type τ and value val its label should have at revision rev.

We now need to analyze the cases in which Q and this specific first SA can disagree. In the following, we assume that $\mathsf{rev} > 0$ and reset the case numbering for (slightly) more clarity.

Case A (no promise). First, let us analyze the case where Q returns O^Q based on a tree proof, i.e. not as a promise. For clarity, we omit the prefix **ii.A** in the case numbering.

Case 1 (different revision subtrees). Assume Q computes leaf index $idx^Q = VRF.proofToHash(\pi^Q)||\mathsf{rev}$, and similarly A computes $idx^A = VRF.proofToHash(\pi^A)||\mathsf{rev}$. If $VRF.proofToHash(\pi^Q) \neq VRF.proofToHash(\pi^A)$, then trivially $idx^Q \neq idx^A$. That is, Q and A believe label to be at different leaves in the label subtree (see subsection 3.5.2). However, this cannot happen due to our assumption that uniqueness of VRFs holds. Thus we must have $VRF.proofToHash(\pi^Q) = VRF.proofToHash(\pi^A)$. This means that Q and A must agree on the label-subtree-leaf, which also means that they see the same revision subtree for label. We will need this fact below.

Case 2 (same revision subtrees). In this case the leaf indices are the same (by Case 2.1 and the definition of leaf indices):

$$idx^Q = (VRF.proofToHash(\pi^Q) \;||\; \mathsf{rev}) = (VRF.proofToHash(\pi^A) \;||\; \mathsf{rev}) = idx^A$$

In other words, Q and SA consider the same leaf (but possibly in different trees).

Case 2.1 ($r = s$). Assume Q and SA happen at the same epoch.

Case 2.1.1 ($roothash_r^Q \neq roothash_s^A$). This is equivocation, which is a contradiction to our assumption that an External Audit has passed at epoch t with $r, s \leq t$, since we concluded that External Audits detect equivocation.

Case 2.1.2 ($roothash_r^Q = roothash_s^A$). In this case, Q and A are seeing the same tree. What can still go wrong?

Case 2.1.2.1 ($leafhash_{idx}^Q \neq leafhash_{idx}^A$). Since the roothashes are the same, there must be a hash collision somewhere on the path from the leaf to the root. This is a contradiction to our assumption that SHA-256 is collision resistant.

Case 2.1.2.2 ($leafhash_{idx}^Q = leafhash_{idx}^A$). Assume the leaf hashes are the same.

Case 2.1.2.2.1 ($\mathsf{val}^Q \neq \mathsf{val}^A$). Assume the leaf hashes are equal but the values are not (and hence $O^Q \neq O^A$).

Case 2.1.2.2.1.1 ($\tau^Q = abs, \tau^A = abs$). Cannot be because then the values are not different: $\mathsf{val}^Q = \mathsf{val}^A = \emptyset$.

Case 2.1.2.2.1.2 ($\tau^Q = abs, \tau^A = incl/obs$). Cannot be because then:
$leafhash^Q_{idx} = \varepsilon \neq h(h(SKL.data/ObsolescenceToken) \mid\mid minEpochId) = leafhash^A_{idx}$
and in this branch we assumed that the leaf hashes are equal.

Case 2.1.2.2.1.3 ($\tau^Q = incl/obs, \tau^A = abs$). Same as the previous case.

Case 2.1.2.2.1.4 ($\tau^Q = incl/obs, \tau^A = incl/obs$). Cannot be because then we must have a hash collision in the leaf hash. (For all the four possible combinations.)

Case 2.1.2.2.2 ($\mathsf{val}^Q = \mathsf{val}^A$). Since $O^Q \neq O^A$ we must have $\tau^Q \neq \tau^A$. Recall from section 3.3 that $\mathsf{val}_{abs} = \emptyset$, $\mathsf{val}_{incl} = \{data, minEpochId\}$, and
$\mathsf{val}_{obs} = \{ObsolescenceToken, minEpochId\}$.

Case 2.1.2.2.2.1 ($\tau^Q = abs, \tau^A \neq abs$). Recall that we have $\mathsf{rev} > 0$. However, ProtonKT.QueryValue checks that $\mathsf{rev} = 0$ for $\tau = abs$, so it would raise a warning. Similarly, ProtonKT.SelfAudit raises a warning for $\tau = abs$ in the final for-loop. That is, Self Audit initializes $verifiedRev = 0$ and only allows non-absent updates. Both are contradictions to our assumption that Q and SA completed successfully.

Case 2.1.2.2.2.2 ($\tau^Q \neq abs, \tau^A = abs$). Same as the previous case.

Case 2.1.2.2.2.3 ($\tau^Q = incl, \tau^A = obs$). Because $\mathsf{val}^Q = \mathsf{val}^U$ we must have $data = ObsolescenceToken$, and Q interprets it as an SKL data and A as an ObsolescenceToken. However, this is a contradiction to the fact that the algorithms checks that the SKL data is JSON-encoded and that ObsolescenceToken is a non-empty hex value.

Case 2.1.2.2.2.4 ($\tau^Q = obs, \tau^A = incl$). Same as the previous case.

Case 2.2 ($r \neq s$). Assume Q and SA happen at different epochs. Recall from Case 2 that $idx = idx^Q = idx^A$.

Case 2.2.1 ($leafhash^Q_{idx} = leafhash^A_{idx}$). Same as above in $r = s$ (Case 2.1.2.2).

Case 2.2.2 ($leafhash^Q_{idx} \neq leafhash^A_{idx}$). Assume the leaf hashes differ between the tree that Q sees and the tree that A sees. Thus we must have $\mathsf{val}^Q \neq \mathsf{val}^A$.

Case 2.2.2.1 (equivocation). Assume Q and A are seeing trees that have diverged. This is a contradiction to the earlier conclusion that an External Audit (which we assumed has run) detects equivocation.

Case 2.2.2.2 (no equivocation). Assume there is a single tree that has been evolved from epoch r to epoch s (through insertions, updates, deletions).

Case 2.2.2.2.1 (leaf modified in-place). Assume that the value of our leaf at idx was maliciously modified in-place. This is a contradiction to the earlier conclusion that an External Audit (which we assumed has run) detects non-append-only-ness.

Case 2.2.2.2.2 (leaf illegally deleted). Same as leaf modified in-place.

Case 2.2.2.2.3 (leaf legally deleted). Assume the leaf at idx was legally deleted because it was superseded by a newer leaf with revision $\mathsf{rev} + 1$ more than DeletionParam time ago. Recall that SA is the first Self Audit that sees rev. Thus there was no Self Audit in the time interval [rev inserted, ($\mathsf{rev} + 1$ inserted) + DeletionParam]. That is, SA must have run more than DeletionParam after $\mathsf{rev} + 1$ (sic!) was inserted. This is a contradiction to our assumption that A has run Self Audits every DeletionParam time.

Case B (promise). Now let us analyze the case where Q returns O^Q as a promise P. By our initial assumption there exists a successful Promise Audit PA that sees P at epoch $p \leq t$. For clarity, we omit the prefix **ii.B** in the case numbering.

Case 1 (not included). Assume the server did not include O^Q in the tree at p.

Case 1.1 ($|time(P) - time(R)| >$ MMD). In this case PA raised a warning, which is a contradiction to the assumption that PA passed.

Case 1.2 ($|time(P) - time(R)| \leq$ MMD). In this case PA passed but saw neither an inclusion nor obsolescence proof for P. This is the contradiction to the assumption that PA is the specific Promise Audit which saw P in the tree.

Case 2 (included). Assume O^Q is included in the tree at p. Here we can repeat the same argument as Case A, setting $r \leftarrow p$ and $Q \leftarrow P$.
Only Case 2.1.2.2.2.1 is modified to instead of ProtonKT.QueryValue raising a warning because the check fails, ProtonKT.PromiseAudit either raises a warning because the MMD is overdue, or PA did not see P (because $\tau = abs$ is the same as Case 1).
With these modifications we can see that all branches of the argument in Case A hold.

Overall all branches lead to a contradiction. Therefore we can conclude that our initial assumption that $O^Q \neq O^A$ was wrong, thus we must have $O^Q = O^A$. Thus Query-to-SelfAudit consistency holds.

Leaf index collisions In the argument above we needed uniqueness of VRFs but not collision resistance. But what if there is a VRF hash collision? That is, what if two different (normalized) email addresses A and B VRF-hash to the same value?

For Self Audits, if a leaf contains the keys for A, then B's Self Audit will not recognize them as its own, and will thus raise a warning. Thus it is in the interest of the server not to trigger a VRF hash collision. Because B's Self Audit will abort raising a warning, this is also not an attack on Query-to-SelfAudit consistency (which only considers successful audits).

Note that in the argument for Query-to-SelfAudit Consistency we don't have such cases because we are only looking at a fixed label.

No Self Audits, No Security Note that Self Audits and their correctness checks are critical for any practical security. If Self Audits are not run, spurious key changes are not detected by KT. Users need to be online regularly and run Self Audits, at least once every three months.

Non-Proton Addresses This need for Self Audits means that Non-Proton Addresses (i.e., addresses that have no Proton account) cannot get any security from KT. The key server can insert malicious keys into the tree, and because they are not audited we cannot be sure that these keys are correct.

However, this problem is not new, it also exists without KT. KT simply does not improve over the status quo if there are no Self Audits. Users should continue to use the Address Verification feature to pin keys of Non-Proton Addresses.

5 Future Work

In this section, we take a look at what's next for ProtonKT, and future improvements that may be done.

5.1 Additional verification of Merkle Tree roots

As mentioned in section 3.6, we currently publish chain hashes to Certificate Transparency logs (via a TLS certificate), committing the Merkle Tree root of every epoch to a public append-only log. The CT log provides a Signed Certificate Timestamp (i.e. a promise to include the TLS certificate in the log), which is verified by the clients. Additionally, auditors verify that the CT log is operating consistently. This ensures that it is impossible for any single party to equivocate and surreptitiously serve a malicious key to a user. Nevertheless, there are still a number of improvements we could make to strengthen the commitment and improve the verification of Merkle Tree roots in the clients, to provide even more avenues for detection of misbehavior of the server.

5.1.1 Publishing to Blockchains

First of all, we may want to additionally publish the chain hashes to a public blockchain, such as that of Bitcoin or Ethereum. Users and auditors could then choose to additionally verify that epochs are included in these blockchains before trusting any given epoch.

The reason this was not done in the initial version is that for the web client, verifying data in a public blockchain is difficult, and for the mobile clients, verifying data in a public blockchain is prohibitively expensive. However, one way to circumvent this would be for users to run their own blockchain node, which the client could then communicate with, if the user configures it to. This functionality would benefit advanced users with particularly strong security requirements.

5.1.2 Verifying SCT inclusion proofs

An alternative security improvement, that could potentially be done without involvement from the user, would be to verify that TLS certificates which a given Certificate Transparency log promised to include (via a Signed Certificate Timestamp), were indeed included within the "maximum merge delay" of Certificate Transparency (24 hours).

This could be done by storing Signed Certificate Timestamps locally, and later requesting an inclusion proof, potentially indirectly (via Proton's servers). If the certificate was not included, the user should be warned, similarly to if a Signed Key List was not included in Key Transparency within its maximum merge delay (currently 72 hours).

In the web client, this functionality could alternatively be left to the browsers, which may also implement this verification. In this case, simply making a request to the relevant domain (over HTTPS) would be sufficient for the browser to verify the TLS certificate, its Signed Certificate Timestamp, and later its inclusion in the Certificate Transparency logs.

5.2 Standardization

Our current implementation of Key Transparency is specific to Proton, and intended to protect communications between Proton users. In the future, it may be desirable to also protect communications between Proton and non-Proton users in a similar way. To achieve this, (a version of) Key Transparency will need to be standardized and implemented by various providers.

To this end, there is a Key Transparency Working Group at the Internet Engineering Task Force (IETF)[1], aiming to standardize a variant of Key Transparency. We plan to contribute to this work whenever relevant, and may implement the standardized version of Key Transparency once finalized.

[1] https://datatracker.ietf.org/wg/keytrans/about/

Bibliography

[1] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. Cryptology ePrint Archive, Paper 2014/1004, 2014. https://eprint.iacr.org/2014/1004.

[2] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. Cryptology ePrint Archive, Paper 2018/607, 2018. https://eprint.iacr.org/2018/607.

[3] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. Cryptology ePrint Archive, Paper 2023/081, 2023. https://eprint.iacr.org/2023/081.

[4] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science (FOCS '99)*, pages 120–130, 1999. https://dash.harvard.edu/handle/1/5028196.

[5] Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Včelák. Verifiable Random Functions (VRFs). RFC 9381, August 2023. https://datatracker.ietf.org/doc/html/rfc9381.

[6] Werner Koch. OpenPGP Web Key Directory. Internet-Draft draft-koch-openpgp-webkey-service-16, Internet Engineering Task Force, May 2023. https://datatracker.ietf.org/doc/draft-koch-openpgp-webkey-service/16/.