ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Security Analysis of Proton Key Transparency

Master Thesis

Thore Carl Göbel

20. November 2023

Advisors: Prof. Dr. Kenny Paterson, Felix Linker (ETH Zürich),
Daniel Huigens (Proton AG)

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Cryptography is a tool to turn any problem into a key management problem. More often than not, the answer to the question "How can I obtain keys authentically?" is "Trust the key server." This makes the key server a single-point-of-failure for end-to-end encrypted communication systems. Previous approaches such as PGP's Web of Trust or Signal's Safety Number don't scale to millions of users.

Key Transparency (KT) addresses this problem by making server actions transparent. KT logs all public keys in a Merkle hash tree, producing a consistent, verifiable key directory. Clients can check that what they believe to be their contacts' keys is consistent with the tree, and clients can monitor their own keys to be correctly included in the tree. KT removes the need to trust the key server (and moves it to the need to trust one of the auditors who check that the tree is correctly constructed).

ProtonKT is Proton's design of a KT scheme. It is being rolled out not only in their email service but also across their product suite. In this thesis, we analyse ProtonKT. We start by giving a specification of how ProtonKT works, providing high-level intuition as well as detailed algorithm descriptions. We also define the security property that ProtonKT should provide: Query-to-SelfAudit Consistency. We analyse the protocol manually and argue why this property holds. Finally, we formally model ProtonKT with the Tamarin Prover, making use of the recently introduced subterms and natural numbers. Unfortunately, we did not manage to find a formal proof because we encountered a limitation in Tamarin's induction mechanism.

Acknowledgements

Thank you Daniel Huigens for proposing an exciting topic, and for the opportunity to collaborate on analysing ProtonKT. And thank you to the entire Proton cryptography team – Aron, Daniel, Lara, Lukas, and Marin – for the helpful discussions, and for the clarifications when I was trying to find out why ProtonKT does something and what the historical context is.

A big thank you to Felix Linker, for your ability to ask the right questions, for guiding me in the right direction, and for giving more structure to this work. Your advice and feedback really helped to understand Tamarin better and write less messy models. I appreciate your willingness to join half-way through this thesis.

Also thank you Prof. Kenny Paterson for allowing me to do this thesis in the Applied Cryptography Group. I am grateful for the pointers you gave me, in terms of papers to look at, but also in terms of people to reach out to (Felix!).

Finally, thank you to my girlfriend, my family, and all my friends for supporting me. I hope the Buchstabensuppe analogy helped more than it confused you.

Contents

Chapter 1

Introduction

End-to-end encrypted (E2EE) communication systems have become ubiquitous. Instant messaging apps like Signal or WhatsApp which offer E2EE through the Signal Protocol have millions or even billions of users. Email providers like Proton Mail offer easy-to-use E2EE email using PGP.

However, the Achilles' heel of E2EE is key management: How do users get an *authentic* copy of their contacts' public keys? A malicious key server colluding with the message transport server could launch a MITM attack by serving fake public keys, breaking E2EE. For example, a key server could be hacked or legally coerced.

To detect such attacks, users currently need to verify public keys manually out-of-band. Mobile messengers make this relatively easy. Signal and WhatsApp users can compare their Safety Number by scanning a QR code. Threema gamified public key verification by prominently displaying coloured dots in the chat view: red for untrusted keys, green after scanning a QR code with the public key. [1] In email and PGP, however, even out-of-band verification is often not available in a user-friendly fashion. For example, mainstream PGP-based services like Proton Mail offer no way to compare fingerprints in their mobile app.

Unfortunately, out-of-band verification has usability and scalability problems. First, it is one-to-one, meaning that every pair of contacts needs to scan their QR codes. In a large group of friends this quickly becomes tedious and time-consuming. Anecdotally[2], not even cryptographers compare Safety Numbers very often. Second, public keys change, especially in mobile messengers when users switch phones. Every key change results in a notification in Signal. The noise of these key change notifications can lead to users ignoring the warnings over time. Third, out-of-band comparison is impossible when users have no prior communication channel (e.g. a whistleblower contacting a journalist).

[1] https://threema.ch/en/faq/levels_expl

[2] https://twitter.com/matthew_d_green/status/1646532201948012545

Key Transparency Key Transparency (KT) promises to address the problem of obtaining authentic public keys with a more scalable solution. The idea of KT is to build a verifiable key directory. When clients look up keys in the directory, the response will contain both the value and a proof that this value is consistent with the directory state. This ensures that all users of the system can have the same view of the directory. It also makes server actions transparent: If Alice looks up her own keys, she can be sure that when Bob looks them up Bob will see the same keys. Then Alice can simply monitor the directory regularly to check that her correct keys are still there. Alice can also see the history of her keys. This allows Alice to detect when the server is temporarily inserting a bad key and then swapping it back for her real keys.

The beauty of KT is that it requires no user interaction in normal usage. When Bob first communicates with Alice, he does not need to scan the Safety Number. Bob can rely on Alice to make sure her keys are correct, which works thanks to the consistency of the directory. Similarly, Alice's client can monitor her keys in the background, thus ensuring her keys are correct. The only point of user interaction is when Alice's client detects some unexpected keys. Then the client raises a warning to inform Alice.

Another beauty of KT is that it shifts the burden of monitoring for *correct* public keys from the contacts to the owner of the key. Consider the warning you get when you log in to your Google account from a new device. The pop-up "Recent log-in on a new device at IP 195.176.110.112" is only useful to yourself. If it was you, you ignore the pop-up. If it wasn't, you investigate. But if every time your friend Alice logged in on a new device you would get a pop-up "Alice just logged in on a new device", what would you do? This is exactly what is happening with Signal's Safety Number: your contacts receive a warning, yet the only person who can audit whether the key change is legitimate is yourself. In KT, Bob does not get a warning when Alice changes her keys. Only Alice is notified, and it is up to her to verify that the change is legitimate.

Proton Proton is a tech company offering a suite of products consisting of Mail, Calendar, Drive (cloud storage), Pass (a password manager), and VPN. Its primary selling point is privacy. Using end-to-end encryption Proton wants to ensure that only users themselves can access their data, but not the company or anyone with access to their servers. Proton runs a public key server, which is used both to look up keys when emailing someone, but also when sharing a file or a calendar invite. Thus Proton is naturally interested in KT: in Proton's overall threat model its own servers are untrusted, including its key server. KT would allow Proton users to automatically verify each others' keys, removing the need to trust the server or do out-of-band verification.

1.1 Organisation

In this thesis, we describe and analyse Proton's Key Transparency design. Chapter 1 surveys related work, and chapter 2 introduces some preliminaries. After that, chapter 3 provides a specification of Proton's Key Transparency scheme. We then

define security properties and analyse the scheme in chapter 4. Chapter 5 provides recommendations for improving the scheme. Finally we conclude in chapter 6.

For the rest of this chapter, we first discuss the related work, then look at KT in the real world, and end by stating our contributions with this thesis.

1.2 Certificate Transparency

Before we discuss Key Transparency in more detail, let us look at a related scheme that inspired KT: Certificate Transparency (CT). This will allow us to draw parallels and analyse the differences between CT and KT.

CT was introduced by Google in 2013 [16] and updated in 2021 [17]. Google wanted to address the problem of malicious Certificate Authorities (CAs) issuing certificates for someone other than the real domain owner, which allowed MITM attacks. CT makes the actions of CAs "transparent" by having append-only logs of (nearly) all the certificates ever issued by CAs. [3] *Append-only* means that entries can only be added but not modified or deleted. The logs are operated by CAs themselves but also by other organisations such as Cloudflare. The log entries are what we call *non-repudiable proof* of a CAs' behaviour: If a certificate is logged, the CA cannot later claim that it didn't issue that certificate. Importantly, CT does not prevent CAs from wrongly issuing certificates, but it does make it transparent and publicly detectable.

Concretely, CAs insert the certificates that they issue into a Merkle tree (the "log"). To distribute trust, CAs must insert the certificate into at least two logs operated by two different entities. During a TLS handshake, browsers can require that the server presents both a certificate and a proof of inclusion of the certificate in the two logs. Third-party monitors continuously scan the logs, a) checking that the Merkle tree is consistent (i.e. correctly constructed and append-only), and b) scanning for unexpected certificates on behalf of domain owners and notifying them.

However, in CT there is no relationship between a certificate and its leaf index in the tree. It is up to the log operator to place certificates, usually in the order that they come in. Thus everytime example.com requests a new certificate, this will create a new leaf somewhere in the tree. Therefore, any third-party monitor who wants to find all certificates for example.com needs to scan the entire tree and look at every single leaf, which is expensive.

A second downside of CT is that there is no privacy. The certificates in the tree are public and the certificates contain the domain name in the Subject Alternative Name field. Because of that, CT leaks basically the entire list of domain names that are using

[3] Nearly all because CAs can still issue certificates without including them in a log. However, these certificates won't be accepted by Google Chrome and Apple Safari.

certificates on the web. This can reveal the existence of a websites that would like to stay hidden such as secretcorporateportal.com.

Finally, CT relies on inclusion proofs with respect to a tree and the fact that this tree is append-only. What if the log operator secretly creates two trees, and e.g. inserts a certificate only into one of them? Depending on who is querying the log, the operator could either use one or the other tree. This is a split-world-view attack, also called *equivocation*. To ensure the server is not equivocating and creating multiple trees, monitors need to communicate and compare their views of the tree. This can be done by comparing the root hash of the Merkle tree. However, as the RFC itself admits: "This technique, known as 'gossip', is an active area of research and not defined here." [17, Section 11.3]

1.3 Related Work

In this section we discuss the history of Key Transparency designs. The most relevant papers are CONIKS [21], SEEMless [7], and Parakeet [19]. CONIKS is the first academic KT design. SEEMless formalises the concepts behind KT, and makes privacy and efficiency improvements over CONIKS. Parakeet makes further storage improvements over SEEMless and introduces a consistency protocol to prevent split-world view attacks.

1.3.1 CONIKS

Melara et al. published CONIKS in 2014 [21]. CONIKS is considered the first Key Transparency design. It was inspired by Certificate Transparency, taking the idea of making CA behaviour transparent and mapping it to key servers [4]. CONIKS creates *authenticated bindings* from usernames to public keys.

Design Like CT, CONIKS is based on a Merkle tree. The leaf nodes are not hashed certificates but instead hashed public keys. The index of a leaf node is not chosen arbitrarily but instead derived from the username. Given a leaf, an inclusion proof is the path from the leaf to the root. The key server signs and publishes the tree root (Signed Tree Root (STR)). Key changes are incorporated in batches, and a new STR is published every epoch.

Consistency CONIKS' functional goal is to achieve *consistency*. That is, every user sees the same username–key bindings [21, Section 1]. The bindings might not be *correct* (a stronger property), i.e. there might be a key bound to user Alice which is actually controlled by Bob. However, since the bindings are consistent, everyone including Alice, will see the same key bindings (and Alice can complain).

[4] CONIKS calls the key servers "identity providers".

Security Goals CONIKS' main security goals are [21, Section 2.2]:

- *Non-equivocation:* If a key server presents a split-world view to two parties, it has to maintain this equivocation forever. With enough auditors who communicate with each other and compare tree roots, the chance of equivocation being detected is high.
- *No suspicious keys:* Fake keys added by a malicious key server are quickly detected and there is non-repudiable proof.
- *Privacy:* Given some consistency and/or inclusion proofs, an adversary learns nothing about which other usernames are contained in the tree, or which public keys are linked to these usernames.

The first two properties are straightforward, and similar to CT. Privacy, however, is a new property specific to CONIKS and KT.

Privacy The attack against privacy is the following: Assume that leaf indices are simply $idx = \text{Hash}(username)$. Also assume that the key server is rate-limiting lookups. That is, the adversary can query a few but not all usernames to obtain some public keys and inclusion proofs. That is, the adversary knows a subset of the tree (the paths in the proofs). It can now launch an *offline attack* and brute-force the rest of the tree. This is possible since the hash function is public (the adversary knows how to compute it). By distinguishing whether a leaf is empty or not, and also brute-forcing which username is located at which leaf index, the adversary can learn which usernames exist. Note that usernames often have low entropy (e.g. phone numbers or email addresses).

To defend against this, CONIKS computes the leaf index as $idx = \text{VRF}_{sk}(username)$. See section 2.1 for details about Verifiable Random Functions (VRFs). Importantly, the VRF is keyed with a secret key only known to the server. The adversary can still complete the hash tree, but it can no longer compute which leaves belong to which usernames. Meanwhile, normal users can validate inclusion proofs (Does the path start at the correct leaf?) because the server provides a VRF proof proving that idx was calculated correctly. This VRF proof can be verified using the server's public key.

In addition, CONIKS hides the public keys by not storing the keys themselves, but instead storing a commitment to the keys. Thus knowing the leaf content (through brute-force) does not leak anything about the keys.

Overall, privacy is improved compared to CT, where all certificate data is public.

Monitoring Monitoring in CONIKS is more private and also more efficient than in CT. In CT, monitors need to check every single leaf. Otherwise the monitor could miss a misissued certificate. In CONIKS scanning the entire tree is not desired for privacy. Luckily, the uniqueness property of VRFs ensures that each username has a single unique index. Thus to monitor their keys, a user only needs to do a single query for

their own username. They need to verify a single VRF proof (Is the index correct?) and a single inclusion proof (Is the path consistent with the tree root?).

This allows CONIKS to place the burden of monitoring public keys on the users. Every user monitors their own entry for suspicious keys by querying the key server for their own username every epoch.

Third-party monitors only need to monitor for non-equivocation. They verify the tree roots (STRs) and that they are correctly chained across epochs. If the tree roots ever diverge, this is non-repudiable proof of bad server behaviour. However, this requires a large network of monitors, or users that can gossip and compare tree roots. Otherwise, equivocation attacks are possible.

Formalism The CONIKS paper does not formally specify the protocol and the security properties. It also lacks formal proofs, analysing only non-equivocation and giving intuitive arguments for the other properties.

1.3.2 SEEMless

Building upon CONIKS, SEEMless was published in 2018 [7]. SEEMless is the first to formalise KT and provide formal proofs. It also makes auditing the directory more private and more efficient compared to CONIKS. Finally, it greatly reduces the server-side storage requirements and allows for shorter epochs.

The main difference between CONIKS and SEEMless is that SEEMless maintains an append-only tree across epochs, whereas CONIKS maintains a unique tree for every epoch.

Formalism: VKD SEEMless formally defines the *Verifiable Key Directory (VKD)* as a primitive for KT. A VKD is a set of algorithms [7, Section 2]: Publish, Query, QueryVer, KeyHistory, HistoryVer, Audit. These algorithms allow the server to publish commitments to the directory, clients to query usernames and their own key change history, and auditors to audit consistency.

For this VKD definition, the paper formally defines and proves completeness, soundness, and privacy. Privacy is defined through a leakage function: how much information is leaked by the server responses (the commitments, the proofs)?

Formalism: aZKS SEEMless also defines another primitive: *append-only Zero Knowledge Sets (aZKS)*. A normal ZKS is a commitment to a fixed set of (label, value) pairs, such that (1) the prover/committer can prove label (non-) membership with respect to the commitment, but also such that (2) neither the commitments nor the proofs leak any information about the set. aZKS are a generalisation of ZKS such that the set is not fixed and labels can be added (but not deleted or edited).

SEEMless uses the aZKS to build the VKD. In the VKD, labels are usernames with revisions, and values are public keys. Key changes are insertions of the same username with a later revision.

Concrete Construction After these formal foundations the paper describes a concrete instantiation of a VKD called "SEEMless" (hence the paper name). The construction is designed to be modular. The VKD is built from aZKS, and the aZKS is built from a Patricia trie (a compact binary tree, also known as radix tree). Like in CONIKS, the leaf indices and leaf values are protected with VRFs and commitments respectively.

The Patricia trie construction is very intricate. It is designed to be space-efficient, i.e. to grow with the number of key changes as opposed to growing with the number of epochs. It is also designed to allow for efficient insertions, and efficient Query and KeyHistory queries. The tree construction achieves this by not naïvely replicating the entire tree every epoch, and instead building a single tree that stores the historical delta of the tree inside the tree itself.

The SEEMless construction introduces several new ideas compared to CONIKS: First, usernames have revisions. So the labels of the (label, value) pairs in the VKD are not `username` but `username:revision`. This is necessary to fit the aZKS model where labels cannot be edited and only new labels can be inserted. It also prevents a tracing attack (see below).
Second, SEEMless maintains two aZKS per epoch (rather than one): an "all" aZKS and an "old" aZKS. $\mathsf{aZKS}_{\mathsf{all}}$ contains all the username:revisions. $\mathsf{aZKS}_{\mathsf{old}}$ contains all the username:revisions minus the ones that were newly added in the current epoch. The server can now prove that a revision is the latest one by giving a membership proof for $\mathsf{aZKS}_{\mathsf{all}}$ and a non-membership proof for $\mathsf{aZKS}_{\mathsf{old}}$. [5]

Auditing This construction requires that every epoch is audited by at least one honest auditor who verifies the append-only property. Offloading this work to an auditor allows clients to perform less work because they only have to check the latest tree (and not all trees like in CONIKS).

Tracing Attack SEEMless also identifies an attack on privacy in CONIKS (Appendix C [7]). Consider the case where Alice and Bob have two leaf nodes that share the same direct parent. Bob queries for Alice's key only once, thus learning her position in the tree. Now Bob can simply query his own key regularly. Alice's leaf is included in the inclusion proof for Bob's key, thus Bob can observe when Alice's key changes without querying for Alice.

This is a *tracing attack* on privacy, because key changes are sensitive. For example if Bob compromises Alice's private key, the fact whether or not Alice rotates her key tells Bob whether the compromise was detected.

[5] Assuming that users audit that their own keys are correctly inserted in both aZKS and that there are no additional unexpected keys.

SEEMless is not vulnerable to this, because key updates are not edits of an existing label `alice`, but rather the insertion of a new label `alice:3` with a revision. Thus to learn Alice's key changes, Bob must directly query for her key. Direct querying can be subject to server policies such as rate limits or Alice blocking Bob.

1.3.3 Parakeet

Parakeet, published in early 2023, is one of the most recent academic works on KT [19]. It builds and improves upon SEEMless in several ways: Firstly, Parakeet proposes a more efficient data structure for the VKD that scales to billions (and not just millions) of users. This includes the ability to safely prune the tree over time and delete old entries. Secondly, the authors propose a consistency protocol to publish and agree on the tree roots, something which previously required an undefined gossip protocol.

oZKS Parakeet uses the concept of an *ordered append-only zero-knowledge set oZKS*. An oZKS is a generalisation of an aZKS, with the additional property that there is a strict ordering of the inserted elements.

Compaction (Deletion) Parakeet extends the VKD definition to a "VKD with compaction". Compaction allows the server to delete old unused keys. This works by first marking elements as *stale* during *tombstone epochs*. In subsequent *deletion epochs* these stale elements are removed. The interval between deletion epochs is a system parameter.

Clients are required to come online at least once between two deletion epochs and check their own entries. An external honest auditor, in addition to checking append-only-ness, now also checks that only old values are marked as stale, and that only stale values are deleted, and that these actions only occur in the dedicated tombstone and deletion epochs.

Consistency protocol The security of all KT schemes relies on the fact that equivocation is eventually detected.

In CONIKS clients are expected to proactively contact "enough" auditors and ask them for their observed root commitments [21, Section 4.1.4]. Auditors and clients effectively gossip between each other.

SEEMless skips over the consistency protocol entirely and simply assumes that some way to achieve consistency exists (e.g. through gossip, a blockchain, or some other mechanism [7, Section 2.1]).

Parakeet approaches this problem from a distributed systems perspective. It proposes the following consistency protocol: the server sends each VKD update to some well-known *witnesses*. The witnesses verify the update proof and if it passes sign the updated root commitment. The server collects the witnesses' signatures into a *certificate*.

Later, the server presents this certificate to the clients alongside the root commitment. This ensures non-equivocation, assuming that there are enough honest, non-Byzantine witnesses (a quorum). This is convenient for clients because they only need to contact the central server, as opposed to having to run a complicated gossip protocol.

1.3.4 Comparison between CONIKS, SEEMless, Parakeet, ProtonKT

In Table 1.1 we compare some metrics for CONIKS, SEEMless, Parakeet, and – looking ahead – also ProtonKT.

We assume that there are n usernames, each with one initial key, t epochs, c key changes for a single user, and C key changes aggregated across all users (in addition to the users' initial keys).

	CONIKS	SEEMless	Parakeet	ProtonKT
Merkle tree	one tree per epoch	one tree forever	one tree forever	one tree forever
Tree size	$\mathcal{O}(n \cdot t)$ [6]	$\mathcal{O}(n + C \cdot \log n)$ [7]	$\mathcal{O}(n + C)$ [8]	$\mathcal{O}(n + C)$
Key History time	$\mathcal{O}(t \cdot \log n)$ [9]	$\mathcal{O}(c \cdot \log n)$ [10]	$\mathcal{O}(c \cdot \log n)$	$\mathcal{O}(c \cdot \log n)$
Consistency mechanism	Comparing with other auditors, gossip	Undefined, assumed to exist	Certificate signed by quorum of witnesses	CT logs
Principals	IdP, clients, auditors	IdP/server, clients/ users, auditors	IdP, users, witnesses	server, clients, auditors

Table 1.1: Comparison between CONIKS, SEEMless, Parakeet, ProtonKT

1.4 KT in the Real World

This section describes some real-world KT implementations that are not just academic proposals. They show the industry's interest in KT between 2014 and now.

Keybase Keybase is a service where users can link devices, public keys, and identities on other platforms (such as Twitter or GitHub) together into a single Keybase identity. All statements ("I own this key", "I control this GitHub account") are cryptographically signed. The server maintains a *sigchain* for each user. New signed statements are appended to the sigchain, effectively providing an immutable history of the user's statements.

[6] There is a new tree every epoch.

[7] There is one tree, and adding a new leaf changes up to all nodes on the path. Need to store old nodes to reconstruct old tree states.

[8] There is one tree. Old tree states can be directly reconstructed from that tree.

[9] The client needs to check the membership proofs for all epochs. Membership proofs are paths from the leaf up to the root.

[10] For each key change there is one membership proof.

To prevent the server from maliciously changing a user's sigchain Keybase uses a KT scheme (albeit not calling it Key Transparency). They issued their first epoch back in 2014. [11] This makes it one of the earliest real-world KT implementations that we are aware of. We don't know whether Keybase was inspired by CONIKS (also published in 2014) or coincidentally working on it at the same time.

First, a new statement is signed and appended to a user's sigchain. Then the sigchain is hashed to form the leaf of a Merkle Tree. The root of the Merkle Tree is inserted into a blockchain (originally Bitcoin, later Stellar). [2]

Clients verify all statements, their signature, the tree inclusion proof, and that the tree root is included in the blockchain. However, there are no third-party auditors that e.g. monitor that the tree is append-only, or that monitor for spurious tree roots in the blockchain. Security relies on the fact that there is cryptographic proof, and that someone *might* find it and complain, thus deterring malicious server behaviour.

Zoom Zoom, which acquired Keybase in 2020, is planning to implement KT with what they call the *Zoom Transparency Tree*. [12] Their design is inspired by Keybase.

Google Google announced Key Transparency as an "open-source prototype" [13] in 2017. Their implementation is inspired by CONIKS. However, as of July 2021 the repository is archived, and it is unclear whether this project ever reached the production stage. [14]

One idea that Google proposed (but didn't implement) is "gossip stapling": [15] monitors sign the tree roots that they see and return it to the server. The server can then give the monitor signatures to clients. This is very similar to Parakeet's idea of witness certificates. The difference is that Parakeet certificates require a quorum, whereas gossip stapling could be a single signature from a monitor that the client trusts.

WhatsApp In WhatsApp, keys are bound to an app installation. Every time the user changes their device or reinstall the app, this leads to a key change. In practice, this leads to a lot of noise for users who have enabled notifications for Safety Number changes.

WhatsApp announced in April 2023 that they were deploying KT based on Parakeet [18]. They initially wanted to implement SEEMless, but in order to scale to two billion users they had to make performance improvements that led to Parakeet [24]. WhatsApp's epochs are in the order of 5 to 10 minutes [24].

[11] https://keybase.io/_/api/1.0/merkle/root.json?seqno=1

[12] Section 4 of https://github.com/zoom/zoom-e2e-whitepaper/raw/master/archive/zoom_e2e_v4_1.pdf and https://datatracker.ietf.org/meeting/116/materials/slides-116-keytrans-keybase-and-zoom-00

[13] https://security.googleblog.com/2017/01/security-through-transparency.html

[14] https://github.com/google/keytransparency

[15] https://github.com/google/keytransparency/issues/178

Apple iMessage Apple announced in October 2023 that it is rolling out "Contact Key Verification" in iMessage in iOS 17.2 and macOS 14.2. [10] Its KT design introduces *Signed Mutation Timestamps (SMTs)*: similar to SCTs in CT, the server signs a promise to include a value in the tree. This allows the server to reply immediately (even if the new value is not yet included in the tree), while still producing not-repudiable proof of server (mis)behaviour.

Proton Proton is also developing KT, primarily for its email service, but also in other contexts where authentic public keys are required, e.g. for sharing calendar invites, sharing files on Drive, or sharing password vaults. Proton is rolling out KT during 2023.

The goal of this thesis is to analyse Proton's KT scheme.

Proton assumes that users have a long-term secret: their password. This is unlike WhatsApp (which assumes users cannot keep passwords or devices), but like Keybase (which assumes that users have at least one trusted device that can sign key updates). The server then stores the private key on behalf of the user, encrypted with a symmetric key derived from the password. This means that there are far fewer key changes: existing users only change their keys when (1) they forget their password and thus cannot decrypt their old keys, or (2) they explicitly generate new ones (e.g. to replace old RSA-2048 with new Ed25519 keys). Notably keys do not change when the mobile app is reinstalled (unlike WhatsApp). Proton's epochs are normally 4 hours long.

OpenPGP KT has been discussed at two OpenPGP summits (2015 and 2019). [16] [17] But so far no email provider except Proton is known to have deployed KT, nor is there any draft standard.

IETF The IETF has formed a Working Group to develop a standard for KT, mainly in the context of Message Layer Security (MLS). Initial discussions were held at a Bird-of-Feather meeting at IETF 116 in March 2023, and the Working Group was formed in October 2023 just before IETF 118. [18]

Other Applications for Transparency Logs The idea of using Merkle Trees to create transparency logs has also been applied to other contexts. For example, Golang uses transparency logs to secure their module ecosystem. [19]

[16] https://wiki.gnupg.org/OpenPGPEmailSummit201512
[17] https://wiki.gnupg.org/OpenPGPEmailSummit201910Notes#Workshop:_Key_Transparency
[18] https://datatracker.ietf.org/wg/keytrans/about/
[19] https://go.googlesource.com/proposal/+/master/design/25530-sumdb.md

Google has developed Trillian, a general Verifiable Log/Verifiable Map implementation based on a Merkle tree. Trillian underlies Google's CT implementation but also their KT prototype. [20] Google also has a wider Transparency initiative. [21]

Finally, Mozilla floated the idea of Binary Transparency in 2017, [22] but it was never productionised in Firefox. [23] Mozilla proposed to build a Merkle tree out of the Firefox release artefacts and request a web-PKI certificate for the domain `hash[0:32].hash[32:64].releaseVersion.binaryTransparencyVersion.fx-trans.net`. Because the certificate is logged in CT and CT is assumed to be immutable, this certificate serves as a commitment to the root hash. This is interesting because Proton later developed the same idea again independently (see section 3.6).

1.5 Contributions

In this thesis we make the following contributions:

1. We provide a pen-and-paper description of Proton's Key Transparency scheme. This is interesting for two reasons:

   First, like SEEMless, ProtonKT builds upon CONIKS. But ProtonKT makes different design decisions that lead to different properties. Take privacy for example: in SEEMless insertions of new users are indistinguishable to auditors from key updates for existing users (see section 1.3.2). This comes at the cost of a much more complicated tree construction and larger inclusion proofs (e.g. to prove that a revision is the latest one). Proton decided that it does not require this indistinguishability (e.g. because the key server is public already). Thus the ProtonKT tree construction is simpler and inclusion proofs more efficient. For standardisation bodies such as the IETF who want to consider broad industry use cases, this is an interesting input.

   Second, while prior work formally defines the *Verifiable Key Directory* as the appropriate primitive for KT (see section 2.2), this definition is only in terms of the algorithm inputs and outputs, and the required properties. But the important parts for a security analysis are the internal steps of the algorithm, and the checks that it does. We provide the first full step-by-step algorithm description of a KT scheme. This format makes it easier to put the algorithm description and the software implementation side-by-side and check that they match. This is not possible with prior work.

2. We give a manual security analysis of ProtonKT. We define security properties and argue why they hold.

[20] https://github.com/google/trillian
[21] https://transparency.dev/ and https://github.com/transparency-dev
[22] https://wiki.mozilla.org/Security/Binary_Transparency
[23] https://bugzilla.mozilla.org/show_bug.cgi?id=1341396

3. We develop a formal model of ProtonKT scheme using the Tamarin Prover. This demonstrates another application of subterms, a feature that was recently introduced in Tamarin [9]. Our model makes heavy use of subterms, e.g. in form of natural numbers for epoch ids and key revisions. Unfortunately, we encountered a limitation in Tamarin's induction mechanism, and because of that did not find a formal proof for our model.

Chapter 2

Background

2.1 Verifiable Random Functions (VRFs)

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.

A *verifiable random function (VRF)*, introduced in 1999 [22], provides such a correctness proof alongside the output value.

A more informal way to think about VRF 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) [12], 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:

- *Pseudorandomnes*: 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. [1]

- *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 keypair is maliciously generated. In EC-based VRFs, clients can validate the key to detect this. See section 7 of RFC 9381 [12].

2.2 Verifiable Key Directory

SEEMless first introduced the *Verifiable Key Directory (VKD)* to formalise KT [7, Section 2]. Parakeet uses a slightly different VKD definition, which we will use [19, Appendix B].

A VKD maps *labels* (usernames) to *values* (public keys). The server maintains the actual directory Dir and some auxiliary state st. Clients can query the directory and verify the response. Auditors can verify that the directory is correctly constructed.

Parakeet defines a VKD as follows:

Definition 2.1 (Verifiable Key Directory (VKD)) *A* Verifiable Key Directory (VKD) *consists of seven algorithms:*

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

- $(\mathsf{Dir}_t, \mathsf{st}_t, \Pi^{\mathsf{Upd}}) / \perp \leftarrow \mathsf{VKD.Publish}(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)$

  Updates the directory Dir *and the internal state* st *for a set of updates* S*. Returns a proof* Π^{Upd} *that the update is correct.*

  The server is assumed to have obtained the set of key updates from clients and to aggregate them into S*.*

- $0/1 \leftarrow \mathsf{VKD.VerifyUpdate}(t, \mathsf{com}, \mathsf{com}', \Pi^{\mathsf{Upd}})$

  Verifies a single VKD.Publish *update with respect to its pre- and post-root commitments* com, com′ *using the update proof* Π^{Upd}*.*

- $0/1 \leftarrow \mathsf{VKD.Audit}(t_1, t_n, (\Pi_t^{\mathsf{Upd}})_{t=t_1}^{t_{n-1}})$

  Verifies the set of updates between the epochs t_1 *to* t_n*.*

- $(\pi, \mathsf{val}, \alpha) \leftarrow \mathsf{VKD.Query}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$

  Returns, for a queried label label*, the latest value* val *and its revision* α*.* val *may be* $\perp$ *if the label is not a member of the directory. The proof* π *verifies that the value is consistent with the directory state, i.e. it is a (non-)membership proof.*

- $0/1 \leftarrow \mathsf{VKD.VerifyQuery}(t, \mathsf{label}, \mathsf{val}, \pi, \alpha)$

  Verifies the (non-)membership proof with respect to the root commitment com_t*.*[2]

- $\left((\mathsf{val}_i, t_i)_{i=1}^{n}, \Pi^{\mathsf{Ver}}\right) \leftarrow \mathsf{VKD.KeyHistory}(\mathsf{st}_t, \mathsf{Dir}_t, t, \mathsf{label})$

  Returns the values that the label *mapped to during its lifetime and a correctness proof* Π^{Ver}*. It contains only the epochs* t_i *where the* label *was updated to a new* val_i*.*

- $0/1 \leftarrow \mathsf{VKD.VerifyHistory}(t, \mathsf{label}, (\mathsf{val}_i, t_i)_{i=1}^{n}, \Pi^{\mathsf{Ver}})$

  Verifies the key history with respect to com_t [3] *using the proof* Π^{Ver}*.*

t *is the epoch. Failure is indicated by* $\perp$ *or by the boolean* $0/1$*.*

These algorithms hide internal client-to-server API calls. For example, VKD.Query and VKD.KeyHistory will make a network request to the server to obtain the actual values. Similarly, VKD.VerifyQuery and VKD.VerifyHistory will reach out to Parakeet's WitnessAPI to obtain the tree root commitments com.

[2] In SEEMless, this algorithm also takes the commitment com_t to the tree root. In Parakeet the commitment is internally retrieved from the WitnessAPI.

[3] See footnote 2.

The server can, and is in fact expected to, apply access controls. For example, it can rate-limit calls to VKD.Query; or it can limit calls to VKD.KeyHistory such that clients can only query their own key history but not the key history of other users.

2.3 Tamarin Prover

Formal methods are used to formally, in rigorous mathematic language, specify and analyse computer systems (algorithms, protocols, programs, hardware, etc.). This is done by defining a "language" with formal syntax (i.e. with a well-defined alphabet and a grammar). A mathematical "theory" is used to introduce semantics. An example of formal languages are process calculi, such as *Communicating Sequential Processes (CSP)* and the *π-calculus*. One can then formally express desired properties of the system, and use manual or automated/computer-assisted techniques to prove these properties. This is called *formal verification*. Two main approaches to formal verification are model checking and automated theorem provers:

In *model checking*, one explores the entire state space of the system, checking that a property is always satisfied. This can be done for finite-state models (by enumerating all states, subject to state explosion), but also for infinite-state models using symbolic model checking (reasoning over collections of states). Model checking is often used in normal software development to check safety and liveness properties.

(Automated) theorem provers on the other hand use deductive reasoning: Theorem provers try to (automatically) generate a proof that a given property holds. They start from the program specification, and try to deduce/infer new statements from it, until they reach a proof. The program specification and the larger theory (the axioms) give rise to a constraint system, acting as guardrails for the inference. The inference is usually guided by heuristics. Because of this, theorem provers are closer to traditional, manual, step-by-step, deductive proofs. In fact, interactive theorem provers are not just used in computer science, but also in mathematics: one such example is Lean.[4]

Tamarin The *Tamarin Prover* is an automated, interactive theorem prover designed for security protocols analysis. It operates in the symbolic model, which allows it to reason about an unbounded number of parallel protocol executions. The language in which protocols are specified are *multiset rewriting rules*. Properties are specified using logical statements over traces of protocol executions. It can check both satisfiability (does there exist a trace satisfying this formula?) and validity (does this formula hold for all traces?). Tamarin checks validity by negating the formula and then checking satisfiability. If the negated formula is satisfiable, then this is an attack trace, which Tamarin can output as a visual graph. Tamarin can try and find proofs automatically, or it can be run interactively to construct a proof by hand and explore the protocol

[4] https://leanprover-community.github.io and https://ethz.ch/staffnet/en/news-and-events/internal-news/archive/2023/08/the-2023-paul-bernays-lectures.html

step-by-step. This interactivity, showing protocol traces as graphs, can be useful for debugging both protocol models and proofs.

Protocols are specified in Tamarin using terms, facts, and rules:

Terms, Functions, Equations Terms, together with functions, are used to represent cryptographic messages (rather than bit-strings). For example, $h(m)$ is the hash function h applied to the message term m. $senc(m,k)$ is a symmetrically encrypted term. Equational theories model properties of functions. For example the equation $adec(aenc(m, pk(sk)), sk) = m$ relates asymmetric encryption and decryption. When using terms, functions, and equations to model cryptographic primitives, Tamarin assumes these primitives to be perfect.

Terms can be of different types, also called *sorts*: Publicly-known terms are prefixed with a dollar sign \$. Freshly generated terms are prefixed with a tilde "$\sim$". Temporal terms (timestamps) are prefixed with a hashtag "#". Natural numbers are prefixed with a percentage sign "%".

Rules and Facts Rules and multisets of facts are used to represent protocol messages. Facts are built from terms. They are the "state" of the protocol. Rewriting rules consume the facts in the multiset on the LHS (*premise*), and produce new facts in the multiset on the RHS (*conclusion*). Rewriting rules are labelled with *actions* containing *action facts*. For example, the following rule takes a `State1` fact containing a message term and a key term and produces a different fact containing the symmetric encryption of m:

```
rule Encrypt:
    [ State1(m, k) ] --[ EncryptAction(m,k) ]-> [ State2(senc(m,k)) ]
```

Linear facts can only be consumed once. *Persistent facts* can be consumed arbitrarily often; they are prefixed with an exclamation mark "!". The generation of fresh, random values is modelled as the special `Fr(~m)` fact.

Network Messages and the Network Adversary Tamarin uses the Dolev-Yao adversary model, i.e. the adversary "is" the network. The network can read, modify, insert, drop, replay, reorder messages. The network is modelled using the special `Out(m)` and `In(m)` linear facts. Every message is sent out to the adversary, and every message is received in from the adversary, and the adversary can handle them arbitrarily.

This goes beyond simply modelling network messages: We can also add rules that explicitly send data out, thus giving the adversary more knowledge. For example, we can simulate the adversary compromising a party by sending out their private key.

In the example below we have three rules: one to model a PKI that generates a long-term secret key for user Alice $\$A$. With the `Out` fact we send out the public key to the network, and hence to the adversary. Note that the adversary cannot look into the `pk()` function, i.e. it cannot deconstruct this function symbol and cannot learn `~ltk`. The persistent fact `!Ltk` can be consumed multiple times. This allows the `InitAlice` rule to start many different protocol threads, identified by a thread `~id`. Finally, the `CompromiseAlice` rule models the adversary compromising Alice's long-term key.

```
rule GenKey:
      [ Fr(~ltk) ] --> [ !Ltk($A, ~ltk), Out($A, pk(~ltk)) ]

rule InitAlice:
      [ Fr(~id), !Ltk($A, ~ltk) ] --> [ StAlice1(~id, $A, ~ltk) ]

rule CompromiseAlice:
      [ !Ltk($A, ~ltk) ] --[ Compromise($A) ]-> [ Out(~ltk) ]
```

Lemmas Lemmas are used to specify security properties. Lemmas are specified using logical statements over protocol traces, in a language similar to first-order logic. `Ex` and `All` are reserved symbols. Timestamps at which facts occur in a trace are prefixed with a hashtag "#" (though sometimes it can be dropped as syntactic sugar).

For example, satisfiability lemmas are often used to prove executability of a protocol. This is useful as a sanity check to catch modelling bugs.

```
lemma Exec_AppendAuditor_Basic:
    exists-trace
    "Ex id roothash #i #j #k .
          AppendInsert(id)@i
        & AppendUpdate(id)@j
        & AppendAudit(id, %1 %+ %1 %+ %1, roothash)@k
        & i < j
        & j < k "
```

Similarly, we can write validity lemmas for security properties. For example, we can check that a value x is secret, unless the adversary has compromised a party A that was assumed to be honest: [5]

```
lemma Secrecy:
      all-traces
      "All x #i. Secret(x)@i
```

[5] This example is taken from Section 7 "Property Specification" of the Tamarin manual [23] and slightly modified.

```
    ==> not (Ex #j. K(x)@j)
        | ( Ex A #j. Honest(A)@i & Compromise(A)@j ) "
```

`K(m)` is a special fact modelling that the adversary *knows* a term m. `Secret`, `Honest`, `Compromise` are action facts that we need to have included in the action of some rule in our model.

Such lemmas of the form "secure or X" also give rise to a threat model: while we grant the adversary the power to do X in the protocol rules, in the lemma we may need to exclude attacks where the adversary actually does X. This creates a threat model where we accept X as an attack that the protocol does not prevent.

We can write auxiliary lemmas by marking a lemma as `[reuse]`. When proving a lemma, Tamarin will assume all previous reuse-lemmas hold (they need to be proven separately). Tamarin can then use the reuse-lemmas' statements to limit its search. This can be used to break down proofs into smaller ones.

Restrictions Restrictions restrict the set of traces that Tamarin will consider. They are specified as formulas over action facts. This allows us to model checks that the protocol does, by restricting when a rewrite rule can be applied. A common example is checking for equality. By writing the below restriction and adding the `Eq(x,y)` action fact to a rule's actions, we can restrict Tamarin's search to only considering traces for which the restriction's formula holds. [6]

```
restriction Equality:
    "All x y #i . Eq(x,y)@i ==> x = y"
```

Subterms Unboundedness in security protocols typically arises from two dimensions: First, unbounded parallel sessions. Tamarin can often handle these. Second, terms within a session that grow unboundedly, for example counters or hashchains. Tamarin has recently received support for these in form of *subterms* [9]. As the paper explains: "Intuitively, if x is a subterm of t, then x is needed to compute t." [9] For example, both x and $h(x)$ are subterms of $h(h(x))$. Subterms are strict, i.e. x is not a subterm of itself. Subterms allow us to write lemmas with a new subterm operator "<<". For example:

```
lemma CounterStrictlyIncreases:
      all-traces
      "All x y #i #j . Counter(x)@i & Counter(y)@j & i < j ==> x << y
```

Using subterms, Tamarin also defines natural numbers using a basis element (`%1`) and addition (`%+`). `%1` is a subterm of `%1 %+ %1`.

[6] This example is taken from Section 7 "Property Specification" of the Tamarin manual [23].

Limitations of Formal Verification No proof technique and no model can capture the entirety of any real system. For example in Tamarin, by using a symbolic model of terms we abstract away bit-strings, thus potentially excluding attacks that use side-channels or bit-flips. There can also be bugs in the model, or the actual implementation could differ from the model. Formal verification only complements other analysis techniques.

Chapter 3

ProtonKT Specification

This section 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 also describe how ProtonKT is different from other KT protocols.

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.

This construction is a trade-off: on one hand it leaks the number of revisions a user has and when they are updated. On the other hand, it makes it trivial to audit that the server is behaving correctly, e.g. that it is appending the revisions correctly and not overwriting existing revisions.

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 hashchain 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.

This concludes our rough overview of ProtonKT. Next, we will look at each of the protocol's components in more detail.

3.2 VKD Labels: Email Addresses

Recall that a Verifiable Key Directory (VKD) is a dictionary of label-value pairs. [1] In this section and in the next, we specify the labels and values used in ProtonKT.

Informally, the labels are email addresses and the values are public keys. However, there are some subtleties. For labels, consider the following use cases that make email addresses complicated:

- *Address normalisation:* Email addresses can have different formats that all map to the same normalised 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 normalisation rules that are applied to a label before it interacts with the VKD algorithms. The normalisation rules

[1] We don't call them key-value pairs to avoid confusion with cryptographic keys.

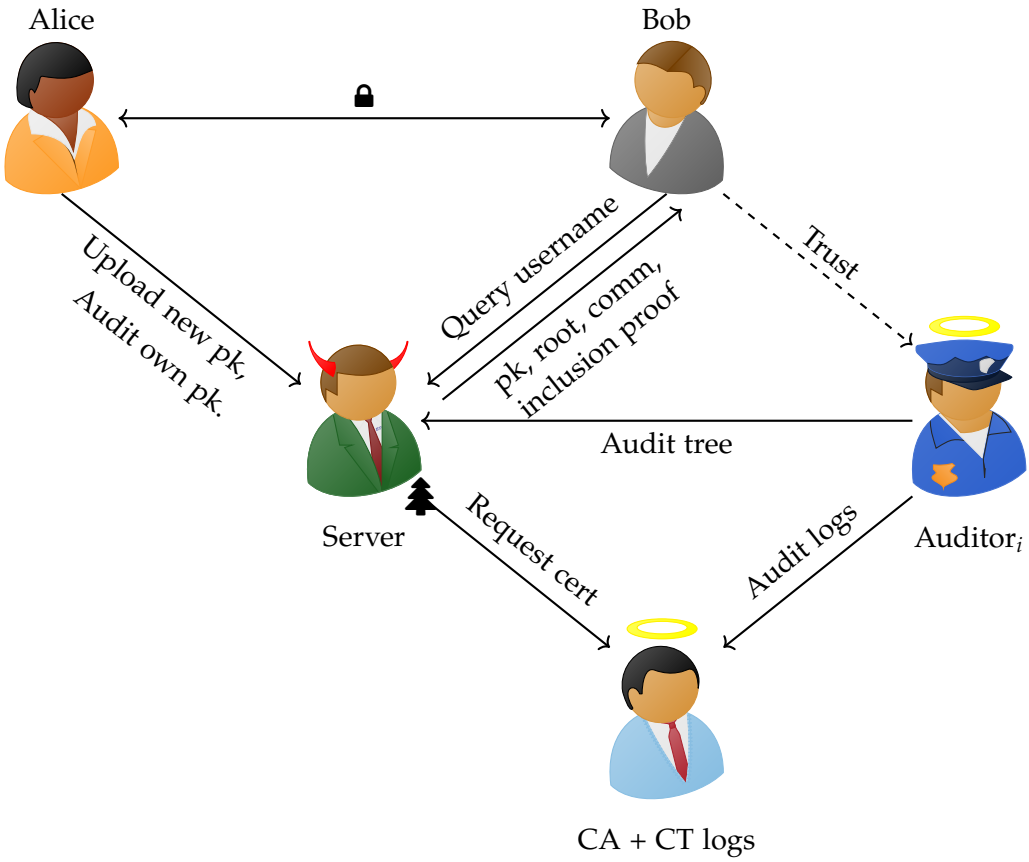

Figure 3.1: ProtonKT Overview

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 the VKD 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.

So formally speaking, we simply extend the VKD definition to take a new parameter *normRules*.

Proton has two normalisation 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 (.). [2]

[2] https://github.com/ProtonMail/protoncore_android/blob/85d316965201/key-transparency/domain/src/main/kotlin/me/proton/core/keytransparency/domain/usecase/NormalizeEmail.kt

- *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 using VKD.Query and VKD.VerifyQuery to look up an address (since from Bob's perspective it does not matter which account controls an address).

  Alice, however, needs to call VKD.KeyHistory and VKD.VerifyHistory for *all* of the addresses that are associated with her account. If she doesn't monitor an address, a malicious server could put a fake address into the VKD 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@gmail.com). An *Internal Address* is a Proton-managed email address (bob@proton.me or custom domains). An *External Address* (e.g. charlie@gmail.com) is an email address that is linked to a Proton account, but whose email is not handled by Proton Mail.[3]

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

  For External Addresses, the server may return email encryption keys that it found in the Web Key Directory (WKD) [13] (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 VKD labels are typed.

- *Custom domains:* In Proton, users can bring their own domain and delegate email handling to Proton. These are called *custom domains*. For example, the private email of the author is managed by Proton: `host -t mx thore.io`. This means that Internal Addresses can have arbitrary domains, and not just @proton.me. The only 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.

[3] This allows users to sign up for a Proton account with a GMail address.

- *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 obsolence 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 VKD label for the catch-all address, i.e. an @ followed by the domain name.

3.3 VKD Values

The Verifiable Key Directory (VKD) 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 the VKD for a label, there are three possible outcomes: *absence*, *inclusion*, and *obsolence*. 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 obsolence, the value is a special *ObsolenceToken*. 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: [4]

```
struct {
    boolean primary;
    uint64 flags;
    string fingerprint; // hex-encoded SHA-1/SHA-256 hash of the public key
```

[4] https://github.com/ProtonMail/WebClients/blob/3ae32a9/packages/shared/lib/interfaces/SignedKeyList.ts

```
    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. [5] It also has a primary bit and some flags. See section A.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 Obsolence 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 organisation on Proton can disable an email address, e.g. when an employee leaves.

[5] 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.

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. [6] 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 recognise this empty *SKL.data* value. For privacy, we want that an auditor cannot distinguish between active and disabled addresses. Thus instead of an empty data field, we insert an *ObsolenceToken*. It is defined as:

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

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 $h(ObsolenceToken)$, 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, Obsolence

In reality, when a client queries the server for a label, the server returns more than just an SKL or an Obsolence Token. It also returns additional metadata, for example

[6] 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.

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/obsolence).

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

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

We can formalise 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/obsolence), 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 from the VKD primitive:

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

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

- *revision* should be set. (It is not part of val, as the VKD definition treats it separately.)
- *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 fulfils the transparency requirement: if the server fails to include the key, the client has evidence of server misbehaviour. (This is local evidence. That is, the client cannot convince other clients of the server's misbehaviour. 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. [7]

$maxEpochId$ 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). [8]

3.5.1 Leaf indices

Naïve approach A core difference between CT and KT (as proposed by CONIKS) is that in KT every label has a *unique index*, whereas in CT a domain's certificate can be logged at any index. This makes queries very efficient (and more private), because clients have to check only a single leaf. A first naïve approach is to map a label label to a fixed length by hashing it, and then using the binary encoding of $h(\mathsf{label})$ as the unique index (up to hash collision). Then the output length of the hash function defines the depth of the tree.

$$idx = h(\mathsf{label})$$

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

[8] *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.

Privacy with VRFs CONIKS proposes *private indices* so that proofs don't reveal anything about other labels that are included in the tree. (Recall that from the co-path from leaf to root one can deduce which neighbouring leaves are present or absent). [21] Privacy is achieved by instead using a VRF output as the index:

$$idx = VRF.hash_{\mathsf{sk}}(\mathsf{label})$$

Recall from section 2.1 that VRF hashes can be publicly verified using pk and privately computed using sk. For brevity we omit the proof π here.

Revisions A problem in CONIKS is that every epoch has a new tree, forcing users to audit every single epoch. ProtonKT, inspired by SEEMless, proposes to use a single append-only tree and revisions to address this. Recall that SEEMless uses the following indices:

$$idx = VRF.hash_{\mathsf{sk}}(\mathsf{label} \mid\mid \mathsf{rev})$$

The advantage of this is that the auditors (who see the new leaves as part of the update proof) learn nothing about which labels were updated. Auditors cannot distinguish between a new label insertion and a label update. The disadvantage is that users need to do extra work to ensure that the revision that the server serves them really is the latest one. This is achieved using markers [7, Section 4].

Proton, however, calculates indices as follows:

$$idx = VRF.hash_{\mathsf{sk}}(\mathsf{label}) \mid\mid \mathsf{rev}$$

With this construction each label (each email address) has a unique subtree (defined by the trailing index bits of the revision, see subsection 3.5.2). This allows for the tracing attack described by SEEMless. However, Proton considers this acceptable leakage. Because Proton runs a public, non-access-controlled key server, an attacker can already trace key changes through repeated querying.

This construction also has some advantages: (1) Auditors can ensure that new revisions are always incrementally increasing. (2) Auditors can ensure that old revisions are correctly deleted. (3) It is easy to prove that a revision is the latest one. In comparison, Parakeet needs to be much more intricate and puts more work on clients that in ProtonKT auditors can do. For example, we will see in section 3.8 that ProtonKT can support deletions without requiring tombstoning like Parakeet.

Summary ProtonKT instantiates VRF with ECVRF-EDWARDS25519-SHA512-TAI [12][9], 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. Each bit of the 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).

[9] As of November 2023, Proton uses Draft 10 and not the final RFC 9381.

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 VKD value. If the VKD value is present, the leaf hash is computed as: [10]

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

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

$$h(h(ObsolenceToken) \mid\mid minEpochId)$$

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 *ObsolenceToken* 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). See also the VKD Values description in section 3.3, and see the discussion on type separation in section 5.2.

Overall, this leaf hashing commits the VKD values *SKL.data*, *ObsolenceToken*, 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.

[10] https://github.com/ProtonMail/pm-key-transparency-go-client/blob/a09cc15/verify_proof.go#L101 and https://github.com/ProtonMail/WebClients/blob/fab01b1d12cd5f9f7d1838f6464d155dd0b8c158/packages/key-transparency/lib/verification/verifyProofs.ts#L147

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 VKD 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 VKD 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

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

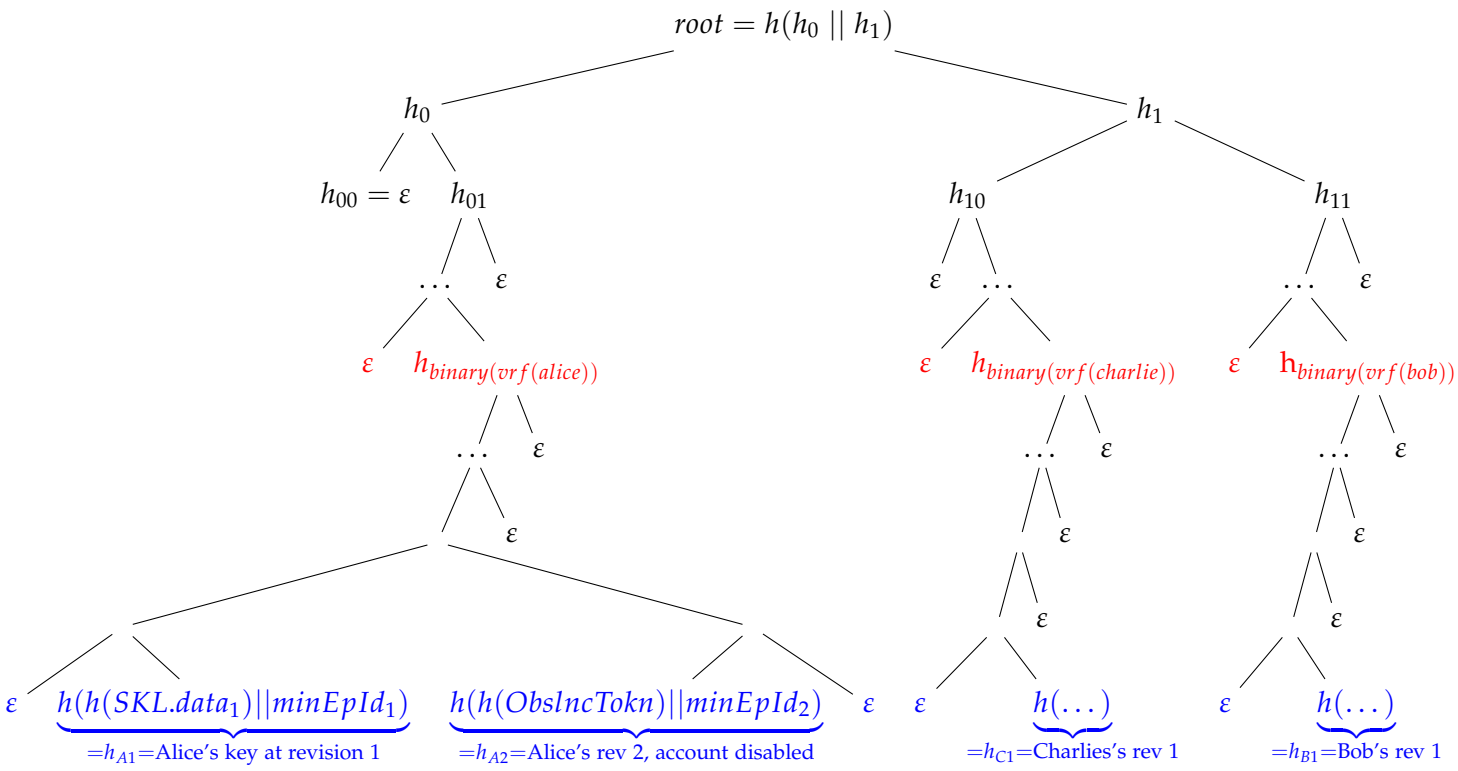

Figure 3.2: Tree structure example. Leaves of the label subtree (at depth 224) are shown in red and leaves of the three revision subtrees (at depth 256) are in blue.

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. π_{copath} is the inclusion/absence proof. [11] It is a list of all the neighbours 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:

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

[11] Obsolence proofs are a variant of inclusion proofs. They prove that an ObsolenceToken is included in the tree.

2. Second, from the outcome type τ and the query outcome values val we can construct the leaf hash (as defined above). Inclusion and obsolence proofs *must* check that the *SKL.data* is JSON-encoded and the *ObsolenceToken* is hex-encoded.
3. Third, we (re-)compute the hash chain from the leaf to the root using the neighbours in π_{copath}. The bits in the leaf index determine whether a neighbour should be $hash_{left}$ (bit 1, path goes right) or $hash_{right}$ (bit 0, path goes left).
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 neighbours on the co-path, and we start hashing only when we hit the first non-null neighbour. (If we have a null/empty neighbour 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 obsolence 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 (neighbours 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 onwards (even if the first few neighbours 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 neighbours 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 hashchain 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.

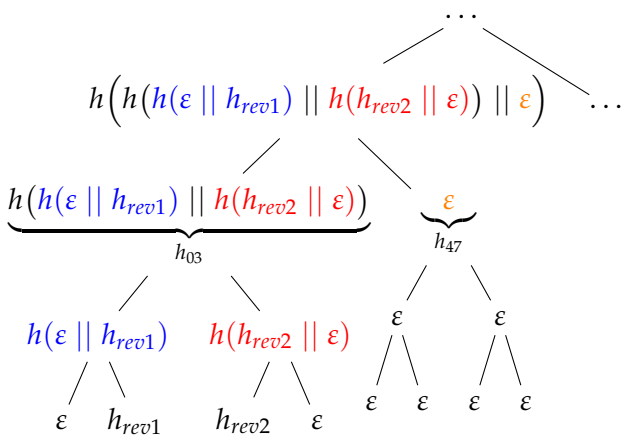

Figure 3.3: Tree structure example (incomplete). Only the lower left part of a single revision subtree is shown. $h_{rev1,2}$ are non-empty leaf hashes of the form $h_{revi} = h(h(SKL.data \,||\, minEpochId))$

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.

Previous approaches Recall that CONIKS achieves consistency by having third-party auditors who monitor the tree roots. CONIKS clients query multiple such auditors at random and ask them for their view of the root. Others, like Keybase [2] and EthIKS [6] write the tree root to a blockchain and use smart contracts, thus outsourcing the problem of global consistency to the blockchain network. SEEMless skips it and assumes a consistency mechanism is given. Parakeet proposes its own consistency protocol involving a quorum of witnesses. Apple uses iMessage itself as a transport mechanism for gossip: clients include the tree roots they see in "a small percentage of messages". [10]

Proton's approach Proton proposes another approach for achieving a consistent view: Instead of piggybacking on a blockchain, Proton 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).[12] `issuanceTime` is the epoch issuance Unix timestamp claimed by the server.[13] `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:

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

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. $n = 256$).

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

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

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

[13] 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.

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.

Comparison Below, we analyse the similarities and differences of the consistency mechanisms used by ProtonKT, Parakeet (WhatsApp), and Apple iMessage.

- *Setup*: Parakeet clients require a set of well-known witnesses. Proton clients require no knowledge about the auditors, but require a set of well-known CA and CT logs.

  For both, the identities and public keys of the witnesses or CAs need to be configured on the clients in advance. This requires a hardcoded setup or a trusted out-of-band channel to update the configuration.

  Apple requires no setup.

- *Trust Assumptions*: In Proton, clients need to trust: a small set of CAs (two or three) who issue the Certificate and the CT log operators who issue the SCTs.

  In Parakeet, clients need to trust that there is no quorum (majority) of colluding witnesses.

In Apple, because clients share root hashes via gossip, the iMessage servers cannot simply MITM a subset of a client's chats. It would either need to MITM none, or all chats (to rewrite all root hashes messages on the fly). Clients need to trust that such a major attack would be detected.

- *Communicating observed hashes*: In Parakeet information about the observed root hash flows from witnesses to clients via the witness certificate. Clients see how many and which witnesses have signed the certificate.

  Similarly, Apple's gossip is a direct exchange of observed root hashes.

  In Proton auditors and clients never exchange any information. Clients check the SCT and trust its promise that the certificate they see will be included in CT. And they trust that some auditor scans the CT logs and will raise a complaint if there are conflicting certificates. However, clients are unaware of whether there are any active auditors.

- *Issuing new epochs*: Parakeet requires enough witnesses to be online (namely $\geq \frac{2}{3}n + 1$ out of n) to issue a new epoch.

  Proton requires any one of the configured, well-known CAs and a log operator to be online.

  Apple has no external dependency and can issue new epochs any time.

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). [14]

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 the VKD as 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.

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

Previous approaches Parakeet [19] was the first to construct a KT scheme that supports deletion. They call the primitive *VKD with compaction (cVKD)*. See also subsection 1.3.3. However, Parakeet needs tombstoning: First, during tombstone epochs elements are marked as to-be-deleted. Then, clients must come online to check that their entries are correctly tombstoned. Finally, during deletion epochs all tombstoned entries are deleted (and auditors verify that nothing else is deleted).

Why tombstones? Parakeet requires this extra step because of the way how their tree is constructed. Their leaf indices are of the form $VRF.hash_{\mathsf{sk}}(\mathsf{label} \mid\mid \mathsf{rev})$. When they insert a new revision, they insert both label || rev and label || rev || stale. The stale marker allows queries to verify that a revision is the latest one (the stale entry must have an absence proof for the claimed latest rev). During Self Audit (VKD.VerifyHistory) clients check that their stale markers are set correctly.

What if there were no tombstones? Then a malicious server could delete label || rev || stale while keeping label || rev. Now the server is able to present an old revision to querying clients. Then before the next Self Audit the server simply inserts label || rev || stale again. Therefore for security, we require that the revision and the revision-stale-marker are always tombstoned together. This needs to be checked by the clients – auditors do not see into the VRF hashes.

This is why Parakeet needs tombstone epochs (after which clients check that the server behaved correctly, i.e. a local check on a few entries per client) as well as deletion epochs (after which auditors check that the server behaved correctly, i.e. a global check on the entire tree).

Proton's approach 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. [15]

Parakeet auditors check this, too.

- Values 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.

  This can be monitored by third-party auditors in ProtonKT, but not in Parakeet.

This also means that the ProtonKT server can delete old-enough items in any epoch (unlike Parakeet which may only delete in deletion epochs, because it needs to wait for clients to run Self Audits after tombstoning).

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.[16] 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.

Algorithm The client stores a $verifiedRev$, initialised to 0.[17] 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.

[15] 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.

[16] 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 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.

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

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. [18]
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. [19]
   - 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. For an alternative approach without a dedicated absence proof, see section 5.4.

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.

[18] 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.)

[19] 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.

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. [20]

Note that even though the ObsolenceToken contains a timestamp, we need not it during Self Audit. Unlike the signature timestamp, the ObsolenceToken'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.

Note that even if there are no Self Audits, ProtonKT is designed to still ensure that queries are consistent and the server cannot behave arbitrarily.

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 organisation 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 recognise.

3.10 Promise Audit

Sometimes the KT server makes a promise to include a value. With a Promise Audit clients verify that the server fulfils 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:[21]

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

[20] Sidenote: 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.

[21] Depending on τ_i, either SKL_data_i or $ObsolenceToken_i$ 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 misbehaviour and a warning is raised. If a different value is included at *expectedRev*, this is also misbehaviour.

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 organisations 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 (like in CONIKS) 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 naïve 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 [22]).

3.12 ProtonKT Subprotocols

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

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

Subprotocols Recall from section 2.2 that the VKD primitive defined by SEEMless consists of seven algorithms: VKD.Publish, VKD.VerifyUpdate, VKD.Audit, VKD.Query, VKD.VerifyQuery, VKD.KeyHistory, VKD.VerifyHistory.

ProtonKT differs slightly:

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.

A major difference between ProtonKT and the VKD definition is that ProtonKT does not have dedicated "Verify"-algorithms. Verification is an internal part of querying and auditing.

Below, we define these subprotocols in more detail. We describe what they do, and we relate them to the VKD primitive.

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

  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.

  In the standard VKD definition, this happens implicitly: VKD.Publish is assumed to have obtained a set of updates from somewhere. We state this explicitly, as an algorithm that runs and exits independently of ProtonKT.Publish.

  In addition, this allows for extensibility: for example, ProtonKT.RequestInsertion could be modified to return a signed promise of inclusion, similar to SCTs in CT or SMTs in iMessage's KT.

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

  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 $cert_t$).

  Corresponds to VKD.Publish.

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

  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.

SEEMless' VKD does not have dedicated algorithms for querying epochs because SEEMless assumes some gossip algorithm exists. Parakeet defines its own consistency algorithm WitnessAPI, which corresponds to ProtonKT.QueryEpoch. The VKD algorithms in Parakeet then internally call their WitnessAPI.

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

  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, obsolence); in the REST API it is called `Type`. val (called `SKL` in the API) contains other necessary fields (depending on τ) such as: $SKL.data$, $ObsolenceToken$, $minEpochId$ (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.

  Corresponds to VKD.Query and VKD.VerifyQuery.

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

  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.

  Corresponds to VKD.KeyHistory and VKD.VerifyHistory.

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

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

Has no correspondence because the VKD primitive has no notion of promised inclusions.

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

  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.

  Corresponds to VKD.Audit and VKD.VerifyUpdate.

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.

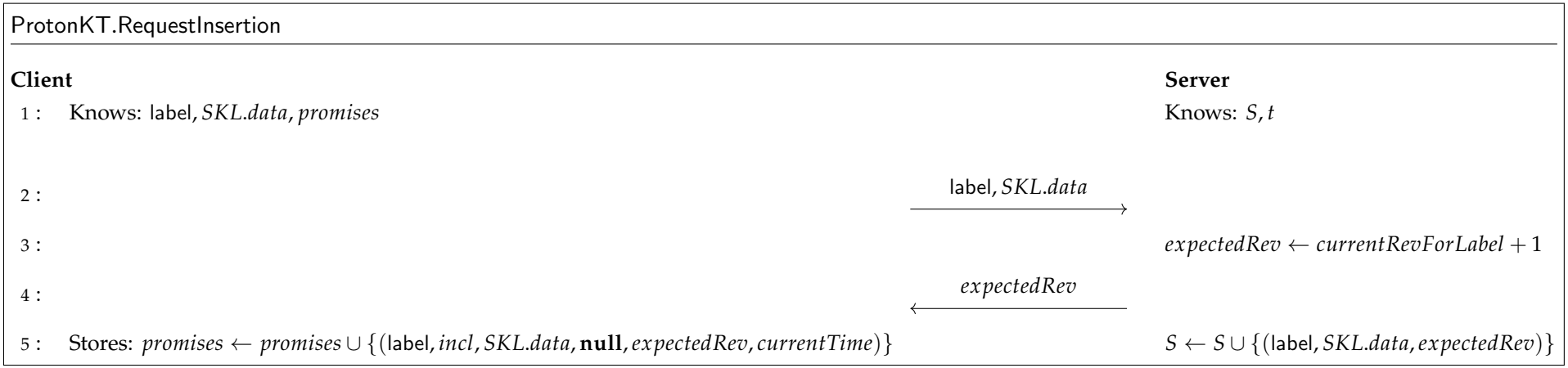

Figure 3.4: ProtonKT.RequestInsertion Sequence Diagram

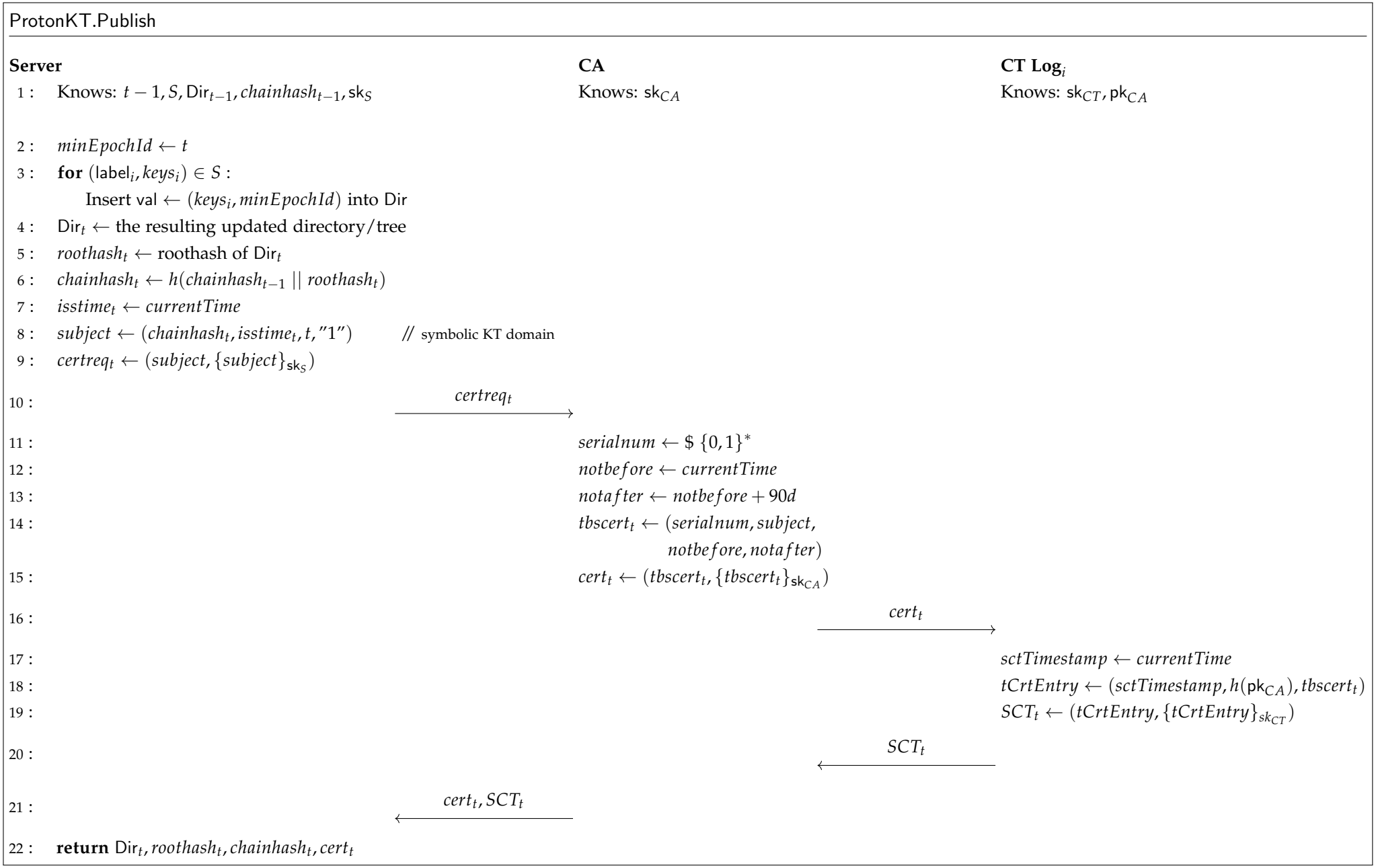

Figure 3.5: ProtonKT.Publish Sequence Diagram

ProtonKT.QueryEpoch

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

Figure 3.6: ProtonKT.QueryEpoch Sequence Diagram

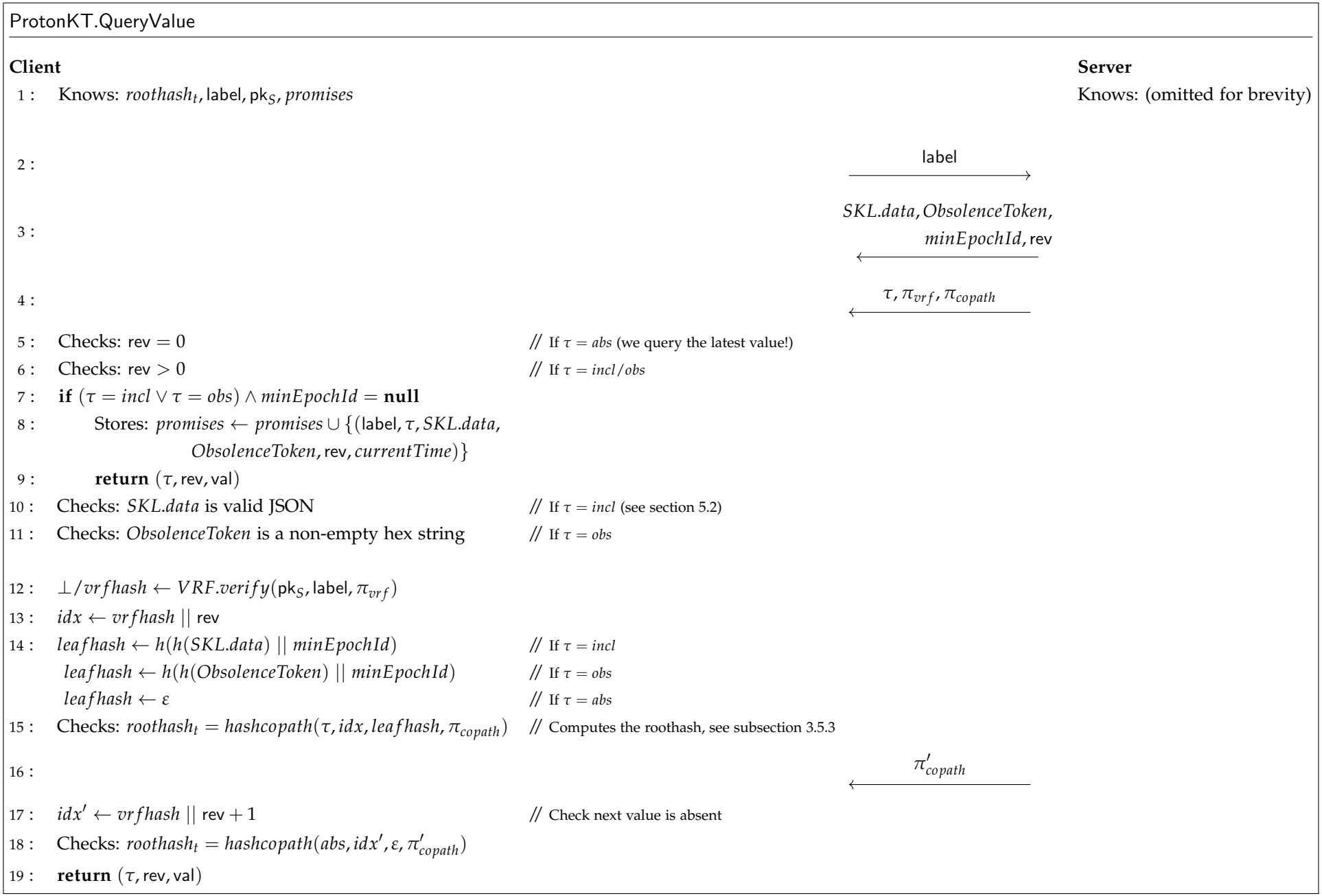

Figure 3.7: ProtonKT.QueryValue Sequence Diagram

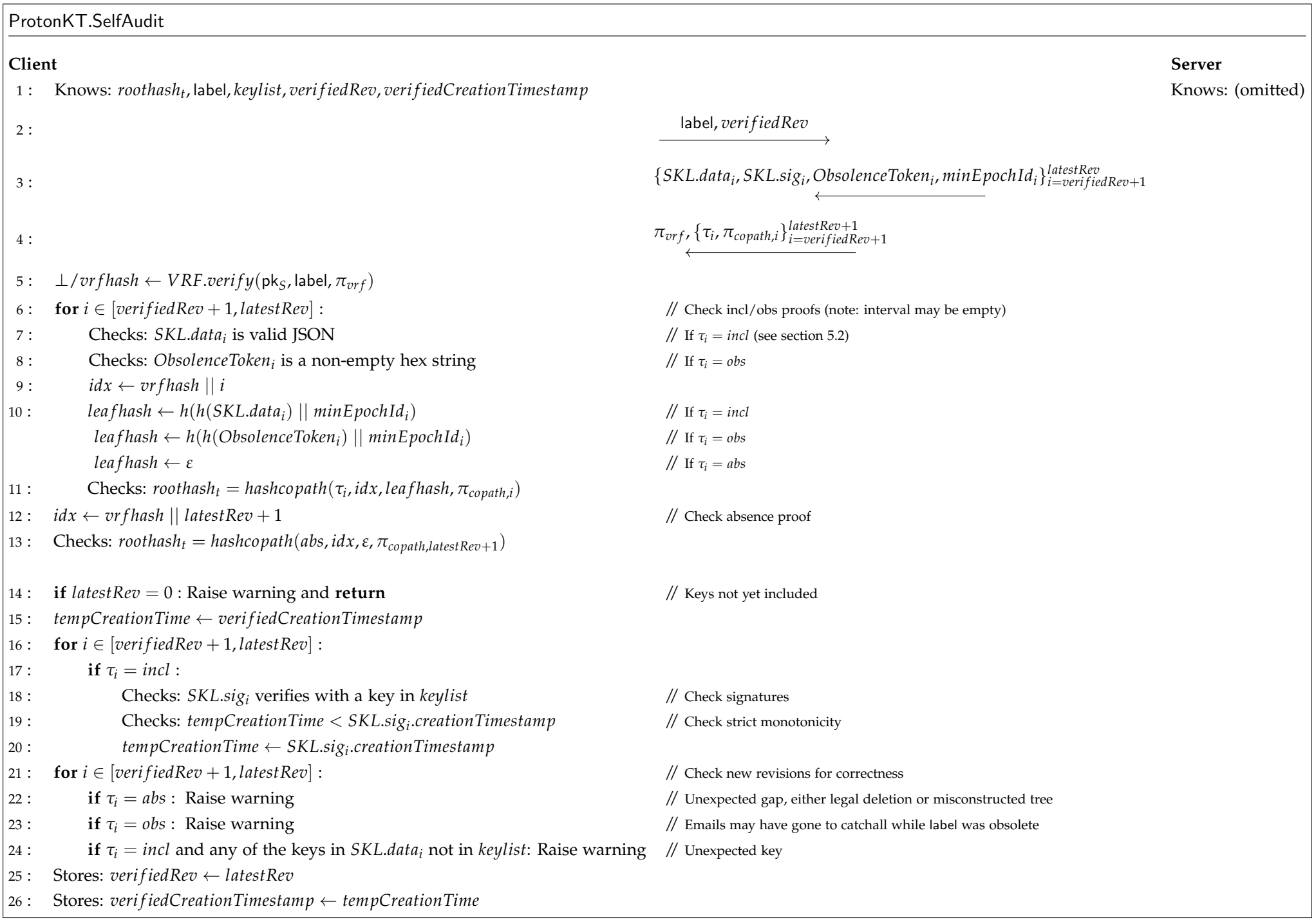

Figure 3.8: ProtonKT.SelfAudit Sequence Diagram

ProtonKT.PromiseAudit

	Client		**Server**
1 :	Knows: $roothash_t, promises$		Knows: (omitted for brevity)
2 :	**for** $(\mathsf{label}, \tau, SKL.data, ObsolenceToken, expectedRev, creationTime) \in promises$		
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 $\tau = incl$ (see section 5.2)	
12 :	Checks: $ObsolenceToken$ 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 $\tau = incl$	
	$leafhash \leftarrow h(h(ObsolenceToken) \mid\mid minEpochId)$	// 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 3.9: 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 $t \in [r, s]$:			
$roothash_t, issTime_t, chainhash_t \leftarrow \mathsf{ProtonKT.QueryEpoch}(t, chainhash_{t-1})$	$\leftarrow\!\!\cdots\!\!\rightarrow$		
Checks: $issTime_{t-1} < issTime_t$			
	$\xrightarrow{*.\{t\}.1.\texttt{keytransparency.ch}}$		
	$\xleftarrow{\{cert_i\}_{i,t}}$		
Checks: only certs with the expected full KT domain and no others			
	$\xrightarrow{t}$		
	$\xleftarrow{\underbrace{\{(\overbrace{vrfhash_i \mid\mid \mathsf{rev}_i}^{idx_i}, valhash_i, minEpochId_i)\}_i}_{insertions}}$		
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}			
	$\xleftarrow{\underbrace{\{(vrfhash_i \mid\mid \mathsf{rev}_i, valhash_i, minEpochId_i)\}_i}_{updates}}$		
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 3.10: ProtonKT.ExtAudit Sequence Diagram (Part 1)

ProtonKT.ExtAudit (continued)

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

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

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

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

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

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

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

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 3.11: ProtonKT.ExtAudit Sequence Diagram (Part 2)

3.13 Relating the Model and the Implementation

Scope Proton offers clients for web, Android, iOS, and desktop. Availability and feature completeness differ across the products (Mail, Calendar, Drive, Pass).[23] KT is an interesting project because it touches a lot of code. Being a suite of end-to-end-encrypted services, public keys are used everywhere: from email encryption, to receiving calendar invites, and sharing password vaults. Every time a contact's public key is used, it must be verified against KT. This makes integrating KT a big undertaking.

In our work we only reviewed the web client, whose source code is available on GitHub.[24] While an Android[25] and a Go library[26] exist, they are both from a previous protocol iteration that differs significantly from the final scheme. Fully deploying ProtonKT across Android and iOS remains future work.

The server code is not open source, and thus we did not review it. However, this is a feature of KT: the server is untrusted, and security should not rely on whether or not the server behaves correctly. We did however interact with the REST-API using curl to test basic functionality. We found one bug where the server, when asked for the proof for revisions equal or higher than 2^{32}, would wrongly reply with the proof for revision $2^{32} - 1$ instead of with an out-of-bounds error.

As we will see in the security analysis, the external auditors play a crucial role in ensuring that the server constructs the tree correctly. Unfortunately, the auditor implementation is still work-in-progress, so we did not review it.

Findings The web client code that we did review seems to match up with the protocol scheme defined in this thesis. There are additional error handling and reporting mechanisms that a real implementation necessitates. The implementation is also spread across different packages and files, and the naming in the code (having grown organically) differs from the naming in this thesis. This makes it harder to just put the message sequence diagrams and the source code side-by-side. A future rewrite should consider adhering to the now well-defined naming scheme.

The main difference between the web source code and our specification is that the implementation lumps the Self Audit and the Promise Audit together. Otherwise, our protocol description and the implementation match functionality-wise.

In section 5.5 we discuss a finding that would have allowed the server to cause promises to never be audited because the MMD could get ignored. This has since been fixed.

[23] VPN does not need KT because it is not looking up any other users' public keys.
[24] https://github.com/ProtonMail/WebClients/tree/f949ce269404/packages/key-transparency
[25] https://github.com/ProtonMail/protoncore_android/tree/0061b0ae37d6/key-transparency
[26] https://github.com/ProtonMail/pm-key-transparency-go-client

Chapter 4

Security Analysis

In this section, we analyse the security of ProtonKT. We first look at the privacy goals. Next, we define the security properties that we want ProtonKT to achieve. We also describe the threat 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. Finally, we formally model ProtonKT with Tamarin.

4.1 Privacy Properties

In this section we describe the privacy goals that ProtonKT has. To limit the scope of this thesis 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' organisation.

- Whether a revision that is present in the tree is active (valid SKL) or disabled (ObsolenceToken). Recall that both the SKL.data and the ObsolenceToken contain randomness (key fingerprints are hashes,[1] random part of ObsolenceToken). The auditor would need to compute the preimage of $h(SKL.data)$ and $h(ObsolenceToken)$ to distinguish them, which is infeasible.

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 neighbouring parts of the tree have changed. This allows for the tracing attack described by SEEMless (see section 1.3.2).

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 analyse 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 misbehaviour. 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 $r < p \leq t$,*

[1] We work in the random oracle model.

- *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 visualises the components that are mentioned in the property, as well as highlighting the timepoints at which they can occur.

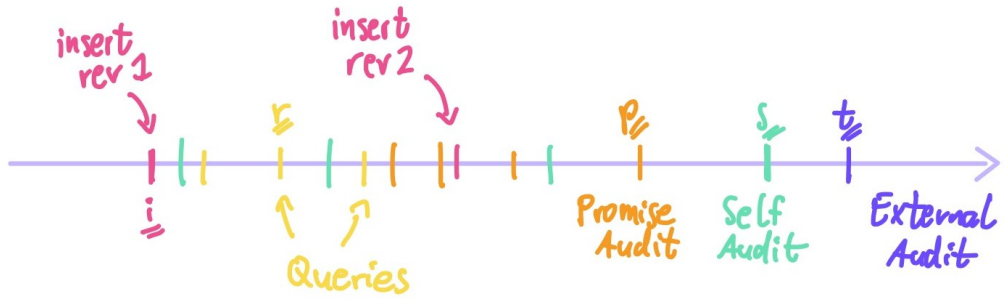

Figure 4.1: Visualisation for Query-to-SelfAudit Consistency
(showing also other possible locations of queries, Self Audits, and Promise Audits)

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 travelling 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 traveller'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 [7, 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 [19, Definition 4, Appendix B], we have added the Promise Audits but do not have tombstones.

Sidenote: "consistency", an overloaded term The term "consistency" is heavily used in transparency protocols, but often with slightly different semantics. Thus for clarity we call our properties "Query-to-SelfAudit Consistency" and not just "consistency".

CONIKS [21] talks about "consistency of name-to-key bindings", which can be thought of as "query-to-query consistency" (since CONIKS does not have explicit a Self Audits algorithm).

Parakeet defines "consistency" of the root hashes, called "commitments" in Parakeet's terminology [19, Section II.A]. That is, Parakeet looks at "root hash consistency" not "query consistency". This is an important distinction: if two clients agree on which tree they see, they may still disagree on the query outcome, e.g. due to a type confusion when interpreting the leaf value (see section 5.2).

In Certificate Transparency the term "consistency" is used for append-only-ness of the tree: "Merkle consistency proofs prove the append-only property of the tree." [17, Section 2.1.4.]. That is, a CT "consistency proof" proves that the tree at epoch t and the tree at epoch $t+1$ are "consistent" with each other. For clarity, we instead use the terms "update proof" and "update consistency" when talking about the tree construction being "consistent" across epochs.

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 Manual Analysis

In this section we provide a manual 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.

[2]For ECVRF this should hold if the curve and group generator are from a trusted source [12, Section 7]. Since ProtonKT hardcodes the parameters of ECVRF-EDWARDS25519-SHA512-TAI, the parameters can easily be checked.

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]

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

[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.

[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.

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 $latestRev \geq \mathsf{rev}$. 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 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 obsolence ProtonKT.QueryValue enforces that $\mathsf{rev} > 0$. We must also have $\tau^A = abs$ because the Self Audit is initialised 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 analyse 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 analyse 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):

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

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:
$leafhash_{idx}^Q = \varepsilon \neq h(h(SKL.data/ObsolenceToken) \mid\mid minEpochId) = leafhash_{idx}^A$
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
$\mathsf{val}_{obs} = \{ObsolenceToken, minEpochId\}$.

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 initialises $verifiedRev = 0$ 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 $data = ObsolenceToken$, and Q interprets it as an SKL data and A as an ObsolenceToken. However, this is a contradiction to the fact that the algorithms checks that the SKL data is JSON-encoded and that ObsolenceToken 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, (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 analyse 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 ($|time(P) - time(R)| > \mathsf{MMD}$). 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 obsolence 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 (normalised) 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 recognise 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.

4.5 Limitations of the Security Properties and of KT

In this section we analyse the limitations that our security property has. We also discuss limitations of KT in general, and where KT does not improve over a key directory without transparency.

Property Assumptions Query-to-SelfAudit Consistency has a lot of assumptions. They constrain the cases in which we can expect security from this property. Conversely, whenever the assumptions are not met, we cannot expect the property to hold, and we cannot expect any security. A trivial case is the External Audit: if there is none, the server can have misconstructed the tree in many different ways that attack consistency but is not noticed by clients.

No Self Audits, No Security Self Audit is more interesting: what if there is none? Our initial idea was that even in the absence of Self Audit it might make sense to have a property we would have called *Query-to-Query Consistency*. Intuitively: all clients

who query label and obtain outcome $O = (\tau, \mathsf{rev}, \mathsf{val})$ see the same outcome O for a fixed revision rev.

It turns out that such a property is not very useful, because the server can still return arbitrary keys as it pleases if there are no Self Audits.

For example, the server can still isolate a single user Charlie. When Charlie queries Alice's key, the server returns the MITM key. For all other queries the server returns Alice's correct key. If Alice does not Self Audit, then this MITM attack works even if Query-to-Query Consistency holds! It proceeds as follows: let Alice's correct key be at revision rev. When Charlie queries, the server returns $(\mathsf{rev}+1, \mathsf{pk}_{MITM})$ as a promise, and inserts it in the next epoch. For all other queries that happen just slightly after Charlie's query the server returns $(\mathsf{rev}+2, \mathsf{pk}_{Alice})$ as a promise, and inserts it as well. The server will never reply with $\mathsf{rev}+1$ to anybody expect to Charlie. Every time Charlie queries, the server can do this. This works because clients always query the latest value (to prevent the server from showing outdated views if Alice legitimately updates her key). Thus even if this potential Query-to-Query Consistency holds, the server can still execute practical attacks. [5]

Alternatively, the server can also insert a malicious key and MITM everyone (not just Charlie).

Therefore we do need Self Audits and their correctness checks for any practical security.

Non-Proton Addresses In practice this 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. [6]

This is problematic because normally emails to Non-Proton Addresses would be unencrypted since there are no keys for them. When the server inserts a MITM key, the client will use it. Then the UI would make the user believe that it is using "end-to-end encryption" whereas previously the UI would show that it is an unexpected email. This lulls the user into a false sense of security.

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.

Catch-All Addresses Another case where missing Self Audits are an issue are catch-all addresses. The first question is: who audits the keys of the catch-all address (identified by VKD label `@example.com`, i.e. only the domain part). This should be done by

[5] Note that in this example, the server is rolling back Alice's key. The Self Audit, if it ran, would detect this when checking the signature timestamps and would raise a warning.

[6] To defend against this, users should use the Trusted Keys feature (see section A.1) to pin keys of Non-Proton Addresses.

the domain owner. If there is no Self Audit, the catch-all address is similarly vulnerable as normal VKD labels.

The second problem is: the server can insert malicious keys for non-existent addresses. Since these addresses are not owned by any user, they won't be Self Audited. Then, instead emails to `nonexistent@example.com` would not be encrypted with the catch-all keys, but with the address-specific malicious keys, allowing the server to MITM the email.

This is a problem that is currently unsolved in ProtonKT. Note that this issue also exists in non-KT-enabled systems.

4.6 Formal Model with Tamarin

In this section we develop a formal model of ProtonKT using the Tamarin Prover. The model files can be found on GitHub.[7] We use Tamarin version 1.8.0. We first describe the model. Afterwards we discuss our attempts at proving the security properties.

4.6.1 The Model

Our model is spread over multiple Tamarin files for clarity. It implements the main algorithms: ProtonKT.RequestInsertion, ProtonKT.QueryEpoch, ProtonKT.QueryValue, ProtonKT.SelfAudit, ProtonKT.ExtAudit. The model matches the specification given in chapter 3 in spirit. However, as with any model, it makes simplifications and abstractions typical for a symbolic setting.

For example, when querying a value we don't model the client sending label out to the network. Since in Tamarin `$label` is already a public term the adversary learns nothing new. Instead, we directly model the client "unpromptedly" receiving the value and the proofs from the network.

We don't model the VRF. In our first model (see below) we have no explicit tree structure because we abstract the tree as facts. Therefore we have no need for bit-string-based indices; instead we match leaf facts directly based on their labels, which are terms. In our second model we use a restriction on subterms to ensure that each label has a unique index in the tree (`restriction LabelNotInHeadOrTail`). [8]

We also don't model the KT server role in Tamarin. In our adversary model (see section 4.3) we gave the adversary the power to control both the network and the server. Thus we can leave it up to Tamarin's constraint solver to construct the messages that the client and auditor roles receive from the network.

We model the CA/CT ecosystem as persistent facts: `!CT(%epoch_id, chainhash)`. "Verifying" a certificate and an SCT in ProtonKT.QueryEpoch then boils down to using

[7] https://github.com/thgoebel/master-thesis-key-trans/

[8] Additionally, recall that the VRF contributes only to privacy by randomising the indices. For the security property that we want to analyse predictable indices are sufficient as long as they are unique.

the persistent fact in the premise of a rule. We also give the (network) adversary the power to insert any entry into the CT log:

```
  rule CT_Insert:
  [ In(<epoch_id, chainhash>) ]
--[ CtInsertChainhash(epoch_id, chainhash) ]->
[ !CT(epoch_id, chainhash) ]
```

We model revisions and epoch ids as natural numbers (`%rev, %epochid`). This is indicated by the prefixed percentage sign. Natural numbers are a new feature that were recently introduced in Tamarin [9], building upon subterms.

The core challenge in modelling ProtonKT is modelling the tree. We tried two different approaches:

Model 1: Tree as Persistent Facts In our first iteration we model the tree as persistent facts: `!TreeLeaf($label, val, %rev, %min_epoch_id)`

On one hand, this is a heavy abstraction, the entire tree structure is not modelled. We also cannot have an explicit roothash or chainhash, nor log them to CT.

On the other hand, this abstraction also has advantages:

- First, since persistent facts can be consumed arbitrarily often, they cannot be "deleted". Thus we trivially have an append-only tree. (This limits us to modelling ProtonKT without the deletion feature.)
- Second, we don't need complicated copath hashing to model proofs. Using the `!TreeLeaf` fact in the premise of a rule already models the verification of the proof: the hashing of the neighbours, and the comparison against the root hash.
- Third, since there is only a single tree, the adversary cannot equivocate on the tree and create split-views of the tree.

Model 2: Tree as Terms In a second iteration we want to approximate the structure of the binary Merkle hash tree more realistically. We model the root hash as follows:
`roothash = h( <'head', h(ut_0), h(ut_1), ..., h(ut_n), 'tail'> )`
`ut_i` are the revision subtrees, also known as user subtrees. Thus we don't have a full binary tree, but a hash tree with a single root and many parallel subtrees. `head` and `tail` are needed for pattern matching.
The user/revision subtree is modelled as follows:
`ut = < $label, <%n, val_n>, ..., <%3, val_3>, <%2, 'empty'>, 'rest' >` [9]
The revision subtree is flat, too, and not binary. Thus overall our hash tree has three levels: root, label subtree leaves, revision subtree leaves.

[9] The fact that it ends at revision 2 not 1 is an artefact of non-ideal modelling.

Recall that we don't have a server role. However, in Model 2, the External Audit (specifically the append-only part) reconstructs the tree to check it against the claimed root hash logged in CT.

For example, this is the Append Audit's rule to insert a new value into the tree:

```
rule AppendAudit_Insert:
    let roothash_old = h( <head, tail> )
        new_ut = <$label, <%1 %+ %1, 'empty'>, 'rest' >
        roothash_new = h( <head, <h(new_ut), tail>> )
        chainhash_new = h(<chainhash_old, roothash_new>)
    in
    [ St_AppendAuditor(id, %epoch_id, chainhash_old, roothash_old)
    // head + tail let the adversary choose the index at which to insert
    , In(head)
    , In(tail)
    , !CT(%epoch_id %+ %1, chainhash_new)
    ]
    --[
      LabelNotInHeadOrTail($label, head, tail)
    ]->
    [ St_AppendAuditor(id, %epoch_id %+ %1, chainhash_new, roothash_new) ]

// limit that $label is not already in head nor tail (i.e. has a unique position)
restriction LabelNotInHeadOrTail:
    "All label head tail #i .
        LabelNotInHeadOrTail(label, head, tail)@i
        ==>
          not (label << head)
        & not (label << tail)
    "
```

The auditor already has the tree in form of the `roothash_old` contained in its state fact `St_AppendAuditor`. Using pattern matching we can separate it into a `head` and a `tail`, thus defining the position at which the new label should be inserted. We let the adversary chose `head` and `tail` by receiving them from the network. Then we use the restriction (and subterm reasoning!) to enforce that the label does not already appear in the tree, i.e. that it will have a unique position and is not a duplicate. Because `$label` is a public term we can use it without it appearing in one of the premise's facts. Also note the usage of the `!CT` persistent fact, which models the verification of the new chainhash against the CT logs.

The Append Audit rule for updating values for labels is very similar. The difference is that instead of creating a new user subtree `new_ut`, it pattern matches an existing user subtree and prepends a new (revision, value) tuple.

In Model 2 querying and Self Audit explicitly receive the root hash (i.e. the tree). They, too, use pattern matching on the root hash and on the user subtree to model inclusion proofs.

By not having server rules that build the tree, we give the adversary (played by Tamarin's constraint solver) the power to construct it. Queries and Self Audits only look at the revision subtree that they are interested in. Theoretically the rest of the tree *could* be wrongly constructed. Here the External Auditor's Append Audit comes in: it, too, receives the tree from the network. It verifies the tree by reconstructing it according to the correct rules and comparing the result against the provided root hash.

Model Limitations Our Tamarin models still have the following limitations:

- The SKL is highly simplified. *SKL.data* is only a single key not a list of keys. There is no primary bit or other key flags. There is no SKL signature.
- We don't model absence proofs. Queries can only return values that are included.
- We don't model obsolence and ObsolenceTokens.
- Deletions are not modelled. Thus the tree is append-only and growing forever.
- Promises and Promise Audits are not modelled. Thus Querying can only return values that are already in the tree.
- We don't model time. The model doesn't have a concept of DeletionParam or MMD. This is also why the CT facts don't need to contain the epoch *issuanceTime*.
- The VRF is not modelled. See above.

4.6.2 Attempts at Proving Security Properties

For both model versions we wrote basic executability lemmas (`lemma Executability_XYZ`). These serve as sanity checks that all subprotocols are executable by Tamarin. If a protocol is not be executable, security properties trivially hold: For example, if a client never sends a secret because the corresponding rule is not executable (i.e. not reachable) then the adversary trivially never learns the secret.

Executability lemmas are relatively straight forward because Tamarin only needs to find a single trace that satisfies the formula, and the property is usually easy to reach.

For security lemmas the formula needs to hold for all traces. Tamarin negates it and tries to find a trace satisfying the negation. Like our manual analysis, Tamarin starts from an attack state and reasons backwards to see how it could have been reached. Ideally this ends in a contradiction, i.e. there is no chain of applying the multiset rewriting rules that lead to the attack state.

We unsuccessfully attempted to prove Query-to-SelfAudit Consistency in Model 2. Because Model 1 was originally intended only as a stepping stone to Model 2, we did not

have enough time to go back to it and try and prove Query-to-SelfAudit Consistency in Model 1 once the challenges in Model 2 became apparent.

Let us look at the challenges in more detail.

Situation: Loops Both the Self Audit and the Append Audit have loops that iterate over revisions. Tamarin's backward search cannot solve general lemmas in looping protocols. Therefore, we need to use induction.

Induction First, we need to understand how induction works in Tamarin. Consider the following example, taken from Chapter 11, "Advanced Features" of the Tamarin manual [23].

```
rule start:
  [ Fr(x) ]
--[ Start(x) ]->
  [ A(x) ]

rule repeat:
  [ A(x) ]
--[ Loop(x) ]->
  [ A(x) ]

lemma AlwaysStarts [use_induction]:
  "All x #i. Loop(x) @i ==> Ex #j. Start(x) @j"
```

The annotation `[use_induction]` tells Tamarin to try and proof this lemma using induction.

Without induction, Tamarin first negates the formula, ending up with:
`"Ex x #i. ( Loop(x) @i & not (Ex #j. Start(x) @j) )"`
If Tamarin can find such a trace, it would be a counterexample for the lemma. In its backwards search, Tamarin starts from an `A(x)` fact (that it knows exists because of the `Loop` action fact). Then Tamarin's search will forever loop trying to apply the `repeat` rule and ending the tree while avoiding the `start` rule.

With induction, Tamarin rewrites the formula to obtain the following induction hypothesis:
`"All x #i. Loop(x) @i ==> (Ex #j. Start(x) @j) | last(#i)"`
In other words, whenever we have a `Loop` action fact either the property holds, or we are in the `last(#i)` timepoint of the trace. This entire induction hypothesis then becomes an additional constraint, with which Tamarin's constraint solver is then (usually) able to prove the original lemma.

Connecting Chain Hashes across Epochs Recall from our manual analysis of Query-to-SelfAudit Consistency that we have the case $r \neq s$, i.e. the query and the Self Audit happen at different epochs. In this case we somehow need to relate the trees at these two epochs. At first sight, the chain hashes make this easy:

$$chainhash_t = h(chainhash_{t-1}||roothash_t)$$

It is clear that $chainhash_{t-1}$ should be a subterm of $chainhash_t$:

$$chainhash_{t-1} \sqsubset chainhash_t$$

Therefore, to connect the two trees a lemma can look as follows (*Ch* is the action fact in the trace, *ch* is the term representing the chain hash):
`"All ch1 ch2 #i #j. Ch(ch1)@i & Ch(ch2)@j ==> ch1 << ch2"`
The induction hypothesis becomes:
`"All ch1 ch2 #i #j. Ch(ch1)@i & Ch(ch2)@j ==> ch1 << ch2 | last(#i) | last(#j)"`

Problem on Case Splits During its backward search, Tamarin does case splits. In the case of the `last(#i)` timepoint, the trace can be completed with some different $Ch(ch3)$ with $ch3 \sqsubset ch2$. $ch3$ may still be a valid subterm of $ch2$, i.e. there is no equivocation, but it is not the subterm $ch1$ we are interested in. Thus the proof gets stuck, and we cannot prove our lemma to relate the trees.

Easy-but-Impossible Solution An easy solution to this would be an auxiliary lemma such as:
`"All ch1 ch2 #j. Ch(ch2)@j & ch1 << ch2 ==> Ex #i. Ch(ch1)@i"`
Intuitively, we want to reason over all subterms $ch1$ of $ch2$, and show that they must have previously appeared in the trace. This would allow us to connect the chain hashes of querying and Self Audit together.

However, this formula is not *guarded* because $ch1$ does not appear in any fact on the left hand side. We cannot have such a formula in Tamarin, it is not valid syntax. This is were our proof attempts are stuck.

Overall, we unfortunately were not able to find a proof that Query-to-SelfAudit Consistency holds in our Tamarin model of ProtonKT. This is due to the outlined challenges with induction, as well as to time constraints.

Benefits of the Formal Modelling Process Nevertheless, the process of formally modelling ProtonKT and trying to find formal proofs already provided us with important insights. Going through this process enabled us to better understand the different parts of the protocol. Sometimes questions came up during modelling which pointed to gaps in our understanding and/or our pen-and-paper description (the message sequence diagrams).

For example, when handling the base case for the Append Audit (Is the first revision `%1` or is it `%1 %+ %1`?) we noticed that we wrongly assumed that ProtonKT inserts new entries at the left-most leaf, which would be revision 0. When we checked the web client code we found that revision 0 is always empty and the first revision is inserted as revision 1.

Chapter 5

Recommendations

In this section we describe recommendations for improving the ProtonKT scheme.

5.1 Discrediting the KT Server through Fake Root Commitments

The goal of any transparency system is to ensure that the server cannot misbehave in a way that is undetectable. If the server is misbehaving, this action should produce non-repudiable proof. If on the other hand the server is acting honestly, we might also like that no one can "defame" the server, i.e. make it look like the server is misbehaving when it isn't.

The authors of CONIKS discuss this problem in a separate blog post [20]. They focus on detecting the source of an inconsistency: Was it a protocol flaw? An implementation bug? Outside compromise of the server? A malicious employee? While they don't state it explicitly, they are trying to establish an accountability property [14].

This sort of "non-discreditability" was not among the original design goals for ProtonKT. Nevertheless, we think it is important (even if only from a business perspective), hence we bring it up as a recommendation.

Attack In turns out that a malicious CA can discredit an honest ProtonKT server. Recall from section 3.6 that the tree root hash is committed by requesting a certificate with a Subject Alternative Name (SAN) of the form:

```
hash[0:32].hash[32:64].timestamp.epochid.1.keytransparency.ch
```

A malicious CA could issue a certificate for the following SAN on its own accord:

```
hash'[0:32].hash'[32:64].timestamp.epochid.1.keytransparency.ch
```

with the same `epochid` but a different `hash'`. This certificate will never be presented by the server to clients, but it will be included in CT logs. To auditors scanning the CT log, this is indistinguishable from the server equivocating on the tree hash.

Importantly, this works because a certificate only binds an identity to a public key. It does not contain a statement signed by the private key that approves this binding. A Certificate Signing Request would be such a statement. However, the CSR is not included in the certificate, and the CA may delete the CSR after issuing the certificate.

Adversary Model In its original adversary model, Proton did not consider CAs. However, if we extend our adversary model to include malicious CAs then this attack on non-discreditability becomes possible.

On one hand, this issue is easy to exploit for a CA: it just has to issue a certificate. On the other hand, CAs are by nature under heavy public scrutiny and they, too, have a business risk at stake. This makes it less likely that a CA at its own volition would execute this attack.

However, there are attacks that trick CAs into misissuing certificates. One such example is circumventing DNS-based domain verification with BGP-hijacking [4]. (Let's Encrypt has mitigated this particular attack by using multiple vantage points for DNS verification.[1] It is unclear if the second CA used by Proton, ZeroSSL, has such mitigations in place.)

Recommended fix We can simply use signatures to prevent this attack. The server needs to prove that it requested the certificate. It can do this e.g. by signing the original target domain name, and including this signature as a second SAN domain in the certificate request:[2]

```
sig[0:32].sig[32:64].sig[64:96].sig[96:128].epochid.1.keytransparency.ch
```

More generally, the server needs to sign a statement containing the tree hash, and it needs to publicly and immutably commit to that signature (using CT, a blockchain, a bulletin board, etc.). Currently Proton's scheme only commits to the tree hash. The origin of the hash is not verifiable.

5.2 Make the Leaf Type Explicit

Recall that the leaf hash for included values is

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

[1] https://community.letsencrypt.org/t/validating-challenges-from-multiple-network-vantage-points/40955

[2] This assumes Ed25519 signatures (EdDSA using Curve25519), which have size 512 bits. One hexadecimal character encodes 4 bits, hence 128 characters.

and that for obsolent values it is

$$h\big(h(\mathit{ObsolenceToken}) \mid\mid \mathit{minEpochId}\big)$$

Note that the proof type (inclusion/obsolence) is *not* included in the leaf hash, and thus also not explicitly included in the tree.

This smells like a good target for a type confusion attack. That is, the key server might cause some clients to interpret the first value as an SKL, and other clients to interpret it as an ObsolenceToken. This would break Query-to-SelfAudit Consistency.

Impact However, this is *not* exploitable. Recall that ProtonKT requires clients to check[3] that the ObsolenceToken is a non-empty hexadecimal string. Thus an SKL can never be interpreted as an ObsolenceToken. And an ObsolenceToken cannot be interpreted as an SKL because it will not decode as valid JSON.

Nevertheless, using an encoding (hex vs. JSON) to separate types is not clean cryptography. It feels fragile. It is not immediately clear that the encoding has a security impact. When reviewing the source code, the check `isHexadecimal()` reads like a sanity check, not like a critical protocol component. In addition, encodings are usually not part of a symbolic model where we reason using terms. (One can manually constrain the symbolic model to account for the encoding of values, but this is not idiomatic.)

Recommended improvement The proof type should be explicitly bound to the value when committed in the tree. We recommend including the proof type in the leaf hash:

$$h\big(\ h(\text{"1"} \mid\mid \mathit{SignedKeyList.data}) \mid\mid \mathit{minEpochId}\big)$$

$$h\big(\ h(\text{"2"} \mid\mid \mathit{ObsolenceToken}) \mid\mid \mathit{minEpochId}\big)$$

Note that this does *not* leak the proof type to the auditors. Auditors only see $h(\text{"2"} \mid\mid \mathit{ObsolenceToken})$ and $\mathit{minEpochId}$. They cannot deconstruct ("look into") the hash output, they can only brute-force all possible values. Thus assuming that $\mathit{ObsolenceToken}$ cannot be guessed, the auditors cannot distinguish $h(\text{"1"} \mid\mid \mathit{SKL.data})$ and $h(\text{"2"} \mid\mid \mathit{ObsolenceToken})$ (like before).

This change also simplifies the argument in subsection 4.4.2. Currently Case 2.1.2.2.2.3 relies on checking the encoding. With our suggested change this instead reduces to collision resistance of the hash function.

[3] https://github.com/ProtonMail/WebClients/blob/94081ab/packages/key-transparency/lib/verification/verifyProofs.ts#L184

5.3 Easier Investigation of KT Warnings

Ideally, ProtonKT will happily run in the background. But what if there is a warning? This is a new UX challenge that is still open. As Kevin Lewi, who works on WhatsApp's Parakeet design, acknowledges in an interview: "We're also working on what the user interface of this will actually look like: what do users see when they wanna verify each other?" [24].

No export ability An important UX aspect is the following: what can the user do if KT does raise a warning? In this case, the user potentially has proof of server misbehaviour. Clearly, users would like to export such proof for backup and later analysis.

However, Proton's web UI does not yet offer such an export feature. It simply displays a warning: "Proton Mail failed to verify that the keys of this address are consistent with Key Transparency." There is only high-level user guidance. No detailed technical cause is given. Just like advanced users may want to view raw email headers, they may also want to view raw KT errors.

Limited ability to compare fingerprints A natural first step in case of a KT warning would be to fall back to manually comparing PGP key fingerprints out-of-band (just like Signal's and WhatsApp's Safety Number scanning).

When verifying Bob's key against KT fails, the web client does show Bob's full expected fingerprint and prompt Alice to compare it manually with Bob. Bob can find his own full key fingerprints in the web client settings.

However, users cannot easily compare fingerprints proactively when there are no KT errors. On web, the fingerprints of contacts are trimmed to only show the first 12 of the total 64 hex characters of the SHA-256 fingerprint (see Figure 5.1). (As a workaround, one can download the keys and use external tools to compute the fingerprint.) On Android and iOS, there is no way at all to display fingerprints or download keys (neither your own nor your contacts'). And with KT not available on mobile, users continue to have to blindly trust the key server.

Additionally, as we have seen in section 4.5, manual fingerprint comparison still remains important for contacts who are not using Proton. This is because they are not running Self Audits and thus don't benefit from KT.

Recommended improvement First, we recommend that KT warning pop-ups have a button to download the underlying KT state as a sort of "evidence export". Second, we believe that even with KT, manual key fingerprint comparison will remain an important fallback. We recommend to properly implement this across all platforms and products.

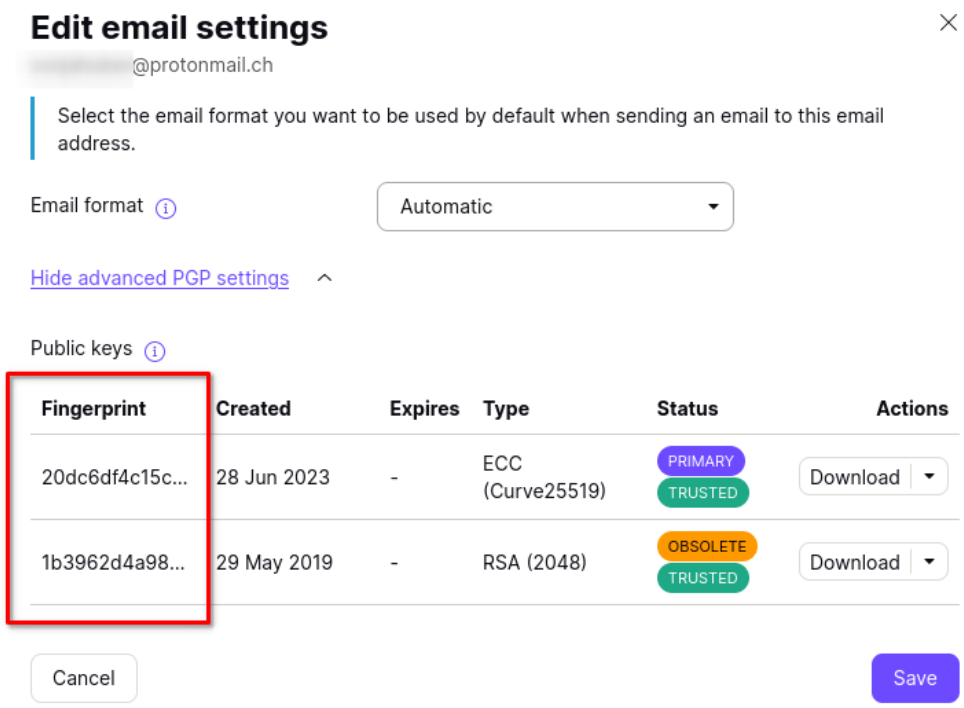

Figure 5.1: Viewing a Contact's Key Fingerprints in Proton Mail Web Client v5.0.30.8

5.4 Improved Check that a Revision is the Latest Revision

Queries that return included or obsolent values need to verify that the revision rev that the server replied with is really the latest available revision, so that the server cannot present an outdated view. Self Audit needs to do the same to check that the server is not hiding newer revisions. Recall from section 3.9 and section 3.12 that to achieve this in ProtonKT, the server also provides an absence proof for $\text{rev} + 1$ and clients verify it. ProtonKT assumes that auditors verify that revisions are continuous. That is, the server must not insert revisions 1, 3, and 5 while leaving 2 and 4 absent.

Recommended change Instead of verifying the absence proof for $\text{rev} + 1$ and relying on auditors to detect server misbehaviour, clients can check for most-recentness directly. Consider the right neighbours on the lower co-path, within the revision subtree. If rev is the highest non-absent revision, then all the right neighbours in the revision subtree co-path must be empty. We recommend that instead of checking an absence proof, ProtonKT clients check this condition.

It is faster: Checking an entire absence proofs requires 256 hash function invocations, while our check only requires up to 32 comparisons. It also saves bandwidth and server computation time.

Note that this is already implemented for absent values in ProtonKT.QueryValue: it checks the tree proof for revision 0 and then checks that the lower neighbours are all empty. [4] This allows the client to conclude that the latest value is absent in the tree. Note that for simplicity Figure 3.7 still shows the old way of checking rev + 1 for absence, too.

5.5 Server can Delay Promise Audits for more than the Maximum Merge Delay

Situation Recall from section 3.4 that the SKL has a field *expectedMinEpochId* which the server uses to signal to the client when it should expect the value to be included in the tree, i.e. when it makes sense to check back for an inclusion proof.

Also recall that for security we require all promises to be included in the tree within the Maximum Merge Delay (MMD) of 72 hours. That is, clients should audit the promise after the MMD has passed, no matter what the non-binding *expectedMinEpochId* says.

Attack Originally the web client checked the *expectedMinEpochId* first and the MMD only later: [5]

```
if (expectedMinEpochID > epoch.EpochID) {
return LocalStorageAuditStatus.RetryLater;
}
/* ... only further down the MMD is checked ... */
if (isTimestampTooOld(creationTimestamp)) {
  return throwKTError('SKL revision was ignored after more '
      + 'than max allowed interval', { email, revision });
}
```

Thus a malicious server could set a very high *expectedMinEpochId*, one that is years in the future. This would cause the client to effectively never audit the promise. Because the promise is not audited for eventual inclusion, the server could have replied to the query with any value that is not in the tree and thus not seen by a Self Audit. This would break Query-to-SelfAudit Consistency. Note that we only reviewed the source code and did not verify this attack in practice.

[4] https://github.com/ProtonMail/WebClients/blob/ebfa3ae416d1ff66/packages/key-transparency/lib/verification/verifyProofs.ts#L128-L137

[5] https://github.com/ProtonMail/WebClients/blob/ebfa3ae/packages/key-transparency/lib/verification/self-audit/verifyLocalStorage.ts#L60

Fix This has since been fixed: [6]

```
if (!isTimestampTooOld(creationTimestamp) && expectedMinEpochID > epoch.EpochID) {
  return LocalStorageAuditStatus.RetryLater;
}
```

Client Time vs Server Time Clients not having the correct time, i.e. having too much clock skew, is a general problem is security. For example, if the device time is off by too much, the notBefore and notAfter timestamps of X.509 certificates cannot be properly verified. The PGP signatures used by Proton also contain timestamps, thus signature verification is another reason for having roughly the correct time. To address this issue, Proton's web client uses the HTTP Date header to obtain the server time. The client maintains a `lastServerTime` field that is updated with every API response received from the server. [7] If the local client time diverges more than 24 hours from the server time, the web client displays a warning. [8]

What does this mean for Promise Audits? They check that:

$$|currentTime - creationTime| > \mathsf{MMD}$$

In the web client implementation both the *currentTime* (set during Promise Audit[9]) and the *creationTime* (set during querying when storing the promise[10]) use the server time. Thus while in our security analysis we assume that these timestamps are trusted, in practice the server controls both of them.

However, this does not lead to an attack because the user will always see a warning: either a KT warning or a time warning. Thus at worst the server can relax the MMD bound of 72 hours by another 48 hours. In our security analysis this is equivalent to changing the MMD parameter.

[6] https://github.com/ProtonMail/WebClients/commit/b55859a1031650abf9c3fb76b84cc8ac19d232f8

[7] https://github.com/ProtonMail/WebClients/blob/9d9fcbd3f6b98135/packages/components/containers/api/ApiProvider.js#L155

[8] https://github.com/ProtonMail/WebClients/commit/ddb1ad3701e127a32fe74401b1ec9996fd7ff5d1

[9] https://github.com/ProtonMail/WebClients/blob/ab32168726081262/packages/key-transparency/lib/helpers/utils.ts#L23-L32

[10] https://github.com/ProtonMail/WebClients/blob/4a5bf82f0f806dc4/packages/components/containers/keyTransparency/useSaveSKLToLS.ts#L38

Chapter 6

Conclusion

This thesis analysed the security of ProtonKT, Proton's Key Transparency scheme.

We started by describing the individual parts of ProtonKT at a high-level. Afterwards we gave a detailed specification of the subprotocols as message sequence diagrams. Notably, ProtonKT uses a different tree structure than previous KT designs, which allows it to shift additional work from clients into auditors, e.g. allowing it to support deletions without requiring tombstoning. ProtonKT also has slightly different subprotocols compared to the existing VKD primitive. For example, ProtonKT has a Promise Audit which allows the server to return values that are not yet in the tree but will be included soon.

The later parts of this thesis defined the security property that ProtonKT is designed to provide: Query-to-SelfAudit Consistency. We then did a manual analysis to argue why it should hold. We also formally modelled ProtonKT using the Tamarin Prover. Using subterms provided a more realistic, albeit still not fully faithful way to model both the Merkle tree and the root hash chain. For example, our hash tree is not binary and has only three levels. Due to limitations of Tamarin's induction, we did not find a formal proof showing that Query-to-SelfAudit Consistency holds in our model.

Finally, we gave some recommendations on how to improve the protocol in future iterations. This included preventing CAs being able to discredit the KT server, making the proof type explicit, making querying more efficient, and pointing out a now fixed attack on promises.

Future Work Analysing ProtonKT from other angles is a natural opportunity for future work. For example, we did not consider the computational model (classic game-based proofs). One could also pick up our Tamarin model to try to find a formal proof. Another possibility is to expand on non-discreditability, and to formally model and prove it as an accountability property, similar to [15].

Appendix A

Appendix

A.1 Keys at Proton

Proton uses a variety of asymmetric PGP keys across its products (which includes not just Mail, but also Calendar and Drive). This section describes the different key types and what they are used for.

A Proton user has a username and a password. We will use the terms "user" and "account" interchangeably. The password is used to authenticate against the server using the *Secure Remote Password (SRP)* protocol (a password authenticated key exchange (PAKE) protocol). [1]

The server stores all key pairs on behalf of the user in PGP-encrypted form. The bcrypt hash of the account password is used as a passphrase (OpenPGP-speak for symmetric key) to decrypt a key pair. [2]

Key Types The first type of key pair are the *account keys* (also called *user keys*). The account keys encrypt account-specific information. They are "private" to a user, i.e. not known to or needed by other users. They are not included in KT.

The second type are the *address keys* (also called *email encryption keys*). They are used for encrypting emails and other users can look them up. They are included in KT to achieve consistency.

There are also other asymmetric key pairs, e.g. *calendar keys* to encrypt calendars or *share keys* and *node keys* to encrypt files on Drive. These key pairs are protected by

[1] https://proton.me/blog/encrypted-email-authentication

[2] https://github.com/ProtonMail/WebClients/blob/4c9d1f09a015/packages/components/containers/login/loginHelper.ts and https://github.com/ProtonMail/WebClients/blob/4c9d1f09a015/packages/srp/lib/keys.ts

a randomly generated key (random "passphrase" in Proton's terminology) which is itself encrypted with the address key. [3]

Users can manage their account keys and address keys in the web client (see Figure A.1). For example, users with old Proton accounts that were created with RSA-2048 keys can generate new Curve25519 keys. The old keys are kept to allow decrypting old data.

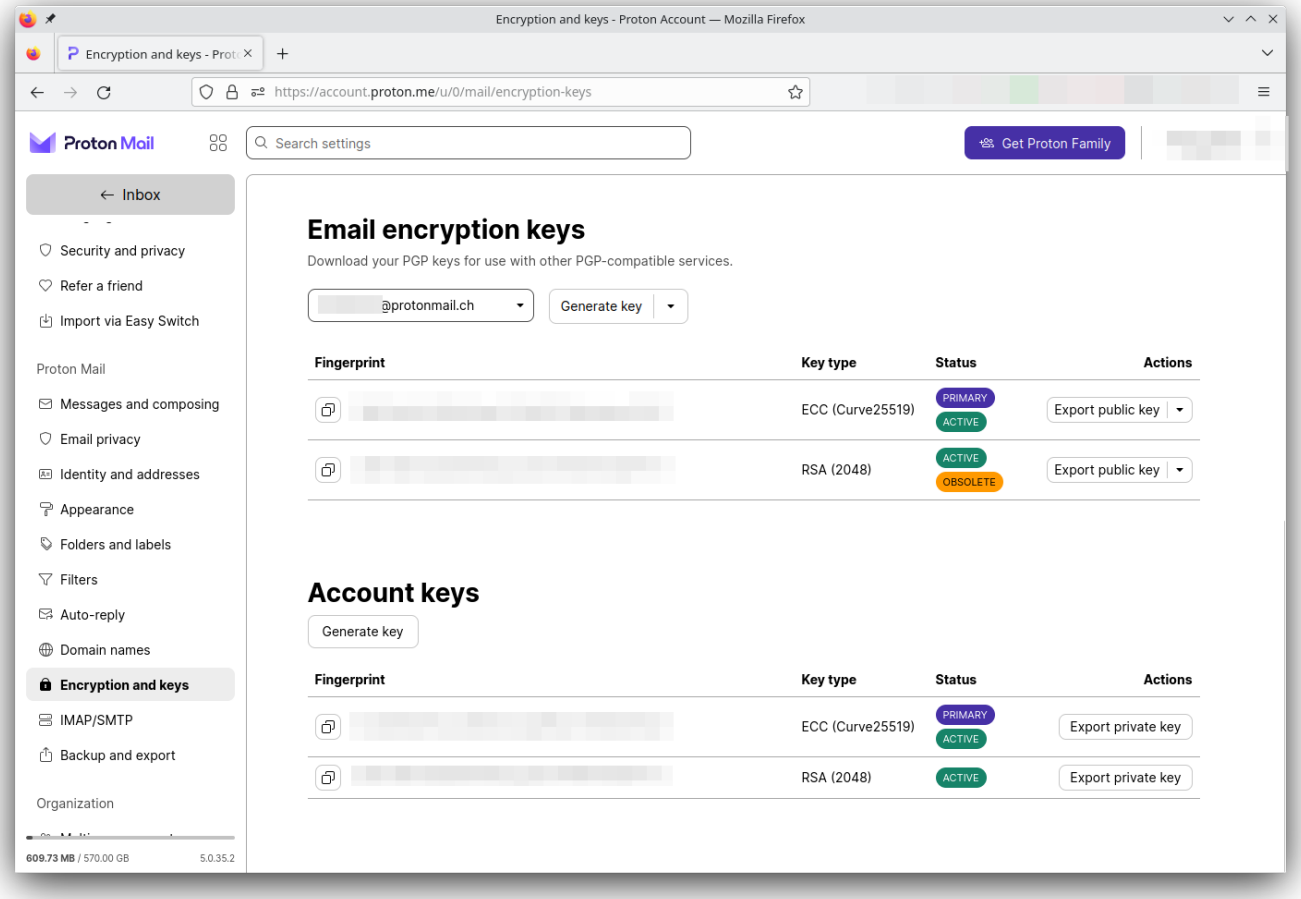

Figure A.1: Proton Account Keys and Email Keys

Trusted Keys Proton allows Bob to mark Alice's public key as "trusted". This effectively pins the public key: Bob will continue to encrypt emails towards Alice with this key, even if Alice changes her primary address key.

[3] https://proton.me/blog/protoncalendar-security-model and https://proton.me/blog/protondrive-security

Key Flags Proton stores some metadata alongside the keys. Note that this metadata is only interpreted by Proton-clients and not part of the OpenPGP standard. However, this metadata is included in the SKL and thus in KT (see section 3.3).

A key can be marked as *primary*, a simple boolean flag. The primary key is used by default for all operations. E.g. when Bob looks up Alice's address key, the server will return the primary key. Alice will also sign her outgoing emails with her primary address key.

There is also an integer $flags$ where the bits have the following meanings (note that the negations can be confusing): [4]

1: *NOT-COMPROMISED*: If this flag is set to 0 (i.e. *not* set) the key is marked as compromised and all signatures made by this key will be considered invalid. Thus if this flag is set to 1 this "key can be used to sign". [5] A compromised key is also obsolete, i.e. no new email may be encrypted to it.

2: *NOT-OBSOLETE*: If this flag is set to 0 (i.e. *not* set) the key is marked as obsolete and no new emails will be encrypted to it. If Bob has previously marked this key as trusted, this flag forces Bob to no longer use this key, and use the new primary one instead. Thus if this flag is set to 1 this "key can be used to encrypt". [6]

4: *EMAIL-NO-ENCRYPT*: Don't encrypt emails with this key. This is used for keys that belong to External Addresses (see section 3.2). For example, if alice@gmail.com signs up for Proton Drive (but not Mail) Proton will create an address key for Alice so that her files can be encrypted. However, this key should not be used by Bob to encrypt email, since GMail won't be able to decrypt it.

8: *EMAIL-NO-SIGN*: Don't expect emails to be signed with this key. See *EMAIL-NO-ENCRYPT*.

Some flags are visible to the user in the Mail web client. [7] For example, the user can mark a key as primary and toggle the *NOT-COMPROMISED* and *NOT-OBSOLETE* flags. The other two flags (*EMAIL-NO-ENCRYPT*, *EMAIL-NO-SIGN*) are not shown in the UI.

[4] https://github.com/ProtonMail/WebClients/blob/1135ef15e95feb5d/packages/shared/lib/constants.ts#L840:L853

[5] See footnote 4

[6] See footnote 4

[7] https://proton.me/support/pgp-key-management#key-flags-and-settings

Bibliography

[1] Cerificate transparency. https://certificate.transparency.dev/.

[2] Keybase server documentation. https://book.keybase.io/docs/server/stellar.

[3] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack resilient public-key infrastructure. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 382–393, November 2014. https://netsec.ethz.ch/publications/papers/ccsfp200s-cremersA.pdf.

[4] Henry Birge-Lee, Yixin Sun, Anne Edmundson, Jennifer Rexford, and Prateek Mittal. Bamboozling certificate authorities with bgp. In *Proceedings of the 27th USENIX Security Symposium*, Proceedings of the 27th USENIX Security Symposium, pages 833–849. USENIX Association, 2018.

[5] Henry Birge-Lee, Yixin Sun, Anne Edmundson, Jennifer Rexford, and Prateek Mittal. Bamboozling certificate authorities with bgp. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 833–849, USA, 2018. USENIX Association.

[6] Joseph Bonneau. EthIKS: Using ethereum to audit a coniks key transparency log. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 95–105, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-53357-4_7.

[7] 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.

[8] Vincent Cheval, José Moreira, and Mark Ryan. Automatic verification of transparency protocols (extended version), 2023. https://arxiv.org/abs/2303.04500.

[9] Cas Cremers, Charlie Jacomme, and Philip Lukert. Subterm-based proof techniques for improving the automation and scope of security protocol analysis. Cryptology ePrint Archive, Paper 2022/1130, 2022. https://eprint.iacr.org/2022/1130.

[10] Apple Security Engineering and Architecture (SEAR). Advancing iMessage security: iMessage Contact Key Verification. Apple Security Research Blog, Oct 2023. https://security.apple.com/blog/imessage-contact-key-verification/.

[11] Mohammad Etemad and Alptekin Küpçü. Efficient key authentication service for secure end-to-end communications. Cryptology ePrint Archive, Paper 2015/833, 2015. https://eprint.iacr.org/2015/833.

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

[13] 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/.

[14] Ralf Kuesters, Tomasz Truderung, and Andreas Vogt. Accountability: Definition and relationship to verifiability. Cryptology ePrint Archive, Paper 2010/236, 2010. https://eprint.iacr.org/2010/236.

[15] Robert Künnemann, Ilkan Esiyok, and Michael Backes. Automated verification of accountability in security protocols. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 397–413, 2019.

[16] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. RFC 6962, June 2013. https://www.rfc-editor.org/info/rfc6962.

[17] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. Certificate Transparency Version 2.0. RFC 9162, December 2021. https://www.rfc-editor.org/info/rfc9162.

[18] Sean Lawlor and Kevin Lewi. Deploying key transparency at WhatsApp. Engineering at Meta, April 2023. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/.

[19] 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.

[20] Marcela Melara. Why making johnny's key management transparent is so challenging, March 2016. https://freedom-to-tinker.com/2016/03/31/why-making-johnnys-key-management-transparent-is-so-challenging/.

[21] 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.

[22] 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.

[23] The Tamarin Team. Tamarin-prover manual, Oct 2023. https://tamarin-prover.github.io/manual/index.html.

[24] Security Cryptography Whatever. WhatsApp Key Transparency with Jasleen Malvai and Kevin Lewi, May 2023. https://securitycryptographywhatever.com/2023/05/06/whatsapp-key-transparency/.

[25] Jiangshan Yu, Vincent Cheval, and Mark Ryan. DTKI: A new formalized PKI with verifiable trusted parties. *The Computer Journal*, 59(11):1695–1713, July 2016. https://arxiv.org/abs/1408.1023.