OPTIKS: An Optimized Key Transparency System

Julia Len[1], Melissa Chase[2], Esha Ghosh[2], Kim Laine[2], and Radames Cruz Moreno[2]

[1]Cornell Tech, New York, NY[*]
[2]Microsoft Research, Redmond, WA

Abstract

Key Transparency (KT) refers to a public key distribution system with transparency mechanisms proving its correct operation, *i.e.*, proving that it reports consistent values for each user's public key. While prior work on KT systems have offered new designs to tackle this problem, relatively little attention has been paid on the issue of scalability. Indeed, it is not straightforward to actually build a scalable and practical KT system from existing constructions, which may be too complex, inefficient, or non-resilient against machine failures.

In this paper, we present OPTIKS, a full featured and optimized KT system that focuses on scalability. Our system is simpler and more performant than prior work, supporting smaller storage overhead while still meeting strong notions of security and privacy. Our design also incorporates a crash-tolerant and scalable server architecture, which we demonstrate by presenting extensive benchmarks. Finally, we address several real-world problems in deploying KT systems that have received limited attention in prior work, including account decommissioning and user-to-device mapping.

1 Introduction

The term *Key Transparency* (KT) refers to a key distribution system – a key directory – that provides transparency guarantees for its operation. These transparency properties are often implemented using cryptographic proof techniques, but may in some cases be implemented using trusted execution environments as well. While KT cannot prevent the system from misbehaving (for example, making an unrequested key update on a user's account), it ensures any incorrect behavior will be detected either immediately, or with a delay.

The importance of KT is particularly evident in end-to-end encrypted (E2EE) communication. If the communication service simply distributes the public keys of communication partners, nothing would prevent it from inserting itself into the conversation as a *meddler-in-the-middle* (MitM) and capturing traffic intended for the victim. Realizing this obvious problem, some communication system providers have implemented mitigations like *security codes*, which require scanning QR codes for text-based messaging or reading out long strings of numbers among participants for calls or meetings. However, using these techniques requires manual interaction. KT provides an automated way of checking that the users are getting the correct keys: as it requires no user interaction, it provides a much more usable and secure solution.

In a KT system, the server maintains the directory of user public keys. It periodically publishes a privacy-preserving[1] commitment to its current directory on a bulletin board. The server produces a cryptographic proof of consistency along with any keys it distributes: the purpose of this proof is to show that the keys distributed are both the latest and consistent with the commitment posted on the bulletin board. A KT system supports two types of queries. The user devices can monitor their own key history by asking for their key history and proof. Users can also ask for the latest key of their contacts.

KT has gained significant traction, both in industry [4, 25] and academia [6, 8, 12, 17, 18, 22, 24]. This increased interest is also evident from the developing IETF standardization effort [19]. Notably, WhatsApp [14], Apple iMessage [1], and Proton Mail [2] recently deployed KT, with WhatsApp's design based on academic systems SEEMless [6] and Parakeet [17], iMessage's design based on CONIKS [18] and Proton Mail's design combining ideas from CONIKS [18], SEEMless [6] and Parakeet [17].

While there has been a rich body of literature on KT systems, most of these prior systems lack important features like scalability, among others. To the best of our knowledge, Parakeet is the first paper tackling scalability challenges that arise

[*] Work partially done at Microsoft Research.

[1] *Privacy:* The keys maintained by a user as well as their key updates can be sensitive information. Accordingly, academic and industry proposals [4, 6, 8, 14, 17, 18] and recent standardization efforts [19] have emphasized privacy as an explicit goal for KT systems. Thus, we also aim for privacy as a goal of our system: in particular, lookups for a client's key should not leak anything about other keys stored by the system outside of what is returned by the lookup itself.

when moving from academic proposals to large-scale deployments. This is an important first step, but the design misses some key attributes as well as optimizations. In this paper, we present OPTIKS, a full-featured and optimized KT system. We show that OPTIKS achieves the same level of security and nearly equivalent privacy as Parakeet using a much simpler and more scalable design.

Prior work: CONIKS, SEEMless, and Parakeet. To better understand how we improve over prior work, we begin with a brief technical overview of previous designs. The first KT system with (content) privacy was CONIKS [18]. At a high level, CONIKS works as follows: at every epoch, the server computes a sparse Merkle tree-based hash digest of its entire directory and publishes it. Notably, CONIKS places the key for a client at the same position in the tree each time. Clients are then required to check their key is correctly represented in the digest each epoch.

SEEMless [6] removes this burden from clients by maintaining one tree storing all current and historical key updates. The client can then check a single tree and see exactly when and to what values their key was updated. SEEMless also improves privacy, by always appending any new key changes to the tree in new (random) positions; these positions are chosen using a verifiable random function (VRF) which guarantees a unique but pseudorandom position for each entry. But, this approach increases storage over time. Moreover, SEEMless's design requires the server to perform complicated bookkeeping that maintains, among other things, two copies of the tree, effectively doubling the storage and update costs. Parakeet [17] improves the storage expansion in SEEMless by optimizing the trees, but it still maintains two trees. Its approach to compaction is to increase the bookkeeping complexity by marking nodes as ready for deletion.

In more detail, SEEMless and Parakeet maintain two trees, which [6] refers to as the *all* and *old* trees. The all tree stores all keys ever added to the system, while the old tree only stores stale versions of keys. When a client looks up a key, the server provides a non-membership proof that the key's latest version is not in the old tree to show it is not stale. When the client monitors its own key, it checks that all outdated key versions are indeed added to the old tree, thus guaranteeing that a key lookup can only return the latest key.

OPTIKS-core: Algorithmic optimizations. We first present OPTIKS-core, a protocol which incorporates algorithmic optimizations over prior work. To this end, we make several key observations. First, we cut storage in half by changing the way the client verifies it is getting fresh keys. Second, we consider how the protocol will be used in practice and optimize for those usage patterns.

Recall that the goal of the server is to prove the key it serves has been logged against the specified username and is the latest version. We observe that the old tree is solely for proving this freshness guarantee. We can therefore completely remove the old tree by changing the freshness proofs. As our first optimization, we note that an equivalent freshness proof is to show that the current version is the latest version in the tree; we can do this by giving a non-membership proof for the next version number. This allows us to completely remove the old tree. Since the trees are nearly equivalent sizes, our approach immediately cuts the storage and update costs by half compared to SEEMless and Parakeet.

For the second optimization, consider how users might interact with the system. We expect that clients will occasionally update keys and look up keys of communication partners, but will regularly, and ideally frequently, monitor their own keys to ensure they have not changed unexpectedly. OPTIKS-core is thus optimized to minimize the cost of this latter, more frequent operation of client monitoring, at the cost of somewhat more expensive key lookups. This also allows us to significantly simplify the construction, which makes it easier to explain and implement.

Specifically, we eliminate the complex marker-based bookkeeping logic in SEEMless and Parakeet by changing the lookup and key history proofs. In particular, each key lookup now returns membership of all key versions and non-membership of the next version. Clients then verify all key updates for the queried user (in Section 5, we discuss how to reduce this overhead via client caching). Notably, client monitoring of its own key is then the same as key lookups, thereby reducing system complexity.

Notice that since the tree is append-only, the client only needs to check once that each key update has been included. After that (unless its key has changed), each time it monitors it only needs to check that the key has not been updated to the next version unexpectedly. This then reduces the typical expected client monitoring cost to that of verifying a single non-membership proof. In contrast, the cost of client monitoring for SEEMless and Parakeet scales with the number of client key updates and (logarithmically) with the number of epochs. Our approach of optimizing for the most common operation thus significantly reduces expected deployment costs and simplifies the protocol. We note that while this does slightly increase privacy leakage, we believe this increase is reasonable, particularly because it is further limited by the other features we propose.

We observe that ELEKTRA [15] takes a similar approach, although it adds the additional complexity of requiring signatures and hash chains. From that perspective, our result can be seen as extracting out and analysing security of the core algorithm beneath ELEKTRA, without the overhead and complexity that that work inherits from Keybase and from their stronger security goals. See Section 8 for further comparison.

The OPTIKS-core protocol described so far roughly matches with the functionality and security guarantees provided by CONIKS, SEEMless, and Parakeet.[2] Importantly,

[2] Parakeet does deal with storage overhead, which is the first of the OPTIKS-ext features that we will discuss.

by itself OPTIKS-core is significantly more efficient in the most frequent expected use case of the system. However, as discussed above, there are still significant limitations to deployment. We thus also present OPTIKS-ext, which aims to address these limitations.

OPTIKS-ext: Further improvements. Perhaps the most significant contribution of OPTIKS-ext is its approach to limiting storage growth. We modify the append-only *all* tree by integrating the notion of a longer time period. For example, if our epoch is on the order of a second, this time period might be on the order of a month. At the beginning of each time period, OPTIKS uses only the most recent key for each client to construct a new tree; it then appends any subsequent updates of the current time period to this tree. Clients are responsible for checking their keys every time period to make sure that the final key in each time period is indeed the same as the first key in the next. Implementers can choose which trees from old time periods to store and which to delete. Our design requires that clients be online frequently enough to perform these checks before the old time periods are deleted, which we view as reasonable since time periods will be long. We further note that [17] makes a similar assumption about its users to enable compaction. Notably, however, their compaction mechanism is more complicated and requires more bookkeeping compared to our time-period-based approach. Another benefit of this design over [17] is that it offers a limited form of post-compromise security for privacy, since each time period uses a new key for ensuring privacy. It also limits query leakage from our design, in that the query reveals the user's updates only in the current time period.

We highlight that the flexibility of deleting data from old time periods provides a trade-off in soundness guarantees. In particular, if implementers choose to delete data from old time periods and a user did not come online during some previous time period to check their keys, the user gets no security guarantee for that period. In other words, fake keys might have been distributed on that user's behalf during the missed time period, and the user could not detect this. However, the benefit of our design is that the time period is a tunable system parameter, so that systems with greater security concerns can elect to store data from older time periods. Thus, OPTIKS provides a way to smoothly trade off security for more storage.

Additionally, OPTIKS addresses vital features for deploying KT that prior work does not, such as user account decommissioning or how to resolve the tension between external facing human-readable usernames and internal facing UUIDs (or equivalent identifiers) for indexing into the KT directory. At a high level, we address these by adding additional data structures, but this must be done carefully to avoid subtle issues. To illustrate the subtlety, let us take the decommissioning feature as an example. When a user stops using the system, they will presumably no longer be monitoring their own key history. Ideally, a KT system should still ensure that a malicious service provider cannot replace their key and impersonate them in future communications.

In OPTIKS, we enable this by maintaining an additional tree to keep track of the decommissioned user accounts. At first glance, this seems wasteful – one might ask why we cannot just add the decommissioned usernames to the same tree for each time period? This is because we want the tree of decommissioned usernames to persist through the lifetime of the system and not just between time periods, regardless of whether the decommissioned user is available to monitor it.

Architectural innovation. Prior work considers a monolithic service provider with no specifics on how it operates. OPTIKS contributes a novel system architecture, where we physically separate the query component from the update component. This distinction enables OPTIKS to use more optimal memory representations for the Merkle tree nodes in each component. The benefits are numerous, including better performance, more efficient parallelized updates, and minimal service downtime at epoch boundaries. Many of these ideas can improve other KT systems as well.

In more detail, for updates, we use a linked node representation, where the cost for looking up child nodes is simply the cost of random memory access. Our experiments show that this is at least 2.5 times more efficient than inserting in a hash table representation. It also lends itself to a much more efficient parallel update algorithm. For example, our experiments show that parallel updates on 16 vCPUs improves our batch update performance by 3.7x, whereas Parakeet (implemented using a hash map for node storage) only achieves a 1.3x speedup. On the other hand, a hash table representation is more practical for serving queries, as it supports very fast partial tree node updates, essentially only requiring pulling updated nodes from a database.

Another advantage is that this separation allows us to tailor the implementation separately to the case of updates or queries, in particular in terms of thread-safety. Instead of a monolithic system like Parakeet, for example, that must use extensive locking to make sure that incomplete updates do not produce inconsistent queries, our system trivially prevents this kind of conflict since update and query operations are performed on different machines. We need only worry about thread-safety within the query server and separately within the update server, allowing us to use much lighter-weight techniques. Indeed, our parallel update algorithm does not use any locks at all. On the query side, we need a single lock to indicate that some of the nodes are being updated with new data downloaded from a backing database, minimizing any downtime at epoch boundaries.

This separation also makes it trivial to scale the query service horizontally, just by adding more machines. In this case update downtime can be entirely eliminated by interleaving the query server updates, so that there are always sufficient machines available to respond to traffic. This does mean,

System	Storage	Lookup	Update	Audit	Client Monitoring
CONIKS [18]	$e \cdot n$	1	$k\log(n)$	1	e
SEEMless [6]	$2n$	3	$2k\log(n)$	$2(k+k\log(n))$	$v+\log(e)^*$
Parakeet [17]	$2n$	3	$2k\log(n)$	$2(k+k\log(n))$	$v+\log(e)^*$
OPTIKS	n	$v+1$	$k\log(n)$	$k+k\log(n)$	1

Figure 1: Asymptotic costs of OPTIKS in comparison with other KT systems. For client monitoring we assume clients may cache proofs they have already checked. We note that (*) represents worst case cost; the exact cost is $2^{\lceil \log(v) \rceil} - v + \log(e)$.

however, that these servers may serve slightly different commitments. We show how to modify our PAHD construction so that transparency can still be maintained – another novel feature.

A final important part in our architecture is a cache to store recently used VRF values on the query side, as evaluating the VRF is a heavy public-key computation. The benefit of a cache hit is immense, providing as much as 9 times higher query throughput for the key directory (ignoring networking and web service overheads).

Comparison to prior KT systems. We concretize our comparison of OPTIKS with prior related systems in Figure 1, which shows the algorithm asymptotics of each system. We assume a directory with n key-value pairs and e epochs. We also assume an update that will add k new key-value pairs, a lookup for a key with v versions, and a client monitoring its key when it has not changed. As we mentioned, this latter operation represents what we expect as the majority of client monitoring cases. For simplicity, we assume no VRF computation. Since the core data structure for each system is a Merkle tree, we measure most operations in the number of nodes affected. Specifically, we measure storage in the number of nodes throughout system life, lookup and client monitoring in the number of (non-) membership proofs to check, update in the number of nodes to modify, and audit in the size of proof provided to each auditor per epoch in the number of nodes to download.

OPTIKS achieves the best performance for storage, client monitoring, and update cost, the latter of which is the same as CONIKS. The algorithmic optimizations of OPTIKS enables it to halve the audit costs of SEEMless and Parakeet, although it is still greater than the constant-time audit cost of CONIKS, in which an auditor only needs to check the latest root of the tree has been added to a hashchain. However, this cost comes at the trade-off of clients needing to come online and verify keys every epoch.

As mentioned, the increased performance of OPTIKS in other categories comes at the trade-off of worse lookup performance. However, given that we expect the typical use case to have a single key version each time period for clients, v is likely to be 1 in most cases, which means we expect most clients to verify only 2 proofs when doing a lookup.

Our experiments further show that OPTIKS is significantly more performant and scalable than prior work. For example, for a key directory of 2^{20} keys, a single instance of our Query Service can serve more than 4000 queries per second at a bandwidth of around 20 MB/s, while our Update Service/Task can process more than 1000 updates per second while sustaining a latency of one second. For a much larger key directory of 2^{26} keys, our Query Service can still serve 2240 queries per second at a bandwidth of 13.89 MB/s, and our Update Service/Task can process still around 280 updates per second; in this case the latency remains still less than 4 seconds on average. All of these experiments include the cost of database access, networking, and REST API overhead. We note that the major bottleneck for our update rate is database write performance, since we use costly multi-table transactions to simplify our implementation.

To summarize, we make the following contributions:

- **OPTIKS-core.** We first present OPTIKS-core, a KT system that incorporates novel algorithmic optimizations which enables it to be significantly more efficient and performant than [6, 17, 18]. The key insight is that we optimize for the typical use case in practice when a client is expected to rotate its long-term key relatively infrequently, but where the client wants to monitor relatively frequently to verify that this key remains unchanged.

- **OPTIKS-ext.** We enhance OPTIKS-core with additional features important for deployment. The crucial features include: keeping the core data structure compact, adding user account decommissioning, and adding support for multiple usernames and user devices.

- **Split architecture.** OPTIKS uses a novel architecture, where the update and query services are physically separate from each other. This allows us to use different memory representations for the Merkle tree nodes in each of these components. We believe that our split architecture is generally applicable to many Merkle tree-based systems where the read and write workloads are unbalanced.

- **Experiments.** We provide detailed benchmarks for our system, dividing them into micro-benchmarks and full-system benchmarks. To demonstrate that our system is more scalable than prior work, we run it on benchmarks significantly larger than any presented in prior work.

2 System Setup and Overview

Our KT system models a central server that stores a directory of usernames and the corresponding public keys. There exist intermittent time intervals that we refer to as *epochs*, which we expect to be on the order of seconds, during which the server updates the directory it stores with new key update requests and posts a commitment to this data. Our system also includes longer time intervals (e.g. a month) that we refer to as *time periods*, which we will discuss in Section 5. Users of the system can query the server to look up another user's key(s) or to get the history of the updates to their own key(s). We also assume that third-party auditors (though the users can play the role of the auditors as well) audits the commitments for consistency. We now describe this process in more detail, with an overview of our assumptions, the security properties we expect such a system to meet, and a summary of our solution.

Participants. Our system has the following participants:

- *Users.* A user can register an account with the server and also may permanently leave the system by decommissioning their account. Associated to each user is a *username*, such as an email or human-readable string, that represents their public-facing identity in the system. Also associated to each user is an internal, unique, and static *user id*, such as a UUID, that is not exposed to human users of the system. Note that in practice a username may change or multiple usernames may be associated with a user (*e.g.*, if they add an extra email to their account), so the user id always uniquely identifies the user within the system.
- *Client devices.* Each user has at least one device which they use to communicate (e.g. phones, computers, etc.), where each device has its own public key that must be stored and distributed by the system on the device's behalf. We do not assume any coordination between the user's devices. Clients may update their public keys, look up the keys of other clients in the system, and also check the history of updates to their keys. Associated to each client is an internal, unique *device id*, such as a UUID, that is not exposed to human users of the system.
- *Server.* The server maintains the directory mapping users to their public keys and distributes these keys among the users of the system when queried. It posts a public commitment to the data each epoch. In this work, we also refer to the server as the "service provider."
- *Auditors.* Auditors verify updates made by the server are well-formed via the publicly posted commitments. To ensure privacy, this verification does not involve checking the public keys themselves are correct (indeed, this task falls onto clients, as we mention above). Auditors can be third-party entities or security-conscious users.
- *Bulletin board.* The server posts the commitments to its directory on a public bulletin board to which other participants of the system have access. The bulletin board should be tamper proof as well as append-only and also all participants should have a consistent view of its contents.

Assumptions. As is standard in any KT system, we assume that the server can be malicious and distribute incorrect keys for its users (in the hope of mounting a MitM attack). However, the server is trusted to exercise access control and not give out every client's public key to everyone else. In other words, the server is trusted for privacy.

The client devices can be malicious in that they may aim to learn private information (public keys, how often a certain user changes her key, etc.) about other clients who are not on their contact list.

We assume there exists at least one honest auditor who verifies each update made by the server via commitments posted to the bulletin board.

Our system also relies on all participants having a consistent view of the commitments posted to the bulletin board, which we highlight is a core requirement in all KT systems. As discussed in [6, 12, 17, 18], this could be implemented, for instance, via a gossip protocol or by posting the commitments to a blockchain. Furthermore, we assume the clients, server, and bulletin board have approximately synchronized clocks.

Although we do not model this, we assume that the server enforces some kind of access control for clients querying its system, e.g. rate limiting key lookups or executing key lookups only if the requesting user is a contact of the user whose key is being queried.

Our system relies on users being able to verify the history of their key updates. Therefore, users must have some way of keeping track of their devices and the approximate times of their key updates. This is an assumption made of other KT systems like SEEMless [6]. One way to facilitate this is to enable users to add notes to their key updates, such as "added new laptop." Also crucial to our system is that clients must be online to check their key history each time period. We utilize this assumption as part of our scalability optimizations, which we discuss in Section 5. Given that time periods are long, we expect most clients will achieve this in practice and, indeed, this is a common assumption of KT systems [12, 17]. Moreover, this is an improvement over many KT systems which assume that a client must be online *each epoch* to check their keys [18].

Lastly, the core data structure underlying our KT system is a dictionary that uses a key-value abstraction. Since our construction involves public keys, wherever possible we disambiguate between the two by referring to them either as dictionary keys or public keys explicitly. However, where it is clear from context, we will simply say "key" to mean either dictionary key or public key.

Security Properties. At a high level, we expect OPTIKS to achieve the following security properties. We present these definitions in more detail in Section 3 and Appendix C.

- *Completeness.* When the server is honest, a user that looks up another user's key should receive the latest value of that key, and this should be consistent with what other users of the system see. This also means that all proofs the server provides during a lookup must verify.

- *Soundness.* Assuming that all epochs are audited, the server cannot lie about a key's value during a lookup without the inconsistency being caught during a history check.
- *Privacy.* A KT system should maintain privacy for the users of the system and updates to their keys. We model this with a definition that says participants of the system (excluding the server) should not learn anything from queries to the server except for some well-defined leakage function. For instance, a key lookup for a user should not leak anything about the keys of *other* users of the system.

3 Building Blocks

In this section, we introduce the primitives *Private Authenticated History Dictionaries* (PAHD) and *ordered Zero-Knowledge Sets* (oZKS), which form the core of our construction.

Private Authenticated History Dictionaries. We define *Private Authenticated History Dictionaries* (PAHD), a new cryptographic primitive which forms the basis for OPTIKS. This primitive extends the authenticated history dictionary introduced and used by VeRSA [23]. At a high level, a PAHD enables storing and committing to data using a dictionary key-value abstraction. A server can update what it stores, which begins a new epoch with a new commitment to the dictionary, by adding new key-value pairs or updating the values for existing keys. Notably, the structure preserves the history of changes for keys. Clients can look up a key to retrieve its latest value, along with a proof that the value is correct. Clients can also check the update history for a particular key to learn when it was updated and to what values—this can be used to verify that the recorded history of changes is accurate. We also assume that associated with a PAHD is a randomness space R from which a random seed can be chosen to initialize a PAHD. We provide an informal overview of this primitive below and a detailed description in Appendix C.

- PAHD.Init: The initialization algorithm outputs the initial commitment to the empty dictionary.
- PAHD.Upd: The update algorithm updates the dictionary with a set of key-value pairs and outputs the updated dictionary and update proof.
- PAHD.Lkup / PAHD.VerLkup: The lookup algorithm retrieves the value v for key k along with a membership proof if k is in the dictionary or non-membership proof if k is not. The lookup verification algorithm then verifies this proof.
- PAHD.Hist / PAHD.VerHist: The history algorithm returns the set of values that key k has been assigned over time, the epochs during which each value was assigned, and the membership proofs for each key-value mapping. The history verification algorithm verifies the proofs that are returned.
- PAHD.Audit: The audit algorithm verifies the update proof between two consecutive commitments.

Security definitions. We present an overview of the security properties that a PAHD should meet below and formalize the definitions in Appendix C.

- *Completeness.* Completeness captures the following correctness properties: if a PAHD is initialized and updated honestly, then auditing between any two epochs should succeed, the lookup for any key k should return its latest value v and should verify, and the history check for k should return the correct history of values and the epochs they were added and should also verify.
- *Soundness.* PAHD soundness guarantees that, assuming the data store has been audited successfully by an honest auditor each epoch, a lookup for a key k cannot return some value v that is inconsistent with what the history algorithm returns for k at that epoch. For a PAHD scheme that meets soundness, this means that the server cannot lie about a key's value during a lookup without the inconsistency being caught during a history check. However, this does mean that the user who added the key must perform such history checks to verify that the key's value is correct.
- *Privacy.* The privacy goal for PAHD is that the outputs of Upd (which is used for auditing), Lkup, and Hist should not leak anything beyond the answer and what is specified by a well-defined leakage function $\mathcal{L}$ on the directory's state. We model this using a real-ideal world computational indistinguishability game where a simulator must simulate the outputs of these algorithms using the given leakage.

To instantiate a PAHD scheme, we make use of *ordered Zero-Knowledge Sets*, which we define next.

Ordered Zero-Knowledge Sets. An *ordered Zero-Knowledge Set* (oZKS) is a primitive that lets a potentially malicious prover to commit to a collection of (label, value)-pairs such that the prover can later prove the membership or non-membership of labels in the collection succinctly. The primitive also enables append-only updates to the collection of pairs. This primitive additionally requires a strict ordering on elements inserted by attaching the epoch of insertion along with the label-value pairs and committing to this as part of the data. This primitive is zero-knowledge because the commitment does not leak information about the collection of data and the proofs do not leak information about any other data in the collection.

oZKS builds on the aZKS primitive introduced in [6]. Primitives closely related to oZKS were defined in [8, 17, 21]. For a detailed description of the related notions, see Appendix B. We provide an informal overview of this primitive below and a detailed description in Appendix B.

- oZKS.Init: The initialization algorithm outputs an initial commitment to the empty datastore.
- oZKS.Update / oZKS.VerifyUpd: The update algorithm adds a set of new label-value pairs to the datastore, outputting the new commitment to the data and an update

proof. The update verification algorithm then verifies the update proof between consecutive commitments.

- oZKS.Query / oZKS.Verify: The query algorithm returns the value associated to the queried label, along with the query proof and the epoch that the label was added (or $\perp$ and a non-membership proof if the label is not a member). The query verification algorithm verifies the value returned by a query using the proof.

Construction. We construct an oZKS from an append-only strong accumulator (aSA), a simulatable verifiable random function (sVRF), and a simulatable commitment scheme (sCS), as in [6, 17]. See definitions in Appendix A.

The aSA is constructed from a Merkle Patricia Trie and serves to commit to a dictionary. The label-value pairs serve as the leaves of the tree, where labels are used to specify the location of the leaf. Instead of using the label directly (which could leak sensitive information), we use the sVRF to compute the positions of the labels in the tree. We then use the sCS to commit to the label's value; this commitment and the epoch when the label was added serve as the value stored for each label. For more detail see Appendix B.

Security Definitions. Just as for an aZKS, we expect an oZKS to meet *completeness*, *soundness*, and *privacy*. We describe these definitions in detail and show that our construction meets them in Appendix B.

4 OPTIKS-core: Core OPTIKS Protocol

In this section, we describe a simple, lightweight PAHD construction which we use as the core of OPTIKS, referred to as OPTIKS-core. For simplicity, we assume that each user has one client device and so we use usernames directly as the dictionary keys and the corresponding cryptographic public keys as the values. (We consider the multi-device setting in Section 5.) Our protocol relies on an oZKS as described in Section 3 for its core building block.

▷ PAHD.Init(r): The server chooses a random seed and initializes an empty oZKS via oZKS.Init by giving r as input. The oZKS commitment is returned as the initial commitment and the oZKS initial state is stored in the server's state. The server also initializes the epoch to 0 and stores this in its state.

▷ PAHD.Upd($\mathsf{st}_{t-1}, [k_j, v_j]_j$): The server adds the key-value pairs that are input to the oZKS to create a new commitment to the dictionary. It first checks that all the keys to be updated are unique; if not, it returns $\perp$. In order to differentiate between versions for a key, the server uses the key concatenated with its version number as the oZKS label. We assume that the server keeps track of the version number for each key in its state. Thus, for each key-value pair (k, v), the server first checks if the key already exists in the oZKS. If it does not, the server uses $(k \mid 1)$ as the label. Otherwise, if the key is already at version n, then the server uses $(k \mid n+1)$. Once all the label-value pairs have been formed, the server adds them to the oZKS via oZKS.Update. The server increments the epoch $t-1$ in its state to t, and the resulting oZKS commitment com_t and epoch t serves as the PAHD commitment for epoch t: (com_t, t). The oZKS update proof π^{upd} serves as the PAHD update proof Π_t^{Upd} for epoch t and is stored in the server's state. The server also stores in its state the new oZKS datastore and state.

▷ PAHD.Lkup(st_t, k): For a lookup request for key k, the server retrieves from its state the latest oZKS commitment com_t and the latest version number α for k (where $\alpha = 0$ if k is not in the PAHD). If k is in the PAHD, then the server forms labels $(k \mid 1), \ldots, (k \mid \alpha)$ and calls oZKS.Query for each label to get back $[(\pi_i, v_i, t_i)]_i^\alpha$. To retrieve the non-membership proof $\pi_{\alpha+1}$ for the next version of the key (or to prove that k is not in the dictionary when $\alpha = 0$), the server calls oZKS.Query for label $(k \mid \alpha+1)$. The server returns either v_α as the value for k if $\alpha > 0$ or $\perp$ otherwise.

The server returns as its lookup proof:

– **Correct version i is set at epoch t_i:** For each $i \in [1, \alpha]$, π_i serves as the membership proof for $(k \mid i)$ with value v_i and associated epoch t_i in oZKS w.r.t. com_t. This means the server must return $[(\pi_i, v_i, t_i)]_i^\alpha$ as part of the proof.

– **Server could not have shown version $\alpha + 1$:** Proof $\pi_{\alpha+1}$ serves as the non-membership proof for $(k \mid \alpha+1)$ in oZKS w.r.t. com_t.

▷ PAHD.VerLkup($\mathsf{com}_t, k, v, \pi$): The client first parses com_t as (com, t). Then it verifies each membership proof for labels $(k \mid i)$ for $i \in [1, \alpha]$ and non-membership proof for $(k \mid \alpha+1)$ w.r.t. com and t via oZKS.Verify. We want to preserve a total ordering of key versions and so wish to prevent this from happening. Lastly, the client verifies that the update epochs $t_1, \ldots, t_\alpha$ are monotonically increasing.

▷ PAHD.Hist(st_t, k): This algorithm proceeds the same as Lkup, except that in its syntax it explicitly returns all key versions rather than including them in the proof. Looking ahead, history checks will be different when we introduce our scalability optimizations in Section 5.

▷ PAHD.VerHist($\mathsf{com}_t, k, [(v_i, t_i)]_i^n, \Pi^{\mathsf{Ver}}$): This algorithm proceeds identically to that of VerLkup.

▷ PAHD.Audit($\mathsf{com}_j, \mathsf{com}_{j+1}, j, j+1, \Pi_{j+1}^{\mathsf{Upd}}$): The auditor verifies the oZKS update proof in Π_{j+1}^{Upd} via oZKS.VerifyUpd and then checks that $j+1 \leq t$, where t is the current epoch.

Security and Privacy of OPTIKS-core. We formally prove the security of privacy of OPTIKS-core in Section C.3. Here, we give an informal description of the leakage of OPTIKS-core. During updates, our protocol leaks the number of keys to be updated and the set of keys that were queried to Lkup or Hist since the previous update. Both lookups and history checks leak the value and epoch of addition for each version of a key. Our leakage profile is therefore nearly the same as that for SEEMless and Parakeet, except that key lookups in their protocols leak only the version number for the key and the value and epoch of addition for the latest key version. Looking ahead, we will describe how to minimize such leakage for lookups in Section 5.

5 OPTIKS-ext: Full Featured OPTIKS

As described in Section 1, there is a lot more to making the system deployable beyond the base protocol. Here we discuss in detail how we address those challenges by describing our full-featured protocol OPTIKS-ext. In particular, we describe scalability and reliability optimizations as well as important feature additions to our core protocol.

Reducing storage. A major downside of OPTIKS-core is that it must store all past key updates, resulting in storage that grows indefinitely. To avoid this, we must find a way to safely delete old data, without compromising the transparency guarantees. Parakeet [17] does this with a complex system of bookkeeping. We propose a much simpler solution: we consider time periods of a fixed length (e.g., a month). At the beginning of each time period, we start a new PAHD structure, copying over each key along with its latest version. We assume that users perform a history check at least once a time period. (The only other system to consider limiting storage, Parakeet, makes a similar assumption.) The user is responsible for verifying that their latest key version from the previous time period is accurately copied to the current time period. The service thus only needs to retain the two most recent PAHDs—all earlier data can be archived or deleted.

Overall, this change means that lookups will only retrieve key updates from the current time period, which may significantly reduce lookup cost, particularly for users with frequent updates. History checks will return key versions from the current time period and the previous one. Finally, note that auditors will not need to audit the transition between time periods. We provide more details in Appendix D.

Post-compromise security. Because we generate a fresh PAHD with a fresh server secret every time period, we get a limited form of post-compromise security. In particular, if the service provider's state is revealed at some point, it will not affect the privacy of key updates from future time periods.

Queries w.r.t. different commitments. If we want to support a very high query throughput, one option (as described in Section 6), is to have multiple servers responding to oZKS queries (i.e., generating oZKS membership and non-membership proofs). However, this introduces the possibility that these servers might be slightly out-of-sync and thus answer queries w.r.t. different epochs. Note also that a PAHD lookup response actually consists of many oZKS query responses. Thus, we must consider the possibility that these oZKS query responses are distributed to different oZKS servers who respond w.r.t. different epochs. One option is to require strong consistency between servers, i.e. that they are always answering queries w.r.t. the same epoch, but this is expensive. Instead, we show in Appendix E that we can relax our PAHD to account for this.

Client caching and reducing bandwidth overhead. At the end of Section 4, we discuss how OPTIKS-core makes storage and efficiency improvements that sacrifice some of the efficiency and privacy of lookups. We now describe some improvements that enable us to improve our lookup costs.

First, we observe that when a client performs a lookup, the client can record the latest version number; on subsequent lookups for the same key, the client only needs to retrieve membership/non-membership proofs for subsequent versions. The append-only property guarantees that the earlier versions will still be in the data structure. For example, if the client has already performed a lookup for a contact's key in the current time period and that key has not changed since, then the client only needs to retrieve and verify a single non-membership proof. If only a single key has been added since then, then the client only needs to check a membership proof for the latest key version and a non-membership proof.

The above cases indicate an efficiency improvement over the lookup protocols of SEEMless and Parakeet, which require always checking three membership/non-membership proofs. We note that for new lookups with many key versions, our algorithm remains more expensive; however we conjecture this is an outlier case, especially given that lookups return key versions only for the current time period. Clients could thus cache the most recent version numbers for their most frequent contacts and extend similar savings to history checks.[3]

For the second optimization, we note that our lookup as described in the core protocol requires sending *all* of the user's previous public keys in order to check membership proofs, which increases the bandwidth required for lookups. We can avoid this by modifying our oZKS primitive so that it checks membership proofs without also verifying the associated value. For a lookup the client just needs to know the current key and that the server stored prior versions of the key; knowing the values of the old keys is unnecessary. This would mean that lookups could send the membership proofs for old key versions without needing to send their associated values, reducing bandwidth.

These optimizations also reduce the leakage of lookups, since only the most recent value of the key and the epoch of addition for the new versions to be checked need to be leaked.

[3] ELEKTRA [15] does some similar client caching; in that case the goal is to reduce the signature verification costs as well as the communication. However, to implement this caching for ELEKTRA, the client must at least cache the set of keys currently authorized to sign updates, hence it would require more client storage than the caching in OPTIKS-ext.

Account decommissioning. When a user stops using the system, they will presumably no longer be auditing their key history. We would still like to make sure that a malicious service provider cannot replace their key and impersonate them in future communications. To do this, we add an additional oZKS[4] which stores the usernames that have been decommissioned. This oZKS will not be reset at each time period; instead, the service provider will continue adding to the same oZKS throughout.[5] This means our storage will need to grow with the number of decommissioned accounts, but this growth will be much slower than the total number of key updates. A lookup for a key will return the usual oZKS proof and an additional proof that the associated username is not in the decommissioned-account oZKS. When the user requests that their account be decommissioned, we add their username to the decommissioned-account oZKS, and return a membership proof when this is done. See Appendix D for more details.

Trade-offs of account decommissioning. Note that account decommissioning does have some significant trade-offs. Once a user decommissions their account, this would be irreversible (by design, since otherwise the server could potentially “reinstate" the account without the user’s knowledge and use it to impersonate the user). This also means that there is no way for usernames to be reused – if the username is a phone number for example, and the user gives up that phone number, they could decommission their account, but if that number were given to another user they would be unable to use it for an account. This is somewhat by design – allowing account identifiers to be transferred between users significantly complicates both the transparency and the privacy guarantees.

In settings where account reuse is determined to be extremely important, we could modify our system to allow that at a cost of requiring some out-of-band checks and giving up some privacy. We describe in Appendix D how this could be done, specifically in combination with our proposals for supporting multiple usernames as described below.

Supporting multiple devices and usernames. While our protocol thus far has assumed that each user has a single client device with a static username that can be mapped to their device’s public key by the server, in practice it is often the case that a user will have multiple devices they wish to use with the same account. Furthermore, a user may wish to change the usernames associated with their account, *e.g.* if they use multiple email addresses, they may wish to associate an additional email with their account.

Because we want a single account corresponding to all of these usernames, it might seem like it makes more sense to index the user accounts based on an internal user id. However, this presents a serious problem for transparency, since users will have no way of knowing whether the internal user ids they are given are correct. It is crucial that these usernames are human-readable and human-memorable identifiers, such as phone numbers or email addresses, which the users can share with each other out-of-band. To understand why, let us consider a toy example. Say, both Alice and Bob are registered with the server with their usernames `alice` and `bob`, respectively. But, the (malicious) server tells Alice that Bob’s username is `bobfake` and tells Bob that Alice’s username is `alicefake`. The server can maintain 4 accounts now: `alice`, `bob`, `alicefake`, and `bobfake`. Since Alice will monitor her own account (`alice`) and Bob will monitor the account he thinks belongs to Alice (`alicefake`), no inconsistency will ever be caught by the KT system. In other words, a KT system can only ensure that a consistent view of the history of the key evolution is maintained for each username and that the username is unique within the system.

To meaningfully translate this guarantee for users, it is of paramount importance that the users know each other by the correct usernames. However, many E2EE communication systems have an internal immutable representation of a username (such as a Uniquely Universal Identifier, or UUID) and this is what they use to index the directory. This UUID is different from the public-facing username [16]. It is meaningless to expose these UUIDs directly to human users. Moreover, many services may choose to allow a user to pick multiple public-facing usernames that internally represent the same UUID (such as multiple email addresses). We argue that KT systems should aim for the stronger goal of providing security no matter which username a user’s contact chooses for key lookups. Systems in deployment have acknowledged this as an issue [4] as well but no prior work exists integrating this into a KT system.

To address this, we change our key-update PAHD to map device ids to public keys and add two additional data structures. The first is an oZKS[6] (called username oZKS) that maps each username to its associated user id, and the other is a PAHD (called device-list PAHD) that maps each user id to a list of its associated device ids. This separation also allows a user to have multiple usernames tied to the same account in a straightforward way.

In response to a lookup for a particular username, the service will return the corresponding user id and a proof w.r.t. the username oZKS, the list of devices and a proof w.r.t. the device-list PAHD, and the current public keys for each device[7] along with proofs according to the key-update PAHD. This provides the desired transparency and has the advantage that changing one device’s public key, adding/removing a de-

[4] We only need an oZKS, not a full PAHD, as we do not want entries in this datastructure to change once they have been added.

[5] Note that this reduces the PCS in that an attacker who gains the secret will be able to check whether accounts have been decommissioned, and this ability will persist even when we move to a new time period.

[6] As discussed above, we assume that usernames are not reused, and that once a username is associated with a particular user, that never needs to change. See Appendix D for discussion of an alternative.

[7] Or if the Lookup specifies a particular device, it can just return the current key and proof for that device. In either case, it will return the list of devices and proof w.r.t. the device-list PAHD.

vice, or adding a username requires an update to only a single oZKS entry. Note that if we want to combine this with the account decommissioning discussed above, we would be concerned with the case where the user wants to decomission his entire account and no longer do any monitoring. In this case we would user a decommissioning PAHD which stores the user ids corresponding to decommissioned accounts. When combined with time periods, we would want the username oZKS to be persistent and not regenerated every time period. See Appendix D for more details.

Privacy analysis of OPTIKS-ext. We assume that the update proofs are publicly available. The leakage in our system comes from the two building blocks: oZKS and PAHD. oZKS updates reveal the number of items added each epoch. an oZKS query for an item reveals when that item was added, or (in case of non-membership) allows the user to recognize if that item is added in the future. PAHD updates reveal the combined total number of items added or updated each epoch. A PAHD query (with the second bandwidth optimization above) for an item reveals the item's current value, and the epochs at which any previous updates where made. It will also allow the user to recognize if/when that item is added later.

Based on this our OPTIKS-ext reveals the following information: the system reveals the number of decommissioned users in the system at every epoch (from the decomissioning oZKS), and the number of usernames, users, and devices in the system at the beginning of each time period (from the username oZKS, device-list PAHD, and key-update PAHD, respectively). For each epoch it also reveals the total number of keys updated/added, the number of device lists modified, and the number of usernames added (from the key-update PAHD, device-list PAHD, and username oZKS respectively). In addition, (from the PAHD query functionality and leakage) when Bob queries for Alice's key, he learns 1) how many devices she has, and what their current public keys are, and 2) any changes to her device list or public keys that have occurred since the beginning of the current time period. He will also be able to tell when she next updates her device list and when she next updates each of her keys if those changes happen during the current time period (again from PAHD query leakage), and recognize if/when she decomissions her account (from the decommisioning oZKS). He will learn nothing about updates to Alice's key or device list that occur outside of the current time period.

This is incomparable to the leakage in systems like SEEMless or Parakeet. One the one hand, those systems only reveal information about the single preceding key update, whereas we may reveal information about multiple updates if they occur in the same time period. On the other hand, they reveal information about the preceding key update no matter how long ago it occurred, while we limit leakage to updates occurring in the same time period. We are also strictly better in terms of leakage about future updates – we reveal when the next update occurs only if it is within same time period, while SEEMless/Parakeet always reveal this information. SEEMless and Parakeet also reveal information about some additional future updates because of the complex "marker" system that they user; we do not reveal any such information.

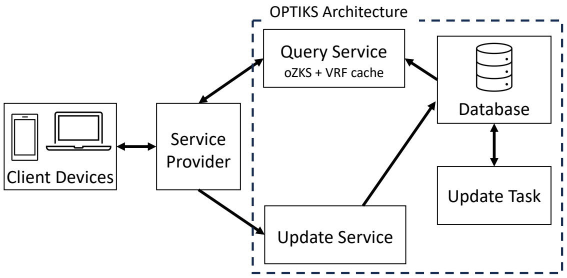


Figure 2: The system overview. The boxed area indicates the system architecture components introduced by OPTIKS.

6 System Architecture and Implementation

Prior works consider only monolithic service providers, avoiding details on how to efficiently handle both query and update requests, as well as how to scale both aspects of the system. We introduce an improved system architecture, which may be applied to other Merkle tree-based KT systems as well. Our oZKS implementation consists of roughly 5000 lines of C++. The rest of OPTIKS consists of roughly 2000 lines of C/C++, and roughly 1600 lines of ASP.NET.

System Components. We present a diagram with an overview of the OPTIKS system and architecture in Figure 2. At a high level, we physically separate the query component, called the *Query Service*, from the update component, composed of the *Update Service* and *Update Task*. This approach has several benefits. To understand why, note that the service provider of a KT system will likely need to support a vast number of concurrent key lookups, where the same key is likely to be queried multiple times within a short interval (*e.g.*, if a new group call or message thread is started). However, we expect that there will be far fewer update requests, which are only handled at epoch boundaries. Our split architecture design has the benefit that it can scale each component separately, responding more appropriately to the different needs of the different components. Since the different components benefit from different data layouts and caching mechanisms, the split architecture enables better use of compute resources, ultimately resulting in improved performance. Importantly, it avoids service interruptions for key lookups when updates take place at each epoch change. Finally, separating components in this way makes it easy to ensure data consistency across the entire system, since the Update Task, as we will describe in more detail, is the only component that performs database transactions. Each component can be separately scaled up through multiple parallel threads, processes, or VMs, as needed.

In more detail, the Query Service and Update Service are implemented as web services providing REST APIs for key lookups and updates, respectively. The Update Task runs periodically to update data and write required changes to the database. When the Service Provider receives key lookup or update requests, it calls the Query Service or Update Service API, respectively, on behalf of client devices.

Next, we go over details of the oZKS and VRF cache, as the design of these includes most novel aspects and has a significant impact on the other components as well. In Appendix F, we provide more details on the Query Service, Update Service, and Update Task.

oZKS and VRF cache. Recall that the core building block for OPTIKS is a PAHD, which itself is built from an oZKS. (See Appendix B for a detailed description.) We implement the PAHD as a combination of the oZKS primitive and logic embedded in the Query Service, Update Service, and Update Task. We implement the oZKS as a C++ library, using BLAKE2 [3] as our cryptographic hash function.

One of the important components for constructing the oZKS is a verifiable random function (VRF), defined in more detail in Appendix A. In short, each key that needs to be stored is added as a label-value pair to the oZKS. The label is computed using the VRF so that all labels appear random. This means that much of the computational overhead of the system will be from VRF and VRF proof computations, which are expensive public key operations. Thus, a crucial challenge of a scalable KT system is handling the ever increasing load from VRF operations.

OPTIKS addresses this issue by integrating a built-in VRF cache for the oZKS to store recently used VRF values and proofs for fast repeated access. Looking ahead, this feature enables far higher key lookup throughput for the key directory. We implement the VRF by adapting the IRTF internet draft ECVRF [11] to use the fast Four$\mathbb{Q}$ curve [10] that allows for an extremely fast hash-to-curve implementation and variable-base scalar multiplication, as discussed in [9].

The Query Service and the Update Task both hold local copies (full or partial) of the oZKS data. However, these components interact with the data in very different ways. In particular, the Query Service needs to support quick horizontal scaling without requiring large database reads, and running instances must be able to quickly apply targeted updates to their internal data structures. The Update Task needs to be optimized for updating the entire oZKS data structure according to pending update requests. OPTIKS thus optimizes the oZKS for each component by running in one of two modes, one tailored to the demands of lookups and the other for updates. This ability to customize the oZKS is another benefit of our split architecture approach. The two oZKS modes are:

- *Stored mode.* In this mode, the Merkle tree nodes are held in a customizable storage system, *e.g.*, a hash table in memory or a database with a memory cache. While this mode is slower and has a higher memory overhead, a major benefit of this implementation is that updates to specific nodes can be easily retrieved from the storage as needed. Thus, the oZKS instance running in the Query Service uses stored mode to enable fast and flexible updates.
- *Linked mode.* Here, the Merkle tree nodes are all allocated in memory in a *linked tree*, which allows for very fast queries and updates. The nodes can still be mapped to storage, but partial updates to the linked tree are difficult to implement, making this approach unsuitable for the Query Service. Instead, our Update Task runs the oZKS instance in linked mode to leverage fast updates.

Our oZKS implementation includes a flexible storage mechanism that enables integration with almost any desired storage back-end. We instantiate this with a Microsoft SQL Server database with an adjustable in-memory cache. The description of the tables used in our implementation is in Appendix G.

7 Performance Evaluation

In this section we discuss the performance of our implementation. Since the oZKS forms the core data structure which commits to keys, we first measure benchmarks of its performance in isolation. Next, we evaluate the full OPTIKS system and then compare our performance to related prior work.

In our evaluation, we wish to answer the following questions about OPTIKS:

- *oZKS costs*: what are the computation costs and memory overhead for lookups and updates of the oZKS?
- *Lookup costs*: what is the maximum number of queries per second OPTIKS can support?
- *Update costs*: what is the maximum number of key updates per second OPTIKS can support? What is the average time required to create a new epoch?
- *Comparison*: how does the performance of OPTIKS compare with prior related systems?

7.1 oZKS Benchmarks

We first measure performance benchmarks of the oZKS in isolation. This includes microbenchmarks of VRF operations, server query and update costs, and client costs. The results are presented in Table 1.

Experimental setup. We run the oZKS benchmarks on an Azure `E16ads_v5` virtual machine, with 16 vCPUs @ 2.60 GHz and 128 GB of RAM. Recall from Section 6 that the oZKS runs in two modes: stored mode (for key lookups in the Query Service) and linked mode (for the Update Task). For measuring oZKS query costs, we run experiments in stored mode on a single thread. For measuring oZKS update costs,

Size	Server Query			Server Update			Client Query		Client Update	
	Memory	Throughput (10^3 queries/s)		Memory	Time	Throughput	Time	Size	Time	Size
(# keys)	(GB)	Cache hit	Cache miss	(GB)	(s)	(10^3 updates/s)	(μs)	(KB)	(μs)	(KB)
2^{20}	0.734	143	16.7	0.517	6.66	140	102.9	2.10	12.6	1.89
2^{22}	3.54	129	15.0	2.00	27.6	144	103.4	2.27	13.7	2.06
2^{24}	14.7	120	13.9	7.95	120	132	104.2	2.44	14.7	2.23
2^{26}	60.2	112	12.4	32.0	520	126	104.8	2.67	15.6	2.40

Table 1: Benchmarks for the oZKS implementation for different numbers of keys. It includes the memory and throughput for the oZKS as instantiated for the Query Service (stored mode); memory, time, and throughput for the oZKS as instantiated for the Update Task (linked mode); client overhead for oZKS query and update proof verification.

we run experiments in linked mode using 16 threads. (For details, see Appendix H.)

VRF microbenchmarks. Since the bulk of the computational overhead of the oZKS comes from VRF operations, we first measure these. The time to compute a VRF value (without a proof) is on average 20.5 μs. Computing the proof takes 47.0 μs, while verifying the proof takes 95.6 μs.

Server query costs. We measure the total memory footprint and query throughput for different oZKS sizes (*i.e.*, number of keys in the oZKS). Recall that the OPTIKS architecture includes a VRF cache to reduce the overhead costs of the server during queries. We thus measure query costs for both VRF cache hits and misses. We performed the experiment twice by querying for keys that are present and not present in the oZKS; we report numbers for the slower case (generally, for keys not present), although the difference is very small. Overall, we find that VRF cache hits enable nearly an order of magnitude greater oZKS query throughput for the server.

We also see from Table 1 that the stored mode oZKS used for queries requires more memory than that of the linked mode instantiation for updates. As explained in Section 6, despite the benefit in smaller memory overhead, linked mode offers less efficient updates to parts of the tree, as our Query Service requires. However, the stored mode oZKS memory overhead could be reduced, such as by storing only the most commonly accessed nodes in memory or dividing the service over several machines.

Server update costs. Our experiments insert new keys into the oZKS in batches of at most 1024 keys starting with an empty structure. The total number of keys added is indicated by the leftmost column in Table 1. For different oZKS sizes, we show the total memory footprint, the total time to insert all keys, and the update throughput in updates per second on 16 threads. These costs include VRF computations.

Our experiments indicate high performance for the server. Namely, with a single machine, the Query Service oZKS can process well over ten thousand (VRF cache miss) or a hundred thousand (VRF cache hit) queries per second. The Update Task oZKS can sustain a similarly high throughput. However, looking ahead, the picture changes significantly when we deploy this in the context of a full system including the overhead from database operations, the REST API, and the networking protocol.

Client costs. Finally, we measure the query and update proof verification time and data size. The experiments for querying include the time to verify the query result, which includes both the VRF proof and Merkle tree proof, on a single thread, and the size of the query response. The experiments for update include the time to verify the update proof for a single added key on a single thread and the data size of the update proof for a single added key. This is intended to measure the overhead for a client to verify that its key update was added to the system. The data sizes do not include networking protocol overhead or the time to download the proofs. We note that since the update proof results apply only to a single added key, they will scale linearly with the size of the batch inserted.

Overall, the client numbers in Table 1 indicate a nearly negligible cost from the oZKS operations. We note that a client would perform these operations only sporadically, *e.g.*, before joining an end-to-end encrypted meeting.

7.2 System Evaluation

We now evaluate a fully implemented system. We perform smaller benchmarks to enable direct comparisons with prior work as well as larger scale measurements to understand the performance of OPTIKS for realistic system loads.

Experimental setup. Our system implementation omits the Service Provider, as its role is to mainly mediate requests and implement authentication logic. We run the Query Service, the Update Service, the Update Task, and the database in Azure in the West US 3 region. We use a stress tester application running in Azure in the West US 2 region as the client.

For more technical details on the Query Service, Update Service, and Update Task implementations, see Appendix F.

Query rate. We measure the maximum query rate, *i.e.*, the maximum number of key lookups per second the Query Service could support. To test this, a small program was written that continuously sends query requests to the REST API. The number of instances of this program running simultaneously was increased until the maximum query throughput was found. We tested the maximum query rate when querying for keys with 1 version and 10 versions in their history. We also measured the average size of the key lookup response. These experiments were performed for key directory sizes ranging from 1M keys to 64M keys. The results are shown in Figure 3.

One takeaway from these experiments is that the Query Service performance is limited by networking overhead. In particular, the key lookup throughput of the full OPTIKS system for a single key version is much lower than that of just the oZKS from our benchmarks in Table 1. Another takeaway is that a key lookup for a key with many versions is less performant than that for a key with a single version. To explain this, we note that the communication cost is linear in the number of key versions and logarithmic in the size of the key directory, which adds to the overhead when querying for keys with longer histories. In practice, however, this overhead can be reduced by allowing client devices to cache previously retrieved key versions. Furthermore, the Query Service can be made more performant by scaling it horizontally to alleviate handling so many simultaneous network connections.

Update Service and Task. We set the Update Task to activate each second so that if no prior update is in progress, it takes a batch of at most 1024 pending updates from the database and starts processing them. This limits the epoch time from below to 1 second.

We measure the maximum key update rate, *i.e.*, the maximum number of key updates per second that can be supported. We also measure the average time it takes to create a new epoch. The results are in Figure 4a. They show the cost of adding keys increases logarithmically. Most of the epochs we observed took 1–5 seconds; some took longer due to unpredictable and fluctuating database response times. The longest epoch observed took 13 seconds.

We next measure the time needed by the Update Task to add 100K keys with different initial directory sizes, and how that time is spent in different operations. The results are in Figure 4b. The Update Service/Task performance is strongly limited by the database performance (compare to the oZKS update performance in Table 1). Indeed, our results show that the bulk of time is spent on database writes. For example, when the key directory has 500K keys, nearly 95% is spent in database operations. This cost is caused by the very expensive (and possibly avoidable) multi-table transactions that we used to simplify the implementation. This percentage decreases slightly when more keys are added, and generally hovers between 94–96%, which means that any improvement in the database (write) performance would almost directly translate to a performance improvement in OPTIKS.

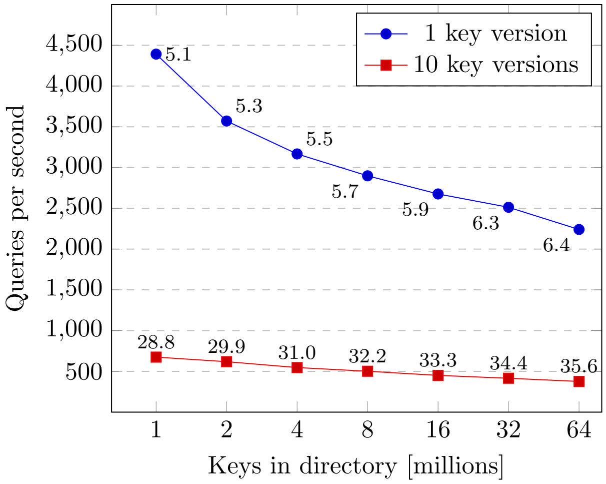


Figure 3: Key query rate as the directory grows in size from 1M to 64M, with a logarithmic scale on the x-axis. The number beside each point indicates the server response size in KB.

7.3 Comparison and Analysis

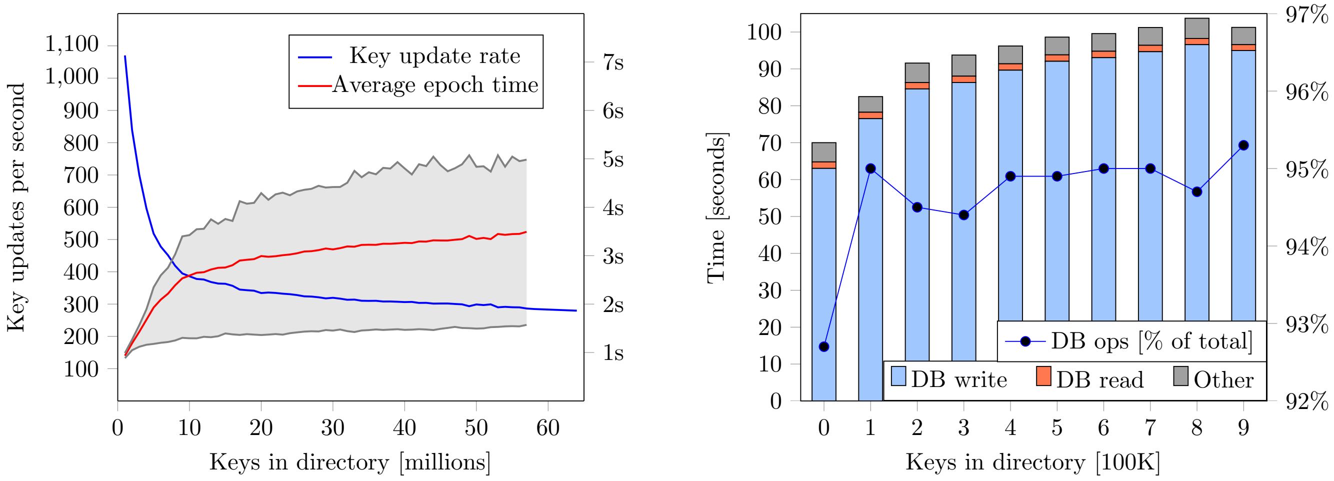
We analyze the results of our experiments and compare OPTIKS to Parakeet [17], the most relevant prior work. We then compare with Merkle^2 [12] and SEEMless [6].We omit comparison to CONIKS [18], as its client monitoring costs are prohibitively expensive for short epochs.

For the most rigorous comparison, we directly run the most recent public version of Parakeet, unless otherwise mentioned, and OPTIKS on the same VM with 16 vCPUs. In all cases, we insert into, or query from, a key directory of size 1M.

Summary. Overall, our results indicate that OPTIKS features a large performance benefit over Parakeet. This improvement comes from several core elements, most notably: (1) a more efficient protocol; (2) the split architecture of OPTIKS leveraging more optimal memory representations and more efficient parallel updates; (3) a faster VRF from using a curve that has particularly fast variable-base scalar multiplication and hash-to-curve implementation; (4) better engineering of performance-critical functions; and (5) a VRF cache that improves the Query Service performance. We next go into more detail on some of these improvement points.

VRF performance. The VRF value computation in Parakeet takes roughly 50μs, whereas OPTIKS takes only 20.5μs, so we are more than 2x faster. For proof generation, Parakeet takes 144μs, whereas OPTIKS takes 47μs; here we are more than 3x faster. In both cases, this is because we are using a curve that is much better suited for the VRF computations.

Update performance. For single-threaded execution on a hash map or hash table storage back-end, implemented both in Parakeet and OPTIKS, our update (into key directories with 1M keys) takes 71ms per 1024 items, whereas Parakeet takes



(a) Keys update rate and average epoch time when the key directory grows in size from 1M to 64M. The shaded region shows the interquartile range for the epoch time measurements.

(b) Time it takes to add 100K keys. The bar graph shows the breakdown into database operations and a category *Other*, which includes the Update Task compute time. The line graph shows the database operations as a percentage of the total time.

Figure 4: Benchmarks for the key update performance.

347ms per 1024 items. This means that, even without improvements from our split architecture, OPTIKS is already 4.8x faster for updates than Parakeet. We believe this is explained primarily by our faster VRF and a better way to compute the common prefix of two labels.

In our split architecture, we store the nodes in a linked representation for the Update Task. For a single-threaded execution this takes our time down to 28ms per 1024 items, demonstrating the benefit of our architecture. This avoids more costly lookups from a hash table or hash map and can be leveraged by most Merkle tree-based transparency systems.

Another benefit is that multi-threaded batch update is much more efficient with our linked mode. Adding multi-threading in OPTIKS takes our update time down to 7.6ms per 1024 items (3.7x improvement), whereas adding multi-threading in Parakeet results in only a 1.3x improvement at 270ms per 1024 items. We believe this is explained by our implementation of multi-threading that requires no locks, while Parakeet implements multi-threading by spawning threads that need to wait for child tasks to be completed before continuing. We note that at this point we outperform Parakeet update performance by more than 35x. The situation is evened out by the full system overhead (*e.g.*, database), but even then, comparing to the results in the Parakeet paper, we achieve in many cases up to (or over) 10x better throughput.

Lookup performance. The Parakeet paper presents no query performance numbers. If we disable our VRF cache and compare single-threaded executions, we reach a rate of roughly 32 queries/s (from a 1M size key directory). Running Parakeet ourselves shows a performance of roughly 1.3 queries/s. Thus, OPTIKS is more than 24x faster.

Note, however, that our experiments are for the most common case, where we expect users' key to not change often. Recall that with v key versions, an OPTIKS key lookup requires $v+1$ proofs (without any client-side caching), whereas Parakeet always requires 3 proofs. When a user's key does not change, a lookup proof for OPTIKS requires only 2 Merkle proofs per query, less than the 3 proofs required by Parakeet. However, when a user's key changes multiple times within a time period, OPTIKS requires more proofs than Parakeet.

Thus, a better way to compare is to observe the cost of each Merkle proof. Measuring per Merkle proof, we are 16x faster, which we hypothesize is due to architectural differences between the two systems; see below for more discussion. This means that OPTIKS is more performant for lookups than Parakeet up to $v = 3 \cdot 16 - 1 = 47$ key versions. With time periods, it is unlikely typical users will require so many versions.

Enabling the VRF cache changes the situation a lot, improving our performance (upon cache hit) further by around 9x. Multi-threading queries is trivial in both works. Note that our architecture allows the query service to keep running seamlessly while an update is being processed, whereas it is not clear at all how Parakeet would handle that.

Engineering differences. We note that although much of the key lookup performance improvements of OPTIKS over Parakeet are from protocol and system improvements, some of this gap is explained by better engineering design. In particular, the monolithic architecture of Parakeet requires it to use excessively thread-safe constructions in its implementation. Notably, the same system that supports queries (occasional locking required) is also used for updates (frequent locking required). These constructions end up being inefficient and do not provide an ideal solution in either case. Furthermore, Parakeet's implementation uses many dynamic arrays for data which creates unnecessary overhead as compared with using statically sized arrays.

Comparison to Merkle^2. Since Merkle^2 does not present full system benchmarks, we cannot directly compare the end-to-end performance of key lookups and updates, which would include database operations. Instead, we compare the append and lookup throughput in [12, Fig. 13] to the oZKS update and query throughput in Table 1. This gives a comparison of our core data structures without the extra system overhead costs.

For key updates, our reported update rates are more than 100 times that of Merkle^2. For key lookups, we note that each query to Query Service requires (with one key per user) two lookups from the oZKS. Thus, for fair comparison we divide our oZKS query rate by two to approximate our Query Service throughput. For 2^{20} keys, assuming VRF cache misses for the worst case performance, OPTIKS supports 8350 queries/s, while the Merkle^2 *Latest value* query supports fewer than 5000 queries/s.

We next compare to the approximate memory cost reported in [12, Fig. 12]. For 2^{20} keys, this is 22 GB – much larger than our 517 MB. The difference grows for larger key directories (2^{20} is our *smallest* example), as Merkle^2 has an asymptotically larger memory cost.

Finally, we compare our proof sizes and verification times to [12, Table III]. In their setting the key directory has 1 million keys; we compare this to a slightly larger 2^{20} size key directory. Merkle^2 has a very small append proof size of 42 B, whereas our update proof is significantly larger at 1.89 KB. Their lookup proof (for *Latest value* query) is 9.8 KB, whereas our proof is smaller at 4.20 KB. Here we have doubled the lookup proof size from Table 1 to account for the two oZKS lookup proofs each query to the Query Service requires.

Comparison to SEEMless. Just as for Merkle^2, SEEMless [6] also presents no full system benchmarks with which we can directly compare the full system experiments of OPTIKS. Thus, we again compare their results to our oZKS benchmarks in Table 1.

We first compare with their key update time [6, Figure 5]. For a key directory with 10 million keys, SEEMless reports an average update time of slightly under 0.3 seconds. At 2^{24} keys our average update time is roughly 7.6 milliseconds, or just 2.5% of the time of SEEMless.

For key lookups, we compare with [6, Table 2], again dividing our query rates by two for fairer comparison. At 2^{24} keys, our average query time is roughly 0.14 milliseconds, or just 2.4% of the 6.03 milliseconds reported for SEEMless. For query verification, our result of 104.2 microseconds is just 1% of the 10.51 milliseconds reported in [6, Table 2].

These performance differences are much larger than the asymptotic differences indicated in Figure 1 would suggest, and is generally explained by our more efficient implementation. In particular, our VRF operations are between 35–63 times faster than in SEEMless due to the more efficient elliptic curve we use and engineering differences in the VRF implementation itself. This has a huge impact, since a single VRF operation in SEEMless is reported to take between 1.3–3.4 milliseconds – a significant fraction of the total time. Note that the machine used in [6] was running at a slightly lower clock rate than ours (2.30 GHz vs. 2.60 GHz), but had more vCPUs. SEEMless was implemented in Java, whereas our implementation is mostly in C++.

In SEEMless, for 10 million keys, the authors report an average query response size of 8.40 KB. At 2^{24} keys our average query size is 4.88 KB, or 58% of that. Again, here we have doubled the lookup proof size from Table 1 to account for the two oZKS lookup proofs each query to the Query Service requires.

8 Related Work

We have already discussed how our KT system OPTIKS compares with Parakeet [17] and SEEMless [6].

Merkle^2 [12] is another KT system and is currently under consideration for standardization by the IETF working group on KT [19], so we briefly compare its protocol to OPTIKS. However, we emphasize that Merkle^2 cannot be truly compared to OPTIKS because their assumptions make it unsuitable for our use case, and it also lacks the strong privacy guarantees required for KT. In particular, [12] strongly relies on owner signing (and long term, non-resettable) signing keys to build a KT system. Since the fundamental goal of a KT system is building a transparent PKI for client keys, basing it on an external PKI does not serve the purpose.

At a high level, Merkle^2 trades off small update proof costs for large storage costs, while we opt for much smaller storage costs and larger update proof costs. We believe ours is the right trade-off because the large storage costs of Merkle^2 prove unscalable in practice, while auditing our update proofs are still practical even at large scale. We confirm this via experimental comparisons in Section 7.3 as well as offer a more comprehensive asymptotic comparison in Appendix I.

Now, we briefly describe the other KT systems from the literature. Keybase [13] was originally designed as an alternative to PGP key distribution and did not target privacy as a goal. CONIKS [18] was the first academic proposal for a KT system. The efficiency and privacy guarantees of CONIKS were improved in SEEMless [6]. VeRSA [22] and Verdict [24] are other KT systems that use SNARKs and RSA accumulators instead of Merkle trees, making them more expensive to deploy in practice. They also do not target privacy as a goal and so leak update patterns of users. Chen *et al.* [8] was the first paper that introduced Post Compromise Security (PCS) for the underlying building block of our primitive: *ordered Zero-Knowledge Sets* (oZKS). We describe how we achieve a limited form of PCS security in OPTIKS-ext in Section 5.

In another recent work, ELEKTRA [15] adds privacy and formal analysis to the Keybase approach. The result is a system that provides stronger security in the multi-device setting by requiring that each change to a user's set of devices be signed by another of their devices; this provides stronger security against a malicious server, as long as the user has multiple devices. ELEKTRA also adds PCS security by using the rotatable zero-knowledge set from [8] mentioned above. (As mentioned in Section 5, OPTIKS provides only a limited form of PCS.) ELEKTRA provides formal definitions and proofs for the entire system, and uses a stronger (extractability) form of soundness definition compared with prior work. However, with the exception of multi-device support, it does not consider any of the issues we address in Section 5. And since the aim of the ELEKTRA implementation was an academic prototype rather than a scalable implementation, they did not consider the architectural optimizations that we discuss in Section 6. For example, their lookup query service reads from the database on every query (as opposed to using an in-memory cache), resulting in dramatically slower query times compared to OPTIKS.

At an algorithmic level, because ELEKTRA adopts the Keybase approach, each lookup for a username returns the full linear history of changes to that username's keys, which is similar to our lookup approach. As a result, similar to our construction, ELEKTRA also avoids the need for a second tree to provide freshness. On the other hand, it incurs some additional storage and computation costs from the need to store and verify public keys and signatures.

9 Acknowledgements

We thank Vadim Eydelman, Kevin Lewi, Harjasleen Malvai and Antonio Marcedone for their helpful feedback and comments. We also thank our anonymous reviewers and shepherd who helped us improve the quality of the paper significantly. The first author would like to acknowledge support from the National Science Foundation under award CNS-2120651.

References

[1] Advancing iMessage security: iMessage contact key verification. https://security.apple.com/blog/imessage-contact-key-verification/ (accessed: 2024-01-08).

[2] What is key transparency? https://proton.me/support/key-transparency (accessed: 2024-01-08).

[3] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, Lecture Notes in Computer Science. Springer, 2013.

[4] Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Armin Namavari, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, and Alex Stamos. E2e encryption for zoom meetings. White Paper – Github Repository zoom/zoom-e2e-whitepaper, Version 4.0, https://github.com/zoom/zoom-e2e-whitepaper/blob/v4/zoom_e2e.pdf (accessed: 2023-06-03), 2023.

[5] Philippe Camacho, Alejandro Hevia, Marcos A. Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In *Information Security, 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings*, Lecture Notes in Computer Science. Springer, 2008.

[6] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security CCS*. ACM, 2019.

[7] Melissa Chase, Alexander Healy, Anna Lysyanskaya, Tal Malkin, and Leonid Reyzin. Mercurial commitments with applications to zero-knowledge sets. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, 2005.

[8] Brian Chen, Yevgeniy Dodis, Esha Ghosh, Eli Goldin, Balachandar Kesavan, Antonio Marcedone, and Merry Ember Mou. Rotatable zero knowledge sets: Post compromise secure auditable dictionaries with application to key transparency. In *Advances in Cryptology - ASIACRYPT 2022*, Cham, 2022. Springer International Publishing. Full version: https://eprint.iacr.org/2022/1264.

[9] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1135–1150, 2021.

[10] Craig Costello and Patrick Longa. Fourq: four-dimensional decompositions on a q-curve over the

mersenne prime. Cryptology ePrint Archive, Report 2015/565, 2015. https://eprint.iacr.org/2015/565.

[11] Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Včelák. Verifiable Random Functions (VRFs). Internet-Draft draft-irtf-cfrg-vrf-15, Internet Engineering Task Force, 2022. Work in Progress.

[12] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle^2: A low-latency transparency log system. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 285–303. IEEE, 2021.

[13] Keybase.io. Keybase chat. https://book.keybase.io/docs/chat (accessed: 2022-08-03).

[14] Sean Lawlor and Kevin Lewi. Deploying key transparency at WhatsApp. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/ (accessed: 2023-06-02).

[15] Julia Len, Melissa Chase, Esha Ghosh, Daniel Jost, Balachandar Kesavan, and Antonio Marcedone. ELEKTRA: efficient lightweight multi-device key transparency. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2915–2929. ACM, 2023.

[16] Rohan Mahy. More Instant Messaging Interoperability (MIMI) message content. Internet-Draft draft-mahy-mimi-content-02, Internet Engineering Task Force, March 2023. Work in Progress.

[17] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean F. Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[18] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium, USENIX Security 2015*, pages 383–398, Washington, D.C., August 2015. USENIX Association.

[19] Alexey Melnikov and Rohan Mahy. IETF Key Transparency (draft charter). https://datatracker.ietf.org/doc/charter-ietf-keytrans/ (accessed: 2023-06-02).

[20] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-knowledge sets. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 2003.

[21] Microsoft. Ordered Zero-Knowledge Set – oZKS. https://github.com/Microsoft/oZKS (accessed: 2023-06-03), 2022.

[22] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. Versa: Verifiable registries with efficient client audits from rsa authenticated dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2793–2807, 2022.

[23] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. VeRSA: Verifiable registries with efficient client audits from RSA authenticated dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022.

[24] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency dictionaries with succinct proofs of correct operation. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2022.

[25] Eric S. Yuan. Zoom acquires keybase and announces goal of developing the most broadly used enterprise end-to-end encryption offering. https://blog.zoom.us/zoom-acquires-keybase (accessed: 2023-06-03), 2020.

A Preliminaries

Our protocol makes use of the following cryptographic primitives, which are also used by SEEMless [6] and Parakeet [17]. For each, we define the primitive and give a brief overview of the security goals each should satisfy and how we instantiate it.

Collision-resistant hash function. We say that a hash function $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^m$ is a *collision-resistant hash function* if it is difficult for an adversary to find two inputs x, y such that $x \neq y$ and $\mathsf{H}(x) = \mathsf{H}(y)$.

Append-only strong accumulator. An *append-only strong accumulator* (aSA) is a primitive used to commit to a collection of label-value pairs and was introduced in [6] as an extension of strong accumulators [5] that was utilized as part of the SEEMless protocol. It is composed of the following algorithms:

- ▷ $(\mathsf{com}_0, \mathsf{D}_0, \mathsf{st}_0) \leftarrow \mathsf{aSA.CommitDS}(1^\lambda, \mathsf{D})$: Given datastore $\mathsf{D} = \{(l_i, v_i)\}_i^n$, which is a collection of label-value pairs, output an initial commitment to D, the data to which it committed, and initial state.
- ▷ $(v, \pi) \leftarrow \mathsf{aSA.Query}(\mathsf{st}_t, \mathsf{D}_t, \mathsf{com}_t, l)$: Given the state, datastore, and commitment to the datastore for current epoch t and a label l to query, the query algorithm returns the associated value v and query proof π, which serves as either a membership or non-membership proof.
- ▷ $0/1 \leftarrow \mathsf{aSA.Verify}(\mathsf{com}_t, l, v, \pi)$: The query verification algorithm verifies that the value v returned from a query for a label l is correct according to the commitment provided.
- ▷ $(\mathsf{st}_{t+1}, \mathsf{com}_{t+1}, \mathsf{D}_{t+1}, \Pi^{\mathsf{Upd}}) \leftarrow \mathsf{aSA.UpdateDS}(\mathsf{st}_t, \mathsf{D}_t, S)$: Given the state and datastore for current epoch t and a set S of label-value pairs, update the datastore with elements in S, outputting a new commitment and datastore as well as an update proof.
- ▷ $0/1 \leftarrow \mathsf{aSA.VerifyUpd}(\mathsf{com}, \mathsf{com}', S, \Pi^{\mathsf{Upd}})$: The update verification algorithm takes in two consecutive commitments, the set S of updates between them, and the update proof and verifies the update.

This formulation is similar to previous work [6], but in that work the soundness requirement is described as being the same as for oZKS - if VerifyUpd accepts, then we are guaranteed that nothing is deleted from the set. However, there is nothing in that definition that makes any guarantees about the set S that is input to VerifyUpd. For our application, we need an additional soundness property that says that the set S provided as input to VerifyUpd actually does capture what is being added to the directory. In particular we capture this with a soundness game where the adversary produces two commitments $\mathsf{com}, \mathsf{com}'$, and a sequence of commitments and update proofs and sets between them, and wins if 1) it produces a non-membership proof for l w.r.t. com and a membership w.r.t. com', but none of the update sets contain l, or 2) it produces a proof for (l, v) w.r.t. com', but one of the update sets contained (l, v') with $v' \neq v$. We call this additional property *explicit update soundness*.

More formally, an SA scheme satisfies explicit update soundness if for all PPT $\mathcal{A}^*$, there exists a negligible function $\nu()$ such that for all $n, \lambda \in \mathbb{N}$:

$$
\begin{aligned}
\Pr\Big[(\mathsf{com}_{\mathsf{t}_1}, \mathsf{t}_1, \{(\mathsf{com}_i, S_i, \pi_i^{upd})\}_{i=\mathsf{t}_1+1}^{\mathsf{t}_2}, l, v, v', \\
\pi, \pi') \leftarrow \mathcal{A}^*(1^\lambda) : \\
\{\mathsf{aSA..VerifyUpd}(\mathsf{com}_{i-1}, \mathsf{com}_i, S_i, \pi_i^{upd})\}_{i=\mathsf{t}_1+1}^{\mathsf{t}_2} \\
\wedge\ \mathsf{oZKS.Verify}(\mathsf{com}_{\mathsf{t}_2}, l, v', \pi') \\
\wedge\Big(\Big[\mathsf{oZKS.Verify}(\mathsf{com}_{\mathsf{t}_1}, l, v, \pi) \\
\wedge(v = \perp) \wedge (v' \neq \perp) \\
\wedge(l \notin S_{\mathsf{t}_1+1} \cup \ldots \cup S_{\mathsf{t}_2})\Big] \\
\vee\Big[((l, v) \in S_{\mathsf{t}_1}) \wedge (v \neq v')\Big]\Big) \leq \nu(\lambda).
\end{aligned}
$$

Instantiation. We use the same aSA instantiation as used by SEEMless [6], which builds a Merkle Patricia Trie over the labels using a collision-resistant hash function H. At a high level, each label in D becomes a leaf node in the Patricia Trie, storing the associated value. A leaf node is hashed by computing H over the label and value. An interior node is hashed by computing H over its two child nodes. The root node then serves as the commitment to the datastore. The membership proof of a label l in the datastore is computed by providing the sibling path of the associated leaf node, while a non-membership proof of a label l' not in the datastore is computed by providing the sibling path and child nodes of the node corresponding to the longest prefix of l' in the Patricia Trie. The aSA is updated by adding the new leaves to the tree at the appropriate positions, changing the appropriate hash values, and computing the updated root hash. The update proof is the set of old hash values that have since been updated, the set of new hash values that have been added, and the set of unchanged hash values. The update is verified by re-computing the root hashes of the old and new trees and verifying that they match the commitments and verifying that the hashes that are changed are the roots of the subtrees formed by the labels in update set S.

Simulatable verifiable random function. A *simulatable verifiable random function* (sVRF) scheme is similar to a pseudorandom function except that a third-party with a public key can verify that a value was computed correctly from the associated secret key. It is referred to as simulatable because such a proof can be simulated by a hypothetical simulator (such as one controlling a random oracle). An sVRF is composed of the following algorithms:

- ▷ $(pk, sk) \leftarrow\!\!\$\ \mathsf{sVRF.KeyGen}(1^\lambda, r)$: Given randomness r, the key generation algorithm outputs a public-secret key pair.

- ▷ $y \leftarrow \mathsf{sVRF.Eval}(sk, x)$: The evaluation algorithm uses the secret key to compute value y from input x.
- ▷ $\pi \leftarrow \mathsf{sVRF.Prove}(sk, x)$: On input the secret key and value x, this algorithm proves that y is indeed the value associated to x.
- ▷ $0/1 \leftarrow \mathsf{sVRF.Verify}(pk, x, y, \pi)$: The verification algorithm uses the public key to verify that y is the value associated to x.

We expect that an sVRF scheme meets the security property of *verifiability*: for each secret key sk there exists a public key pk such that for any $y \leftarrow \mathsf{sVRF.Eval}(sk, x)$, it is possible to compute and verify proof π that verifies that y is the value corresponding to x w.r.t. sk. As described above, the sVRF should also be *simulatable* so that a hypothetical simulator can fake proof π that the output of $\mathsf{sVRF.Eval}(sk, x)$ is some arbitrarily chosen value y.

Simulatable commitment scheme. A *simulatable commitment scheme* (sCS) is composed of the following algorithms:

- ▷ $\mathsf{com} \leftarrow \mathsf{sCS.Commit}(1^\lambda, m\ ;\ r)$: The commitment algorithm takes as input the security parameter λ and randomness r and produces the commitment over message m.
- ▷ $\tau \leftarrow \mathsf{sCS.Open}(m, r, \mathsf{com})$: Given the message m, randomness r, and associated commitment com, this algorithm outputs the decommitment (or opening) value τ.
- ▷ $0/1 \leftarrow \mathsf{sCS.VerifyOpen}(\mathsf{com}, m, \tau)$: This algorithm verifies that commitment com commits to message m using decommitment value τ.

We require that an sCS meets a *hiding* property, which requires that the commitment should not reveal the underlying message to which it commits. It should also meet a *binding* property, which requires that a commitment cannot be opened to two different values. Furthermore, we require that an sCS should be *simulatable* so that there exists a hypothetical simulator (such as one controlling a random oracle) that could form commitments that could later be opened to arbitrary message m.

Instantiation. We construct an sCS using a collision-resistant hash function H as follows: $\mathsf{sCS.Commit}(1^\lambda, m\ ;\ r)$ outputs $\mathsf{H}(m, r)$, $\mathsf{sCS.Open}(m, r, \mathsf{com})$ outputs τ, and $\mathsf{sCS.VerifyOpen}(\mathsf{com}, m, \tau)$ simply verifies that $\mathsf{H}(m, \tau) = \mathsf{com}$. We note that we model H as a random oracle.

B oZKS Definitions and Construction

An *ordered Zero-Knowledge Set* (oZKS) directly extends the notion of an append-only zero-knowledge set (aZKS) introduced in [6]. For some brief background, the aZKS primitive builds on the notion of the zero-knowledge set [7, 20] and is used as the core building block for SEEMless. It enables a potentially malicious prover to commit to a collection of (label, value)-pairs such that the prover can prove the membership or non-membership of labels in the collection and enables append-only updates to the collection of pairs. Additionally, the primitive is zero-knowledge because the commitment does not leak information about the collection of data and the proofs do not leak information about any other data in the collection.

oZKS additionally requires a strict ordering on elements inserted by attaching the epoch of insertion along with the label-value pairs and committing to this as part of the data.[8] We formalize this primitive, which we will use as the core building block of our protocol, although we introduce a slight modification that allows the oZKS to be instantiated with a random seed. This enables us to capture choosing the randomness used by the algorithm, rather than having the protocol choose this, so that the randomness can be distributed or stored (*e.g.*, among a distributed set of servers that run the oZKS).

A variant of this called the *oZKS with compaction* was introduced by [17] and was used as the core building block for Parakeet. Their definition extends that of the oZKS by also enabling secure deletions to handle the increasing storage requirement of oZKS. Another variant called the Rotatable Zero-Knowledge Set was introduced by [8]; instead of secure deletions, it enables the orthogonal guarantee of post-compromise security by restoring privacy guarantees after server compromise. We use a different (and simpler) approach to handle growing storage costs, enabling us to incorporate the original oZKS primitive in our design and simultaneously achieve some limited form of post-compromise security.

In this section, we formalize the cryptographic primitive ordered Zero-Knowledge Set (oZKS) we describe in Section 3, provide security definitions and a construction, and prove the security of our construction.

Definition 1. *An ordered Zero-Knowledge Set (oZKS) comprises the following algorithms:*

- $(\mathsf{st}, \mathsf{com}) \leftarrow \mathsf{oZKS.Init}(1^\lambda, r)$: The initialization algorithm takes as input a security parameter and random seed and outputs internal state and a commitment to the data.
- $(\mathsf{st}', \mathsf{com}', \pi^{upd}) \leftarrow \mathsf{oZKS.Update}(\mathsf{st}, S)$: The update algorithm updates the datastore with the label-value pairs in set S and outputs the updated state, the new commitment to the data, and a proof π^{upd} that the update was correct. The update algorithm may fail and output $\perp$, *e.g.*, if the input key-value pairs are malformed.
- $0/1 \leftarrow \mathsf{oZKS.VerifyUpd}(\mathsf{com}, \mathsf{com}', i, \pi^{upd})$: The update verification algorithm takes in commitments to the datastore before and after the update, and the epoch number of the former commitment, and verifies the update proof.

[8] We would like to thank Antonio Marcedone for early discussions about the notion of oZKS and its construction.

```
$\mathsf{Completeness}^{\mathcal{A}}_{\mathsf{oZKS}}$:
$r \leftarrow\!\$\, R$
$(\mathsf{st},\mathsf{com}) \leftarrow \mathsf{oZKS.Init}(1^\lambda, r)$
$\mathsf{epno} \leftarrow 0$ ; $\mathsf{St} \leftarrow [\,]$
$\mathsf{St}[\mathsf{epno}] \leftarrow (\mathsf{st},\mathsf{com},\emptyset)$
$\mathcal{A}^{\mathsf{Update},\mathsf{Query}}(\mathsf{com})$
Return 1

$\mathsf{Update}(S)$:
$(\mathsf{st},\mathsf{com},\mathsf{D}) \leftarrow \mathsf{St}[\mathsf{epno}]$
$(\mathsf{st}',\mathsf{com}',\pi^{upd}) \leftarrow \mathsf{oZKS.Update}(\mathsf{st}, S)$
If $(\mathsf{st}',\mathsf{com}',\pi^{upd}) = \perp$ and
$\nexists \mathsf{label}_i, \mathsf{label}_j \in \mathsf{D} \cup S$ s.t. $\mathsf{label}_i = \mathsf{label}_j$:
    Halt and return 0
If $\mathsf{oZKS.VerifyUpd}(\mathsf{com},\mathsf{com}',\mathsf{epno},\pi^{upd}) = 0$:
    Halt and return 0
$\mathsf{epno} \leftarrow \mathsf{epno}+1$
$\mathsf{St}[\mathsf{epno}] \leftarrow (\mathsf{st}',\mathsf{com}',\mathsf{D}\cup S)$
Return $(\mathsf{com}',\pi^{upd})$

$\mathsf{Query}(\mathsf{label})$:
$(\mathsf{st},\mathsf{com},\mathsf{D}) \leftarrow \mathsf{St}[\mathsf{epno}]$
$(\pi,\mathsf{val}',t) \leftarrow \mathsf{oZKS.Query}(\mathsf{st},\mathsf{label})$
$b \leftarrow \mathsf{oZKS.Verify}(\mathsf{com},\mathsf{epno},\mathsf{label},\mathsf{val}',t,\pi)$
If $(\mathsf{label},\cdot) \notin \mathsf{D}$: $\mathsf{val} \leftarrow \perp$
Else: $\mathsf{val} \leftarrow \mathsf{D}[\mathsf{label}]$
If $\mathsf{val}' \neq \mathsf{val}$ or $b = 0$:
    Halt and return 0
Return $(\pi,\mathsf{val},t)$
```

```
$\mathsf{oZKS\text{-}PRIV\text{-}REAL}^{\mathcal{A}}_{\mathsf{oZKS}}$:
$r \leftarrow\!\$\, R$
$(\mathsf{st},\mathsf{com}) \leftarrow \mathsf{oZKS.Init}(1^\lambda, r)$
$\mathsf{epno} \leftarrow 0$ ; $\mathsf{St} \leftarrow [\,]$
$\mathsf{St}[\mathsf{epno}] \leftarrow (\mathsf{st}_0,\mathsf{D}_0)$
$b \leftarrow\!\$\, \mathcal{A}^{\mathsf{Update},\mathsf{Query}}(\mathsf{com})$
Return $b$

$\mathsf{Update}(S)$:
$(\mathsf{st},\mathsf{D}) \leftarrow \mathsf{St}[\mathsf{epno}]$
If $\exists(\mathsf{label},\cdot) \in S$ s.t. $(\mathsf{label},\cdot) \in \mathsf{D}$:
    Return
$(\mathsf{st}',\mathsf{com}',\pi^{upd}) \leftarrow \mathsf{oZKS.Update}(\mathsf{st}, S)$
$\mathsf{epno} \leftarrow \mathsf{epno}+1$
$\mathsf{St}[\mathsf{epno}] \leftarrow (\mathsf{st}',\mathsf{D}\cup S)$
Return $(\mathsf{com}',\pi^{upd})$

$\mathsf{Query}(\mathsf{label})$:
$(\mathsf{st},\mathsf{D}) \leftarrow \mathsf{St}[\mathsf{epno}]$
$(\pi,\mathsf{val},t) \leftarrow \mathsf{oZKS.Query}(\mathsf{st},\mathsf{label})$
Return $(\pi,\mathsf{val},t)$
```

```
$\mathsf{oZKS\text{-}PRIV\text{-}IDEAL}^{\mathcal{A},\mathcal{S}}_{\mathsf{oZKS}}$:
$\mathsf{com} \leftarrow\!\$\, \mathcal{S}_{\mathsf{Init}}(1^\lambda)$
$\mathsf{epno} \leftarrow 0$ ; $\mathsf{D} \leftarrow \emptyset$
$b \leftarrow\!\$\, \mathcal{A}^{\mathsf{Update},\mathsf{Query}}(\mathsf{com})$
Return $b$

$\mathsf{Update}(S)$:
If $\exists(\mathsf{label},\cdot) \in S$ s.t. $(\mathsf{label},\cdot,\cdot) \in \mathsf{D}$:
    Return
$(\mathsf{com}',\pi^{upd}) \leftarrow\!\$\, \mathcal{S}_{\mathsf{Upd}}(\mathcal{L}_{\mathsf{Upd}}(S))$
$\mathsf{epno} \leftarrow \mathsf{epno}+1$
$\mathsf{D} \leftarrow \mathsf{D}\cup\{(\mathsf{label},\mathsf{val},\mathsf{epno}) : (\mathsf{label},\mathsf{val}) \in S\}$
Return $(\mathsf{com}',\pi^{upd})$

$\mathsf{Query}(\mathsf{label})$:
If $(\mathsf{label},\cdot,\cdot) \notin \mathsf{D}$: $\mathsf{val}, t \leftarrow \perp$
Else: $(\mathsf{val},t) \leftarrow \mathsf{D}[\mathsf{label}]$
$\pi \leftarrow \mathcal{S}_{\mathsf{Query}}(\mathsf{label}, \mathcal{L}_{\mathsf{Query}}(t,\mathsf{label},\mathsf{val}))$
Return $(\pi,\mathsf{val},t)$
```

Figure 5: **(Left)** The completeness definition for oZKS. **(Center/Right)** The privacy definition for oZKS.

- $(\pi,\mathsf{val},t) \leftarrow \mathsf{oZKS.Query}(\mathsf{st},\mathsf{label})$: The query algorithm returns the value associated to the queried label, along with the query proof π and the epoch t that the label was added. If label is not a member of the set, it returns $\perp$ for the value and epoch along with a proof of non-membership.
- $0/1 \leftarrow \mathsf{oZKS.Verify}(\mathsf{com}, i, \mathsf{label}, \mathsf{val}, t, \pi)$: The query verification algorithm verifies the value returned by a query using the proof against the current commitment com and current epoch number i.

Construction. Our oZKS construction is similar to that of the aZKS from [6], except that it stores extra data about the epochs when labels were added in order to allow for the strict ordering of elements. It uses an append-only strong accumulator (aSA), a simulatable verifiable random function (sVRF), and a simulatable commitment scheme (sCS), all of which are defined in Appendix A.

▷ $\mathsf{oZKS.Init}(1^\lambda, r)$: The algorithm first generates the sVRF key pair using $(pk, sk) \leftarrow\!\$\, \mathsf{sVRF.KeyGen}(1^\lambda, r)$. Next, it builds an aSA from an empty datastore via $(\mathsf{com}_0, \mathsf{D}_0, \mathsf{st}_0) \leftarrow \mathsf{aSA.CommitDS}(1^\lambda, \emptyset)$. It forms commitment $\mathsf{com} \leftarrow (\mathsf{com}_0, pk)$ and new state st composed of the aSA and its state, sVRF key pair, D_0, com_0, and epoch $t = 0$ and then returns $(\mathsf{com},\mathsf{st})$.

▷ $\mathsf{oZKS.Update}(\mathsf{st}, S)$: Given new datastore entries $S = \{(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)\}$, the algorithm first checks that the update would not result in a duplicate label being added and, if it would, returns $\perp$. Otherwise, the algorithm retrieves the sVRF key pair (pk, sk) and epoch t from its state, chooses random values $r_1, \ldots, r_n$, and builds update set $S' \leftarrow \{(l_1, v_1), \ldots, (l_n, v_n)\}$ where each $l_i = \mathsf{sVRF.Eval}(sk, \mathsf{label}_i)$ and $v_i = (\mathsf{com}^*_i, t+1)$ for $\mathsf{com}^*_i \leftarrow \mathsf{sCS.Commit}(\mathsf{val}_i \;;\; r_i)$. Next, it retrieves the aSA state st_t and datastore D_t from its state and updates the aSA with values from S' via $(\mathsf{st}_{t+1}, \mathsf{com}_{t+1}, \mathsf{D}_{t+1}, \Pi^{\mathsf{Upd}}) \leftarrow \mathsf{aSA.UpdateDS}(\mathsf{st}_t, \mathsf{D}_t, S')$. It forms commitment $\mathsf{com}' \leftarrow (\mathsf{com}_{t+1}, pk)$, update proof $\pi^{upd} \leftarrow (\Pi^{\mathsf{Upd}}, S')$, and new epoch $t \leftarrow t+1$, and then updates its state with the new values to form st'. Finally, it returns $(\mathsf{st}', \mathsf{com}', \pi^{upd})$.

Notice that the main difference between this construction and that of the aZKS from [6] is that the values v_i added to the aSA are composed of not just the commitment over the values val_i from the datastore but also the epoch t when they are added.

▷ $\mathsf{oZKS.VerifyUpd}(\mathsf{com}, \mathsf{com}', i, \pi^{upd})$: The algorithm parses $(\mathsf{com}_t, pk) \leftarrow \mathsf{com}$, $(\mathsf{com}_{t+1}, pk') \leftarrow \mathsf{com}'$, and $(\Pi^{\mathsf{Upd}}, S') \leftarrow \pi^{upd}$. It verifies that $pk = pk'$ and that every tuple in S' contains $i+1$ as the second element and returns 0 if not. It then returns what $\mathsf{aSA.VerifyUpd}(\mathsf{com}_t, \mathsf{com}_{t+1}, S', \Pi^{\mathsf{Upd}})$ returns.

- ▷ $\mathsf{oZKS.Query}(\mathsf{st},\mathsf{label})$: This algorithm retrieves the associated val of label from the datastore in its state as well as the sVRF key pair, computes $l \leftarrow \mathsf{sVRF.Eval}(sk,\mathsf{label})$, and computes the sVRF proof $\pi_{vrf} \leftarrow \mathsf{sVRF.Prove}(sk,\mathsf{label})$. It then queries the aSA for l to get its membership/non-membership proof via $(v,\pi_{sa}) \leftarrow \mathsf{aSA.Query}(\mathsf{st}_t,\mathsf{D}',\mathsf{com}_t,l)$, where $\mathsf{st}_t,\mathsf{D}',\mathsf{com}_t$ are retrieved from the state. If l is in the aSA, then it parses $(\mathsf{com}^*,t^*) \leftarrow v$, retrieves the associated randomness r used to commit to the value from its state, and computes the opening to the commitment for the value via $\tau \leftarrow \mathsf{sCS.Open}(\mathsf{val},r,\mathsf{com}^*)$. It forms query proof $\pi \leftarrow (l,v,\pi_{vrf},\pi_{sa},\tau)$ and returns (π,val,t^*). Otherwise, if l is not in the aSA, it forms query proof $\pi \leftarrow (l,\perp,\pi_{vrf},\pi_{sa},\perp)$ and returns $(\pi,\perp,\perp)$.

- ▷ $\mathsf{oZKS.Verify}(\mathsf{com},i,\mathsf{label},\mathsf{val},t,\pi)$: The algorithm parses the oZKS query proof $(l,v,\pi_{vrf},\pi_{sa},\tau) \leftarrow \pi$ and the oZKS commitment $(\mathsf{com}_t,pk) \leftarrow \mathsf{com}$. It verifies the sVRF proof $\mathsf{sVRF.Verify}(pk,\mathsf{label},l,\pi_{vrf})$ and the aSA proof $\mathsf{aSA.Verify}(\mathsf{com}_t,l,v,\pi_{sa})$. If $v \neq \perp$, it also parses the value v as (com^*,t^*), which are the commitment to the value stored in the oZKS and the epoch when it was inserted; verifies that $t = t^*$; and verifies the commitment to the value $\mathsf{sCS.VerifyOpen}(\mathsf{com}^*,\mathsf{val},\tau)$. Finally, if $v \neq \perp$, it verifies that $t \leq i$. If all verification succeeds, it returns 1, else 0.

Completeness. The formal experiment for completeness is presented in Figure 5. We model completeness as a game in which an adversary $\mathcal{A}$ can interact with an honest oZKS instance. The game begins by initializing an oZKS to compute an initial commitment and then running $\mathcal{A}$, which is given the initial commitment and access to stateful oracles that share state to update or query the oZKS. The goal of $\mathcal{A}$ in the experiment is to trigger some incorrect behavior of the oZKS and get the experiment to return 0 (shown in bold in the game pseudocode).

The game keeps track of the current epoch via value epno. It also keeps track of the state, commitment, and datastore for each epoch via dictionary St. The datastore D is a set of label-value pairs; we note that we abuse notation to have $\mathsf{D}[\mathsf{label}]$ return the value val such that $(\mathsf{label},\mathsf{val})$ is in D. The Update oracle enables $\mathcal{A}$ to update the oZKS with set S of label-value pairs. If Update fails, then the game checks that it was because the labels to be added were not all unique; otherwise, it halts and returns 0. The oracle also verifies that the update was successful via $\mathsf{VerifyUpd}$. The Query oracle enables $\mathcal{A}$ to lookup a label, verifies that the value returned matches the latest value recorded in D, and confirms that verification of the query proof succeeds. Otherwise, the game halts and returns 0. We measure the advantage of $\mathcal{A}$ in the completeness experiment by the probability of the experiment returning 0.

Theorem 1. *Our oZKS construction satisfies completeness.*

The above theorem is easy to see is true by inspection so we do not provide the full details.

Soundness. Our oZKS soundness definition is based on that of the aZKS primitive [6]. The goal of our definition is to capture that once a label-value pair has been added to the oZKS, a proof cannot later be computed that shows that either (1) the label is not a member of the oZKS, (2) the label is a member but with a different value, or (3) the label-value pair was added at a different epoch than when it was actually added. We highlight that goal (3) is the new security goal that oZKS soundness captures over what aZKS soundness targeted.

Specifically, the adversary produces commitments $\mathsf{com}_{\mathsf{t}_1},\mathsf{com}_{\mathsf{t}_2}$ from two epochs, a sequence of update proofs between them, and query proofs for (val,t) and (val',t') for the two epochs respectively. The adversary wins if the proofs verify and one of three cases holds: 1) the values and times are inconsistent, i.e. $\mathsf{val} \neq \perp$ and either $\mathsf{val} \neq \mathsf{val}'$ or $t \neq t'$, 2) the insertion times t,t' are inconsistent with the epochs in which the queries occurred, i.e. $t > \mathsf{t}_1$ or $t' > \mathsf{t}_2$, or 3) it appears that the value val' was inserted for this label in between the two queries, but the insertion time is inconsistent with this, i.e. $\mathsf{val} = \perp, \mathsf{val}' \neq \perp$ and $t' \leq \mathsf{t}_1$.

More formally, an oZKS scheme satisfies soundness if for all PPT $\mathcal{A}^*$, there exists a negligible function $\nu()$ such that for all $n,\lambda \in \mathbb{N}$:

$$\begin{aligned}\Pr\Big[&(\mathsf{com}_{\mathsf{t}_1},\mathsf{t}_1,\{(\mathsf{com}_i,\pi_i^{upd})\}_{i=\mathsf{t}_1+1}^{\mathsf{t}_2},\mathsf{label},\mathsf{val},\mathsf{val}',\\ &\qquad t,t',\pi,\pi') \leftarrow \mathcal{A}^*(1^\lambda):\\ &\{\mathsf{oZKS.VerifyUpd}(\mathsf{com}_{i-1},\mathsf{com}_i,i,\pi_i^{upd})\}_{i=\mathsf{t}_1+1}^{\mathsf{t}_2}\\ &\wedge\ \mathsf{oZKS.Verify}(\mathsf{com}_{\mathsf{t}_1},\mathsf{t}_1,\mathsf{label},\mathsf{val},t,\pi)\\ &\wedge\ \mathsf{oZKS.Verify}(\mathsf{com}_{\mathsf{t}_2},\mathsf{t}_2,\mathsf{label},\mathsf{val}',t',\pi')\\ &\wedge\ \Big([(\mathsf{val} \neq \perp) \wedge ((\mathsf{val} \neq \mathsf{val}') \vee (t \neq t'))]\\ &\vee[(\mathsf{val} \neq \perp) \wedge (t > \mathsf{t}_1)]\\ &\vee[(\mathsf{val}' \neq \perp) \wedge (t' > \mathsf{t}_2)]\\ &\vee[(\mathsf{val} = \perp) \wedge (\mathsf{val}' \neq \perp) \wedge (t' \leq \mathsf{t}_1)]\Big)\Big] \leq \nu(\lambda).\end{aligned}$$

Theorem 2. *Let our oZKS construction as defined above in this section be parameterized with an sVRF scheme that meets the verifiability property, a sCS scheme that meets the binding property, and an aSA scheme that meets soundness and explicit update soundness. Then our oZKS construction satisfies oZKS soundness.*

Proof. Recall that based on our construction, $\mathcal{A}^*$ must provide two commitments $\mathsf{com}_{\mathsf{t}_1} = (\mathsf{com},pk)$, $\mathsf{com}_{\mathsf{t}_2} = (\mathsf{com}',pk')$ and update proofs between them, and two query proofs $\pi = (l,v,\pi_{vrf},\pi_{sa},\tau)$ and $\pi' = (l',v',\pi'_{vrf},\pi'_{sa},\tau')$, where

1. $\mathsf{sVRF.Verify}(pk,\mathsf{label},l,\pi_{vrf})$,
   $\mathsf{sVRF.Verify}(pk',\mathsf{label},l',\pi'_{vrf})$,

$\mathsf{aSA.Verify}(\mathsf{com}, l, v, \pi_{sa})$, and $\mathsf{aSA.Verify}(\mathsf{com}', l', v', \pi'_{sa})$ all accept, and

2. either $v = \mathsf{val} = \perp$ or $v = (\mathsf{com}^*, t)$ s.t. $\mathsf{sCS.VerifyOpen}(\mathsf{com}^*, \mathsf{val}, \tau)$, and

3. either $v' = \mathsf{val}' = \perp$ or $v' = (\mathsf{com}^{*\prime}, t')$ and $\mathsf{sCS.VerifyOpen}(\mathsf{com}^{*\prime}, \mathsf{val}', \tau')$, and

4. $\mathsf{val}, t, \mathsf{val}, 't'$ satisfy the conditions in the soundness game.

We then have the following four cases.

- **Case 1**: $pk \neq pk'$.
  This is impossible since all of the update proofs must be accepting, and each VerifyUpd checks that the pk in the commitments is unchanged.

- **Case 2**: $pk = pk'$, $l \neq l'$.
  This means that the sVRF verification for label, $pk = pk'$ accepts different values, which breaks the verifiability of the sVRF scheme. We can thus directly reduce to the verifiability property of the sVRF scheme.

- **Case 3**: $pk = pk'$, $l = l'$, $\mathsf{val} \neq \perp$ and $\mathsf{val}' \neq \perp$, $\mathsf{val} \neq \mathsf{val}'$, and $\mathsf{com}^* = \mathsf{com}^{*\prime}$.
  This means that there two different openings to the same commitment, which breaks the binding property of the sCS scheme. We can thus directly reduce to the binding property of the sCS scheme.

- **Case 4**: $pk = pk'$, $l = l'$ and either (a) $\mathsf{val} = \mathsf{val}'$, (b) $\mathsf{val} \neq \mathsf{val}'$ and one of them is $\perp$, or (c) $\mathsf{val} \neq \mathsf{val}'$, neither of them is $\perp$, and $v \neq v'$. Then we can consider the game winning conditions:

  a. $(\mathsf{val} \neq \perp) \wedge ((\mathsf{val} \neq \mathsf{val}') \vee (t \neq t'))$
  Here we have 2 subcases: If $\mathsf{val}' = \perp$, then we must have a nonmembership proof for $l = l'$ with respect to $\mathsf{com}_{\mathsf{t}_2}$ and a membership proof with respect to $\mathsf{com}_{\mathsf{t}_1}$, which breaks soundness of the SA. If $\mathsf{val}' \neq \perp$, by this winning condition we know that either $(t \neq t')$ or $\mathsf{val} \neq \mathsf{val}'$; in the latter case, the since we are in Case 4, we know that $v \neq v'$. Either way, we again break soundness of the SA, this time by providing proofs for two different entries (v, t) and (v', t') associated with the same label $l = l'$.

  b. $[(\mathsf{val} \neq \perp) \wedge (t > \mathsf{t}_1)] \vee [(\mathsf{val}' \neq \perp) \wedge (t' > \mathsf{t}_2)]$
  These are both impossible by construction since our oZKS.Verify checks that the claimed insertion epoch is at most the epoch number of the current commitment.

  c. $(\mathsf{val} = \perp) \wedge (\mathsf{val}' \neq \perp) \wedge (t' \leq \mathsf{t}_1)$
  Here we again have 2 subcases: If there was no entry of the form $(l, \cdot)$ in any of the update sets S provided as part of the update proofs π_i^{upd}, then we can break the explicit update soundness property. If there was an entry of the form $(l, (\hat{v}, \hat{t}))$ for some v^* in at least one of the sets S, then it must be the case that $\hat{v} = v'$ and $\hat{t} = t'$, again by explicit update soundness. Finally, because our VerifyUpd algorithm checks the insertion epoch attached to every item in S, we have that $\hat{t}$ must correspond to the time of the update in which set S was included, and thus that $\mathsf{t}_1 < \hat{t}$. Thus we again get a contradiction.

□

Privacy. We capture the $\mathcal{L}$-privacy of an oZKS using a real-ideal world computational indistinguishability game, which is parameterized by leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Upd}}, \mathcal{L}_{\mathsf{Query}})$. An oZKS is private for the leakage function $\mathcal{L}$ if there exists a simulator $\mathcal{S} = (\mathcal{S}_{\mathsf{Init}}, \mathcal{S}_{\mathsf{Upd}}, \mathcal{S}_{\mathsf{Query}})$ such that for any adversary $\mathcal{A}$, the outputs of the experiments oZKS-PRIV-REAL and oZKS-PRIV-IDEAL presented in Figure 5 are computationally indistinguishable.

$\mathcal{A}$ is given access to two stateful oracles with shared state to update and query the oZKS. In the real world experiment oZKS-PRIV-REAL, the values returned to $\mathcal{A}$ are those generated by the actual oZKS algorithms. In contrast, in the ideal world experiment oZKS-PRIV-IDEAL, $\mathcal{A}$ receives outputs generated by $\mathcal{S}$, which receives leakage from leakage function $\mathcal{L}$. In both games for updates, if the queried set S gives an invalid update by adding a label already in the oZKS, the oracle simply returns without performing the update.

Leakage. The concrete leakage for our oZKS construction is:

- $\mathcal{L}_{\mathsf{Upd}}(S)$: outputs $|S|$ and the set Q of labels of items in this update for which there had been a previous nonmembership query
- $\mathcal{L}_{\mathsf{Query}}(t, \mathsf{label}, \mathsf{val})$: outputs val, t.

Theorem 3. *Let our oZKS construction as defined above in this section be parameterized with an sVRF scheme that meets the simulatibility property and a sCS scheme that meets the hiding property. Then our oZKS construction satisfies oZKS privacy with the specified leakage function above.*

Proof. We define the simulator $\mathcal{S} = (\mathcal{S}_{\mathsf{Init}}, \mathcal{S}_{\mathsf{Upd}}, \mathcal{S}_{\mathsf{Query}})$ as follows.

- $\mathcal{S}_{\mathsf{Init}}(1^\lambda)$: The oZKS simulator calls the sVRF simulator to generate the sVRF public key pk. Next, it builds an aSA from an empty datastore via $(\mathsf{com}_0, \mathsf{D}_0, \mathsf{st}_0) \leftarrow \mathsf{aSA.CommitDS}(1^\lambda, \emptyset)$. It forms commitment $\mathsf{com} \leftarrow (\mathsf{com}_0, pk)$ and new state st composed of the aSA and its state, pk, D_0, com_0, and epoch $t = 0$ and then returns com.

- $\mathcal{S}_{\mathsf{Upd}}(|S|, Q)$: For each label in Q, the simulator looks up the simulated sVRF output (which we refer to as the "leaf string") that was previously assigned to that label. For the remaining $|S|-|Q|$ number of labels in the update, it chooses random leaf strings as the output of the sVRF computation and stores these in its state. It then uses the sCS simulator to form the commitments com_i^* and forms aSA values $v_i = (\mathsf{com}_i^*, t+1)$. The simulator now has set S' to add to the aSA. It then proceeds as the rest of the Update algorithm to call the aSA UpdateDS algorithm to update the data store and get back the new commitment com_{t+1} and update proof Π^{Upd}. It stores the set of label-value pairs from the updated aSA and then returns $((\mathsf{com}_{t+1}, pk), (\Pi^{\mathsf{Upd}}, S'))$.

- $\mathcal{S}_{\mathsf{Query}}(\mathsf{label}, \mathsf{val}, t)$: The simulator retrieves the set S_t of label-value pairs in the aSA that were added during epoch t when label was added. It also retrieves the list of labels with their associated leaf strings. If $\mathsf{val} \neq \perp$, meaning the label is actually in the set, and a leaf string has not already been chosen for label, then it picks an unused leaf string l from S_t and uses the sVRF simulator to get the VRF proof π_{vrf} that label outputs to that leaf string. Let $v = (\mathsf{com}^*, t)$ be the associated value for l stored in the aSA. It then calls the sCS simulator to get an opening τ for com^* that outputs to the expected value val. Next the simulator calls the aSA Query algorithm to get the membership proof for l, forms final proof $\pi \leftarrow (l, (\mathsf{com}^*, t), \pi_{vrf}, \pi_{sa}, \tau)$, records the leaf string used for label, and returns π. If $\mathsf{val} = \perp$, meaning the label is not a member of the set, and it has not already chosen a leaf string for label, then it chooses a random leaf string (that is not in the aSA), uses the sVRF simulator to simulate the sVRF proof that it is the correct value for the label, calls the aSA Query algorithm to get the non-membership proof for l, forms final proof $\pi \leftarrow (l, \perp, \pi_{vrf}, \pi_{sa}, \perp)$, records the leaf string used for label, and returns π. Note that recording the leaf string is so that the simulator can be consistent with future query requests.

We prove that simulator $\mathcal{S}$ satisfies the privacy definition with leakage $\mathcal{L}$ from the following sequence of game hops.

Game 0: The same as oZKS-PRIV-REAL.

Game 1: The same as Game 0, except that the commitments stored in the aSA and their associated openings are simulated. Game 1 is indistinguishable from Game 0 by the hiding property of the sCS scheme.

Game 2: The same as Game 1, except that the sVRF public key and VRF proofs are generated by the sVRF simulator and the leaf strings stored in the aSA are chosen randomly instead of computed by the sVRF scheme. Game 2 is indistinguishable from Game 1 by the simulatibility property of the sVRF scheme.

By inspection, Game 2 is identical to oZKS-PRIV-IDEAL, completing our proof. □

C Private Authenticated History Dictionary

In this section, we formalize the cryptographic primitive Private Authenticated History Dictionary (PAHD) we describe in Section 3, provide security definitions, and prove the security of our PAHD construction from Section 4.

C.1 Formal Definition

Definition 2. *A Private Authenticated History Dictionary (PAHD) is defined by the following set of algorithms:*

- $(\mathsf{st}_0, \mathsf{com}_0) \leftarrow\$ \mathsf{PAHD.Init}(r)$: The initialization algorithm takes as input random seed r and outputs the initial commitment to the empty dictionary and the initial state.
- $(\mathsf{st}_t, \mathsf{com}_t, \Pi_t^{\mathsf{Upd}}) \leftarrow\$ \mathsf{PAHD.Upd}(\mathsf{st}_{t-1}, [k_j, v_j]_j)$: The update algorithm which takes as input the current state for epoch $t-1$ and a set of key-value pairs and outputs a commitment to the updated dictionary, the update proof, and state for epoch t. The update algorithm may fail and output $\perp$, e.g. if the input key-value pairs are malformed.
- $(v, \pi) \leftarrow \mathsf{PAHD.Lkup}(\mathsf{st}_t, k)$: The lookup algorithm retrieves the value v for key k along with a membership proof π. If k is not a key in the dictionary, it returns value $\perp$ along with a proof of non-membership.
- $0/1 \leftarrow \mathsf{PAHD.VerLkup}(\mathsf{com}_t, k, v, \pi)$: The lookup verification algorithm verifies that the value returned from a lookup is correct according to the commitment provided.
- $([(v_i, t_i)]_{i=1}^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{PAHD.Hist}(\mathsf{st}_t, k)$: The history algorithm returns the set of values that key k has been assigned over time, the epochs during which each value was assigned, and the membership proofs for each key-value mapping. If k is not a key in the dictionary, it returns value $\perp$ along with a proof of non-membership.
- $0/1 \leftarrow \mathsf{PAHD.VerHist}(\mathsf{com}_t, k, [(v_i, t_i)]_{i=1}^n, \Pi^{\mathsf{Ver}})$: The history verification algorithm verifies that the set of values and epochs returned from the history algorithm is correct according to the commitment provided.
- $0/1 \leftarrow \mathsf{PAHD.Audit}(\mathsf{com}_j, \mathsf{com}_{j+1}, j, j+1, \Pi_{j+1}^{\mathsf{Upd}})$: The audit algorithm takes in two commitments, their associated epochs, and the update proof Π^{Upd} published by the server and outputs a boolean indicating whether the audit was successful.

C.2 Security Definitions

Completeness. The formal experiment for completeness is presented in Figure 6. We model completeness as a game in which an adversary $\mathcal{A}$ can interact with an honest server as a client or auditor. The game begins by initializing an empty PAHD and then running adversary $\mathcal{A}$, which is given the initial commitment and access to stateful oracles that share state to update the dictionary, lookup or check the history of

```
$\mathsf{Completeness}^{\mathcal{A}}_{\mathsf{PAHD}}$:
$r \leftarrow\!\!\$\; R$
$(\mathsf{st}, \mathsf{com}) \leftarrow\!\!\$\; \mathsf{PAHD.Init}(r)$
$\mathsf{epno} \leftarrow 0$
$\mathsf{Dir} \leftarrow [\,]$ ; $\mathsf{Com} \leftarrow [\,]$
$\mathsf{Com}[\mathsf{epno}] \leftarrow (\mathsf{st}, \mathsf{com}, \perp)$
$\mathcal{A}^{\mathsf{Update},\mathsf{Lkup},\mathsf{Hist},\mathsf{Audit}}(\mathsf{com})$
Return 1

$\mathsf{Update}([k_j, v_j]_j)$:
$(\mathsf{st}, \mathsf{com}, \cdot) \leftarrow \mathsf{Com}[\mathsf{epno}]$
$(\mathsf{st}', \mathsf{com}', \Pi^{\mathsf{Upd}}) \leftarrow\!\!\$\; \mathsf{PAHD.Upd}(\mathsf{st}, [k_j, v_j]_j)$
If $(\mathsf{st}', \mathsf{com}', \Pi^{\mathsf{Upd}}) = \perp$ and $\nexists m, n \in [j]$ s.t. $k_m = k_n$:
    **Halt and return 0**
$\mathsf{epno} \leftarrow \mathsf{epno} + 1$
$\mathsf{Com}[\mathsf{epno}] \leftarrow (\mathsf{st}', \mathsf{com}', \Pi^{\mathsf{Upd}})$
For $(k, v) \in [k_j, v_j]_j$:
    $\mathsf{Dir}[k].\mathsf{append}((v, \mathsf{epno}))$
Return $\mathsf{com}', \Pi^{\mathsf{Upd}}$

$\mathsf{Lkup}(k)$:
$(\mathsf{st}, \mathsf{com}, \cdot) \leftarrow \mathsf{Com}[\mathsf{epno}]$
$(v', \pi) \leftarrow \mathsf{PAHD.Lkup}(\mathsf{st}, k)$
$b \leftarrow \mathsf{PAHD.VerLkup}(\mathsf{com}, k, v', \pi)$
If $k \notin \mathsf{Dir}$: $v \leftarrow \perp$
Else: $(v, \cdot) \leftarrow \mathsf{Dir}[k][-1]$
If $v' \neq v$ or $b = 0$:
    **Halt and return 0**
Return $v$

$\mathsf{Hist}(k)$:
$(\mathsf{st}, \mathsf{com}, \cdot) \leftarrow \mathsf{Com}[\mathsf{epno}]$
$([(v'_i, t'_i)]_i^{n'}, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{PAHD.Hist}(\mathsf{st}, k)$
$b \leftarrow \mathsf{PAHD.VerHist}(\mathsf{com}, k, [(v'_i, t'_i)]_i^{n'}, \Pi^{\mathsf{Ver}})$
$[(v_i, t_i)]_i^n \leftarrow \mathsf{Dir}[k]$
If $[(v'_i, t'_i)]_i^{n'} \neq [(v_i, t_i)]_i^n$ or $b = 0$:
    **Halt and return 0**
Return $[(v_i, t_i)]_i^n$

$\mathsf{Audit}(j, j+1)$:
If $j \geq \mathsf{epno}$: Return
$(\cdot, \mathsf{com}_j, \cdot) \leftarrow \mathsf{Com}[j]$
$(\cdot, \mathsf{com}_{j+1}, \Pi^{\mathsf{Upd}}_{j+1}) \leftarrow \mathsf{Com}[j+1]$
$b \leftarrow \mathsf{PAHD.Audit}(\mathsf{com}_j, \mathsf{com}_{j+1}, j, j+1, \Pi^{\mathsf{Upd}}_{j+1})$
If $b = 0$:
    **Halt and return 0**
Return
```

```
$\mathsf{PAHD\text{-}PRIV\text{-}REAL}^{\mathcal{A}}_{\mathsf{PAHD}}$:
$r \leftarrow\!\!\$\; R$
$(\mathsf{st}_0, \mathsf{com}_0) \leftarrow\!\!\$\; \mathsf{PAHD.Init}(r)$
$\mathsf{epno} \leftarrow 0$ ; $\mathsf{St} \leftarrow [\,]$
$\mathsf{St}[\mathsf{epno}] \leftarrow \mathsf{st}_0$
$b \leftarrow\!\!\$\; \mathcal{A}^{\mathsf{Update},\mathsf{Lkup},\mathsf{Hist}}(\mathsf{com}_0)$
Return $b$

$\mathsf{Update}([k_j, v_j]_j)$:
If $\exists m, n \in [j]$ s.t. $k_m = k_n$:
    Return
$\mathsf{st} \leftarrow \mathsf{St}[\mathsf{epno}]$
$(\mathsf{st}', \mathsf{com}', \Pi^{\mathsf{Upd}}) \leftarrow\!\!\$\; \mathsf{PAHD.Upd}(\mathsf{st}, [k_j, v_j]_j)$
$\mathsf{epno} \leftarrow \mathsf{epno} + 1$
$\mathsf{St}[\mathsf{epno}] \leftarrow \mathsf{st}'$
Return $(\mathsf{com}', \Pi^{\mathsf{Upd}})$

$\mathsf{Lkup}(k)$:
$\mathsf{st} \leftarrow \mathsf{St}[\mathsf{epno}]$
$(v, \pi) \leftarrow \mathsf{PAHD.Lkup}(\mathsf{st}, k)$
Return $(v, \pi)$

$\mathsf{Hist}(k)$:
$\mathsf{st} \leftarrow \mathsf{St}[\mathsf{epno}]$
$([(v_i, t_i)]_i^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{PAHD.Hist}(\mathsf{st}, k)$
Return $([(v_i, t_i)]_i^n, \Pi^{\mathsf{Ver}})$
```

```
$\mathsf{PAHD\text{-}PRIV\text{-}IDEAL}^{\mathcal{A},\mathcal{S}}_{\mathsf{PAHD}}$:
$\mathsf{com}_0 \leftarrow\!\!\$\; \mathcal{S}_{\mathsf{Init}}()$
$\mathsf{epno} \leftarrow 0$ ; $\mathsf{Dir} \leftarrow [\,]$
$b \leftarrow\!\!\$\; \mathcal{A}^{\mathsf{Update},\mathsf{Lkup},\mathsf{Hist}}(\mathsf{com}_0)$
Return $b$

$\mathsf{Update}([k_j, v_j]_j)$:
If $\exists m, n \in [j]$ s.t. $k_m = k_n$:
    Return
$(\mathsf{com}', \Pi^{\mathsf{Upd}}) \leftarrow\!\!\$\; \mathcal{S}_{\mathsf{Upd}}(\mathcal{L}_{\mathsf{Upd}}([k_j, v_j]_j))$
$\mathsf{epno} \leftarrow \mathsf{epno} + 1$
For $(k, v) \in [k_j, v_j]_j$:
    $\mathsf{Dir}[k].\mathsf{append}((v, \mathsf{epno}))$
Return $(\mathsf{com}', \Pi^{\mathsf{Upd}})$

$\mathsf{Lkup}(k)$:
If $k \notin \mathsf{Dir}$: $v \leftarrow \perp$
Else: $[(v_i, t_i)]_i^n \leftarrow \mathsf{Dir}[k]$
$\pi \leftarrow \mathcal{S}_{\mathsf{Lkup}}(k, \mathcal{L}_{\mathsf{Lkup}}(k, [(v_i, t_i)]_i^n))$
Return $(v, \pi)$

$\mathsf{Hist}(k)$:
$[(v_i, t_i)]_i^n \leftarrow \mathsf{Dir}[k]$
$\Pi^{\mathsf{Ver}} \leftarrow \mathcal{S}_{\mathsf{Hist}}(k, \mathcal{L}_{\mathsf{Hist}}(k, [(v_i, t_i)]_i^n))$
Return $([(v_i, t_i)]_i^n, \Pi^{\mathsf{Ver}})$
```

Figure 6: **(Left)** The completeness definition for PAHD. **(Center/Right)** The privacy definition for PAHD.

keys in the dictionary, and audit updates. The goal of $\mathcal{A}$ in the experiment is to trigger some incorrect behavior of the PAHD and get the experiment to return 0 (shown in bold in the game pseudocode).

The game keeps track of the current epoch via value epno. It also keeps track of other values via dictionaries Dir and Com. Dir maps a key to a list $[(v_i, t_i)]_i^n$ of values and epochs it has been assigned. Com maps an epoch to a tuple of the server state, commitment, and update proof associated with that epoch. We also make use of the following notation for lists and dictionaries. For a list L, $\mathsf{L}[-1]$ represents the last element of the list and $\mathsf{L}.\mathsf{append}(v)$ represents that value v has been appended to the list. If a key k is not in dictionary D, then $\mathsf{D}[k]$ returns $\perp$.

The Update oracle enables $\mathcal{A}$ to add a set of keys and their values to the dictionary. If Update fails, then the game checks that it was because the keys to be added were not all unique; otherwise, it halts and returns 0. The Lkup oracle enables $\mathcal{A}$ to lookup key k, verifies that the value returned matches the latest value recorded in Dir, and confirms that verification of the lookup proof succeeds. Otherwise, the game halts and returns 0. Likewise, the Hist oracle enables $\mathcal{A}$ to lookup the history of key k and halts and returns 0 if either this does not match what is recorded in Dir or verification of the history proof fails. And finally, the Audit oracle enables $\mathcal{A}$ to verify the update proof between any two epochs and returns 0 if auditing fails. We measure the advantage of $\mathcal{A}$ in the completeness experiment by the probability of the experiment returning 0. A PAHD satisfies completeness if for all PPT adversaries $\mathcal{A}$, the probability that the game described outputs 0 is negligibly small.

Soundness. Our soundness definition is based on that of the Verifiable Key Directory (VKD) primitive from SEEMless [6]. The goal of our definition is to capture that, assuming the dictionary has been honestly audited at every epoch, a malicious server S^* cannot lie about the value v for a key K s.t. it is inconsistent with what is returned by a history check.

To model this, we have server S^* return a key K and the history of n values that have been assigned to K as well as the epochs at which these assignments occurred. S^* also returns the list of all commitments and update proofs starting from t_1 when K was first added to the dictionary to t_{curr}, the dictionary's current epoch, and the history proof Π^{Ver} that proves the history of updates for K. Finally, S^* returns an epoch t^*, a version number j for key k, and a value v as well as lookup proof π.

We then want to capture that if the dictionary is audited successfully from t_1 to t_{curr}, the history of K is verified successfully at epoch t_{curr} for values and epochs $[(v_i,t_i)]_{i=1}^{n}$, and a lookup is performed at epoch t^*then the following guarantees hold:

- If $t_j \leq t^* < t_{j+1}$, then a malicious server could give out a lookup proof that verifies that $v \neq v_j$ is the value for K at t^* only with negligible probability.
- If $t_n \leq t^* \leq t_{\mathsf{curr}}$, then a malicious server could give out a lookup proof that verifies that $v \neq v_n$ is the value for K at t^* only with negligible probability.
- Finally, if $t^* < t_1$, then a malicious server could give out a lookup proof that verifies that $v = \perp$ is the value for K at t^* only with negligible probability.

More formally, a PAHD scheme satisfies soundness if for all PPT S^*, there exists a negligible function $\nu()$ such that for all $\lambda \in \mathbb{N}$:

$$\begin{aligned}
\Pr\Big[&(K,t^*,[(v_i,t_i)]_{i=1}^{n},\Pi^{\mathsf{Ver}},[(\mathsf{com}_k,\Pi_k^{\mathsf{Upd}})]_{k=min(t_1,\,t^*)}^{t_{\mathsf{curr}}},\\
&\qquad j,(v,\pi)) \leftarrow S^*(1^\lambda):\\
&\qquad \mathsf{PAHD.VerLkup}(\mathsf{com}_{t^*},K,v,\pi)\\
&\wedge_{k=min(t_1,\,t^*)}^{t_{\mathsf{curr}}-1}\ \mathsf{PAHD.Audit}(\mathsf{com}_k,\mathsf{com}_{k+1},k,k+1,\Pi_{k+1}^{\mathsf{Upd}})\\
&\wedge \mathsf{PAHD.VerHist}(\mathsf{com}_{t_{\mathsf{curr}}},K,[(v_i,t_i)]_{i=1}^{n},\Pi^{\mathsf{Ver}})\\
&\wedge [[(j \in [1,n-1]) \wedge (t_j \leq t^* < t_{j+1})\ \wedge\ (v \neq v_j)]\\
&\vee[\ (t_n \leq t^* \leq t_{\mathsf{curr}})\ \wedge\ (v \neq v_n)]\\
&\vee[(t^* < t_1)\ \wedge\ (v \neq \perp)]]\\
&\wedge t_1 < \ldots < t_n \leq t_{\mathsf{curr}}\Big] \leq \nu(\lambda).
\end{aligned}$$

$\mathcal{L}$-Privacy. We capture what we call the $\mathcal{L}$-privacy of a PAHD scheme using a real-ideal world computational indistinguishability game, which is parameterized by leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Upd}}, \mathcal{L}_{\mathsf{Lkup}}, \mathcal{L}_{\mathsf{Hist}})$. A PAHD scheme is private for the leakage function $\mathcal{L}$ if there exists a simulator $\mathcal{S} = (\mathcal{S}_{\mathsf{Init}}, \mathcal{S}_{\mathsf{Upd}}, \mathcal{S}_{\mathsf{Lkup}}, \mathcal{S}_{\mathsf{Hist}})$ such that for any adversary $\mathcal{A}$, the outputs of the experiments PAHD-PRIV-REAL and PAHD-PRIV-IDEAL presented in Figure 6 are computationally indistinguishable.

$\mathcal{A}$ is given access to three stateful oracles with shared state to update the dictionary, lookup a key, or get the history of a key. In the real world experiment PAHD-PRIV-REAL, the values returned to $\mathcal{A}$ are those generated by the actual PAHD algorithms. In contrast, in the ideal world experiment PAHD-PRIV-IDEAL, $\mathcal{A}$ receives outputs generated by $\mathcal{S}$, which receives leakage from leakage function $\mathcal{L}$. To avoid trivial wins, in both games for queries to the Update oracle, the oracle first checks that all the keys to be updated are unique and, if not, returns $\perp$.

C.3 Security of PAHD from Section 4

We now formally state and prove that our PAHD construction from Section 4 meets completeness, soundness, and privacy.

Theorem 4. *Let* oZKS *be an ordered Zero-Knowledge Set satisfying oZKS completeness. Then our PAHD construction using* oZKS *satisfies completeness.*

The above theorem is easy to see is true by inspection so we do not provide the full details.

Theorem 5. *Let* oZKS *be an ordered Zero-Knowledge Set satisfying oZKS soundness. Then our PAHD construction using* oZKS *satisfies PAHD soundness.*

Proof. We show that if the PAHD adversary wins, then we can construct an adversary against the soundness security of oZKS. Recall that each proof contains the following:

- π contains membership proofs for all versions up to α and a non-membership proof for version $\alpha+1$ for epoch t^* w.r.t. com_{t^*}.
- Π^{Ver} contains membership proofs for all versions up to n (with insertion times $t_1,\ldots,t_n$), and a non-membership proof for version $n+1$ for epoch t_{curr} w.r.t. $\mathsf{com}_{t_{\mathsf{curr}}}$.

We have the following cases, where for each case we describe how the soundness of oZKS is broken.

- $t^* < t_1 \wedge v \neq \perp$: In this case, we have that π contains a valid membership proof for version $\alpha \neq 0$ and a non-membership proof for version $\alpha+1$ at query time $t^* < t_1$. Let us split this in the following cases:

1. $\alpha \in [1,n]$: This would imply the underlying oZKS has valid membership proofs for $(K|\alpha, t_\alpha, v)$ (as part of π) and $(K|\alpha, t_j, v_j)$ (as part of Π^{Ver}) where $t_\alpha < t_1 \leq t_j$, $j \in [1,n]$. This is impossible by oZKS soundness.
2. $\alpha > n$: This would imply that the underlying oZKS has a valid non-membership proof for version $n+1$ at query time t_{curr} while a valid membership proof for version $n+1$, at query time $t^* < t_{\mathsf{curr}}$. This is impossible by the soundness of oZKS as well.

- $t_n \leq t^* \leq t_{\mathsf{curr}} \wedge v \neq v_n$: In this case we have that π either contains a valid membership proof for version $\alpha \geq 1$ and a non-membership proof for version $\alpha+1$ at query epoch t^* or contains a non-membership proof for version $\alpha = 1$ (and $v = \perp$). We can split this in the following cases:

  1. $v = \perp$: In this case, we have a valid non-membership proof for $\alpha = 1$ for query epoch t^* while a valid membership proof for $\alpha = 1$ with insertion epoch $t_1 \leq t_n \leq t^*$. This contradicts oZKS soundness. When $v \neq \perp$, we have the following cases.
  2. $\alpha \in [1, n-1]$: In this case, we have a valid membership proof w.r.t. $\mathsf{com}_{t_{\mathsf{curr}}}$ for version n with insertion time t_n, but we have a non-membership proof for version n w.r.t com_{t^*}, where $t^* \geq t_n$. This contradicts oZKS soundness.
  3. $\alpha = n$: In this case, we have two valid membership proofs for version α for two different values $v \neq v_n$.This contradicts oZKS soundness.
  4. $\alpha > n$: In other words, $\alpha \geq n+1$. In this case, we have a non-membership proof for version $n+1$ at query epoch t_{curr}, while a membership proof for version $n+1$ at query epoch $t^* \leq t_{\mathsf{curr}}$. This contradicts oZKS soundness as well.

- $(j \in [1, n-1]) \wedge (t_j \leq t^* < t_{j+1}) \wedge (v \neq v_j)$: In this case, we either have that π contains a valid membership proof for version $\alpha \geq 1$ and a non-membership proof for version $\alpha+1$ at query epoch t^* or it contains a non-membership proof for version $\alpha = 1$ (and $v = \perp$). We can split this in the following cases:

  1. $v = \perp$: In this case, we have a valid non-membership proof for $\alpha = 1$ for query epoch t^*. But, as part of Π^{Ver}, there is a valid membership proof for version $\alpha = 1$ for insertion epoch $t_1 \leq t^*$. This contradicts the oZKS soundness.

     When $v \neq \perp$, we have the following cases.
  2. $\alpha \in [1, j-1]$: This means there is a non-membership proof for version $\alpha+1$ at query epoch t^* (from π). However, there is also a membership proof for version $\alpha+1 \leq j$ (by our assumption that $\alpha \in [1, j-1]$) with insertion epoch $t_{\alpha+1} \leq t_j \leq t^* < t_{\mathsf{curr}}$ provided as part of Π^{Ver} at t_{curr}. This contradicts the oZKS soundness.
  3. $\alpha = j$: In this case, for the same version α we have two valid membership proofs:one for v and one for v_j, $v \neq v_j$. This contradicts oZKS soundness.
  4. $\alpha > j$: This means we have a valid membership proof for version $j+1$ (in Π^{Ver}) with insertion epoch $t_{j+1} > t^*$. However, we also have a valid membership proof for version $j+1$ (as part of π) with some insertion epoch $t \leq t^*$. This means, we have two valid membership proofs for version $j+1$ with respect to two epoch numbers $t, t_{j+1}, t \neq t_{j+1}$. This contradicts oZKS soundness.

□

Privacy leakage. The concrete leakage for our PAHD construction is:

- $\mathcal{L}_{\mathsf{Upd}}([k_j, v_j]_j)$: outputs the number of updates j and set Q, which is the set of keys that were queried to Lkup or Hist and for which this is the first update since they were queried
- $\mathcal{L}_{\mathsf{Lkup}}(k, [(v_i, t_i)]_{i=1}^n)$: outputs $[(v_i, t_i)]_{i=1}^n$
- $\mathcal{L}_{\mathsf{Hist}}(k, [(v_i, t_i)]_{i=1}^n)$: outputs $[(v_i, t_i)]_{i=1}^n$

Theorem 6. *Let* oZKS *be an ordered Zero-Knowledge Set satisfying privacy with leakage specified in Appendix B. Then our PAHD construction satisfies privacy with the specified leakage function above.*

Proof. We define the simulator $\mathcal{S} = (\mathcal{S}_{\mathsf{Init}}, \mathcal{S}_{\mathsf{Upd}}, \mathcal{S}_{\mathsf{Query}})$ as follows. We note that the simulator stores state, which in particular includes a table T that maps keys to their version numbers. Furthermore, the simulator utilizes the oZKS simulator $\mathcal{S}^{\mathsf{ozks}}$ to simulate the underlying oZKS used by the protocol.

- $\mathcal{S}_{\mathsf{Init}}()$: The simulator initializes a new oZKS by calling the oZKS simulator via $\mathsf{com} \leftarrow \mathcal{S}^{\mathsf{ozks}}_{\mathsf{Init}}(1^\lambda)$. It initializes the epoch in its state to 0 and then returns $(\mathsf{com}, 0)$.

- $\mathcal{S}_{\mathsf{Upd}}(\mathcal{L}_{\mathsf{Upd}}([k_j, v_j]_j))$: The simulator first creates set Q', which will serve as part of the leakage it will provide to the oZKS simulator $\mathcal{S}^{\mathsf{ozks}}$. For each key $k \in Q$ (which recall is the leaked set of keys for which there had been a Lkup or Hist call since the prior update), the simulator looks up k in table T to get the key's version number n and then adds label $(k \mid n+1)$ to Q'. Then the simulator simulates the oZKS commitment via $(\mathsf{com}', \pi^{upd}) \leftarrow\!\$\, \mathcal{S}^{\mathsf{ozks}}_{\mathsf{Upd}}(j, Q')$. Finally, the simulator increments the epoch in its state and returns $((\mathsf{com} = (\mathsf{com}', \mathsf{epno}), \pi^{upd})$.

- $\pi \leftarrow \mathcal{S}_{\mathsf{Lkup}}(k, [(v_i, t_i)]_{i=1}^n)$: If k is in the table T (meaning $n > 0$), then for $i \in [n]$ the simulator creates oZKS label

$\mathsf{label} \leftarrow (k \mid i)$ and calls $\pi_i \leftarrow \mathcal{S}^{\mathsf{ozks}}_{\mathsf{Query}}(\mathsf{label}, (v_i, t_i))$ to simulate the membership proof for each key version. It then simulates the non-membership proof via $\pi_{n+1} \leftarrow \mathcal{S}^{\mathsf{ozks}}_{\mathsf{Query}}((k \mid n+1), \perp)$. It returns as the lookup proof $[(\pi_i, v_i, t_i)]^n_i$ and π_{n+1}. Otherwise, if k is not in the table, then it simulates a single non-membership proof via $\pi \leftarrow \mathcal{S}^{\mathsf{ozks}}_{\mathsf{Query}}((k \mid 1), \perp)$ and returns π. The simulator also stores in table T that the version for key k is n.

- $\Pi^{\mathsf{Ver}} \leftarrow \mathcal{S}_{\mathsf{Hist}}(k, \mathcal{L}_{\mathsf{Hist}}(k, [(v_i, t_i)]^n_{i=1}))$: This proceeds identically to $\mathcal{S}_{\mathsf{Lkup}}$ except that if $n > 0$, then the lookup proof is simply $[\pi_i]^{n+1}_{i=1}$ since the key values and the insertion epochs are explicitly returned by the algorithm.

We prove that simulator $\mathcal{S}$ satisfies the privacy definition with leakage $\mathcal{L}$ with a direct reduction to the privacy of oZKS with the leakage specified in Appendix B. In particular, we show that if there exists adversary $\mathcal{A}^{\mathsf{PAHD}}$ that can distinguish between games PAHD-PRIV-REAL and PAHD-PRIV-IDEAL, then we can construct adversary $\mathcal{A}^{\mathsf{oZKS}}$ that can distinguish between games oZKS-PRIV-REAL and oZKS-PRIV-IDEAL. We construct $\mathcal{A}^{\mathsf{oZKS}}$ as follows:

- Recall that $\mathcal{A}^{\mathsf{oZKS}}$ gets as input the initial oZKS commitment com as input. $\mathcal{A}^{\mathsf{oZKS}}$ begins by running $\mathcal{A}^{\mathsf{PAHD}}$ with input $(\mathsf{com}, 0\mathsf{epno} = 0)$. $\mathcal{A}^{\mathsf{oZKS}}$ also keeps as part of its state the version number for each key that is added and updated by $\mathcal{A}^{\mathsf{PAHD}}$ and the current epoch number epno.
- When $\mathcal{A}^{\mathsf{PAHD}}$ makes a query to Update, then $\mathcal{A}^{\mathsf{oZKS}}$ retrieves the version for each key to be updated, constructs the appropriate oZKS label for the key, and adds the label together with its value to set S. It then calls its own Update oracle on input S to get back $(\mathsf{com}', \pi^{upd})$. It increments the epno and returns $(\mathsf{com} = (\mathsf{com}', \mathsf{epno}), \pi^{upd})$ to $\mathcal{A}^{\mathsf{PAHD}}$. $\mathcal{A}^{\mathsf{oZKS}}$ also updates the version number for the keys that were updated in its state.
- When $\mathcal{A}^{\mathsf{PAHD}}$ makes a query to Lkup, then $\mathcal{A}^{\mathsf{oZKS}}$ retrieves the version number n for the key from its state and creates the appropriate oZKS label for each key version. It then queries its oracle Query with each label to get back (π_i, v_i, t_i) as the membership proofs and queries for label $(k \mid n+1)$ to get the non-membership proof. If the key is in the dictionary, it returns the latest key version v_n and the membership proofs and non-membership proof; otherwise, it returns $\perp$ as the key value and the single non-membership proof.
- For queries to Hist, $\mathcal{A}^{\mathsf{oZKS}}$ proceeds the same as for queries to Lkup except that the values and the insertion epochs for each key version is returned explicitly instead of as part of the lookup proof.

When $\mathcal{A}^{\mathsf{oZKS}}$ plays game oZKS-PRIV-REAL, then notice that $\mathcal{A}^{\mathsf{PAHD}}$ gets back outputs from the oZKS algorithms, and so this is indistinguishable to game PAHD-PRIV-REAL for $\mathcal{A}^{\mathsf{PAHD}}$. Likewise, when $\mathcal{A}^{\mathsf{oZKS}}$ plays game oZKS-PRIV-IDEAL, then notice that $\mathcal{A}^{\mathsf{PAHD}}$ gets back outputs from the oZKS simulator, and so this is indistinguishable to game PAHD-PRIV-IDEAL for $\mathcal{A}^{\mathsf{PAHD}}$. Thus, if $\mathcal{A}^{\mathsf{PAHD}}$ can distinguish between the two games, then $\mathcal{A}^{\mathsf{oZKS}}$ can distinguish between oZKS-PRIV-REAL and oZKS-PRIV-IDEAL, completing the proof. □

D Details on OPTIKS-ext Protocol

Here we describe more details to protocol extensions made for OPTIKS-ext, which were first described in Section 5.

Reducing storage via time periods. We reduce storage and improve scalability by creating a new PAHD each time period. This requires the following modifications to OPTIKS-core:

- ▷ *Update*: When the time period changes, the service will perform a separate update operation. This operation will not include any new updates, but will only serve to initialize the new time period. The service will look up the most recent public key for each client, add these values into the new PAHD, and then post the resulting commitment as the first commitment of the new time period. Subsequent updates will proceed as in the core protocol.
- ▷ *Lookup*: is as in the core system. However, note that now the PAHD only includes updates from the current time period. Since the core lookup is linear in the number of key updates that the queried user has performed, this may significantly reduce the cost, particularly for users with frequent key updates.
- ▷ *History*: We will only provide history for the current and previous time period. History checks will return the key history for the current time period as well as a key history w.r.t. the last commitment in the previous time period. The client will verify the proofs and verify that the last key in the previous time period matches the initial key in the current time period.
- ▷ *Audit*: Is as before, with the exception that the auditors do not have to audit the transition between the last commitment of one time period and the first commitment of the next.

Account decommissioning. We modify the protocol as follows to enable account decommissioning:

- ▷ *Update*: Each epoch the service will publish two commitments—a PAHD commitment to the current public key directory and history for this time period, and an oZKS commitment to the set of decommissioned usernames. Also note that when a new time period starts, the initial PAHD for the new time period does not need to include any decommissioned usernames.
- ▷ *Lookup*: If the username has been decommissioned, Lookup will return a proof that the username is included in the decommissioned oZKS. Otherwise, it will return a

proof for the current public key according to the PAHD and a proof of non-membership in the decommissioned oZKS.

- ▷ *History*: History will return the key history according to the PAHDs as before, as well as a membership/non-membership proof in the decommissioned-account oZKS. History verification will ignore any history after the epoch at which the username was added to the decommissioned-account oZKS.
- ▷ *Audit*: Will verify proofs for transitions between each pair of PAHD commitments and between each pair of oZKS commitments. In the transition between time periods, the auditors need to check that the oZKS remains unchanged.

Account decommissioning with username reuse. Here we propose a modification for the case where we want to allow usernames to be reused after they are decommissioned (e.g. if the user wants to reinstate a decommissioned account or if the username is something like a phone number which can be transferred to a different user).

First, we would replace the username oZKS with a username PAHD. Recall, our username oZKS maintains a mapping between usernames and user ids, where the user ids are the unique internal representation for a user account. Now, we will replace this with a PAHD which would store a mapping from each username to either a user id or a special NULL value indicating that this username has been removed from the user's account. All users when querying the username PAHD would be given the full history of values associated with the username and would verify that they alternate between user ids and the NULL value, thus guaranteeing that each user removes the user id before it is transferred to a new user. The client would need to be modified to show a very strong notification whenever it finds a username which has been removed and re-added. In that case the querying user should be instructed to verify out-of-band whether the queried user had decommissioned and reactivated their account and to ensure that queried the user (re)started their account after the most recent time that the username was re-added.

This does have the privacy consequence of letting the querying user and the new owner of the username learn exactly when the username was previously added and removed by other users (although not which users this corresponds to or anything about their associated devices or keys). We leave open the problem of minimizing leakage further, but note that some leakage of this sort seems to be an inevitable trade-off for allowing username reuse in this context.

Supporting multiple devices and usernames. Recall that we change our key-update PAHD to map device ids to public keys and add two additional structures: the username OZKS which maps each username to its associated user id and the device-list PAHD which maps each user id to a list of its associated device ids.

We then modify the protocol as follows:

- ▷ *Update*: Each epoch the service will publish three commitments[9]: one for the username oZKS, one for the device-list PAHD, and one for the key-update PAHD. These will reset each time a new time period starts. Assuming that usernames are not recycled, we note that a username is only ever mapped to a single user id; see the account decommissioning with account reuse discussion above for an alternative.
- ▷ *Lookup*: Lookup will return a proof for the current public key for each device according to the key-update PAHD, a proof for the list of devices according to the device-list PAHD, and a proof for the user id according to the username OZKS.[10]
- ▷ *History*: History will return the user id mapping, the history of device-lists, and the history of key updates for each device.
- ▷ *Audit*: Will verify proofs for transitions between each pair of key-update PAHD commitments, each pair of device-list PAHD commitments, and each pair of username OZKS commitments.

E PAHD Using Different Commitments

As we describe in Section 5, a PAHD lookup response is composed of many oZKS query responses. One optimization is to have multiple servers handle these oZKS responses, but it could be that the oZKS servers are slightly out of sync and thus might form proofs with respect to different commitments. In this section, we show that we can relax our PAHD formalization to allow for lookup proofs using different commitments and still achieve soundness. We first update the definition for PAHD to include these commitments and describe changes to the protocol from that described in Section 4. We then provide an updated soundness definition for this new syntax and prove that our updated algorithm meets this definition. Since the updated definitions and proofs of completeness and privacy are nearly identical to those in Appendix C, we only provide the updated soundness analysis.

Definition and Construction. We describe the changes to our PAHD definition and construction, namely for lookups and history checks. Updates and audits remain the same as in our original construction. Since oZKS membership and non-membership proofs may be w.r.t. different commitments, part of the lookup proof contains which commitments were used for the oZKS proofs. Then instead of the single latest commitment, the lookup verification algorithm takes as input all commitments up to the latest epoch t.

[9] When combined with account decommissioning as described above, this will total 4 commitments, with the additional commitment from the decommissioned-account oZKS.

[10] This does reveal some extra information to the querier in that learning the list of devices is not strictly necessary. However, it allows for an efficient solution and the privacy loss seems tolerable.

The other major change is that lookup outputs an additional value t', which represents the epoch for which the latest key version may be considered valid, since the commitments used for the lookup proof may be slightly stale. In essence, this is the epoch of the commitment for which the non-membership proof is proven. To see why, consider the following example. Bob looks up Alice's key, which is at version 5. Bob receives oZKS membership proofs for versions 1 through 5 and a non-membership proof for version 6. Let us say that the membership proof for version 5 is proven w.r.t. the commitment at epoch 10, yet the non-membership proof is computed slightly later so that it is proven w.r.t. the commitment at epoch 11. Assuming that the append-only property of the oZKS is maintained, then we can say that version 5 is also valid for epoch 11 since we know there is no version 6 at that point.

Now, conversely, consider if the membership proof for version 5 is proven for epoch 11, while the server computing the non-membership proof is slightly stale and computes this for epoch 10. Then we can only say that version 5 is valid for epoch 10 and *we cannot say anything* for epoch 11 because the non-membership proof does not specify that there is no version 6 at epoch 11. In particular, version 6 could have been added precisely at epoch 11, making version 5 slightly stale. Therefore, we specify for the user during a lookup request at which epoch the version returned may be considered valid.

- ▷ $(v, \pi, t') \leftarrow \mathsf{PAHD.Lkup}(\mathsf{st}_t, k)$: Upon receiving a lookup request for key k, the server retrieves from its state the latest version number α for k (where $\alpha = 0$ if k is not in the PAHD). If k is in the PAHD, then the server forms labels $(k \mid 1), \ldots, (k \mid \alpha)$ and calls $\mathsf{oZKS.Query}$ for each label to get back $[(\pi_i, v_i, t_i)]_i^{\alpha}$ w.r.t. latest commitment com^i at the time of retrieval. To retrieve the non-membership proof $\pi_{\alpha+1}$ for the next version of the key (or to prove that k is not in the dictionary when $\alpha = 0$), the server calls $\mathsf{oZKS.Query}$ for label $(k \mid \alpha + 1)$ w.r.t. commitment $\mathsf{com}^{\alpha+1}$. The server returns either v_α as the value for k if $\alpha > 0$ or $\perp$ otherwise.

  The server returns as its lookup proof:

  - **Correct version i is set at epoch t_i:** For each $i \in [1, \alpha]$, π_i serves as the membership proof for $(k \mid i)$ with value v_i and associated epoch t_i in oZKS w.r.t. com^i. This means the server must return $[(\pi_i, v_i, t_i)]_i^{\alpha}$ as part of the proof.
  - **Server could not have shown version $\alpha + 1$:** Proof $\pi_{\alpha+1}$ serves as the non-membership proof for $(k \mid \alpha + 1)$ in oZKS w.r.t. $\mathsf{com}^{\alpha+1}$.

  Note that as part of this proof, the server indicates which commitments were used.

  It also indicates the epoch t' associated with $\mathsf{com}^{\alpha+1}$, with respect to which, the non-membership proof was produced. Notice that t' also represents the epoch for which the key version α is valid. The server checks that $t' \geq t_{\alpha+1}$. if not, the server reruns the algorithm to generate the oZKS proofs. Similarly, if generation fails for any of the membership proofs (i.e. if the server handling that proof is sufficiently out of date that it doesn't have the latest updates to this label), then similarly we query the server again to get a more updated proof.

- ▷ $0/1 \leftarrow \mathsf{PAHD.VerLkup}([\mathsf{com}_i]_{i=1}^{t}, k, v, t', \pi)$: The lookup verification algorithm now takes as input all prior commitments up until the latest epoch t. (See remark below.)

  The client verifies each membership proof for labels $(k \mid i)$ w.r.t. com^i for $i \in [1, \alpha]$ and non-membership proof for $(k \mid \alpha + 1)$ w.r.t. $\mathsf{com}^{\alpha+1}$ via $\mathsf{oZKS.Verify}$.

  As before, the client verifies that the update epochs $t_1, \ldots, t_\alpha$ are monotonically increasing. Lastly, the client parses $\mathsf{com}^{\alpha+1}$ as (com, t), and verify $t = t'$ and that for epoch t_α when version α was added, it is true that $t_\alpha \leq t'$. This latter check verifies that key version α was added before the non-membership proof was formed.

- ▷ $([(v_i, t_i)]_{i=1}^{n}, t'', \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{PAHD.Hist}(\mathsf{st}_t, k)$: This algorithm proceeds the same as Lkup, except that in its syntax it explicitly returns all key versions rather than including them in the proof and outputs t'' as the epoch with respect to which, the history is valid.

- ▷ $0/1 \leftarrow \mathsf{PAHD.VerHist}([\mathsf{com}_i]_{i=1}^{t}, k, [(v_i, t_i)]_{i=1}^{n}, t'', \Pi^{\mathsf{Ver}})$: This algorithm proceeds identically to that of $\mathsf{VerLkup}$.

Soundness. We now present the updated soundness definition. Recall that this definition requires malicious server S^* to return values such that if a user has n key versions and a lookup was performed between versions j and $j + 1$, then the lookup returns a key version that is inconsistent with what a history check would later return. To update this definition, we now also require S^* to return epoch t', which represents the epoch for which the version returned by the lookup may be considered valid and t'', which represents the epoch for which the version returned by the history may be considered valid.

More formally, a PAHD scheme satisfies soundness if for all PPT S^*, there exists a negligible function $\nu(\cdot)$ such that for all $\lambda \in \mathbb{N}$:

$$\begin{aligned} \Pr\Big[&(K, [(v_i, t_i)]_{i=1}^{n}, \Pi^{\mathsf{Ver}}, [(\mathsf{com}_k, \Pi_k^{\mathsf{Upd}})]_{k=0}^{t_{\mathsf{curr}}}, \\ &t', t'', t^*, j, (v, \pi)) \leftarrow S^*(1^\lambda) : \\ &\mathsf{PAHD.VerLkup}([\mathsf{com}_i]_{i=0}^{t^*}, K, v, t', \pi) \\ &\wedge_{k=0}^{t_{\mathsf{curr}}-1} \mathsf{PAHD.Audit}(\mathsf{com}_k, \mathsf{com}_{k+1}, k, k+1, \Pi_{k+1}^{\mathsf{Upd}}) \\ &\wedge \mathsf{PAHD.VerHist}([\mathsf{com}_i]_{i=0}^{t_{\mathsf{curr}}}, K, [(v_i, t_i)]_{i=1}^{n}, t'', \Pi^{\mathsf{Ver}}) \\ &\wedge t' \leq t^* \\ &\wedge [[(j \in [1, n-1]) \wedge (t_j \leq t' < t_{j+1}) \ \wedge \ (v \neq v_j)] \\ &\vee[\ (t_n \leq t' \leq t'') \ \wedge \ (v \neq v_n)] \\ &\vee[(t' < t_1) \ \wedge \ (v \neq \perp)]] \\ &\wedge t_1 < \ldots < t_n \leq t'' \leq t_{\mathsf{curr}}\Big] \leq \nu(\lambda). \end{aligned}$$

Remark 1. *Here we define* VerLkup *to take all the commit-*

ments starting from the beginning for simplicity. This can be compressed to only take the a subset of those commitments needed to verify the proof. In that case the above soundness definition could be relaxed to only run Audit *starting from the first commitment used in* VerLkup. *Alternatively, if we modify the construction to form a vector commitment to the roots as in [8], we can allow the* VerLkup *to take only the most recent commitment.*

Theorem 7. *Let* oZKS *be an ordered Zero-Knowledge Set satisfying oZKS soundness. Then our PAHD construction w.r.t. different commitments using* oZKS *satisfies PAHD soundness.*

Proof. We show that if the PAHD adversary wins, then we can construct an adversary against the soundness security of oZKS. Recall that each proof contains the following:

- π contains membership proofs for all versions up to α and a non-membership proof for version $\alpha+1$ for epoch t' w.r.t. $\mathsf{com}_{t'}$
- Π^{Ver} contains membership proofs for all versions up to n and a non-membership proof for version $n+1$ for epoch t'' w.r.t. $\mathsf{com}_{t''}$.

We have the following cases, where for each case we describe how the soundness of oZKS is broken.

- $t' < t_1 \wedge v \neq \perp$: In this case, we have that π contains a valid membership proof for version $\alpha \neq 0$ and a non-membership proof for version $\alpha+1$ for epoch $t' < t_1$. Let us split this in the following cases:
  1. $\alpha \in [1,n]$: This would imply the underlying oZKS has valid membership proofs for $(K|\alpha, t_\alpha, v)$ (as part of π) and $(K|\alpha, t_j, v)$ (as part of Π^{Ver}) where $t_\alpha < t' < t_1 \leq t_j$, $j \in [1,n]$. This is impossible by oZKS soundness.
  2. $\alpha > n$: This would imply that the underlying oZKS has a valid non-membership proof for version $n+1$ for query epoch t'' while a valid membership proof for version $n+1$, with insertion epoch $\leq t'$. This is impossible by the soundness of oZKS as well since we have $t' \leq t''$.
- $t_n \leq t' \leq t'' \wedge v \neq v_n$: In this case we have that π either contains a valid membership proof for version $\alpha \geq 1$ and a non-membership proof for version $\alpha+1$ at query epoch t^* or contains a non-membership proof for version $\alpha = 1$ (and $v = \perp$). We can split this in the following cases:
  1. $v = \perp$: In this case, we have a valid non-membership proof for $\alpha = 1$ for the epoch returned by the server for query: t' while a valid membership proof for $\alpha = 1$ with insertion epoch $t_1 \leq t_n \leq t'$. This contradicts oZKS soundness.
  2. $v \neq \perp$ and $\alpha \in [1, n-1]$: In this case, we have a valid membership proof w.r.t. $\mathsf{com}_{t''}$ for version n with insertion time t_n, but we have a non-membership proof for version n w.r.t $\mathsf{com}_{t'}$, where $t' \geq t_n$. This contradicts oZKS soundness.
  3. $v \neq \perp$ and $\alpha = n$: In this case, we have two valid membership proofs for version α for two different values $v \neq v_n$.This contradicts oZKS soundness.
  4. $v \neq \perp$ and $\alpha > n$: In other words, $\alpha \geq n+1$. In this case, we have a non-membership proof for version $n+1$ at query epoch t'', while a membership proof for version $n+1$ at query epoch $t' \leq t''$. This contradicts oZKS soundness as well.
- $(j \in [1, n-1]) \wedge (t_j \leq t' < t_{j+1}) \wedge (v \neq v_j)$: In this case, we either have that π contains a valid membership proof for version $\alpha \geq 1$ and a non-membership proof for version $\alpha+1$ at query epoch t' or it contains a non-membership proof for version $\alpha = 1$ (and $v = \perp$). We can split this in the following cases:
  1. $v = \perp$: In this case, we have a valid non-membership proof for $\alpha = 1$ for query epoch t'. But, as part of Π^{Ver}, there is a valid membership proof for version $\alpha = 1$ for insertion epoch $t_1 \leq t'$. This contradicts the oZKS soundness.
  2. $v \neq \perp$ and $\alpha \in [1, j-1]$: This means there is a non-membership proof for version $\alpha+1$ for query epoch t' (from π). However, there is also a membership proof for version $\alpha+1$ with insertion epoch $t_{\alpha+1} \leq t_j \leq t'$ provided as part of Π^{Ver} . This contradicts the oZKS soundness.
  3. $v \neq \perp$ and $\alpha = j$: In this case, for the same version α we have two valid membership proofs: one for v and one for v_j, $v \neq v_j$. This contradicts oZKS soundness.
  4. $v \neq \perp$ and $\alpha > j$: This means we have a valid membership proof for version $j+1$ (in Π^{Ver}) with insertion epoch $t_{j+1} > t'$. However, we also have a valid membership proof for version $j+1$ (as part of π) with the insertion epoch $t \leq t'$. This means, we have two valid membership proofs for version $j+1$ with respect to two epoch numbers $t, t_{j+1}, t \neq t_{j+1}$ This contradicts oZKS soundness.

□

F Further Details on Query and Update Components

Query Service. The data used by the Query Service, including public keys and other metadata, is stored in the database, from which updates are retrieved to memory when the epoch changes. Holding a copy separate from that of the Update Task is important to avoid service interruptions on epoch changes. The Query Service also includes an in-memory cache for faster responses, in which it stores some subset of the oZKS and associated data. Looking ahead, in our experiments the cache is large enough to hold the entire oZKS.

Since the Query Service needs to be able to handle a high volume of queries, it is essential for it to have a low computational overhead. To aid in this, the Query Service utilizes the VRF cache that stores the most recently requested VRF proofs. The Merkle tree proofs themselves are impractical to cache, but we note that they are much faster to construct than the VRF proofs even for very large oZKS instances.

In practice, one can run an arbitrary number of Query Service instances to improve scalability. For example, each instance could serve a different subset of the oZKS labels. In this case, the different Query Services may respond with respect to slightly different epochs, since they are not guaranteed to be exactly synchronized. While we limit our experiments to running a single instance, we describe how OPTIKS may be updated to accommodate multiple Query Service instances while still maintaining transparency guarantees in Appendix E.

Experimental setup. The Query Service was implemented in two components. The first is a front-facing web server that provides the Query REST API, hosted in Azure in a P3V3 service plan (1 machine with 8 vCPUs and 32 GB of RAM). The second is a back-end component that holds the oZKS in memory. This runs in an Azure virtual machine (`E16ads_v5`, with 16 vCPUs @ 2.60 GHz and 128 GB of RAM) that runs a web server providing an internal REST API that the front-facing server calls to obtain lookup proofs.

Both Query Service components were multi-threaded by the ASP.NET runtime. They access a Microsoft SQL Server 2022 Enterprise database, running on an Azure `E16ds_v4` virtual machine with 16 vCPUs and 128 GB of RAM.

Update Service and Task. The update component of the OPTIKS architecture is responsible for handling update requests to the key directory of the system. We split this into the Update Service and Update Task, where each handles different parts of the update process. In particular, the Update Service is a REST API that receives update requests from the Service Provider and processes them by writing them to the database. It is completely independent of the oZKS. Instead, the Update Task is responsible for interfacing with the oZKS to update it. It reads the incoming updates from the database and adds them as a batch to the oZKS to form a new epoch. The updated oZKS is then saved to the database, along with additional information. The Update Task runs over scheduled intervals, which can be configured depending on the service load. The Merkle tree insert operations in the oZKS update process are parallelizable across multiple threads, allowing the Update Task to (almost) fully leverage the compute power available to it. For more details, see Appendix H.

The main reason to divide updates into these two components is because this enables better scaling of each individual component based on system requirements. For instance, the Update Service is a very simple REST API that is scaled simply by adding more servers to handle incoming requests. However, scaling the Update Task is far more challenging. Since it writes updates to the oZKS, the only way to scale the Update Task is to partition the oZKS so that different machines may handle updates to each partition.

Experimental setup. The Update Service was implemented in a web server that provides the Update REST API, hosted in Azure in a P3V3 service plan (1 machine with 8 vCPUs and 32 GB of RAM). The Update Task was implemented in a separate Azure virtual machine (`E16ads_v5`, with 16 vCPUs @ 2.60 GHz and 128 GB of RAM). The Update Service was multi-threaded by the ASP.NET runtime and the Update Task was run on 16 threads.

G Database Details

The following are the database tables used for our implementation of OPTIKS:

- The User-Versions table contains one record per client device and key version, describing the full key history information and other relevant metadata.
- The Batch-to-Update table contains one record per each pending key update. The Update Service writes these and the Update Task reads and clears them.
- The Cached-Updates table is written to by the Update Task. It contains information for the Query Service for updating its local oZKS.
- The Update-Proofs table stores update proofs as computed and stored there by the Update Task. It also holds the corresponding oZKS root commitment.
- The Tree-Nodes table ensures crash resilience of the system. It stores a complete copy of the oZKS that allows any in-memory representations to be easily rebuilt. The Update Task updates the Tree-Nodes as it finishes processing.

H Parallelizing oZKS Insertions

Insert operations into the oZKS are parallelizable. As the Merkle tree gets populated, the node labels at the top of the tree will not change. For example, if we have an empty tree and insert labels 0x0000, 0x0001, 0x8000 and 0x8001, the root will have two children with labels 0 and 1. No matter how many more labels are inserted to the tree, the labels of the children of the root will not change. We group the labels that we intend to insert into the oZKS by their first bit, and launch two threads: one that inserts the labels whose first bit is 0, and one that inserts the labels whose first bit is 1. As more labels are inserted, the tree becomes populated with more top-level nodes whose labels will not change, and we may be able the next time to launch four threads instead of two. The first part of the update process is then to check how many threads are possible to launch. Next, we group the labels to insert and insert them using their assigned threads. Finally, the top nodes are recomputed after the update threads are complete.

Using this same principle it would be possible to launch multiple Update Tasks to update different parts of the tree, if the tree was very large and could not fit in the memory of a single machine. The tree would be partitioned as it grows and its top nodes become constant. This would of course require a synchronization mechanism between the different Update Tasks to coordinate the computation of the top node hashes and publishing the root hash and append proofs. In our experiments, the tree is not that large, so we only parallelize updates within a single Update Task instance.

I Comparison of OPTIKS with Merkle^2

The design for Merkle^2 incorporates a forest of chronological Merkle trees where each key-value pair forms a leaf. Each internal node of this tree is associated with a separate prefix tree that stores key-value pairs arranged in lexicographic order that appear in the subtree rooted at the internal node.

The benefit of this design is that update proof sizes remain smaller in their structure because every epoch is an extension of those before it with nodes in the same order as before. For n key-value pairs, this means that auditing requires checking $O(\log(n))$ hashes. To minimize storage costs, however, our tree changes between epochs and internal nodes are also updated. Thus, for the audit proof in OPTIKS, auditors must download both the k nodes that have been added and the $O(k \log(n))$ roots of unchanged subtrees, which represent the nodes that have not been changed.

Nevertheless, these small update proof sizes come at the cost of significantly larger storage requirements. Merkle^2 requires $O(n^2 \log(n))$ of unoptimized storage which can be optimized to $O(n \log(n))$ storage. In contrast, OPTIKS only ever needs $O(n)$ storage. Indeed, our experimental results in Section 7 support that OPTIKS requires far less storage.

For lookup costs, if a key has ℓ versions, then the proof size for Merkle^2 is $O(\ell + \log^2(n))$ in the unoptimized case, where the user must check ℓ signatures and $\log^2(n)$ membership or non-membership proofs. Merkle^2 offers an alternative algorithm that avoids checking the ℓ signatures but this assumes that each client must have a separate master key pair which it can never lose or change. We do not view such an assumption as tenable in practice.

OPTIKS instead requires lookup proofs of size $O(\ell \log(n))$ (excluding VRF proofs, which Merkle^2 does not use because it does not target privacy). When a client has few key versions such that $\ell < \log(n)$, then our lookup cost is less than that of Merkle^2. However, we do note that if a client has a particularly large number of key updates for a given time period, then the lookup cost for OPTIKS can be more expensive than Merkle^2. Our experimental results in Section 7 measure the former case and show that our lookup proof size is indeed smaller in this case.