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, `jl3836@cornell.edu`
[2]Microsoft Research Redmond `{melissac, esha.ghosh, kim.laine, radames.cruz}@microsoft.com`

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. By itself, KT does not imply security or privacy for key management; different applications may impose different constraints on these aspects of the system.

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.

In recent years, KT has gained much traction, both in industry [2, 19] and academia [3, 4, 7, 12, 13, 16, 18]. The increased interest in KT is also evident from the recently developing IETF standardization effort [14]. Very recently, WhatsApp announced its plan to deploy KT [10], based off of the academic papers SEEMless [3] and Parakeet [12].

While there has been a rich body of literature on KT systems, most of these prior systems suffer from scalability concerns and miss out on important features that we address in the present work. To the best of our knowledge, Parakeet is the first paper tackling some of the scalability challenges that arise when moving from academic proposals to large-scale

[1]*Privacy:* The keys maintained by a user as well as their key updates can be sensitive information. Accordingly, academic and industry proposals [2–4, 10, 12, 13] and recent standardization efforts [14] 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.

deployments. While Parakeet is an important first step, in this work, we show that we can achieve the same level of security and almost the same level of privacy as Parakeet using a much simpler KT system that is highly scalable. The main scalability and deployability challenges we address are:

Improving storage cost. Most KT systems are built using a core data structure that is append-only—its state grows forever. The append-only property is crucial for these KT systems to achieve security but obviously presents a severe limitation in practice in terms of massive storage costs. Parakeet is the only KT system to date that addresses this issue, but their solution is very complex and non-trivial to implement.

Post-Compromise Security. In any large-scale deployment, compromises may happen eventually. Post-Compromise Security (PCS) for KT [4] helps recover from such compromises. In particular, PCS helps a KT system recover privacy for all updates which occur after the server's (private) state compromise. [4] is the only work to consider this issue, again at the cost of significant complexity.

Flexible and reliable architecture. Designing a crash-tolerant and scalable server architecture is essential for a KT system, since the bulk of the work of a KT system is performed by the server. The server's design should ideally include a flexible storage API so that data can be persisted in any external storage (database) smoothly. As long as the data in the database is consistent, any part of the server can fail and be recovered by simply reloading the state of the directory from the database. It is also vital that the the server's work is easily parallelizable for performance. Previous KT works present only the protocol or, at best, a prototype. We provide a full architecture and implementation for a scalable, reliable system.

In addition to the above, we introduce the following important features that prior work on KT systems has not addressed.

User account decommissioning. The security of a KT system heavily relies on the assumption that a user comes online intermittently to monitor their own key history. But the obvious question of what happens when a user deletes their account is often overlooked. If a user deletes her account, she will never come back online; this means a malicious party can keep impersonating that user account to her contacts. This behavior will go undetected unless the user communicates with her contacts through some out-of-band channel to inform them that she is no longer using the system. Existing KT systems do not account for this, which presents another serious limitation of current work in this space.

Public-facing username. All KT systems assume some user identifier, which we call the username. The key directory then maps each username to a device-key, or a list of device keys.

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 [11]. 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. No KT systems to date allow for this.

Supporting multiple user devices. In practice, users often have multiple devices they use with an E2EE service, so any practical KT system will need to incorporate mechanisms for multi-device support. While this feature has been considered in Keybase [9] and Parakeet [12], they do not address the challenge of handling device names, which are also typically UUIDs. In particular, introducing devices begs the question of what exactly is mapped to public keys. For instance, it would be impractical to simply map device names to keys since such device names are meaningless to human users and are therefore harder to check for inconsistencies by users. A KT system would therefore need to have a meaningful way of mapping public-facing usernames to lists of device names.

Our Contributions. In this work, we present OPTIKS, a new KT system that addresses the challenges mentioned above. To keep our design simple and easy to parse, we split the design into two parts: OPTIKS-core and OPTIKS-ext.

OPTIKS-core. We begin by constructing OPTIKS using an append-only data structure (a simplified version of ordered Zero-Knowledge Sets (oZKS) [3,4,12]). The resulting simplified protocol, which we refer to as OPTIKS-core, already outperforms the current state-of-the-art system Parakeet, while achieving the same level of security and privacy (it leaks a tiny bit more than Parakeet, which we also show how to reduce). Crucially, OPTIKS-core reduces the storage cost by almost a half when compared to Parakeet[2]. In particular, for a key directory of 1 million keys, Parakeet has a storage cost of roughly 1.1 GB, whereas OPTIKS-core requires only 517 MB.

OPTIKS-ext. While our OPTIKS-core protocol is already very performant, it does not have all the features and optimizations that are desirable for a full-fledged KT system. We describe how to add these features to OPTIKS-core in Section 5. In particular, we show how to modify OPTIKS-core to keep the core data structure compact, so that it does not grow indefinitely. In our new compaction strategy, the server's private state automatically refreshes periodically. This means we get a limited form of post-compromise security. To the best of our knowledge, ours is the first full-featured KT system that provides a limited form of PCS security ([4] defined and built PCS for the core data structure: Zero-Knowledge Sets. But they did not build a PCS-secure KT system). We also add all the additional features we described above in OPTIKS-ext.

Flexible and reliable architecture. One of the core contributions of OPTIKS is its flexible and extensible system architecture, which is crucial for maintaining a large-scale deployment. We split the design of OPTIKS into three main components: a *Query Service*, a lightweight *Update Service*, and an *Update Task*. The two services are instantiated as web services with REST APIs. The Update Service simply receives and stores key update requests in a database, from which the Update Task periodically pulls them for processing. The Query Service responds to query requests and periodically receives updates from the Update Task. The Query Service leverages our storage API that allows a portion of the data to reside in RAM and the rest in a database. We show that a model of eventual consistency, where replicated databases may lag behind the primary database and not reflect the latest changes immediately, suffices for OPTIKS. With this, the Query Service's operations can be parallelized even across multiple virtual machines.

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. 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 (Figure 2), while our Update Service/Task can process more than 1000 updates per second while sustaining a latency of one second (Figure 3a). 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.

Comparison with prior systems. To contextualize the performance of our system with prior work, we provide detailed comparison with state-of-the art KT systems. The system most closely comparable to ours is Parakeet [12]. We show that we outperform Parakeet in both performance and storage cost. For example, it takes OPTIKS less than 100 seconds to update 100K keys to a key directory with 500K keys, whereas Parakeet reports a time of 19 minutes.

We also compare with Merkle^2 [7], demonstrating that our update rate is more than 100 times that of Merkle^2 and our query rate is 1.67 times that of Merkle^2. For a rather small key directory of 2^{20} keys, our memory cost is only 517 MB, whereas Merkle^2 requires as much as 22 GB. For larger directories the difference would be even more dramatic. We note, however, that due to its lack of privacy and full end-to-end benchmarks, the scope of comparison between Merkle^2 and OPTIKS is limited.

Finally, we compare with SEEMless [3], which also does not present full end-to-end benchmarks, limiting our ability to compare. For a key directory of 2^{24} keys, our average query and key update times are less than 2.5% of those reported for SEEMless. Our average query verification time (Table 3) is less than 1% of that reported for SEEMless, while our query is on average 71% smaller in size.

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

[2]Parakeet already outperforms older KT systems in terms of storage cost

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 [3, 7, 12, 13], 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 [3]. 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 [7, 12]. Moreover, this is an improvement over many KT systems which assume that a client must be online *each epoch* to check their keys [13].

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 the full version.

- *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 [17]. 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 the full version.

- 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 the full version.

- *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 [3]. Primitives closely related to oZKS were defined in [4, 12, 15]. We provide an informal overview of this primitive below and a detailed description in the full version.

- 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 [3, 12].

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.

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 the full version and provide a brief overview of soundness and privacy below.

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.

- ▷ $\mathsf{PAHD.Init}(r)$: The server chooses a random seed and initializes an empty oZKS via $\mathsf{oZKS.Init}$ 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.
- ▷ $\mathsf{PAHD.Upd}(\mathsf{st}_{t-1}, [k_j, v_j]_j)$: 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 $\mathsf{oZKS.Update}$. The server increments the epoch $t-1$ in its state to t, and the resulting oZKS commitment com_t serves as the PAHD commitment for epoch 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.
- ▷ $\mathsf{PAHD.Lkup}(\mathsf{st}_t, k)$: 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 $\mathsf{oZKS.Query}$ 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 $\mathsf{oZKS.Query}$ 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.
- ▷ $\mathsf{PAHD.VerLkup}(\mathsf{com}_t, k, v, \pi)$: The client 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_t via $\mathsf{oZKS.Verify}$. The client also checks that version α is less than the current epoch t, since otherwise this would imply multiple versions were added in the same epoch [3]. 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.
- ▷ $\mathsf{PAHD.Hist}(\mathsf{st}_t, k)$: 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. Looking ahead, history checks will be different when we introduce our scalability optimizations in Section 5.
- ▷ $\mathsf{PAHD.VerHist}(\mathsf{com}_t, k, [(v_i, t_i)]_i^n, \Pi^{\mathsf{Ver}})$: This algorithm proceeds identically to that of $\mathsf{VerLkup}$.
- ▷ $\mathsf{PAHD.Audit}(\mathsf{com}_j, \mathsf{com}_{j+1}, j, j+1, \Pi_{j+1}^{\mathsf{Upd}})$: The auditor verifies the oZKS update proof in Π_{j+1}^{Upd} via $\mathsf{oZKS.VerifyUpd}$ and then checks that $j+1 \le t$, where t is the current epoch.

Security and Privacy of OPTIKS-core. We formally prove the security of privacy of OPTIKS-core in the full version.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.

Comparison with SEEMless [3] and Parakeet [12]. While our protocol has some similarities to that of SEEMless and Parakeet (which itself is based off SEEMless), so we present a detailed comparison here. The first major difference is that, the data needed to be stored by our core protocol is half that of SEEMless and Parakeet (before compaction). This is because when a key is updated in SEEMless and Parakeet, they must add an extra label to the oZKS indicating that the prior key version is stale, in addition to the label storing the new key version. This means that two labels must be added to the oZKS during each key update in their protocols, while our protocol requires only one label to be added.

[3]Since we have very short epochs, this is not a limitation

Another major benefit of our protocol is that a history check in our protocol is far more efficient. Both SEEMless and Parakeet require adding special marker nodes to the oZKS for key versions that are powers of 2, which require checking approximately three proofs for each key version in addition to checking a number of proofs that are logarithmic in the number of *total epochs*. Our protocol simply requires checking a single proof for each key version and one non-membership proof. We also note that audit costs for our protocols are equivalent, with the caveat that Parakeet requires slightly more checks due to verifying the deletion of appropriate elements.

These improvements come at the cost of more expensive lookups. Our lookup algorithm is equivalent to a history check, meaning that the number of proofs needed to be sent and verified is linear in the number of key versions. In contrast, SEEMless and Parakeet require verifying only three proofs. As we describe above, this also means that our protocol leaks more for lookups as well, with the benefit of this being a simpler and more efficient protocol.

In the next section, we describe how to enhance our protocol to gain the best of both worlds: overall efficiency improvements that improve scalability and reliability while also in turn reducing extra leakage and overhead costs for lookups.

Comparison with Merkle^2 [7]. Merkle^2 [7], another KT system, is currently under consideration for standardization by the IETF working group on KT [14], 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 (see Section 1 for more discussion on this). In particular, [7] strongly relies on an external PKI 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.

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 [12] 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.

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 the full version of the paper 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.

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.

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. 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. We provide more details on specific algorithm updates in the full version of the paper.

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.

[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.

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, as discussed in Section 1, 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. To address this, we change our key-update PAHD to map device ids to public keys and add two additional PAHD structures. The first PAHD (called username PAHD) maps each username to its associated user id, and the other (called device-list PAHD) maps each user id to a list of its associated device ids. 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 PAHD, the list of devices and a proof w.r.t. the device-list PAHD, and the current public keys for each device[5] 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 device, or adding/removing a username requires an update to only a single PAHD entry. We present more details on these changes to our protocol in the full version of the paper.

6 System Architecture

We now describe the details of our system architecture.

Overview. To benchmark OPTIKS in realistic scenarios, we implemented a system comprising the following components:

- *Query/Update Service.* Web services providing REST APIs for key queries and key updates.
- *Update Task.* A periodically running task that updates the oZKS and writes required changes to a database.
- *oZKS and VRF Cache.* We implement the oZKS primitive as a C++ library. The Query Service and the Update Task both hold local copies of the oZKS data. The oZKS has a built-in cache to store VRF proofs and values for fast repeated access.
- *Storage/Database.* As an append-only data structure, the oZKS can grow to be very large, and may not fit in RAM of most machines. We have built a flexible storage mechanism into the oZKS that allows the user to set up almost any kind of storage back-end they want. We instantiate this with a Microsoft SQL Server database with an adjustable in-memory cache.
- *Service Provider.* A Service Provider calls the Query Service and Update Service APIs on behalf of client devices.

[5]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.

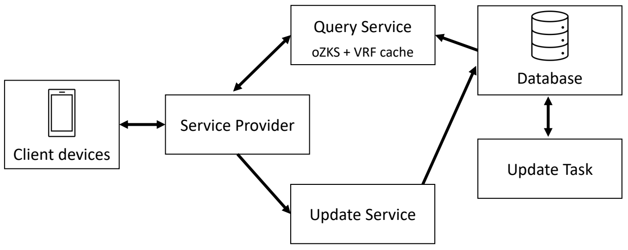


Figure 1: Our system architecture.

- *Bulletin Board.* At the change of each epoch, the Update Task pushes the newly computed oZKS root commitment and update proofs to the public bulletin board. We did not implement the bulletin board, as it is a simple component and is not a bottleneck in the system.

In practice, the Service Provider will need to handle access control using any standard methods. This is beyond the scope of OPTIKS, so in our research prototype the REST APIs do not implement any kind of access control. The architecture of our implementation is depicted in Figure 1 and we will discuss it next in more detail, component by component.

We also note that we did not implement a PAHD as a separate component. Rather, it is implemented as a combination of the oZKS and logic embedded in the Query Service, Update Service, and Update Task.

oZKS and VRF cache. Our implementation of the oZKS is a C++ library, which both the Query Service and the Update Task depend on. For a cryptographic hash function, we use BLAKE2 [1]. The library is publicly available at `https://GitHub.com/Microsoft/oZKS`.

The oZKS can run in two modes, with different pros and cons: *stored mode* and *linked mode*.

- *Stored 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. In stored mode, updates to specific nodes can be easily retrieved from the storage as needed. The downside is that the stored mode is slower, both for queries and for updates, and has a memory overhead due to hash tables maintaining the nodes in memory. The oZKS instance running in the Query Service uses the stored mode to allow for fast and flexible updates.
- *Linked mode.* 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 a storage, such as a database, but partial updates to the linked tree are difficult to implement. This makes linked mode unsuitable for the Query Service. Instead, our Update Task runs the oZKS instance in linked mode to leverage fast updates.

The VRF cache is implemented as an LRU (Least Recently Used) cache and is built into the oZKS library. Its size is a system parameter and depends on the expected load and query distribution. Since the Query Service needs to repeatedly produce both existence and non-existence proofs, its instance of the oZKS uses the VRF cache. The Update Task only needs the VRF values once (it does not even need the proofs) for the updates it processes and has no need for the VRF cache.

Storage/Database. The size of the oZKS (and other associated data) can grow to be very large, and in some cases may not fit entirely in RAM. The large size is particularly problematic for the Query Service machines that will need to handle a lot of traffic and quickly scale horizontally according to demand. In particular, it is impractical for the Query Service to have to read the full oZKS data into memory, all at once, before being able to respond to queries.

We built a flexible abstract storage mechanism into the oZKS that allows the user to create almost any kind of storage back-end they prefer. The storage mechanism supports batched stores as well as flexible per-node reads to enable efficient communication of data between our system components.

We instantiated the storage mechanism using Microsoft SQL Server as a backing database with an adjustable in-memory cache. The database tables used in our implementation are as follows:

- 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.

A note on consistency and reliability. Microsoft SQL Server provides an ACID consistency model to ensure transactions are atomic, consistent, isolated, and durable. Worldwide availability is achieved using active geo-replication, where the database is replicated in the same or different regions. This is a type of read-replication, where the replicated databases are only available for reading. In this case, SQL Server provides a model of eventual consistency, where replicated databases may lag behind the primary database and might not reflect the latest changes immediately. However, the data on the secondary databases is guaranteed to be transactionally consistent, which means that only committed changes will be replicated. This suffices for our purposes.

The reliability of our system relies on the reliability of the backing database component. As long as the data in the database is consistent, any part of the system can fail and be recovered by simply reloading the state of the directory from the database. This is the main reason why our system relies on updating each epoch through a database transaction: if the transaction completes successfully, the system is known to be in a consistent state. If the transaction failed for any reason, the system would still be in the previous consistent state. Modern databases provide a host of techniques to ensure the integrity of the data stored in them, as well as known techniques and procedures to recover from a catastrophic event. In the case of SQL Server's active geo-replication, for example, any of the replicas can take over the role of an instance that suffered a catastrophic failure, providing quick disaster recovery. Relying on the data integrity that a modern database provides makes our system reliable as well.

Query Service. This responds to clients' query requests as they are forwarded to it by the Service Provider. It also returns the epoch number that the query response is valid for, so that the client can verify the proof against the correct oZKS root commitment. The oZKS runs in stored mode.

The Query Service holds (in memory) a copy of the oZKS and associated data, such as the full public keys and other metadata. All the data is backed by 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.

Since the Query Service needs to be able to handle a high volume of queries, it is essential for it to have a low computation overhead. To aid in this, the Query Service utilizes the VRF cache that stores most commonly requested VRF proofs for both existence and non-existence proofs. The Merkle proofs themselves are impractical to cache, but we note that they are much faster to compute 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 serving a different subset of the oZKS labels. In this case, the different Query Services may respond with respect to slightly different epochs, because they are not guaranteed to be exactly synchronized. In our experiments we limit to a single instance. We describe how our protocol may be updated to accommodate multiple Query Service instances in the full version of the paper.

Update Service. The Update Service is entirely independent of the oZKS. It receives key update requests from the Service Provider and processes each as follows: (1) retrieve the latest existing version of the key from the database; (2) increment the key version by 1, or use 0 if no prior version was found; (3) write the update request data to the database.

Update Task. The Update Task 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. This process runs every certain period of time, which can be configured depending on the service load. The oZKS runs in linked mode. The Update Task executes the following steps: (1) read the existing update requests from the database; (2) compute VRF values for the new keys; (3) insert the requests as a batch in the oZKS, keeping track of the nodes that were modified; (4) update the Merkle tree nodes in the database; (5) write in a separate database table the list of updated nodes; (6) write entries in the database for each added key; (7) write update proofs and the new oZKS root commitment in the database; and (8) delete from the database the processed update requests. This operation runs as a single atomic multi-table transaction, which ensures consistency of the system. Once the transaction is committed, the update proofs and new oZKS root commitment are published to the bulletin board.

The Merkle tree insert operations in the oZKS update process 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.

7 Performance

In this section we discuss the performance of our implementation. First, we look at the performance of the oZKS and VRF implementations in isolation. Next, we evaluate a fully implemented system, including the Query Service, Update Service,

Keys	RAM	VRF caching	QPS
2^{20}	734 MB	Cache hit	143,000
		Cache miss	16,700
2^{22}	3.54 GB	Cache hit	129,000
		Cache miss	15,000
2^{24}	14.7 GB	Cache hit	120,000
		Cache miss	13,900
2^{26}	60.2 GB	Cache hit	112,000
		Cache miss	12,400

Table 1: Benchmarks for the oZKS implementation as instantiated for our Query Service. Columns in order from left to right indicate: (1) total number of keys in the oZKS; (2) RAM consumption to hold the oZKS and all associated data in memory; (3) whether the VRF proof was in the VRF cache; (4) the query rate (Queries Per Second) on a single thread.

and Update Task. Finally, we compare our performance to Parakeet [12], Merkle^2 [7], and SEEMless [3].

oZKS and VRF Benchmarks. As we explained in Section 6, the oZKS runs in two modes: stored mode (for Query Service) and linked mode (for Update Task). We perform two sets of experiments, one for the stored mode and one for the linked mode. For our oZKS and VRF benchmarks we used an Azure `E16ads_v5` virtual machine, with 16 vCPUs @ 2.60 GHz and 128 GB of RAM.

Since the oZKS computations the Query Service performs are easily parallelizable, we run these experiments on a single thread. To demonstrate the importance and potential benefit of the VRF cache (recall Section 6), we rig the experiments in two ways: ensuring VRF cache hits or cache misses. For different oZKS sizes (*i.e.*, number of keys in the oZKS), we show the total memory footprint and the query throughput. We performed the test twice by querying either for keys that are present and for keys that are not present in the oZKS; we report numbers for the slower case (generally, for keys that are not present), although the difference is very small. The results are in Table 1.

Keys	RAM	Total time (s)	UPS
2^{20}	517 MB	6.66	140,000
2^{22}	2.00 GB	27.6	144,000
2^{24}	7.95 GB	120	132,000
2^{26}	32.0 GB	520	126,000

Table 2: Benchmarks for the oZKS implementation as instantiated for our Update Task. Columns in order from left to right indicate: (1) total number of keys inserted in the oZKS; (2) RAM consumption to hold the oZKS and all associated data in memory; (3) total time to insert the keys; (4) the key update rate (Updates Per Second) on 16 threads, when the oZKS has the denoted size.

The Update Task processes updates using 16 threads. We insert new keys in batches of 1024 to be consistent with the full system benchmarks below. For different oZKS sizes, we show the total memory footprint, the total time it took to insert all of the keys, and the update throughput. The results are in Table 2.

We note that while both instantiations of the oZKS keep the full node, key, and necessary metadata in RAM, the linked mode instantiation is more efficient in terms of compute memory, as we explained in Section 6. However, this can be addressed by using a more fine-tuned hash table implementations for the in-memory data structures, splitting the responsibility of the service into several machines, or storing only the most commonly accessed nodes in memory using our flexible storage system. Despite its performance, linked mode is unsuitable for applying efficient updates to parts of the tree, as our Query Service requires.

Our VRF is implemented by adapting the IRTF internet draft ECVRF [6] to use the fast Four$\mathbb{Q}$ curve [5]. The time to compute a VRF value (without a proof) was on average 20.5 μs. Computing the proof took more than twice as much, 47.0 μs. Verifying the proof is the costliest operation at a measured 95.6 μs.

Finally, we measure the query result and update proof verification time and data size. The update proof results apply only to a single added key and will scale linearly with the size of the insert batch. These are presented in Table 3.

Keys	Query		Update	
	Time (μs)	Size (KB)	Time (μs)	Size (KB)
2^{20}	102.9	2.10	12.6	1.89
2^{22}	103.4	2.27	13.7	2.06
2^{24}	104.2	2.44	14.7	2.23
2^{26}	104.8	2.67	15.6	2.40

Table 3: Benchmarks for the oZKS proof verification time and data size. The experiments run on a single thread. Columns in order from left to right indicate: (1) total number of keys inserted in the oZKS; (2) the time to verify the query result (VRF proof and Merkle proof) on a single thread; (3) the data size of the query response; (4) the time to verify the update proof for a single added key on a single thread; (5) the data size of the update proof for a single added key. The data sizes do not include networking protocol overhead.

System Benchmarks. In this section we describe our full system benchmarking process and present results for multiple scenarios. We will focus on smaller benchmarks to enable clear comparisons to prior work. However, we want to also scale up the benchmarks to be more realistic in size and load.

We omit 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 call the service from a stress tester application running in Azure in the West US 2 region.

The Query Service was implemented in two components:

- 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);
- A back-end component that holds the oZKS in memory. This component 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.

We wanted to find the maximum query rate, *i.e.*, the maximum number of queries 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. Two different tests for the query rate where run: (1) a test for finding the maximum query rate when querying for keys with a single version in their history, and (2) a test for finding the maximum query rate when querying for keys with 10 versions in their history. We also show the average sizes of the query responses. The results are in Figure 2.

The Query Service performance is limited by networking (compare to Table 1). The communication cost is linear in the number of key versions and logarithmic in the size of the key directory, which explains the lower performance when the history contains 10 key versions. In practice, client devices can keep track of which key versions they have verified, eliminating the need to check all prior key versions repeatedly. Furthermore, the Query Service can be scaled horizontally to alleviate the burden of handling so many simultaneous network connections.

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 running on 16 threads. We measured the maximum key update rate, *i.e.*, the maximum number of key updates per second the Update Service and Update Task could support. We also measured the average time it took to create a new epoch. The epoch time was limited below to 1 second. The results are in Figure 3a.

We also measured 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 Update Task

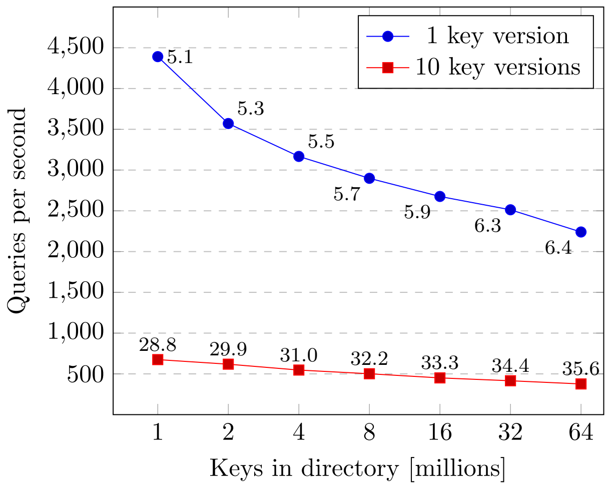


Figure 2: Key query rate as the directory grows in size from 1M to 64M. Note the logarithmic scale on the horizontal axis. Multi-threading on the front-end and back-end components was controlled by the ASP.NET runtime. The number next to each data point indicates the server response size in KB.

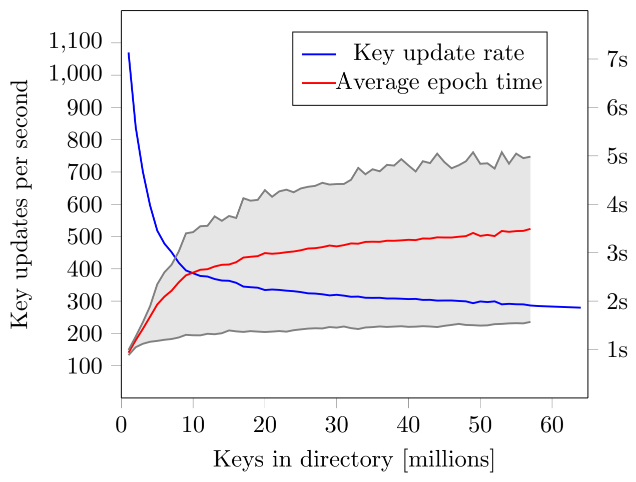
was configured to read 1,000 pending key updates at a time from the database. The results are in Figure 3b.

The logarithmically increasing cost of adding keys to the directory is evident from Figure 3a. Most of the epochs we observed took 1–5 seconds, but some took much longer due to unpredictable and fluctuating database response times. The longest epoch we observed took 13 seconds.

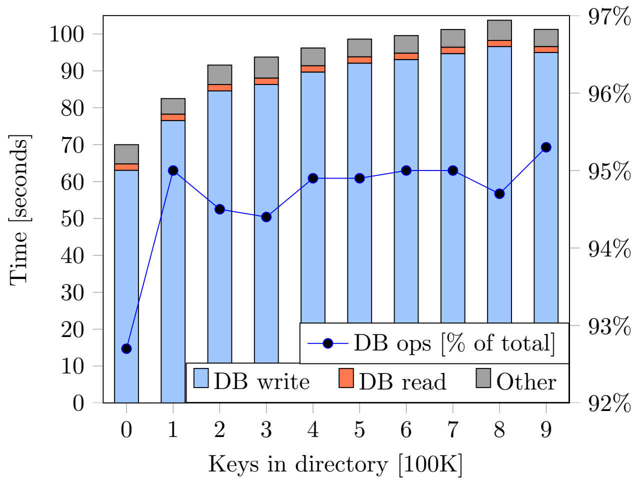
The Update Service/Task performance is strongly limited by the database performance (compare to Table 2). As can be seen from Figure 3b, most of the time is spent writing updated information to the database. For example, when the key directory has 500K keys, nearly 95% is spent in database operations. 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 4–6%, which means that any improvement in the database (write) performance would almost directly translate to a performance improvement in Figure 3a and Figure 3b.

Comparison with Prior Work. We compare and contrast our results to most relevant prior works: Parakeet [12], Merkle^2 [7], and SEEMless [3]. We omit comparison to CONIKS [13], as it cannot scale in a meaningful way to the kinds of large key directories we are targeting. These systems all differ from each other and from ours in various ways: scenario, security model, and optimization goals. Hence, we compare them only in specific aspects where the the comparison is fair and meaningful.

Comparison to Parakeet. Parakeet [12] provides the closest point of comparison. The authors motivated it as a practical



(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. Multi-threading on the Update Service was controlled by the ASP.NET runtime. The Update Task ran on 16 threads.



(b) Time it takes to add 100K keys. The current size of the key directory is on the horizontal axis. 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 3: Benchmarks for the key update performance.

way to bring key transparency to WhatsApp. They also presented an estimate that the system would need to be able to handle at least 120 key updates per second.

The only full system benchmarks for Parakeet that include database operations are in [12, Fig. 6], which is comparable to Figure 3b.[6] For example, starting from an empty key directory, Parakeet takes more than 10 minutes to insert the first 100K keys, whereas OPTIKS takes only 70 seconds. Inserting into a key directory of 500K keys, Parakeet takes 19 minutes to insert 100K more keys, whereas OPTIKS takes less than 100 seconds. The authors mention that of the 19 minutes *only* 12 – or roughly 63% – were taken by database operations, whereas for us the database operations took as much as 95%. This means that our overall performance, already better than Parakeet, benefits much more from any engineering improvements to the database performance.

There are no key update rate numbers in a full system scenario for Parakeet,[7] but we can estimate an average from [12, Fig. 6]. Thus, for an empty directory, Parakeet performs an estimated 167 updates per second, whereas we reach more than 1,000 updates per second (Figure 3a). At 500K keys, Parakeet performs an estimated 90 updates per second, which is below their stated goal of 120. Even for 2^{26} keys our update rate is 280 per second.

For key directories of 1 million and 4 million keys, [12] reports a total storage cost of roughly 1.1 GB and 3.5 GB, respectively, whereas our total storage cost for 2^{20} keys and 2^{22} keys is only 517 MB and 2.00 GB (Table 2).[8] Extrapolating, [12] estimates to require 850 GB for 1 billion users, whereas for us the estimated storage is roughly 477 GB.

Comparison to Merkle^2. Merkle^2 [7] is difficult to compare to OPTIKS; we discuss this further in the full version of the paper. They also do not present full system benchmarks, so we cannot compare end-to-end performance in any case.

We compare the append and lookup throughput in [7, Fig. 13] to our Table 2 and Table 1. For key updates, our reported update rates are more than 100 times that of Merkle^2. For queries, we note that each Query Service query requires (with one key per user) two lookups from the oZKS. Thus, it seems fair to divide our query rate in Table 1 by two. For 2^{20} keys, assuming VRF cache misses, we outperform with 8,350 queries per second the Merkle^2 *Latest value* query with less than 5,000 queries per second.

We compare to the approximate memory cost reported in [7, Fig. 12]. For 2^{20} keys, this is 22 GB – much larger than our 517 MB (Table 2). 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 [7, Table III]. In their setting the key directory has 1 mil-

[6] Note that [12] uses an AWS `t3.2xlarge` virtual machine, with 8 vCPUs @ 3.10 GHz, whereas we use a virtual machine with 16 vCPUs @ 2.60 GHz.

[7] We note that [12, Fig. 9–11] are not comparable to our Figure 3a, as they omit the full system (*e.g.*, database) overhead.

[8] Comparing to the oZKS implementation running in the stored mode (Table 1) with all data loaded in memory would be incorrect, as this figure includes overhead from generic hash table implementations. We also note that for a very large key directory we can split both the Query Service and the Update Task (with slightly more complexity) across multiple machines to avoid any RAM limitations.

lion 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 query proofs (for *Latest value* query) is 9.8 KB, whereas our proof is smaller at 2.10 KB.

Comparison to SEEMless. While SEEMless [3] presents no full system benchmarks directly comparable to our results, we can compare their key update time [3, Figure 5] to our Table 2. For a key directory with 10 million keys, [3] reports an average update time of slightly under 0.3 seconds. Adapting the results of Table 2, at 2^{24} keys our average update time is roughly 7.6 milliseconds, or just 2.5% of the time of SEEMless.

For queries, we can compare [3, Table 2] to our Table 1. Again, we divide our query rates in Table 1 by two to establish a fair 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 in Table 3 is just 1% of the 10.51 milliseconds reported in [3, Table 2].

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 just 2.44 KB, or 29% of that.

8 Related Work

We have already discussed how our KT system OPTIKS compares with Parakeet [12], SEEMless [3] and Merkle^2 [7] in Section 4 and Section 7. Here, we briefly describe the other KT systems from the literature.

Keybase [8] is the only KT system deployed in practice to our knowledge. It was originally designed as an alternative to PGP key distribution and did not target privacy as a goal. CONIKS [13] was the first academic proposal for a KT system. The efficiency and privacy guarantees of CONIKS were improved in SEEMless [3]. VeRSA [16] and Verdict [18] 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.* [4] 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] Keybase.io. Protocol security review. https://rwc.iacr.org/2019/slides/keybase-rwc2019.pdf (accessed: 2023-06-03), 2019.

[10] 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).

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.