Parakeet: Practical Key Transparency for End-to-End Encrypted Messaging

Harjasleen Malvai*†, Lefteris Kokoris-Kogias‡§, Alberto Sonnino‡¶, Esha Ghosh‖,
Ercan Oztürk**, Kevin Lewi**, and Sean Lawlor**
*UIUC, †IC3, ‡Mysten Labs, §IST Austria, ¶University College London (UCL), ‖Microsoft Research, **Meta

***Abstract*—Encryption alone is not enough for secure end-to-end encrypted messaging: a server must also honestly serve public keys to users. Key transparency has been presented as an efficient solution for detecting (and hence deterring) a server that attempts to dishonestly serve keys. Key transparency involves two major components: (1) a username to public key mapping, stored and cryptographically committed to by the server, and, (2) an out-of-band consistency protocol for serving short commitments to users. In the setting of real-world deployments and supporting production scale, new challenges must be considered for both of these components. We enumerate these challenges and provide solutions to address them. In particular, we design and implement a memory-optimized and privacy-preserving verifiable data structure for committing to the username to public key store.**

To make this implementation viable for production, we also integrate support for persistent and distributed storage. We also propose a future-facing solution, termed "compaction", as a mechanism for mitigating practical issues that arise from dealing with infinitely growing server data structures. Finally, we implement a consensusless solution that achieves the minimum requirements for a service that consistently distributes commitments for a transparency application, providing a much more efficient protocol for distributing small and consistent commitments to users. This culminates in our production-grade implementation of a key transparency system (Parakeet) which we have open-sourced, along with a demonstration of feasibility through our benchmarks.

I. Introduction

The use of end-to-end encrypted (E2EE) messaging ([31], [32]) ensures that conversations between users remain private. It is just as vital, though, to make sure these interactions happen with the right people. To avoid being compromised by a man-in-the-middle attack, users must discover the necessary public keys to identify each other. In reality, this is accomplished either by scanning QR codes in person to validate recipient public keys or by depending on a service provider to generate the relevant public key of a communication partner. The term *key transparency* refers to a system in which a service provider stores the public keys of individuals on a publicly accessible key directory server so that users can query this server for messaging individuals in their contact list.

However, since the server can supply users with outdated or fabricated keys, this strategy makes the server a single point of failure. To counteract this, three complimentary methods have been discussed in existing literature. Firstly, users must be able to verify public keys served by the identity provider as well as their own key(s) on a regular basis. Secondly, the key changes must be publicly auditable, to ensure that the server is adhering to certain update rules. Finally, a consistency enforcement protocol must be used to prevent the server from serving different versions of the directory to different users.

To address the problem of providing auditable identity bindings, Chase et al. [16] introduced a primitive called *verifiable key directory* (VKD) and defined its required security properties. Their system, called SEEMless, instantiates a VKD and proves the correctness of the underlying scheme. Although the evaluation of SEEMless indicates that it veers toward practicality, SEEMless makes the following simplifying assumptions, omitting issues that must still be resolved in order to be applicable to a large-scale, real-world deployment. SEEMless assumes that there exists a mechanism for clients to obtain consistent views of a small server commitment, at every server epoch. Their implementation relies on local RAM storage to store all of the data ever required for key transparency. Scalable implementations need a separate, optimized storage layer. The data structure that SEEMless uses to keep state, which they call a *history tree*, assumes that it is feasible to perpetually store enough data to reconstruct the state of the server's cryptographic data structure from any epoch. Even though their data structure is well-optimized to satisfy the requirement of storing this much data, it becomes a bottleneck when scaling to billions of users.

Hence, while a good starting point, these assumptions render SEEMless, as is, infeasible for large-scale deployments. For instance, WhatsApp, one of the most popular end-to-end encrypted messaging apps, has a monthly active user base of 2 billion [2] and has been downloaded on average 0.66 billion times [3] in the last five years. With the rollout of multi-device [1] where each user device has a separate key, the daily new key creations can be estimated to be roughly 5.4 million (based on an average of 3 devices/browsers per user and download numbers). If we assume that existing users update their keys at a similar volume, a rough estimate doubles this to approximately 10 million. This leads SEEMless's data structure to require approximately 27TB (counting existing and new keys based on number of key updates/creations) of storage within the first year (See Appendix I).

A. Our Contributions

In this work, we present Parakeet, a key transparency system which is designed specifically to overcome the scalability challenges that limit prior academic works from being practical for real-world encrypted messaging applications.

We achieve this by constructing a more efficient VKD, extending the VKD to support compaction, and then leveraging a simple consistent broadcast protocol to avoid the reliance on clients having to connect to blockchains (and the scalability issues associated with this dependence) to achieve consistency. Below, we elaborate on these three central contributions:

1) **Building an efficient VKD.** A VKD is a cryptographic primitive, defined and implemented by [16], which allows an identity provider to store an evolving set of label-value pairs, commit to this set, and respond (using cryptographic proofs) to client queries about this committed set and its updates. At a scale of billions of users, however, the efficiency requirements of a system implementing a VKD become more stringent than [16]'s implementation is able to meet. We address these limitations by implementing a cryptographic construction called an *ordered zero-knowledge set* (oZKS), which allows us to more efficiently realize a VKD with storage improvements of up to an order of magnitude when compared against existing solutions. In addition to the storage optimizations for the VKD, we also present a modular and flexible data-layer API, called StorageAPI, which can be implemented using any distributed database solution. Our VKD implementation, written in Rust, has been published as an open-source library [5].
2) **Supporting compaction.** Many works on verifiable data structures that support updates require append-only data structures ([41], [16]). In large-scale practical contexts, requiring the support of an ever-growing storage and in-memory system that supports append-only data structures can be a barrier for the deployment of key transparency. We introduce the term "compaction" to refer to an operation that allows reducing the data stored on the server by purging ancient and obsolete entries. Secure compaction loosens the requirement for append-only data structures, as introduced in [16], through a minimal additional assumption. By extending the existing oZKS functionality to enable secure deletions, we present our *two-phase compaction* paradigm of a VKD, which can be brought on par to existing works with comparable performance. The novelty of our oZKS construction consists of the mechanism for enabling secure deletions. We include a theoretical discussion of our construction here as we believe it is of independent interest for any applications which may be limited by the storage requirements of append-only verifiable data structures.
3) **Serving commitments.** All of the equivocation-based security definitions for transparency require users being able to access a small shared commitment. Some works assume a shared public ledger, such as a blockchain [41], [13]. Others rely on out-of-band gossip (see, e.g. [33]). However, at a scale of billions of users, these mechanisms all have drawbacks. For example, across platforms and geographic regions, picking good out-of-band mechanisms, which do not result in disconnected partitions of sets of users, is a challenge. As [41] analyzed, using a blockchain to respond a large number of queries could eventually result in flooding the open ports of a large fraction of the nodes of this chain. The alternative described by [41] is to use a *header relay network*, i.e. a small number of nodes serving a specific transparency application, essentially, a centralized service. Since none of these solutions have tackled this problem at a scale of billions of clients, finding the minimal set of requirements for a consistency protocol remains an open question. To this end, we propose a lightweight *consensusless* consistency protocol that provides these guarantees with low performance overhead.

Paper organization. In Section II, we present an overview of the Parakeet system, which consists of the VKD construction, the consistency protocol and an interface between the two. In Section III, we review the original VKD definition and present a more storage-efficient construction based on an oZKS. In Section IV, we extend VKDs to support the compaction operation to address storage limitations in practical systems. In Section V, we describe the consistency protocol used by Parakeet to serve commitments. In Section VI, we provide microbenchmarks and comparisons to prior works, run against our open-source implementation of the system. We discuss related works in Section VII and conclude in Section VIII.

II. Overview

A central entity called the *identity provider* (IdP) keeps a local database linking users' identifiers (*e.g.* phone numbers) to their public keys. It periodically sends a commitment of its latest state to a distributed set of authorities, called *witnesses*, that ensure its correct behavior. The witnesses store the latest commitment and also communicate with clients to make sure the identity provider is not censoring or eclipsing them. The *users* query the identity provider to look up the public key associated with a specific user identifier. Each user can additionally monitor the history of their own public key. The identity provider is trusted for ensuring that only permissioned users can perform particular actions.

A. System & Threat Model

Participants. Parakeet is run by the following participants:

- **Identity provider (IdP):** The entity keeping the identifier-to-key binding for every user and replying to users' queries.
- **Witnesses:** A distributed set of authorities that cross-check the IdP.
- **Users:** Ask the IdP to store a specific key bound to their identifier, and query it to look up the key bound to specific identifiers. Users can also periodically check the history of their own key, to ensure that the server did not make any unwarranted changes to their key.

The IdP and each of the witnesses generate a key pair consisting of a private signature key and the corresponding public verification key, so that their identity is known.

By definition, an *honest* witness always follows the Parakeet protocol, while a *faulty* (or Byzantine) one may deviate arbitrarily. We present the Parakeet protocol for $3f+1$ equally-trusted witnesses, assuming a fixed (but unknown) subset of at most f Byzantine witnesses. In this setting, a quorum is defined as any subset of $2f + 1$ witnesses. (As for many BFT protocols, our proofs only use the classical properties of quorums thus apply to all Byzantine quorum systems [30].) When a protocol message is signed by a quorum of witnesses, it is said to be certified: we call such a jointly signed message a *certificate*. Additionally, we assume the network is fully asynchronous [21]. The adversary may arbitrarily delay and reorder messages; however, messages are eventually delivered.

We discuss the various guarantees for these parties below.

Properties. The consistency protocol of Parakeet satisfies the following properties:

- **Consistency:** For a given epoch t, if a user outputs a commitment com_t and a different user outputs a commitment com'_t, then com_t=com'_t.
- **Validity:** If an honest witness outputs com_t as valid, then com_t was proposed by the identity provider.
- **Termination:** If the identity provider runs the protocol honestly for epoch t, commitment com_t, eventually the identity provider will produce a certificate cert_t for t.

The security properties of the VKD of Parakeet are more complicated to formally state, so we defer a more formal treatment to appendix B and section III. At a high level, we require that the construction of a VKD satisfy:

- **Completeness:** If an identity provider honestly serves values, then, for any label label, registered with the identity provider, all users should receive (and accept) consistent views of the value associated with label.
- **Soundness:** As stated before, the soundness definition used in this paper is that of non-equivocation, i.e., if, at an epoch t, Bob accepts a value val as Alice's key, the server cannot convince Alice that her key was val' for the same epoch t with $\mathsf{val}' \neq \mathsf{val}$. Note, this is in the presence of auditors.
- **Privacy:** Privacy for a VKD is defined with respect to all parties who are not the identity provider itself. The responses to all API calls made by parties which are not the identity provider are zero-knowledge with a well-defined, permissible leakage function.

B. Overview of our VKD Solution

Our VKD solution, aimed at real-world, large-scale applications, removes a majority of the assumptions made by previous works in this domain, such as, [16], [34]. We modify the constructions and definitions presented in [16] to use a primitive called ordered zero-knowledge set (oZKS) [6], which, together with a secure commitment scheme (cCS) and a verifiable random function (sVRF), allow us to instantiate a VKD in space linear in the number of updates for labels in the VKD. This is in contrast to [16]'s construction, that additionally requires space linear in the number of server epochs, with a high constant factor (linear in the security parameter). We further augment these constructions to provide a method to allow compacting the underlying data-structures, i.e., deleting values which are no longer in use, while still requiring very little monitoring from the user to provide security. This combines the space efficiency gains of a VKD construction which relies on the assumption that users monitor their keys constantly (e.g. [34]), with the security gains of a construction based on append-only data structures (e.g.[16]).

C. Overview of our Consistency Protocol

Our first result is to debunk the common belief that such systems require consensus [36], [41], [16]. A first correct but inefficient solution would be to use simple Reliable Broadcast [14] to achieve all required properties in full asynchrony, a setting where deterministic consensus is actually impossible [22]. This, however, is inefficient. Instead, we design a tailor-made solution for Parakeet (Section V). The resulting system has low-latency, and additionally the identity provider can be natively sharded across many machines (unlike consensus-based solutions) to allow unbounded horizontal scalability.

D. Bridging the VKD & the Consistency Protocol

For the most part, we handle the consistency and VKD components of Parakeet separately. However, they do need to communicate with each other. Here, we provide an *application programming interface* (API) to bridge this gap. The IdP must provide small commitments to the witnesses and the users need to communicate with the witnesses to receive these up-to-date commitments. To this end, we define a simple witness API.

As stated above, the witnesses run the consistency protocol to certify a commitment to the internal state of the VKD. So, our witness API includes an algorithm for the IdP to propose a commitment at each server epoch and prove that it correctly updated its VKD ($\mathsf{WitnessAPI.ProposeNewEp}$). We also allow any party to query the witnesses to retrieve the commitment and certificate for an epoch ($\mathsf{WitnessAPI.GetCom}$). Finally, the API call $\mathsf{WitnessAPI.VerifyCert}$ verifies the certificate for a commitment. The details of this API are in appendix D.

III. Verifiable Key Directory

In this section, we will discuss the primitive *Verifiable Key Directory* (VKD), defined by [16], and its properties. Recall that a VKD consists of three types of parties: (1) an identity provider (or, server), (2) users, and (3) independent auditors. In a VKD, each user has an associated label, denoting their username. The server stores a directory Dir of label-value pairs. Each value corresponds to a public key. The clients can request updates to their own public keys – equivalent to requesting a change to the state of Dir. For efficiency, many such requests are batched together, with updates going into effect at discrete timesteps (*epochs*). So, Dir is stateful, has of an ordered sequence of states Dir_t, one state per epoch t.

To support verifiability in a VKD, each state of the directory needs a corresponding commitment com_t. The commitment com_t is made public using the $\mathsf{WitnessAPI}$ defined in appendix D. This model assumes that any changes to the directory go into effect when the corresponding epoch t goes into effect and the commitment, com_t, for this epoch is published.

The security of a VKD crucially relies on at least one honest auditor per epoch checking the latest update for validity. This assumption is common across a majority of the work in this area (see, e.g. [43], [16]), in order to maintain client-side efficiency. In this work, we realize this assumption using the witnesses, each of which runs the auditing operation as part of executing WitnessAPI.ProposeNewEp. Note that the WitnessAPI solution can be used *in addition to* any other solution, such as gossip for distributing commitments [33] or client auditing [43], also discussed in Section VII. When serving billions of users across different geographical regions and platforms, it is not always reasonable to expect clients to run audit operations or to participate in a connected gossip network, so using a set of witnesses only increases security.

A. Outline of this Section

The rest of this section is devoted to revisiting the VKD primitive, with improvements targeted for production. In section III-B, we recall the various algorithms for this primitive and its properties. For our storage-optimized VKD, we need the oZKS primitive, whose motivation and properties we describe in section III-C. Then, in section III-D, we describe our VKD construction using the oZKS, which achieves properties identical to [16]. Even with improved efficiency, several practical problems remain, including: (1) allowing users to have multiple devices linked to their accounts, and multiple updates for the same user in an epoch, (discussed in section III-E), and (2) introducing a separate, efficient storage layer for a scalable VKD implementation (discussed in section III-F).

B. VKD Definition

The primitive verifiable key directory (VKD) was first defined by Chase, et. al. [16]. As stated before, the VKD server holds a directory mapping labels to values with one state per epoch. The commitment to this state (and the proof of correct update) is published by the server, using VKD.Publish.

When users want to lookup the public key for a particular label, they request this by calling VKD.Query. The response to a lookup comes with a proof of correctness, which the requesting user can verify (with VKD.VerifyQuery). Each user can also check the history of her own key, implicitly, for her label, getting a mapping $t \rightarrow \mathsf{val}_t$ of the state of the associated value at every epoch. The user can get this history and its proof (VKD.KeyHistory), as well as verify it (VKD.VerifyHistory). Any party can verify the output of a publish (VKD.VerifyUpd) or a sequence of publish operations (VKD.Audit).

Soundness and privacy of a VKD. The soundness definition for the VKD primitive is exactly the same as that of [16]. This definition captures the non-equivocation property stated in section II-A, assuming that the server never deletes any existing records. We also define the privacy of a VKD the same way as [16]: all operations are zero knowledge with a well defined leakage function.

C. Ordered (Append-Only) Zero-Knowledge Set with Deletion

As a step towards a more space-efficient implementation of a VKD than that of SEEMless, we replace their aZKS building block with a primitive which we call an *ordered append-only zero-knowledge set* (oZKS). Here, we discuss the properties of oZKS and briefly describe our implementation.

An oZKS is actually a further generalization of the append-only zero-knowledge set (aZKS) primitive, which was first introduced in [16]. [6] presents an implementation (and corresponding informal definition) of an oZKS construction. The oZKS primitive is used in a setting where a party (often called a *server*) holds a data-store of label-value pairs (with unique labels), and is trusted for privacy, but not to serve consistent views of label-value pairs. The party uses oZKS to commit to label-value pairs where the labels are all unique.

Initial work on zero-knowledge sets (e.g. [17], [35], [15]) committed to static data stores. The aZKS primitive extended this to include insertions only. The recent implementation of an append-only oZKS [6] extends the aZKS to include a strict ordering on when elements are inserted. Thus, an oZKS should support verifiable algorithms for: initially committing to a datastore (oZKS.CommitDS), insertions (oZKS.InsertionToDS), membership/non-membership queries (oZKS.QueryMem and oZKS.QueryNonMem). We use the oZKS primitive to build our VKD construction in section III-D.

Comparing with the oZKS from [6]. Note that the append-only property of the implementations of both [6]'s oZKS and [16]'s aZKS require an ever-growing storage requirement on the server. In large-scale practical applications, never purging obsolete data can be infeasible. The novelty of the oZKS construction in our work consists specifically of the mechanism for *enabling secure deletions*. We present this as a middle ground solution between fully mutable auditable data structures that require users to be always online (e.g. CONIKS), and append-only auditable data structures that lead to an unreasonable server storage cost for long-running systems (e.g. prior and concurrent work on oZKS and SEEMless).

[18] also formalizes the protocol presented in the implementation [6] and extends it to provide post-compromise security. Their construction is still append-only and does not support secure deletion. So, our contribution is orthogonal to that of [18].

We introduce the secure deletion extension here for completeness and apply it in section IV. For verifiability, the deletion operation of an oZKS with compaction is actually a two-step process: marking nodes as candidates for deletion (a process we call tombstoning, through oZKS.TombstoneElts) and deletion of tombstoned elements (oZKS.DeleteElts). In appendix B, we provide a formal definition of all of these algorithms and in appendix C, we describe our oZKS construction.

Our oZKS implementation. Similarly to [16]'s aZKS, our oZKS instantiation uses a Merkle Patricia Trie (MPT) to commit to label-value pairs. However, instead of using a complicated persistent data-structure, which requires storing all states of every MPT node, for the oZKS data structure, we simply include the epoch a leaf was inserted as part of the value committed for a node. This allows us to store one MPT which mutates over time and old states of nodes in this tree can be garbage collected, instead of persisting forever. Also, note that for privacy, the MPT-based oZKS implementation computes leaf labels using a *verifiable random function* (VRF) (defined in appendix C), which is a deterministic function computable only by the holder of a secrete key for a publicly known public key, PK. Any party can, however, check the correct computation of a VRF using PK. For privacy of the actual value associated with a label, we use a hiding commitment scheme. Concretely, this means that to add label with value val to the datastore and update the corresponding commitment at epoch t, the owner of the datastore adds a leaf with label $\mathsf{VRF}(\mathsf{label})$ and value $(\mathsf{com}(\mathsf{val}); t)$ to the MPT. Figure 1 shows an example of the MPT-based aZKS used by [16] as it evolves through various insertions. Figure 2 shows an example of our oZKS construction, with the same leaves being inserted as in Figure 1.

Note that this constitutes only a simple oZKS, without the tombstone or deletion algorithms. We discuss tombstoning and deletion in section IV.

Soundness and privacy. The oZKS without the $\mathsf{TombstoneElts}$ and $\mathsf{DeleteElts}$ algorithms is said to be sound if, given at least one honest auditor in every epoch, the server cannot delete any elements which were previously committed. I.e., the oZKS without tombstones or deletions is append-only. At first glance, the terms "append-only" and "with deletion" for an oZKS may seem like contradictions. What we intend to capture is a mechanism to commit to a set, in an append-only manner, but with the additional ability to remove very old values which may no longer be needed by the calling application. Hence, the soundness property of this data structure, defined in appendix B, is that values may only be marked for deletion if they are older than an epoch permitted by a system parameter. Other than the fixed epochs for tombstoning old-enough nodes, and correspondingly for deleting nodes marked as tombstoned, the data structure should only allow insertions for labels not present in the datastore already. As in the privacy definition for the aZKS of [16], we require all functions for the oZKS to be zero-knowledge, with a well-defined leakage function.

Leakage for our implementation. Our oZKS construction, when committing to an initial data store ($\mathsf{oZKS.CommitDS}$), leaks the size of the datastore. The $\mathsf{oZKS.InsertionToDS}$ also leaks the size of the datastore before and after the update. For each inserted element, the adversary also learns whether it queried for this element before and if it did, this tells the adversary when this element was added. $\mathsf{oZKS.TombstoneElts}$ and $\mathsf{oZKS.DeleteElts}$ leak when the tombstoned (resp. deleted) elements were inserted and the size of the datastore before and after each call. The responses to $\mathsf{oZKS.QueryMem}$ and $\mathsf{oZKS.QueryNonMem}$, in addition to the actual response, also leak the size of the datastore and for $\mathsf{oZKS.QueryMem}$, it leaks when this element was added.

D. Revisiting the SEEMless VKD Construction

In this section, we construct a VKD (without compaction) using an oZKS, instead of an aZKS and also summarize the concrete implementation strategy for our VKD implementation. This replacement of an aZKS with an oZKS results in functionality which is equivalent to that of SEEMless [16] with significant space savings. The formal description of this construction is in appendix E.

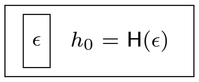
Concrete implementation. As mentioned in section III-C, our concrete implementation of the oZKS consists of a Merkle Patricia Trie (MPT), used to commit to a leaf's value and bind it to the epoch in which it was inserted. This MPT-based oZKS is an important component of our VKD construction. In addition to this oZKS, the VKD requires a database to store actual username to value mappings as well as when each value was added. Suppose a user with username Alice first joined the system at a time t with public key value val_1. Thus, val_1 is the first version of Alice's public key and the server adds the label '$\mathsf{Alice}|1$' with value val_1 to the oZKS at epoch t. If at a later epoch t', Alice's key is updated from version i to $i+1$ with new value val', the server inserts the labels '$\mathsf{Alice}|i|\mathsf{stale}$' with value equal to the empty string, and '$\mathsf{Alice}|(i+1)$' with value equal to val' to the oZKS. At a lower level, this means that at epoch t', the server adds to the MPT the leaves whose labels are '$\mathsf{VRF}(\mathsf{Alice}|(i+1))'$ and '$\mathsf{VRF}(\mathsf{Alice}|i|\mathsf{stale})'$, with values $(\mathsf{com}(\mathsf{val}'); t')$ and $(\mathsf{com}(\epsilon); t')$, respectively.

Soundness and privacy. The soundness definition of this VKD construction is identical to that of [16]. At a high level, as long as for epochs up till epoch t, (1) a client with label label checks the states of key using the $\mathsf{VKD.VerifyHistory}$ algorithm, (2) the $\mathsf{WitnessAPI}$ is honest, and (3) at least one honest auditing party verifies each update using $\mathsf{VKD.Audit}$, then the identity provider could not have output a diverging view of the val associated with label at any epoch less than t. In other words, in the presence of auditing and witnesses, Alice and Bob must always agree on the view of $\mathsf{val}_{\mathsf{Alice}}$—the value associated with the label Alice. The leakage functions of this construction match those of [16]'s construction exactly.

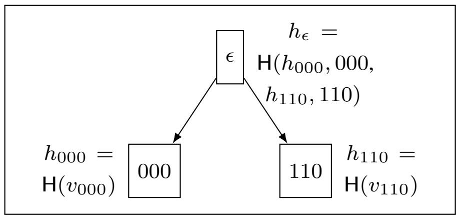
Space efficiency improvements. The aZKS construction of [16] allows reconstructing the state of verification data for a data store at any time, by storing all intermediate states ever generated. Both our work and [16] implement a compressed version of a Merkle Patricia Trie, where, for a random set of leaves, the expected depth of a leaf in a tree with n leaves is $\log(n)$. Thus, if a new node is added to the MPT of [16], this results in adding $\mathcal{O}(\log(n))$ new states to persistent storage. Even with batching, this means that the space complexity of their implementation depends on the number of epochs, in addition to the number of leaves added. However, the space complexity of our oZKS implementation only depends on the total number of leaves added. The impact of this difference is quite significant, as shown in Section VI.

E. Other Practical Considerations

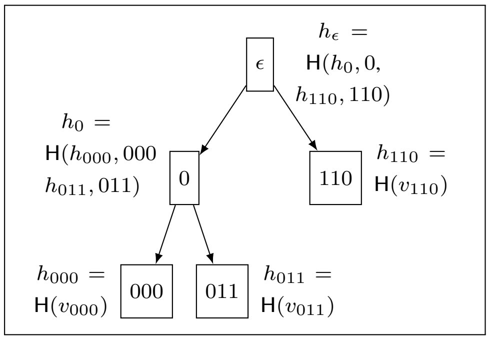
In addition to the compaction extension, we discuss the following additional considerations for practical deployments of key transparency for messaging applications.



Epoch 0

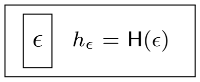

ϵ
$h_\epsilon = \mathsf{H}(h_{000}, 000, h_{110}, 110)$
000
110
$h_{000} = \mathsf{H}(v_{000})$
$h_{110} = \mathsf{H}(v_{110})$

Epoch 1: After inserting label-value pairs $(000, v_{000})$ and $(110, v_{110})$.

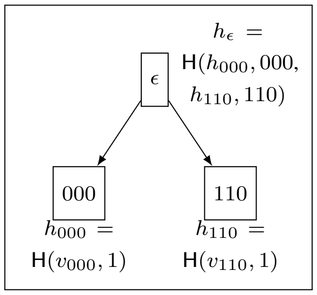

ϵ
$h_\epsilon = \mathsf{H}(h_0, 0, h_{110}, 110)$
$h_0 = \mathsf{H}(h_{000}, 000\ h_{011}, 011)$
0
110
$h_{110} = \mathsf{H}(v_{110})$
000
011
$h_{000} = \mathsf{H}(v_{000})$
$h_{011} = \mathsf{H}(v_{011})$

Epoch 2: After inserting label-value pair $(011, v_{011})$.

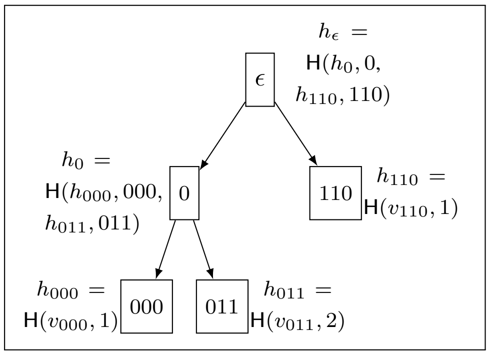
Fig. 1: An example of the evolution of the SEEMless Merkle Patricia Trie-based aZKS construction with 3-bit labels. The aZKS starts with no entries at epoch 0, which is committed in the tree with a single node, with label ϵ and commitment $\mathsf{H}(\epsilon)$. At the first epoch, two new leaves are inserted and at epoch 2, a third leaf is inserted. Note that the values inserted in this tree correspond to the entries inserted in Figure 2. However, in contrast to the oZKS construction, the tree at epoch 2 as is cannot be used to reconstruct any of its previous states. SEEM*less* optimizes the ability to get values from previous states using storage compression.



Epoch 0


ϵ
$h_\epsilon = \mathsf{H}(h_{000}, 000, h_{110}, 110)$
000
110
$h_{000} = \mathsf{H}(v_{000}, 1)$
$h_{110} = \mathsf{H}(v_{110}, 1)$

Epoch 1: After inserting label-value pairs $(000, v_{000})$ and $(110, v_{110})$.


ϵ
$h_\epsilon = \mathsf{H}(h_0, 0, h_{110}, 110)$
$h_0 = \mathsf{H}(h_{000}, 000, h_{011}, 011)$
0
110
$h_{110} = \mathsf{H}(v_{110}, 1)$
000
011
$h_{000} = \mathsf{H}(v_{000}, 1)$
$h_{011} = \mathsf{H}(v_{011}, 2)$

Epoch 2: After inserting label-value pair $(011, v_{011})$.

Fig. 2: An example of the evolution of our Merkle Patricia Trie-based oZKS construction with 3-bit labels. The oZKS starts with no entries at epoch 0, which is committed in the tree with a single node, with label ϵ and commitment $\mathsf{H}(\epsilon)$. At the first epoch, two new leaves are inserted and at epoch 2, a third leaf is inserted. Note that the tree at epoch 2 can be used to reconstruct any of its previous states.

User interfaces. In this work, we assume that the server is trusted for bootstrapping users. For example, if a user loses their phone, there is a mechanism for them to recover their account. We also assume that the client application has interfaces to inform users about various events including: if witnesses are offline for extended periods of time, if a proof failed to verify, or if a public key they requested to add to their account did not take effect within a specified time period. [38] discusses some of the design considerations for a user-facing transparency system. We leave further study to future work.

Multiple devices per user. A common property of most mainstream messaging applications is the ability to associate one user account with multiple devices owned by the user (e.g. computer, mobile phone), each with a distinct E2EE keypair. This means that a VKD must support a set of public keys belonging to a user, as opposed to just a single key. This simply involves using the set of public keys as the values in the VKD, ensuring that validating client queries involves a membership check instead of an exact match of public key.

Multiple updates per epoch. Depending on the length of the epochs, clients may submit multiple updates to their public key values within the span of a single epoch. Only publishing the latest key received for a user within an epoch can lead to issues with consistency for clients which may have been messaged within the fast key updates. Instead, when sequencing these updates, the server can designate the ordering by adding an ordered list of public keys as opposed to a single key per epoch. Then, clients can check for list membership when querying for proofs, and the server can use the ordering to ascertain the latest key after the next epoch begins.

Together, these adjustments to client public keys result in an ordered list (multiple updates per epoch) of sets (multiple devices) of public keys being hashed in our VKD construction.

F. Storage API

As discussed in Section I, a large-scale VKD implementation requires a separate storage solution which we call StorageAPI.

Warmup. As a first attempt at defining StorageAPI, we may want only two operations on top of a simple database:

- $\mathsf{val}/\bot \leftarrow \mathsf{StorageAPI.GetFromStorage}(\mathsf{key})$: This call takes as input a key (key). If the key is in the database, it returns the associated value (val). Otherwise, it outputs $\bot$.
- $1/\bot \leftarrow \mathsf{StorageAPI.SetToStorage}(\mathsf{key}, \mathsf{val})$: This call takes as inputs a key key and corresponding value val. It outputs 1, if this value is successfully in the database and $\bot$ otherwise.

We could use this simple API for all data types by defining the storage keys for each type as an encoded binary vector with a prefix to demarcate types. The implementer of StorageAPI determines the underlying storage layer to handle each type. This also maintains unique keys for individual records without changing the storage key type signature.

Handling storage latency. Of course, in a practical implementation, storage will have to consist of a set of geographically spread-out, duplicated nodes. This means that one of the biggest bottlenecks in timely update of the server's verifiable data structures, as well as client queries, is memory latency. For large-scale end-to-end encrypted messaging identity providers, what may *not* be a problem, however, is bandwidth. Hence, we add batched storage APIs to leverage the high-bandwidth to reduce latency. We define these APIs as:

- $\{\mathsf{val}_i\}_i \leftarrow \mathsf{StorageAPI.BatchGetFromStorage}(\{\mathsf{key}_i\}_i)$: This call should take as input a set of keys key_i and return values val_i stored in the database associated with the corresponding key. If label_i is not in the database, $\mathsf{val}_i = \bot$.
- $1/\bot \leftarrow \mathsf{StorageAPI.BatchSetToStorage}(\{(\mathsf{key}_i, \mathsf{val}_i)\}_i)$: This call takes as inputs a set of key-value pairs $(\mathsf{key}_i, \mathsf{val}_i)$, and outputs 1, if all values are successfully set and $\bot$ if any errors occur. Note that this implicitly implies atomacity of a single $\mathsf{BatchSetToStorage}$ operation.

Batching-friendly oZKS. Note that batching storage operations is not enough on its own, unless the algorithms we implement are also amenable to batched memory accesses. In particular, underlying our implementation of an oZKS is a compressed Merkle Patricia Trie, whose construction satisfies the following invariant:

Each node is the longest common prefix of its children.

This implies that, for example, just looking at the label for a leaf being inserted does not automatically allow the server to know which nodes (and corresponding key_i's) to use with $\mathsf{BatchGetFromStorage}$. To solve this problem, our implementation starts with assuming that the storage solution has caching capabilities. Under the assumption of a large enough cache, if we can preload nodes for a batched operations, say, $\mathsf{VKD.Publish}$, the operation itself can be done as if in RAM. Later, the cache can be flushed to persistent memory as a single transaction operation. oZKS operations requiring tree-traversals are called with a leaf label as input, batched versions of these operations are called with a set of leaf labels `leaves`. To optimize remote persistent storage accesses, at a high level, we implement procedures which operate following two steps: (1) compute a set `prefixes` of all prefixes of values in `leaves`, (2) starting at the tree's root node, begin a breadth-first search for nodes with labels in the set `prefixes`, batch-fetching labels at the same depth. This ensures that all of the nodes required to run algorithms for the labels in `leaves` will be loaded at the end of the preload operation with only one access to persistent storage per layer of the Merkle Patricia Trie, without overwhelming the cache by having to load the entire tree. In fact, the set of nodes retrieved to cache is exactly the set required for the original oZKS batched operation.

So far, we have omitted any discussion of batching for set (write) storage operations. This is easier, since batching writes is equivalent to flushing an in-memory cache of the updated nodes, an operation our storage interface requires. The only writing operation is $\mathsf{VKD.Publish}$, and once it has completed its changes, we can commit all of the changes in the transaction cache as a single atomic operation (assuming the storage layer supports this). Since all of the modified data will be in the cache, flushing the cache in a timely manner is all that is needed to ensure up-to-date views of storage.

Other storage considerations. Even with our cache-based solution, there is an underlying assumption that the storage layer can support a cache-sized atomic write operation. For very large systems, we would like to further loosen this requirement, by adding support for the situation in which the storage layer cannot atomically (or at least efficiently) commit the entire transaction of changes in a single operation. We make some simple modifications to the data-structure layer to prevent inconsistencies in read operations during an update.

All proof generation operations use an epoch value stored in the oZKS data-structure to determine the latest epoch, denoted ($\mathsf{LatestEp}$). $\mathsf{LatestEp}$ must be updated last in order to ensure that the data corresponding to $\mathsf{LatestEp}$ has been written for all parts of the data-structure, before any reading is permitted. Once the oZKS's latest epoch is updated, all operations will take the new epoch as truth and access consistent values.

With this in mind, we need to change the stored record for a tree node to store two values at a time: the previous and the current value. For atomicity, this means storing a dual-value struct, each value with an associated epoch. Read and write operations read the epoch stored at these values to determine which to designate as current and which as previous. The read operation always reads the value whose epoch is less than or equal to $\mathsf{LatestEp}$, and the write operation overwrites values whose epoch is less than $\mathsf{LatestEp}$. This allows us to update tree nodes without impacting proof generation operations.

IV. VKD WITH COMPACTION

Previous constructions, such as [16] and ([43], [34]) seem to fall into two extreme categories: (1) works that assume the server's storage to be linear in the number of updates, and (2) works that assume users are either always online or can retroactively check the server's view of their keys for any epochs they may have missed. This means that the client has to do linear work in the number of server epochs to monitor their keys. Also, at production scale, we must account for the storage implications of being able to serve users who come online infrequently—e.g., in case (2), the server would need to store a large trove of history, perhaps even all of its data.

Even the soundness of our oZKS-based VKD construction depends on the append-only property of the oZKS. In our implementation, we instatiated the oZKS described above using the Merkle Patricia Trie-based SA, as in [16]. For each version added by the user, the server adds a label of the form $(\mathsf{uname}|i)$ to the MPT and for each version retired by the user, it adds a label of the form $(\mathsf{uname}|i|\mathsf{stale})$. So, the space complexity of st for our VKD grows linearly in the total number of updates, i.e. $|\mathsf{st}| \in \mathcal{O}(\sum_{t=0}^{n} |S_t|)$ where S_t is the input to the tth call to $\mathsf{VKD.Publish}$. This poses two problems:

- the labels committed for a given user may never be deleted, so it becomes impossible for a user to request that *all* data associated with their account be deleted. This is (arguably) in conflict with "right to be forgotten" regulations, such as those laid out in the GDPR [4].
- eventually, traversing a tree which is monotonically growing leads to unreasonable blowups in proof sizes as well as the time taken to respond to queries and make server updates.

This motivates the need for a process of secure (and transparent) compaction of storage on the VKD server. Naturally, the best way to do this would be to delete very old data stored on the VKD server, should it no longer be useful.

Overview of our paradigm. To address the issue of ever-growing storage, we extend the VKD functionality to a VKD with compaction, denoted cVKD. The compaction consists of two phases: the *tombstone phase*, a special epoch when some of the server's data is marked for garbage collection, followed, after a period of time, by, the *compaction phase*, a special epoch where values marked as tombstoned are garbage collected i.e. deleted. All other epochs are expected to function as before, i.e., internal data structures remain append-only. The auditors contribute global correctness checks to the tombstone and compaction processes by verifying that only data which is "old enough" is tombstoned (VKD.VerifyTombstone), that only tombstoned values are garbage collected (VKD.VerifyCompact) and, that in all epochs which are not used for tombstoning or deletion, the server updates its commitments correctly, as before. To ensure that values are properly garbage collected, we require that users monitor their own key history after each tombstone phase, prior to the following compaction phase. Through VKD.KeyHistory, the user checks any deletions were correct.

Soundness of a VKD with compaction. We provide a formal definition of a cVKD in appendix B. The soundness definition for a VKD is more general than a single tombstone or deletion epoch, but rather, refers loosely to the following intuition. We start with some requirements on the user for verifying her own key. Specifically, we require that between any consecutive tombstone and deletion, a user checks her key at least once. Then, "*if* a user Alice verifies her key according to the requirements and believes her own key at an epoch t was PK, *then*, any other user must also believe that Alice's key at epoch t was PK". Note that Alice checking her keys between all "tombstone-deletion" epochs induces a mapping from "epoch t to the freshest public key and version number at t". For this mapping to be unambiguous, and strictly increasing, a version number cannot be deleted and reinserted without detection.

A. Construction

We extend the oZKS-based VKD construction from appendix E to include a compaction phase, to allow garbage collection and verification of data which is no longer needed.

The following example and Figure 3 illustrate what kind of data the server may want to delete in order to compact its storage. Recall that when a user with username Alice first joins the system, with public key PK_1, the VKD adds the label Alice|1 with value PK_1 to the oZKS. When Alice updates their key from the first version PK_1 to PK_2, the server simultaneously adds the labels Alice|2 and Alice|1|stale with values PK_2 and ϵ to the oZKS. This pattern generalizes to any further updates by the user Alice. Suppose, after several years of joining, Alice is on their 10th key version, and has come online and checked their key history a few times in the interim. The data for Alice|1, PK_1 may not be useful and the server may want to reduce storage costs by deleting such entries. We motivate our final cVKD construction with preliminary attempts to support compaction directly from a VKD.

Attempt 1. The most straightforward attempt at adding a compaction functionality to the VKD built using an oZKS involves simply allowing the deletion of arbitrary, "old enough" oZKS entries by introducing a oZKS.DeleteElts functionality. We define "old enough" as a system parameter called StaleParam, with the requirement that any entries deleted by the server must have been inserted at least StaleParam epochs ago. That is, if oZKS.DeleteElts is called in some epoch t, the only entries the auditors will verify as correctly deleted were added before the epoch $t - \mathsf{StaleParam}$. In our MPT-based oZKS implementation, this is easy to support, since the epoch a leaf was added is included in the leaf's hash. From the auditors' point of view, as long as a deleted node is old enough, its deletion was valid. For privacy, the auditors should not access the plaintexts used to compute the VRF for the MPT.

What if certain users check their key history infrequently? Suppose a user Alice is at version 20 of their public key and version 10 was added a long time ago. The server could mount an attack where it deletes a label Alice|10|stale but not Alice|10. If the user Alice does not come online for a while, the server could serve the stale value at version 10 for Alice, in response to VKD.Query, then delete Alice|10. The server could do all of this before Alice comes online to check their public key. The only way to prevent this seems to be to store all states for a long time, or to have users be always online: both of which significantly degrade efficiency.

Attempt 2. We could try to patch the issues with the oZKS.DeleteElts functionality in the previous attempt by introducing a system parameter called DeletionEpochs such that an oZKS.DeleteElts proof only verifies if it is called in an epoch in the set DeletionEpochs with the important invariant that the oZKS remains append-only at all non-deletion epochs. We could then rely on the assumption that a user comes online between any two consecutive elements of DeletionEpochs to run VKD.KeyHistory on their own label. Now, if a user Alice comes online, they can ensure that (1) the epochs for version numbers are correctly ordered, for example, if the label Alice|9 was inserted at epoch t_9 and Alice|10 was inserted at t_{10}, then $t_9 < t_{10}$, (2) versions are marked stale at the appropriate epoch, in the previous example, this means that Alice|9|stale was inserted at t_{10}, and (3) for an honest IdP, labels of the form Alice|version are always inserted *before* the corresponding label Alice|version|stale, they should ensure that

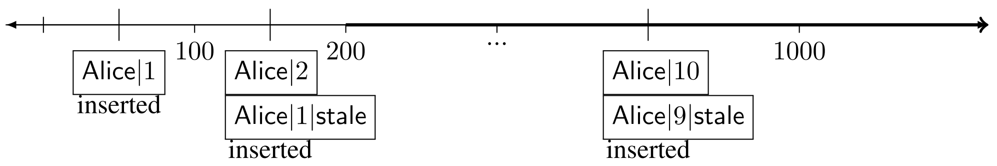


Fig. 3: An example of labels corresponding to user Alice's key updates. Since Alice has changed their key enough times since their initial entry into the system and epochs before epoch 200 are considered *old enough*, Alice|1, Alice|1|stale can be deleted.

if Alice|version|stale is deleted, then so is Alice|version.

However, an issue still remains: suppose the server has two consecutive deletion epochs del_1 and del_2, with n non-deletion epochs between them. Suppose, Alice comes online at epoch $\mathsf{del}_1 + 1$ and checks their key history and then again at epoch $\mathsf{del}_2 + 1$ checks their key history. Suppose at epoch $\mathsf{del}_1 + 1$ their latest key's version number was 10, and the server now inserted 3 fake keys for their between epochs $\mathsf{del}_1 + 2$ and del_2, i.e. their key's version right before the next deletion is fallaciously 13. As part of del_2, the server could delete the labels it added for Alice's versions 11 through 13 and the label Alice|10|stale. Such an attack could go undetected, unless Alice came online in exactly the epoch $\mathsf{del}_2 - 1$. In the presence of network delays and users who are possibly offline for long periods of time, we consider this requirement too restrictive.

Hence, we propose adding the restriction that StaleParam is at least as large as the space between two deletions. Meaning, auditors would not accept deletion of labels which were inserted more recently than the most recent deletion. In the above example, this would mean that $\mathsf{StaleParam} > \mathsf{del}_2 - \mathsf{del}_1$ and any values inserted between del_1 and del_2 would remain for Alice to check when they comes online at epoch $\mathsf{del}_2 + 1$.

Even with the restriction that $\mathsf{StaleParam} > \mathsf{max}(\mathsf{del}_{i+1} - \mathsf{del}_i)$, if the user does not immediately come online at each epoch $\mathsf{del}_i + 1$, and her key is old enough, the server could temporarily rollback her key's version. Consider the following case: a user Alice updates their key very infrequently and their most recent key, say, version 10, is old enough to fit the criteria for deletion. That is, Alice|10 was added at an epoch older than StaleParam. At the next deletion epoch del, the server could delete all record of their most recent key, i.e. the labels Alice|10 and Alice|9|stale, effectively rolling back their key to version 9. It could then re-insert the labels Alice|10 and Alice|9|stale before Alice checks their key history for this round of deletions. All of the checks we mentioned above would pass and yet, for a period of time, if another user Bob queries for Alice, they would see a different version and value than what Alice will see later. This is a subtle technical issue but preventing such an attack is integral for a construction to be considered sound!

The auditors could mitigate this issue by storing old labels and ensuring deleted labels were not reinserted. This requires auditors to be stateful and have linear storage complexity in the size of the updates. This would make the auditor's storage as large as the server's – which, our construction avoids.

We argue that the requirement that the user come online immediately after every deletion epoch is also too strong. In the next construction, we propose a design that fixes this issue.

Final construction. In our final construction, we patch this last issue by including a set of *tombstone epochs*, denoted TombstoneEpochs in the public parameters. A tombstone epoch is an epoch when items are marked for deletion, i.e. *tombstoned*, but not actually deleted. This allows users to check their own key history and ensure that values marked for deletion are appropriate, *before* they are actually deleted.

For example, if the IdP wants to delete the label Alice|10 at a deletion epoch del, it would first have to set the value of Alice|10 to TOMBSTONE at the preceding tombstone epoch TombEp. Between TombEp and del, Alice can run KeyHistory and see if this is appropriate at the VKD level.

Meanwhile, at TombEp, the auditors ensure that the only oZKS elements which are modified at TombEp are *old enough*, according to the parameter StaleParam. At all epochs between TombEp and del, the auditors continue checking append-only proofs. At del, auditors check that the only oZKS elements which are deleted have the value TOMBSTONE .

The final construction requires a client to come online between deletion epochs and check the following set of conditions. If the minimum valid version number it receives is min and its current version number is current, then:

- *Correct tombstoning or deletion.* For all versions version below min, it gets either (a) a non-membership proof for uname|version, or (b) membership proofs for uname|version and uname|version|stale, with either both of them or neither of them having the value TOMBSTONE .
- *Correct ordering.* For any version version in the range min to current $- 1$, that the epoch t_{version} when uname|version was added is less than the epoch $t_{\mathsf{version}+1}$, when uname|version $+ 1$ was added.
- *Correct version changes.* For any version version in the range min to current $- 1$, that uname|version $+ 1$ and uname|version|stale were added at the same epoch.
- *Freshness of current value.* For the current version current, it checks for the non-membership of uname|current|stale.
- *Non-membership of next few entries.* For any version $\in [\mathsf{current} + 1,\ 2^{\lfloor \log(\mathsf{current}) \rfloor + 1})$, uname|version wasn't added.
- *Non-membership of much further entries.* For L, the most recent server epoch, for any $j \in [2^{\lfloor \log(\mathsf{current}) \rfloor + 1},\ 2^{\lfloor \log(L) \rfloor}]$, it checks a non-membership proof for uname$|2^j$.

Soundness. We give a detailed proof for the soundness of this construction in appendix B. The intuition for soundness is as follows: For the most part, this construction with deletion is identical to the one in section III, since all but a designated set of epochs is append-only. The only divergence occurs at tombstone or deletion epochs, and we show that dishonest server behavior cannot go undetected, even in these epochs.

We require that a user must check their key history once for each tombstone epoch, prior to the following deletion. If a user's key is at $\mathsf{version}$, the server can show the wrong key for a user in one of the following ways. The server could try to change the public key committed to the label $\mathsf{version}$: this is mitigated by the auditors checking that the only change being made is tombstoning or deletion of already tombstoned values. If the server tombstones the label $\mathsf{uname}|\mathsf{version}$, the user will detect that it is tombstoned since they will come online before the next deletion. The second option is that the server could show an older key, i.e. show a key for some version $\mathsf{version}' < \mathsf{version}$ but this should be detected by the one of the key history checks. The third option is that it could try to add a higher version $\mathsf{version}' > \mathsf{version}$: this is already detected by the construction in section III. The fourth option is that it could try to re-insert an old version number, which was deleted, with a different key: this particular attack is unique to the scenario with deletions and is detected by the key history check, since if the server re-inserts an old version number, the user will see this the next time they comes online, which will be before the server has the opportunity to hide this change.

Assumptions. Recall that previous works, as well as our construction in section III, make two major assumptions. First, they assume that the server's underlying data structure is append-only, i.e. the identity provider could never delete any verification data. Second, that users are not always online. The guarantee is that if a user checks their key history for a period of time, they will catch the a cheating server. In fact, there is no upper bound on the time the user can wait before checking their key history but still get the same guarantee. This construction changes these assumptions as follows:

1) We assume that the system only considers a value stale-enough to tombstone, if it was added *before* the previous tombstone. The ability to tombstone and delete values is already an improvement over previous, append-only verifiable data structures. We argue that the compromise of keeping data for a certain period of time is still a major improvement for long-term storage costs.
2) Instead of allowing the user to go for an unbounded amount of time before checking their key history, we require that they come online between each pair of tombstone and corresponding deletion epochs and check their key history to get guarantees for that period. Since large servers only need to run deletions infrequently, while more strict that the previous assumption, this still provides a user with leeway. In fact, this approach also has the added benefit that there is an upper bound on the amount of time the server can cheat for a particular user and go undetected.

Leakage. The leakage of the subset of algorithms of this cVKD construction, which are inherited from the definition of a VKD is identical for this construction. For $\mathsf{cVKD.TombstoneElts}$ and $\mathsf{cVKD.Compact}$, they both leak the size of the cVKD at the start of the operation. Also, $\mathsf{cVKD.TombstoneElts}$ leaks the number of tombstoned values and $\mathsf{cVKD.Compact}$ leaks the updated size of the cVKD.

Space efficiency. A compressed MPT is constructed with the invariant that any node with only one child is deleted and replaced by its child. This means that when a new leaf is inserted, somewhere in the tree an additional new node will have to be inserted to accommodate this leaf. When a leaf is deleted, its immediate parent can also be deleted. Hence, for each leaf deleted from this MPT-based VKD, the equivalent of 2 MPT nodes worth of data can be erased from the server.

V. The Parakeet Consistency Protocol

Existing transparency overlays typically fall in two categories:

- Reactive Client-side Auditing: These protocols [34] offload the consistency protocol to the clients. Clients are expected to gossip among themselves about their views of the system and construct fraud proofs showing that the IdP has misbehaved. Unfortunately, this class of protocols is too optimistic for a security-critical infrastructure and too heavy for the end user. As a result, there is a high risk that users do not eagerly gossip proofs or that the adversary launches targeted eclipse attacks [24] to isolate targeted users. Finally, gossip relies on always-connected nodes forming a connected graph, perpetually routing messages. This is impractical if the many of nodes in the gossip network are small devices with intermittent network connectivity and rely on battery power.
- Using a black-box blockchain as a trust anchor: This approach [41], [16] guarantees consistency as long as the blockchain is secure, i.e., it does not have forks. This alleviates any risks for clients who only need to trust the black-box blockchain, however, (i) it introduces a significant extension of the trust assumptions, which might not be correct given the multitude of attacks blockchains suffer [11], (ii) it limits the update speed of the system to the finality speed of the blockchain (which is tens of minutes in Bitcoin), and (iii) it doesn't solve the problem of eclipse attacks against clients as they remain a challenge in blockchains. Also, no existing protocols has provided a detailed security analysis for the interactions with any existing blockchain.

One way to avoid these issues is having auditors run consensus on each update of the IdP, essentially replacing the trust assumptions of the blockchain with a custom-made blockchain just for the transparency layer. However, this is simply an overkill. As a final contribution, Parakeet shows a consensus-less protocol that achieves all the desired properties for defending against split-view attacks. Section V-C further extends this protocol to provide censorship resistance and read-freshness.

A. *Consensus-less Strong Consistency*

The Parakeet protocol consists of network messages exchanged between the participants (Section II-A). The users communicate with the identity provider (IdP) and witnesses. The IdP communicates with the witnesses and users, but the witnesses *need not communicate directly with each other*.

Updating the state. All updates to the state of the IdP start with an *update request* sent by a user to the IdP. The request contains a user identifier id and the new key $\mathsf{pk}_{\mathsf{new}}$ associated with id. The IdP collects several user requests and runs the

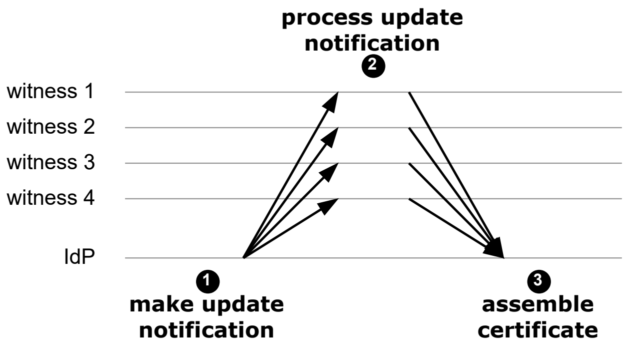


Fig. 4: Illustration of the key update protocol.

key-update protocol illustrated in Figure 4. The IdP broadcasts an *update notification* to the witnesses to notify them that the state of the key directory has been updated (❶). This notification contains the following fields: (i) a commitment to its state com_t, (ii) the epoch t, (iii) a proof π^{Upd} that com_t is a valid commitment and that the state is a valid update of the previous state, and (iv) a signature by the identity provider over this data. Each witness locally verifies that the epoch number t has been incremented by 1 and that the proof π^{Upd} is valid; it then counter-signs the update notification (❷). Finally, the IdP collects at $2/3f+1$ witnesses' signatures into a *certificate* (❸) which is attesting that a quorum of witnesses verified each past update and the IdP could not have equivocated.

Reading the state. The user sends a *read request* to the IdP to request the key associated with a specific identifier. This request contains the identifier key to query. The identity provider replies to read requests with a message containing the following fields: (i) the key key associated with the identifier, (ii) a certificate from the witnesses over its latest update request, (iii) a proof that key is included in the certified state, and (iv) a signature by the identity provider over this data.

B. Proofs Sketches

We provide the intuition for the proofs of consistency, validity, and termination of the protocol defined in Section II.

Consistency. We prove consistency by contradiction. Let's assume two correct users output different commitments com_t and $\mathsf{com}_{t'}$ for the same epoch t. Then com_t is signed by $2f+1$ witnesses out of which $f+1$ are assumed honest. Similarly, $\mathsf{com}_{t'}$is signed by $2f+1$ witnesses out of which $f+1$ are honest and did not sign $\mathsf{com}_{t'}$. But then there should be $f+1+f+1$ honest and f malicious witnesses. But $n=3f+1<3f+2$, hence a contradiction.

Validity. Validity directly follows from the integrity property of the signature scheme used by the IdP and the soundness of the proof π^{Upd}. Honest witnesses only counter-sign update notifications if they are correctly signed by the IdP and the proof π^{Upd} verifies. As a result, there cannot exist a valid certificate over an invalid update.

Termination. Assuming all messages are eventually delivered and there exist $2f+1$ honest witnesses then the honest IdP will send a single commitment for each epoch t which all honest witnesses will counter-sign and produce a valid certificate for.

C. Anti-Censorship and Freshness Subprotocol

We provide an optional enhancement of the consistency protocol presented in Section V providing censorship resistance from malicious IdP. This protocol allows users to tie the liveness of their update requests with the liveness of the entire system effectively defending against selective censorship attacks. The protocol works as follows:

- Users whose updates keep not appearing in the IdP's state send their update request to the witnesses.
- Every epoch witnesses collect these requests and forward them to the IdP.
- Once the witness has forwarded a user update request, it stops signing any future IdP's update notification that do not come with a proof that the user update has been included in the state.

As a result the moment the censored client has sent their update request to $f+1$ honest witnesses, the liveness of the protocol is tied to the update being included. Hence the IdP can now only act as crashed and halt the whole system if it wants to censor the update (at which point no other requests can be processed).

A second enhancement of the consistency protocol is that during an update the witnesses send their signature over not only the hash of the tree but also their local timestamp. Then clients are given the option to ask for $2f+1$ timestamps during a read operation. Given that a median timestamp is robust to f faults and assuming bounded clock-drift clients can deduce the freshness of the state tree received as reply. They can thus choose to ignore the reply if it is too old (e.g., more than a day old).

Given that now the updates are coupled to anti-censorship and clients are aware of the time the state is updated in Parakeet, we guarantee that key updates are included in a timely manner in the state and every client quickly receives the new update (otherwise a malicious IdP is forced to halt the whole system).

VI. Implementation and Benchmarks

We implement our VKD scheme presented in Section III in Rust. We also implement a networked multi-core eventually synchronous Parakeet based on our VKD. It uses `tokio` [7] for asynchronous networking, `ed25519-dalek` [8] for elliptic curve based signatures, and data-structures are persisted using `Rocksdb` [9] (unless otherwise specified). It uses TCP to achieve reliable point-to-point channels, necessary to correctly implement the distributed system abstractions. We have open-sourced both our VKD [5] and consistency protocol [5] implementations (which together form Parakeet), including our Amazon Web Services (AWS) orchestration scripts and measurements data to enable reproducible results [10].

Benchmarks. We performed several benchmarks to thoroughly assess the practicality of Parakeet. To this end, we used two kinds of AWS instances, both with up to 5Gbps of bandwidth, running Linux Ubuntu server 20.04, running on 2.5GHz Intel Xeon Platinum 8259CL machines: (1) `t3.medium` instances with 2 virtual CPUs (1 physical core) and 4GB memory. These machines are relatively cheap due to their low specs and we chose them to assess how performant the witnesses of Parakeet are. (2) `t3.2xlarge` instances with 8 virtual CPUs and 32GB of memory. These machines are used to benchmark the most heavyweight IdP operations, namely publish. We implemented IdP and Witnesses in Rust and for signatures we used `ed25519_dalek` with 32-byte public/private keys pairs.

A. Microbenchmarks

We microbenchmark the overhead of the consistency protocol on the AWS `t3.medium` instance and the IdP capabilities of our implementation on the `t3.2xlarge` instances. We set the committee size to 4 witnesses, the size of keys and values to 32 bytes each (i.e. a key-value pair is 64 bytes), and a batch size of 1,024 key-values per batch. In the tables below, every measure displays the mean, standard deviation over 10 runs.

CPU analysis. Table II shows the results of microbenchmarks for the following VKD operations:

1) *Create notification:* The IdP generates a batch of (random) public keys and publishes them. Then it generates an audit (append-only) proof over this publish operation. The audit proof along with the the new tree root and the sequence number are signed by the IdP. This message constitutes a notification. Note that the result in table II only includes the cost of the consistency and audit related operations, which we derive by subtracting the server-side publish cost from the total cost of publishing *and* creating a notification.
2) *Verify notification:* A witness verifies a notification. This step consists of the verification of the IdP-generated signature and the audit proof.
3) *Create vote:* A witness signs the verified notification and creates a vote.
4) *Verify vote:* A witness verifies a vote.
5) *Aggregate certificate:* A list of votes are combined to form a certificate. The number of votes needed to create a certificate is $2f + 1$ (see Section V). With a four-witness setup, this means combining 3 votes.
6) *Verify certificate:* All 3 votes in a certificate are verified.

The slowest operations are the generation and signing of audit proofs, dispatching them to witnesses, *Create notification*, and the verification of the signature and audit proof presented in Create notification, i.e. *Verify notification*. Similarly, verification of these two contributed to the second slowest operation *Verify notification*. Comparatively, remaining operations were very fast – completed under one millisecond.

Storage analysis. To examine the storage costs of Parakeet, as well as SEEMless, we inserted 100,000 users at a time to the respective VKDs with a MySQL-based storage layer, running locally in a docker container. Figure 5 shows the storage cost and its breakdown for Parakeet's VKD, as the number of users increases. For 5M users, the VKD requires about 4.5GB and this cost grows linearly in number of users.

Measure	Mean (ms)	Std. (ms)
Verify notification	68.70	1.50
Create vote	0.02	0.00
Verify certificate	0.16	0.01

TABLE I: Microbenchmark of single core CPU overhead of the consistency protocol operations of Parakeet on witness machines. Committee of 4 witnesses; notifications batch 1,024 updates of 64 bytes each; average and standard dev. of 10 measurements.

Measure	Mean (ms)	Std. (ms)
Create notification	47.12	9.30
Verify vote	0.07	0.00
Aggregate certificate	0.00	0.00

TABLE II: Microbenchmark of witness CPU overhead of the consistency protocol operations of Parakeet on the IdP machine; notifications batch 1,024 updates of 64 bytes each; average and standard dev. of 10 measurements.

At this rate, we would expect roughly 850GB for 1 billion users – demonstrating the scalability of our solution.

Figure 6, shows the time it takes to run a publish operation on Parakeet's VKD. The storage layer of this experiment implements a cache and uses a MySQL database for persistent storage. Each of these update operations, as well as the ones in fig. 5 required only one or two persistent storage accesses each, and the storage accesses made up a majority of the time it took for the publish operation to complete. For example, for inserting 100k users on top of a VKD containing 500k existing users, it took about 19 minutes, of which 12 were for MySQL writes. We attempted to run similar benchmarks for SEEMless. Our implementation of an aZKS included all the same caching optimizations as the oZKS implementation we used for Parakeet. With persistent memory, however, this data structure became infeasible at fairly small scale. For example, inserting 100k new users into a set of 300k users and this operation took $\approx$ 66 minutes. Of this, 51 were spent writing to persistent storage and about 7 were spent on reading from it. This took 47 persistent storage accesses. In appendix I, we show that as the number of epochs increases, for the same set of users, SEEMless sees a large storage size blowup, versus Parakeet, whose storage size remains constant.

B. End-to-end benchmarks

We evaluate the throughput and latency of our implementation of Parakeet through experiments on AWS. We particularly aim to demonstrate the following claims:

(C1) Parakeet achieves enough throughput to operate at planetary scale.
(C2) Parakeet achieves low latency even under high load, in the WAN, and with large committee sizes.
(C3) Parakeet runs efficiently on cheap machines with low specs (comparable to common HSMs).
(C4) Parakeet is robust when some parts of the system inevitably crash-fail. Note that evaluating BFT protocols in the presence of Byzantine faults is still an open question [12].

We deploy a testbed on AWS, using `t3.medium` instances across 5 different AWS regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm

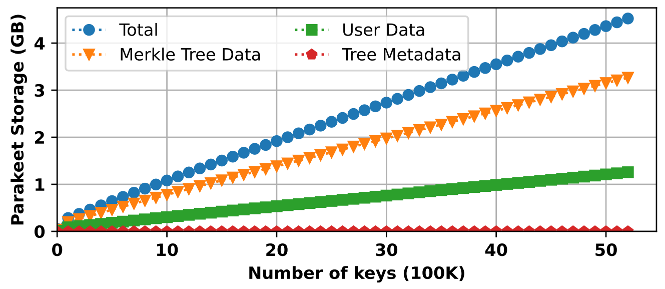


Fig. 5: Memory consumption of Parakeet's VKD. Measurements at intervals of 100k new users and up to 4.5M users. Total memory consumption is the sum of memory required by the Merkle Patricia Trie data, some metadata and the original database of usernames and public keys.

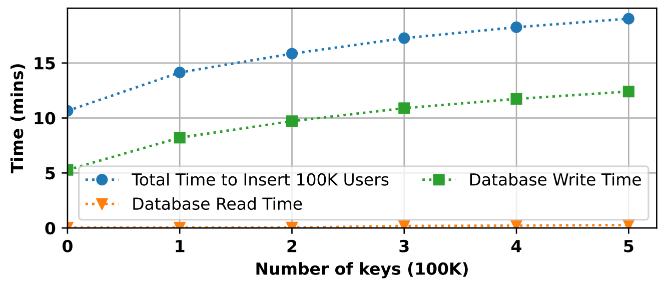


Fig. 6: Time to publish with insertion of a batch of 100k new users into Parakeet's VKD, with an existing VKD of sizes 0-500k. This graph also includes the time for database reads and writes for each insertion.

(eu-north-1), and Tokyo (ap-northeast-1). Witnesses are distributed across those regions as equally as possible.

In the following sections, each measurement in the graphs is the average of 3 independent runs, and the error bars represent one standard deviation; errors bars are sometimes too small to be visible on the graphs. Our baseline experiment parameters are 10 honest witnesses, a batch size of 1024, and an update size of 64B. We instantiate one benchmark client (collocated on the same machine as the IdP) submitting transactions at a fixed rate for a duration of 5 minutes. When referring to *latency*, we mean the time elapsed from when the client submits a request to when the IdP receives confirmation that the request is successfully processed. We measure it by tracking sample requests throughout the system.

Benchmark in the common case. Figure 7 illustrates the latency and throughput of Parakeet for varying numbers of witnesses. The maximum throughput we observe is around 800 updates/s while keeping the latency below 3.5 seconds. Based on the system usages estimates for the large-scale end-to-end encrypted messaging service WhatsApp (Section I), we would arrive at the requirement to process around 120 updates/s. Parakeet exceeds by over 6x the throughput required to operate at this scale, and thus satisfies claims (C1) and (C3).

Figure 7 also illustrates that performance do not vary with 10, 20 or even 50 witnesses. This observation concurs with Section VI-A showing that the bottleneck of Parakeet is the IdP. Increasing the number of geo-distributed witnesses up to 50 doesn't impact the end-to-end performance of the system; Parakeet thus satisfies claim (C2). We however expect that keeping increasing the number of witnesses will eventually make the network to become the system's bottleneck.

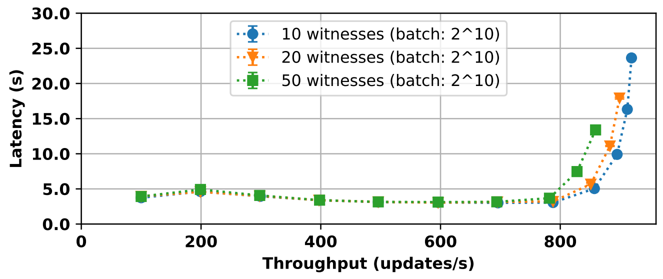


Fig. 7: Throughput-latency performance of Parakeet. WAN measurements with 10, 20, 50 witnesses. No faulty witnesses, 1024 maximum batch size, and 64B updates size.

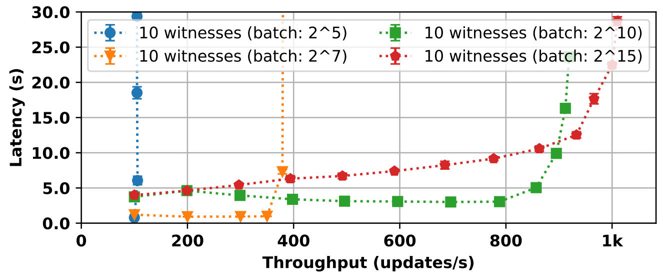


Fig. 8: Throughput-latency performance of Parakeet. WAN measurements with 10 witnesses. No faulty witnesses, various batch sizes, and 64B updates size.

Figure 8 illustrates the performance of Parakeet when varying the batch size form 2^5 to 2^{15}. The maximum throughput we observe for batches sizes of 2^5 and 2^7 is respectively 100 updates/s and 350 updates/s. This is much lower than the 800 updates/s that Parakeet can achieve when configured with a batch size over 2^{10}. Small batch sizes, however, allow Parakeet to trade throughput for latency. Parakeet configured with a batch size of 2^5 can process up to 100 updates/s in under 800ms, setting the batch size to 2^7 allows Parakeet to operate at scale while robustly maintaining sub-second latency.

Benchmark under crash-faults. Figure 9 depicts the performance of Parakeet when a committee of 10 witnesses suffers 1 to 3 (crash-)faults (the maximum that can be tolerated in this setting). It shows that Parakeet's performance is not affected by (crash-)faults, thus satisfying claim (C4).

Contrarily to BFT consensus systems [28], Parakeet maintains a good level of throughput under crash-faults. The underlying reason for steady performance under crash-faults is that Parakeet doesn't rely on a leader to drive the protocol. This is in sharp contrast with related work (e.g. [16], [13], [41]) that rely on an external blockchain for consistency.

VII. Related Work

Key transparency. As discussed in Section I, this work extends SEEMless [16] by instantiating a light-weight consistency protocol to prevent server equivocation in a practical setting, while also extending SEEMless to handle real-world constraints on storage capacity, efficiency, and scalability to billions of users. While Keybase ([23], [26]) was the first deployment of an auditable public key directory (created as a user-friendly alternative for PGP), CONIKS [34], was the

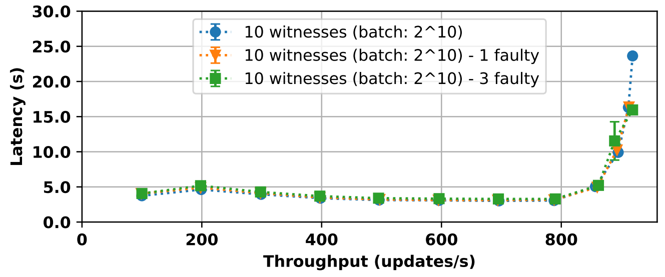


Fig. 9: Comparative throughput-latency under crash-faults of Parakeet. WAN measurements with 10 witnesses. Zero, one, and three crash-faults, 1024 maximum batch size, and 64B updates size.

first academic work that formalized the notion of a service that maintains and periodically commits to a public key directory. SEEMless itself can be seen as an extension to CONIKS. CONIKS essentially relies on a synchronous gossip protocol among users in order to detect server equivocation. Unfortunately, this assumption is both hard to scale efficiently for millions of users and easily breakable over the internet.

Since then, multiple works have proposed different ways of publishing directory commitments and serving audit proofs to clients. EthIKS [13] and Catena [41] demonstrate how to leverage a public blockchain (Ethereum and Bitcoin, resp.) as a ledger to provide non-equivocation guarantees and support auditability for the commitment to the CONIKS directory, while minimizing bandwidth overhead for its clients. In this paper we show that the strong assumption of using consensus is unnecessary and instead propose a lightweight consistency protocol. Mog [33] takes a different approach by proposing a gossip protocol and a verifiable registry that allows individual clients to perform their own auditing. The gossip protocol still requires synchrony, however, it relies on the assumption of sleepy committees to solve the consensus problem. In theory, this would enable scalability but could hinder liveness for long periods when there are not enough available witnesses or when the network is unstable. In this work, instead, we have shown that consensus is unnecessary for guaranteeing a consistent view of the tree to the clients which enables us to provide consistent views without relying on a good network. Finally, KTACA [45] relies on clients to (1) gossip through out-of-band channels to share consistent commitments to a directory, (2) run any global client audits. Our work could be extended to using sleepy committees or gossip, however we opted to use highly available witnesses that should be deployed by professional services (e.g., professional blockchain validators) and will provide timely security to the users. KTACA additionally combines the gossip approach with anonymous key history checks and lookups to prevent an adversarial server from causing targetted forked views of its directory. Allowing anonymous lookups, however, goes counter to our privacy requirement, that only authenticated and permitted users be permitted to query.

Several recent works on key transparency have focused on improving auditor efficiency. Merkle^2 [25] proposes a solution which reduces the amount of work required for an auditor to verify a series of key updates from linear to logarithmic, assuming that key updates can be signed by clients through the use of "signature chains". However, it is unclear what integrity protection the system will provide if the signature keys of the clients are lost. Assuming that the clients can maintain long term cryptographic secret keys is unrealistic, especially in the setting of key transparency, where the focus is on building a PKI for clients who cannot remember cryptographic key material. AAD [40], Aardvark [29], Tomescu et. al. [42], Tyagi et. al. [43], and Verdict [44] have proposed using accumulators (bilinear and RSA) and SNARKs as commitments in order to make auditor verification more efficient. However, the computational overhead incurred from relying on the algebraic assumptions themselves can outweigh the asymptotic improvements over the number of key updates per epoch.

Atomic transactions. Atomic transactions [27] allow all-or-none type of execution for a set of operations. In large systems, they are often a necessity since the underlying database can end up in an inconsistent state if operations are not sequentially executed (e.g., withdrawing money from account A and depositing it to account B). To this end, several solutions have been proposed in the literature—[37], [39], [20] to name a few. Although they are a strong primitive for building concurrent applications, transactions come with their cost; locks might leave the systems in a dead-locked state whereas failure in a single operation can cancel a transaction and might require re-execution of the whole set; or alternatively they might not be supported cross-shard [19].

In VKD, we side-step such issues by (1) executing a publish operation by a single writer and (2) preserving the previous value of a node. (1) is needed to ensure that concurrent publish operations not overwrite nodes' updated states and (2) allows us to allow concurrent reads (e.g., lookup proof generation) and writes (i.e., publish operation). In result of these two properties, we only require that the update order is preserved only between the node updates and the latest epoch, i.e., we allow the nodes to be updated in any order.

The downside of this approach is that the storage cost is effectively doubled. Yet, we believe this is an acceptable trade-off due to drastic storage reduction compared to existing key transparency solutions such as SEEMless [16] and the flexibility to use any storage as the underlying key directory.

VIII. Conclusion

While much recent effort has focused on various aspects of key transparency, large-scale applications on the order of billions of users have not previously been considered. We expose the gaps in purely academic scale implementations and bridge these gaps is our design of Parakeet, a key transparency system with large-scale deployment in mind. Our production-grade implementation of Parakeet shows the feasibility of our approach, which we further demonstrate through experiments.

Acknowledgments

This work is supported by the Novi team at Meta. Harjasleen Malvai was funded in part by IC3 industry partners and NSF grant 1943499.

References

[1] "How whatsapp enables multi-device capability," https://engineering.fb.com/2021/07/14/security/whatsapp-multi-device/, [Online; accessed 5-July-2022].

[2] "Most popular global mobile messenger apps as of January 2022, based on number of monthly active users," https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/, [Online; accessed 5-July-2022].

[3] "Whatsapp revenue and usage statistics (2022)," https://www.businessofapps.com/data/whatsapp-statistics/, [Online; accessed 5-July-2022].

[4] https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN#d1e2606-1-1, 2015.

[5] https://github.com/facebook/akd, 2022.

[6] https://github.com/Microsoft/oZKS, 2022.

[7] https://github.com/tokio-rs/tokio, 2022.

[8] https://github.com/dalek-cryptography/ed25519-dalek, 2022.

[9] https://rocksdb.org/, 2022.

[10] https://github.com/asonnino/key-transparency/tree/main/scripts, 2022.

[11] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Sok: Consensus in the age of blockchains," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 183–198.

[12] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," in *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[13] J. Bonneau, "Ethiks: Using ethereum to audit a coniks key transparency log," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 95–105.

[14] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[15] D. Catalano, D. Fiore, and M. Messina, "Zero-knowledge sets with short proofs," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2008, pp. 433–450.

[16] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai, "Seemless: Secure end-to-end encrypted messaging with less trust," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1639–1656.

[17] M. Chase, A. Healy, A. Lysyanskaya, T. Malkin, and L. Reyzin, "Mercurial commitments with applications to zero-knowledge sets," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2005, pp. 422–439.

[18] B. Chen, Y. Dodis, E. Ghosh, E. Goldin, B. Kesavan, A. Marcedone, and M. E. Mou, "Rotatable zero knowledge sets: Post compromise secure auditable dictionaries with application to key transparency," Cryptology ePrint Archive, Paper 2022/1264, 2022, https://eprint.iacr.org/2022/1264. [Online]. Available: https://eprint.iacr.org/2022/1264

[19] A. Cheng, X. Shi, L. Pan, A. Simpson, N. Wheaton, S. Lawande, N. Bronson, P. Bailis, N. Crooks, and I. Stoica, "Ramp-tao: layering atomic transactions on facebook's online tao data store," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 3014–3027, 2021.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[21] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[22] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[23] N. Group, https://keybase.io/docs-assets/blog/NCC_Group_Keybase_KB2018_Public_Report_2019-02-27_v1.3.pdf, 2019.

[24] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *USENIX 2015*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 129–144. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman

[25] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa, "Merkle^2: A low-latency transparency log system," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 285–303.

[26] Keybase, https://keybase.io/_/api/1.0/merkle/root.json?seqno=1, 2014.

[27] B. W. Lampson, "Atomic transactions," in *Distributed Systems—Architecture and Implementation*. Springer, 1981, pp. 246–265.

[28] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru, "Turret: A platform for automated attack finding in unmodified distributed system implementations," in *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 660–669.

[29] D. Leung, Y. Gilad, S. Gorbunov, L. Reyzin, and N. Zeldovich, "Aardvark: A concurrent authenticated dictionary with short proofs," Cryptology ePrint Archive, Report 2020/975, 2020, https://eprint.iacr.org/2020/975.

[30] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed computing*, vol. 11, no. 4, pp. 203–213, 1998.

[31] M. Marlinspike, "Advanced cryptographic ratcheting," https://signal.org/blog/advanced-ratcheting/, 2013.

[32] ——, "Whatsapp's signal protocol integration is now complete," https://signal.org/blog/whatsapp-complete/, 2016.

[33] S. Meiklejohn, P. Kalinnikov, C. S. Lin, M. Hutchinson, G. Belvin, M. Raykova, and A. Cutter, "Think global, act local: Gossip and client audits in verifiable data structures," *arXiv preprint arXiv:2011.04551*, 2020.

[34] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 383–398.

[35] S. Micali, M. Rabin, and J. Kilian, "Zero-knowledge sets," in *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 2003, pp. 80–91.

[36] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "{CHAINIAC}: Proactive software-update transparency via collectively signed skipchains and verified builds," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1271–1287.

[37] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[38] K. Pothong, L. Pschetz, R. Catlow, and S. Meiklejohn, "Problematising transparency through LARP and deliberation," in *DIS '21: Designing Interactive Systems Conference 2021, Virtual Event, USA, 28 June, July 2, 2021*, W. Ju, L. Oehlberg, S. Follmer, S. E. Fox, and S. Kuznetsov, Eds. ACM, 2021, pp. 1682–1694.

[39] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 385–400.

[40] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas, "Transparency logs via append-only authenticated dictionaries," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1299–1316.

[41] A. Tomescu and S. Devadas, "Catena: Efficient non-equivocation via bitcoin," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 393–409.

[42] A. Tomescu, Y. Xia, and Z. Newman, "Authenticated dictionaries with cross-incremental proof (dis)aggregation," Cryptology ePrint Archive, Report 2020/1239, 2020, https://eprint.iacr.org/2020/1239.

[43] N. Tyagi, B. Fisch, J. Bonneau, and S. Tessaro, "Client-auditable verifiable registries," *Cryptology ePrint Archive*, 2021.

[44] I. Tzialla, A. Kothapalli, B. Parno, and S. Setty, "Transparency dictionaries with succinct proofs of correct operation," *IACR Cryptol. ePrint Arch.*, p. 1263, 2021. [Online]. Available: https://eprint.iacr.org/2021/1263

[45] T. K. Yadav, D. Gosain, A. Herzberg, D. Zappala, and K. Seamons, "Automatic detection of fake key attacks in secure messaging," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3019–3032.

Appendix

A. VKD with Compaction Glossary

In order to discuss details about our approach to the compaction of server-side storage, we will need some notation. We gather notation for our construction here:

- TOMBSTONE : A special string to replace the value of an oZKS entry which is valid to delete.
- TombstoneEpochs: The set of epochs when tombstone marking is permitted. No insertions should be allowed in this epoch, only tombstoning.
- CompactionEpochs or DeletionEpochs: The set of epochs when entries with value TOMBSTONE are permitted to be deleted from the oZKS.
- StaleParam: A system parameter for calculating the upper bound for the epochs in which a node must have been inserted in order to be valid for tombstoning.
- DeletionParam: A system parameter defining the number of epochs between a tombstone epoch and the subsequent compaction epoch. Assuming CompactionEpochs and TombstoneEpochs are sorted in increasing order, CompactionEpochs $= \{t_i +$ DeletionParam $\mid t_i$ is the ith element of TombstoneEpochs$\}_i$.

Also note that we use the terms identity provider and server interchangeably.

B. Formal Definitions

Definition 1. *Given a datastore,* DS, *i.e. a set of tuples* $(\mathsf{label}_i, \mathsf{val}_i)$*,* DS.Labels*, denotes the set* $\{\mathsf{label} \mid \exists(\mathsf{label}, \cdot) \in \mathsf{DS}\}$.

Definition 2. *An ordered append-only zero-knowledge set (with compaction) (*oZKS*) is a data-structure with a set of public parameters* p*, where* p *includes a set* TombstoneEpochs *of epochs, and parameters* StaleParam, DeletionParam*. An* oZKS *supports the following algorithms* (oZKS.CommitDS, oZKS.QueryMem, oZKS.QueryNonMem, oZKS.VerifyMem, oZKS.VerifyNonMem, oZKS.InsertionToDS, oZKS.VerifyInsertions, oZKS.TombstoneElts, oZKS.VerifyTombstone, oZKS.DeleteElts, oZKS.VerifyDel) *described below:*

- $(\mathsf{com}, \mathsf{st_{com}}, \mathsf{DS}_1) \leftarrow \mathsf{oZKS.CommitDS}(1^\lambda, \mathsf{DS})$*: This algorithm takes in a datastore* DS *consisting of label-value pairs* $(\mathsf{label}_i, \mathsf{val}_i)$ *with unique keys, and a security parameter* λ *and outputs:* $\mathsf{st_{com}}$*, an internal state;* com*, a commitment to* DS*;* $\mathsf{DS}_1 = \{(\mathsf{label}_i, \mathsf{val}_i, 1) \mid (\mathsf{label}_i, \mathsf{val}_i) \in \mathsf{DS}\}$*, here* 1 *is the* oZKS *epoch, at which each tuple from* DS *was inserted and* aux *is any extra information needed for for this tuple.*
- $(\mathsf{com}', \mathsf{st_{com'}}, \mathsf{DS}_{t+1}, \pi_S, t+1) \leftarrow \mathsf{oZKS.InsertionToDS}(\mathsf{st_{com}}, \mathsf{DS}_t, S, t)$*: This algorithm takes in the current datastore* DS_t*, the internal server state* $\mathsf{st_{com}}$*, the current server epoch* t *and a set* S *of label-value pairs* $\{(\mathsf{label}_i, \mathsf{val}_i)\}$ *for an update, such that* $(\mathsf{label}_i, \cdot, \cdot) \notin \mathsf{DS}_i$*. The algorithm computes* $S' = \{(\mathsf{label}, \mathsf{val}, t+1) \mid (\mathsf{label}, \mathsf{val}) \in S\}$*. It returns* $\mathsf{DS}_{t+1} = \mathsf{DS}_t \cup S'$, com$'$*, the commitment to* DS_{t+1}*,* $\mathsf{st_{com'}}$ *the updated internal state of the server and a proof* π_S *that* com$'$ *is a commitment to* DS_{t+1} *such that* $\mathsf{DS}_t \subseteq \mathsf{DS}_{t+1}$ *and each entry of* $\mathsf{DS}_{t+1} \setminus \mathsf{DS}_t = \{(\mathsf{label}, \mathsf{val}, t+1)$ *for some* label *and* val$\}$.
- $0/1 \leftarrow \mathsf{oZKS.VerifyInsertions}(\mathsf{com}, \mathsf{com}', \pi_S, t)$*: Verifies the proof* π_S *that* com, com$'$ *commit to some datastores* DS, DS$'$ *respectively, such that* $\mathsf{DS} \subseteq \mathsf{DS}'$ *and that* $\mathsf{DS}' \setminus \mathsf{DS} = \{(\mathsf{label}, \mathsf{val}, t)$ *for some* label, val$\}$.
- $(\pi, \mathsf{val}, t)/\bot \leftarrow \mathsf{oZKS.QueryMem}(\mathsf{st}, \mathsf{DS}, \mathsf{label})$*: This algorithm takes a datastore* DS *and a corresponding internal state* st*, and a label* label*. If there exists an entry* $(\mathsf{label}, \mathsf{val}, t) \in \mathsf{DS}$*, then the algorithm returns* (π, val, t) *where* t *is the* oZKS *epoch when the tuple with the label* label *was inserted. If* $(\mathsf{label}, \cdot, \cdot) \notin \mathsf{DS}$*, return* $\bot$.
- $0/1 \leftarrow \mathsf{oZKS.VerifyMem}(\mathsf{com}, \mathsf{label}, \mathsf{val}, t, \pi)$*: This algorithm takes in a triple* label, val, t*, the membership proof* π *for* $(\mathsf{label}, \mathsf{val}, t)$ *with respect to* com*. It outputs* 1 *if the proof verifies and* 0 *otherwise.*
- $\pi/\bot \leftarrow \mathsf{oZKS.QueryNonMem}(\mathsf{st}, \mathsf{DS}, \mathsf{label})$*: This algorithm takes a datastore* DS *and a corresponding internal state* st*, and a label* label*. It returns* π*, a proof of non-membership of* label *in* DS.
- $0/1 \leftarrow \mathsf{oZKS.VerifyNonMem}(\mathsf{com}, \mathsf{label}, \pi)$*: This algorithm takes in a* label, com *and a non-membership proof* π*. It outputs* 1 *if the proof verifies and* 0*, if not.*
- $(\mathsf{com}', \mathsf{st_{com'}}, \mathsf{DS}_{t+1}, \pi_S, t+1) \leftarrow \mathsf{oZKS.TombstoneElts}(\mathsf{st_{com}}, \mathsf{DS}_t, S, t, t_{\mathsf{stale}})$*: This algorithm should only be called if* $t \in$ TombstoneEpochs *and* $t_{\mathsf{stale}} = t -$ StaleParam*. It algorithm takes in the current datastore* DS_t*, the internal server state* $\mathsf{st_{com}}$*, the current server epoch* t*, a stale epoch parameter* t_{stale} *and a set* S *of triples* $\{(\mathsf{label}_i, \mathsf{val}_i, t_i)\}$ *for an update, such that* $(\mathsf{label}_i, \mathsf{val}_i, t_i) \in \mathsf{DS}_t$*. The algorithm checks that for each* $(\mathsf{label}_i, \mathsf{val}_i, t_j) \in S$*,* $t_j \leq t_{\mathsf{stale}}$*. It initializes* $\mathsf{DS}_{t+1} = \mathsf{DS}_t$ *and for each* $(\mathsf{label}_i, \mathsf{val}_i, t_j) \in S$*, it replaces the entry* $(\mathsf{label}_i, \mathsf{val}_i, t_j)$ *of* DS_{t+1} *with* $(\mathsf{label}_i,$ TOMBSTONE $, t_j)$*. It returns this updated* DS_{t+1}*,* com$'$*, the commitment to* DS_{t+1}*,* $\mathsf{st_{com'}}$ *the updated internal state of the server and a proof* π_S *that* com$'$ *is a commitment to the data store* DS_{t+1} *such that (1) if* $(\mathsf{label}_i,$ TOMBSTONE $, t_j) \in \mathsf{DS}_{t+1}$*, then* $(\mathsf{label}_j, \mathsf{val}_j, t_j) \in \mathsf{DS}_t$*, for some* val_j*, with* $t_j \leq t_{\mathsf{stale}}$*; (2) if* $(\mathsf{label}_k, \mathsf{val}_k, t_k) \in \mathsf{DS}_t$ *with* $t_k > t_{\mathsf{stale}}$*, then* $(\mathsf{label}_k, \mathsf{val}_k, t_k) \in \mathsf{DS}_{t+1}$ *and, (3)* $\mathsf{DS}_{t+1}.\mathsf{Labels} = \mathsf{DS}_t.\mathsf{Labels}$.
- $0/1 \leftarrow \mathsf{oZKS.VerifyTombstone}(\mathsf{com}, \mathsf{com}', \pi_S, t, t_{\mathsf{stale}})$*: Verifies the proof* π_S *that* com, com$'$ *commit to some datastores* DS, DS$'$ *respectively, such that* $\mathsf{DS}'.\mathsf{Labels} = \mathsf{DS}.\mathsf{Labels}$*, and that for any* $\mathsf{label} \in \mathsf{DS}'.\mathsf{Labels}$ *such that* $(\mathsf{label}, \mathsf{val}, t_j) \in \mathsf{DS}$ *and* $(\mathsf{label}, \mathsf{val}', t_j) \in \mathsf{DS}'$*, with* $\mathsf{val} \neq \mathsf{val}'$*, then* $t_j \leq t_{\mathsf{stale}}$*. If all these checks pass, and* com *is the commitment to the epoch* t *and* $t_{\mathsf{stale}} = t -$StaleParam*, then this algorithm outputs* 1*, otherwise it outputs* 0.
- $(\mathsf{com}', \mathsf{st_{com'}}, \mathsf{DS}_{t+1}, \pi_S, t+1) \leftarrow \mathsf{oZKS.DeleteElts}(\mathsf{st_{com}}, \mathsf{DS}_t, t)$*: This algorithm is only run if* $t -$ DeletionParam $\in$ TombstoneElts*. If so, takes in the current datastore* DS_t*, the internal server state* $\mathsf{st_{com}}$*, the current server epoch* t *and computes the set* $S =$

$\{(\mathsf{label}, \cdot, \cdot) | (\mathsf{label}, \text{TOMBSTONE}, \cdot) \in \mathsf{DS}_t\}$. *It returns* $\mathsf{DS}_{t+1} = \mathsf{DS}_t \setminus S'$, com', *the commitment to* DS_{t+1}, $\mathsf{st}_{\mathsf{com}'}$ *the updated internal state of the server and a proof* π_S *that* com' *is a commitment to* DS_{t+1} *such that* $\mathsf{DS}_{t+1} \subseteq \mathsf{DS}_t$ *and each entry of* $\mathsf{DS}_t \setminus \mathsf{DS}_{t+1} = \{(\mathsf{label}, \text{TOMBSTONE}, \cdot) | \mathsf{label} \in \mathsf{DS}_t.\mathsf{Labels}\}$.

- $0/1 \leftarrow \mathsf{oZKS.VerifyDel}(\mathsf{com}, \mathsf{com}', \pi_S, t)$*: If* $t - \mathsf{DeletionParam} \notin \mathsf{TombstoneElts}$, *then this algorithm outputs* 0. *Otherwise, it verifies the proof* π_S *that* $\mathsf{com}, \mathsf{com}'$ *commit to some datastores* $\mathsf{DS}, \mathsf{DS}'$ *respectively, such that* $\mathsf{DS}' \subseteq \mathsf{DS}$ *and that* $\mathsf{DS} \setminus \mathsf{DS}' = \{(\mathsf{label}, \text{TOMBSTONE}, \cdot) | \mathsf{label} \in \mathsf{DS}_t.\mathsf{Labels}\}$.
- $0/1 \leftarrow \mathsf{oZKS.VerifyUpd}(t_1, t_n, \{\mathsf{com}_i, \pi_S^i, \mathsf{aux}_i\}_{i=1}^{n-1}, \mathsf{com}_n, \pi_S)$*: First, this algorithm checks that* $t_n - t_1$ *is the number of commitments it received. It denotes* $t_{i+1} = t_i + 1$, *for* $i \in [1, n-1]$. *Then it outputs* 1, *if all of the following checks pass. For* $i \in [1, n-1]$*: if* $t_i \in \mathsf{TombstoneEpochs}$, *it parses* $\mathsf{aux}_i = t_{\mathsf{stale}}$ *and runs the check* $\mathsf{oZKS.VerifyTombstone}(\mathsf{com}_i, \mathsf{com}_{i+1}, \pi_S^i, t_i, t_{\mathsf{stale}})$. *Else, it ignores* aux_i *and if* $t_i - \mathsf{DeletionParam} \in \mathsf{TombstoneEpochs}$, *it runs* $\mathsf{oZKS.VerifyDel}(\mathsf{com}_i, \mathsf{com}_{i+1}, \pi_S^i, t_i)$, *else, it runs the check* $\mathsf{oZKS.VerifyInsertions}(\mathsf{com}_i, \mathsf{com}_{i+1}, \pi_S^i, t_i)$.

Soundness for oZKS **with Compaction.**

Definition 3. *An* oZKS *with compaction, defined in section III-C is said to be sound if, for any PPT adversary* $\mathcal{A}$, *each of the following is less than or equal to* $\mathsf{negl}(\lambda)$*:*

$$
\begin{aligned}
&\Pr[(\mathsf{label}, \{t_i, \mathsf{com}_i, \Pi_i^{\mathsf{Upd}}, \mathsf{aux}_i\}_{i=1}^{n-1}, \mathsf{com}_n, t_{\mathsf{stale}},\\
&\Pi_1^{\mathsf{Ver}}, (\mathsf{val}_1, \mathsf{Epoch}_1), \Pi_2^{\mathsf{Ver}}, (\mathsf{val}_2, \mathsf{Epoch}_2)) \leftarrow \mathcal{A}(1^\lambda, \mathsf{p}) :\\
&\{\mathsf{oZKS.VerifyUpd}(\mathsf{com}_i, \mathsf{com}_{i+1}, \Pi_i^{\mathsf{Upd}}, t_i + 1, \mathsf{aux}_i)\}_{i=1}^{n-1}\\
&\bigwedge \{t_{i+1} = t_i + 1\}_{i=1}^{n-1}\\
&\bigwedge \forall i \in [1, n], t_i \notin \mathsf{TombstoneEpochs}\\
&\bigwedge \forall i \in [1, n], t_i - \mathsf{DeletionParam} \notin \mathsf{TombstoneEpochs}\\
&\bigwedge 1 \leq l \leq k \leq n)\\
&\bigwedge \mathsf{oZKS.VerifyMem}(\mathsf{com}_l, \mathsf{label}, \mathsf{val}_1, \mathsf{Epoch}_1, \Pi_1^{\mathsf{Ver}})\\
&\bigwedge (\\
&(\\
&(([l, k] \cap \mathsf{TombstoneEpochs} = [l, k] \cap \mathsf{DeletionEpochs} = \emptyset) \wedge\\
&((\mathsf{val}_1, \mathsf{Epoch}_1) \neq (\mathsf{val}_2, \mathsf{Epoch}_2) \wedge \mathsf{oZKS.VerifyMem}(\mathsf{com}_k,\\
&\mathsf{label}, \mathsf{val}_2, \mathsf{Epoch}_2, \Pi_2^{\mathsf{Ver}}))\\
&\bigvee (\mathsf{oZKS.VerifyNonMem}(\mathsf{com}_k, \mathsf{label}, \Pi_2^{\mathsf{Ver}}))\\
&))\\
&\bigvee (\\
&[l, k] \cap \mathsf{TombstoneElts} = \{j\} \wedge [l, k] \cap \mathsf{DeleteElts} = \emptyset \wedge\\
&t_{\mathsf{stale}} = t_j - \mathsf{StaleParam} \wedge\\
&(((\mathsf{val}_1, \mathsf{Epoch}_1) \neq (\mathsf{val}_2, \mathsf{Epoch}_2) \wedge \mathsf{Epoch}_1 > t_{\mathsf{stale}}\\
&\wedge \mathsf{oZKS.VerifyMem}(\mathsf{com}_k, \mathsf{label}, \mathsf{val}_2, \mathsf{Epoch}_2, \Pi_2^{\mathsf{Ver}}))\\
&\bigvee ((\mathsf{val}_1, \mathsf{Epoch}_1) \neq (\mathsf{val}_2, \mathsf{Epoch}_2)\\
&\wedge \mathsf{oZKS.VerifyMem}(\mathsf{com}_k, \mathsf{label}, \mathsf{val}_2, \mathsf{Epoch}_2, \Pi_2^{\mathsf{Ver}})\\
&\wedge \mathsf{Epoch}_1 \leq t_{\mathsf{stale}} \wedge \mathsf{val}_2 \neq \text{TOMBSTONE})\\
&\bigvee (\mathsf{oZKS.VerifyNonMem}(\mathsf{com}_k, \mathsf{label}, \Pi_2^{\mathsf{Ver}}))\\
&))\\
&\bigvee (\\
&[l, k] \cap \mathsf{DeleteElts} = \{j\}\\
&\wedge (\mathsf{oZKS.VerifyNonMem}(\mathsf{com}_k, \mathsf{label}, \Pi_2^{\mathsf{Ver}})\\
&\wedge \mathsf{val}_1 \neq \text{TOMBSTONE})\\
&))].
\end{aligned}
$$

Definition of a VKD. [16] defined a VKD as a collection of the following algorithms (included here for completeness):

- $(\mathsf{Dir}_t, \mathsf{st}_t, \Pi^{\mathsf{Upd}})/\bot \leftarrow \mathsf{VKD.Publish}(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)$: This algorithm takes as input a directory Dir_{t-1}, an internal server sstate st_{t-1} and a set of updates S_t consisting of $(\mathsf{label}, \mathsf{val})$ pairs. It updates Dir_{t-1} and st_{t-1} to reflect the updates in S_t. If this update is successful, the algorithm updates the commitment using the WitnessAPI and returns the next state st_t, the updated directory Dir_t and a proof that this update was correct. Otherwise it outputs $\bot$.

- $(\pi, \mathsf{val}, \alpha) \leftarrow \mathsf{VKD.Query}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: This algorithm takes as input a server internal state st_t, a directory Dir_t and a label label and if label is in Dir_t, it returns the value val, the version number α and the proof π that the provided information is correct with respect to the commitment for the epoch t.
- $0/1 \leftarrow \mathsf{VKD.VerifyQuery}(t, \mathsf{label}, \mathsf{val}, \pi, \alpha)$: This algorithm takes as input an epoch t, a label label, a purported version number (α) and value val, corresponding to label and a proof π that val and α are correct. It retrieves the commitment com_t for the epoch t using the $\mathsf{WitnessAPI}$ and verifies π with respect to this commitment. It returns 1 if the proof verifies and 0 in any other case.
- $((\mathsf{val}_i, t_i)_{i=1}^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{VKD.KeyHistory}(\mathsf{st}_t, \mathsf{Dir}_t, t, \mathsf{label})$: This algorithm takes as input the internal state (st_t) and directory (Dir_t) at epoch t and a label label. It outputs an ordered set of tuples $(\mathsf{val}_i, t_i)_{i=1}^n$, where val_i is the purported value corresponding to $that came into effect at epoch \mathsf{t}_i$. It also returns the proof Π^{Ver} to attest to these state changes for the label label.
- $0/1 \leftarrow \mathsf{VKD.VerifyHistory}(t, \mathsf{label}, (\mathsf{val}_i, t_i)_{i=1}^n, \Pi^{\mathsf{Ver}})$: This algorithm takes as input the ordered set of tuples $(\mathsf{val}_i, t_i)_{i=1}^n$ and proof Π^{Ver}, retrieves the commitment for the epoch t and the label label, and verifies Π^{Ver} with respect to the commitment.
- $0/1 \leftarrow \mathsf{VKD.VerifyUpd}(t, \mathsf{com}, \mathsf{com}', \Pi^{\mathsf{Upd}})$: This algorithm verifies the output of a single publish operation with respect to an initial and final commitment.
- $0/1 \leftarrow \mathsf{VKD.Audit}(t_1, t_n, \left(\Pi_t^{\mathsf{Upd}}\right)_{t=t_1}^{t_{n-1}})$: This algorithm takes as input a starting epoch t_1 and an ending epoch t_n, retrieves the required commitments to verify the proofs Π_t^{Upd} attesting to the correct evolution of the server's state.

Definition of a VKD with compaction. Our modified primitive, which we call a VKD with compaction, denoted as cVKD, is defined below. Note the additional set of public parameters: $(\mathsf{TombstoneEpochs}, \mathsf{CompactionEpochs}, \mathsf{StaleParam})$. These determine when tombstoning (ordered set $\mathsf{TombstoneEpochs}$) or compaction (ordered set $\mathsf{CompactionEpochs}$) are permitted, and how old a piece of data needs to be, in order to be eligible for tombstoning ($\mathsf{StaleParam}$).

A VKD with compaction (cVKD) is a VKD with public parameters $(\mathsf{TombstoneEpochs}, \mathsf{CompactionEpochs}, \mathsf{StaleParam})$ and following additional algorithms:

- $(\mathsf{com}_t, \mathsf{Dir}_t, \mathsf{st}_t, \Pi^{\mathsf{Upd}}, t) \leftarrow \mathsf{cVKD.TombstoneElts}(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, t-1, S, t_{\mathsf{stale}})$: This algorithm takes as input an epoch $t-1$, the server's directory Dir_{t-1} at this epoch, its internal state st_{t-1}, a set S of items it wants to tombstone and a parameter t_{stale} to determine the cutoff for tombstoning. It updates the data in the state and directory by tombstoning the appropriate elements of S and returns the updated $\mathsf{Dir}_t, \mathsf{st}_t$ and corresponding commitment com_t.
- $0/1 \leftarrow \mathsf{cVKD.VerifyTombstone}(\mathsf{com}_t, \mathsf{com}_{t+1}, \Pi^{\mathsf{Upd}}, t_{\mathsf{stale}})$: This algorithm takes as input two commitments from consecutive epochs and a parameter t_{stale} which tells it which values are old enough for tombstoning, and verifies the proof Π^{Upd} of correct tombstoning.
- $(\mathsf{com}_{t+1}, \mathsf{Dir}_{t+1}, \mathsf{st}_{t+1}, \Pi^{\mathsf{Upd}}, t+1) \leftarrow \mathsf{cVKD.Compact}(\mathsf{Dir}_t, \mathsf{st}_t, t)$: This algorithm takes in a directory Dir_t, an internal state st_t, the epoch t and outputs the updated directory, state, commitment and epoch, with tombstoned values deleted. It also includes a proof for this update.
- $0/1 \leftarrow \mathsf{cVKD.VerifyCompact}(\mathsf{com}_t, \mathsf{com}_{t+1}, \Pi^{\mathsf{Upd}})$: This algorithm takes as input a server's commitment before and after a compaction and checks the proof Π^{Upd} that the only change made by the server was removing data associated with tombstoned values.

The constraints on $\mathsf{TombstoneEpochs}$, $\mathsf{CompactionEpochs}$ and $\mathsf{StaleParam}$ for a cVKD are as follows:

- If t_i and t_{i+1} are two epochs in the ordered set $\mathsf{TombstoneEpochs}$, then, there exists a deletion epoch $t_{\mathsf{del}} \in \mathsf{DeleteElts}$, between them. That is, $t_i < t_{\mathsf{del}} < t_{i+1}$.
- A piece of data can only be marked as tombstoned in a particular epoch $t_i \in \mathsf{TombstoneEpochs}$ if it was inserted prior to the last tombstone epoch t_{i-1}. That is, $\mathsf{StaleParam} > \mathsf{min}\{|t - t'| : t, t' \in \mathsf{TombstoneEpochs}\}$.

Since we have introduced two new kinds of updates to the server, we modify the key history and audit algorithms for a cVKD as follows:

- $0/1 \leftarrow \mathsf{cVKD.VerifyHistory}(t, \mathsf{label}, \mathsf{min}_t, \mathsf{max}_t, (\mathsf{val}_i, t_i)_{i=\mathsf{min}_t}^{\mathsf{max}_t}, \Pi^{\mathsf{Ver}})$: This algorithm takes as input an epoch t, with respect to which history is being verified and a label label whose values are being checked, min_t, which represents a minimum version number which is not deleted or tombstoned at epoch t and max_t, the maximum version number for label. It also includes the values and epochs at which each version in $[\mathsf{min}_t, \mathsf{max}_t]$ was inserted. It obtains the server commitment at epoch t and verifies the proof Π^{Ver} that all versions below min_t were correctly tombstoned or deleted and that that the history presented for versions from min_t onwards is correct.
- $0/1 \leftarrow \mathsf{cVKD.Audit}(t_1, t_n, \{\Pi_t^{\mathsf{Upd}}\}_{t=t_1}^{t_n-1})$: For each epoch $t \in [t_1, t_n - 1]$, if $t \in \mathsf{TombstoneEpochs}$, this algorithm gets the corresponding commitments and proof, passes it to $\mathsf{cVKD.VerifyTombstone}$, if $t \in \mathsf{CompactionEpochs}$, it passes the corresponding commitments and proof to $\mathsf{cVKD.VerifyCompact}$, else, it passes the appropriate inputs to $\mathsf{cVKD.VerifyUpd}$.

Definition 4. *A VKD with compaction is said to be sound if,* *for any PPT adversary $\mathcal{A}$,*

$$\begin{aligned}
&\Pr[(\mathsf{label}, \{t, \{(\mathsf{val}_k^t, \mathsf{Epoch}_k^t)\}_{k=\mathsf{min}_t}^{\mathsf{max}_t}, \Pi_t^{\mathsf{Ver}}\}_{t\in\{t_1,...,t_n\}},\\
&\{(\mathsf{com}_k, \Pi_k^{\mathsf{Upd}})\}_{k=t_1}^{t_{\mathsf{current}}}, t^*, j, (\pi, \mathsf{val}, \beta)) \leftarrow \mathcal{A}(1^\lambda) :\\
&\wedge\{\mathsf{com}_k \leftarrow \mathsf{WitnessAPI.GetCom}(k)\}_{k=t_1}^{t_{\mathsf{current}}}\\
&\wedge\mathsf{VKD.Audit}(t_1, t_{\mathsf{current}}, \{(\Pi_k^{\mathsf{Upd}})\}_{k=t_1}^{t_{\mathsf{current}}-1})\\
&\wedge\forall t \in [t_1, ..., t_n], \mathsf{VKD.VerifyHistory}(t, \mathsf{label},\\
&\{(\mathsf{val}_i^t, \mathsf{Epoch}_i^t)\}_{i=\mathsf{min}_t}^{\mathsf{max}_t}, \Pi_t^{\mathsf{Ver}})\\
&\wedge\forall t \in \mathsf{TombstoneEpochs},\ \textit{s.t.}\\
&[t, t+\mathsf{DeletionParam}] \cap [t_1, ..., t_{\mathsf{current}}] \neq \emptyset\\
&\implies \{t_1, ..., t_n\} \cap [t, t+\mathsf{DeletionParam}] \neq \emptyset\\
&\wedge t_1 \leq t_2 \leq ... \leq t_n \leq t_{\mathsf{current}}\\
&\wedge\mathsf{StaleParam} >\\
&\mathsf{min}\{|t-t'| \mid t, t' \in \mathsf{TombstoneEpochs}\}\\
&\wedge(\\
&(\exists t, t' \in \{t_1, ..., t_n\}, \gamma, (\mathsf{val}_\gamma^t, \mathsf{Epoch}_\gamma^t) \neq (\mathsf{val}_\gamma^{t'}, \mathsf{Epoch}_\gamma^{t'}))\\
&\vee\\
&(t_{j-1} \leq t^* < t_j\\
&\wedge(\mathsf{Epoch}_\alpha^{t_j} \leq t^* < \mathsf{Epoch}_{\alpha+1}^{t_j} \vee \alpha = \mathsf{max}_{t_j})\\
&\wedge\mathsf{val}_\alpha^{t_j} \neq \mathsf{val} \wedge t^* < t_j'\\
&\wedge\mathsf{VKD.VerifyQuery}(t^*, \mathsf{label}, \mathsf{val}, \pi))\\
&)] \leq \mathsf{negl}(\lambda).
\end{aligned}$$

C. Constructing an oZKS

1) Strong Accumulators

- $(\mathsf{com}_1, \mathsf{DS}_1, \mathsf{st}_1) \leftarrow \mathsf{SA.CommitDS}(1^\lambda, \mathsf{DS})$: This algorithm takes in a data store DS, a security parameter 1^λ and outputs a commitment com_1 to DS, a copy DS_1 of the data store and st_1, the internal state corresponding to DS_1 and com_1.
- $(v, \pi) \leftarrow \mathsf{SA.Query}(\mathsf{DS}_t, \mathsf{st}_t, \mathsf{com}_t, l)$: This algorithm takes in a data store DS_t, the corresponding internal state and commitment to it $\mathsf{st}_t, \mathsf{com}_t$, and a queried label l. If there exists a pair $(l, v) \in \mathsf{DS}_t$, the algorithm outputs v and the proof π that $(l, v) \in \mathsf{DS}_t$. Otherwise it returns $v = \perp$ and π is a non-membership proof for l in DS_t.
- $0/1 \leftarrow \mathsf{SA.Verify}(\mathsf{com}, l, v, \pi)$: Given a commitment com to a datastore, a label l and corresponding string v, if $v = \perp$, the algorithm parses π as a non-membership proof for l, with respect to the commitment com, otherwise, it parses it as a membership proof for (l, v). The algorithm outputs 1 if π verifies and 0 otherwise.
- $(\mathsf{com}_{t+1}, \mathsf{DS}_{t_1}, \mathsf{st}_{t+1}, \Pi^{\mathsf{Upd}}) \leftarrow \mathsf{SA.UpdateDS}(\mathsf{DS}_t, \mathsf{st}_t, S)$: This algorithm takes in a datastore DS_t, corresponding $\mathsf{st}_t, \mathsf{com}_t$, as well as a set S of updates. It initializes $\mathsf{DS}_{t+1} = \mathsf{DS}_t$. For all $(l, v) \in S$, if $(l, v') \in \mathsf{DS}_t$, for some v', it replaces this string in DS_{t+1} with (l, v), else, it adds (l, v) to DS_{t+1}. It computes the corresponding updated internal state st_{t+1} and commitment com_{t+1} and returns $\mathsf{com}_{t+1}, \mathsf{DS}_{t+1}, \mathsf{st}_{t+1}$ as well as a proof Π^{Upd}, that com_{t+1} is indeed the commitment to an update to the data store com_t commited to, with update set S.
- $0/1 \leftarrow \mathsf{SA.VerifyUpd}(\mathsf{com}, \mathsf{com}', S, \Pi^{\mathsf{Upd}})$: This algorithm takes as input a commitment com to a data store DS, another commitment com' to an update to DS, the set of updates S and the proof that com' was indeed the result of updating DS with S. It outpus 1 if Π^{Upd} verifies, otherwise, it outputs 0.
- $(\mathsf{com}_{t+1}, \mathsf{DS}_{t_1}, \mathsf{st}_{t+1}, \Pi^{\mathsf{Upd}}) \leftarrow \mathsf{SA.DeleteElts}(\mathsf{DS}_t, \mathsf{st}_t, S)$: This algorithm takes in a datastore DS_t, corresponding $\mathsf{st}_t, \mathsf{com}_t$, as well as a set S of deletions. It initializes $\mathsf{DS}_{t+1} = \mathsf{DS}_t$. For all $(l, v) \in S$, if $(l, v) \in \mathsf{DS}_t$, for some v, it removes (l, v) from DS_{t+1}. It computes the corresponding updated internal state st_{t+1} and commitment com_{t+1} and returns $\mathsf{com}_{t+1}, \mathsf{DS}_{t+1}, \mathsf{st}_{t+1}$ as well as a proof Π^{Upd}, that com_{t+1} is indeed the commitment to an update to the data store com_t commited to, with deletion set S.
- $0/1 \leftarrow \mathsf{SA.VerifyDel}(\mathsf{com}, \mathsf{com}', S, \Pi^{\mathsf{Upd}})$: This algorithm takes as input a commitment com to a data store DS, another commitment com' to an update to DS, the set of deletions S and the proof that com' was indeed the result of deleting the entries of S from DS. It outputs 1 if Π^{Upd} verifies, otherwise, it outputs 0.

2) oZKS from a $(\mathsf{SA}, s\mathsf{VRF}, s\mathsf{CS})$ triple.

Our oZKS construction, first implemented by [6] is very similar to the aZKS construction of [16], based on a strong accumulator (SA), a simulatable commitment scheme (sCS) and a strong VRF (sVRF).

The main distinction between the oZKS construction and the aZKS construction of [16] becomes clear when we consider a call of the form $\mathsf{oZKS.UpdateDS}(\mathsf{st_{com}}, \mathsf{DS}_t, S, t)$. The algorithm computing oZKS.UpdateDS computes $S' = \{(\mathsf{label}, \mathsf{val}, t+1) \mid (\mathsf{label}, \mathsf{val}) \in S\}$, then it computes $S'' = \{(l, v) \mid l = s\mathsf{VRF.Compute}(SK, \mathsf{label}), v = (s\mathsf{CS}(\mathsf{val}; r), t+1),$ for $(\mathsf{label}, \mathsf{val}) \in S'\}, \mathsf{DS}_{t+1} = \mathsf{DS}_t \cup S', \mathsf{DS}_{t+1}^{\mathsf{SA}} = \mathsf{DS}_t^{\mathsf{SA}} \cup S''$. Finally, it calls $\mathsf{SA.UpdateDS}(\mathsf{DS}_t^{\mathsf{SA}}, S'')$. [16]'s aZKS simply omits including the epoch $t+1$ in the value committed in the SA.

The proof of correct update includes the newly inserted pairs $S'' = \{(l, v)\}$ and the auditor of the update must additionally parse each v to ensure that it contains the correct epoch $(t+1)$.

The main novelty of our oZKS construction is in the tombstone and deletion paradigm. The additional functions to support compaction are as follows.

- $(\mathsf{com}', \mathsf{st}_{\mathsf{com}'}, \mathsf{DS}_{t+1}, \pi_S, t+1) \leftarrow \mathsf{oZKS.TombstoneElts}(\mathsf{st_{com}}, \mathsf{DS}_t, S, t, t_{\mathsf{stale}})$: This algorithm should only be called if $t \in \mathsf{TombstoneEpochs}$ and $t_{\mathsf{stale}} = t - \mathsf{StaleParam}$. It algorithm takes in the current datastore DS_t, the internal server state $\mathsf{st_{com}}$, the current server epoch t, a stale epoch parameter t_{stale} and a set S of triples $\{(\mathsf{label}_i, \mathsf{val}_i, t_i)\}$ for an update, such that $(\mathsf{label}_i, \mathsf{val}_i, t_i) \in \mathsf{DS}_t$. The algorithm checks that for each $(\mathsf{label}_i, \mathsf{val}_i, t_j) \in S$, $t_j \leq t_{\mathsf{stale}}$. Then, this algorithm computes $S' = \{(s\mathsf{VRF}(SK, \mathsf{label}_i), \text{TOMBSTONE}, t_j) | (\mathsf{label}_i, \mathsf{val}_i, t_j) \in S, r \leftarrow^{\$} \{0,1\}^\lambda\}$. It calls $\mathsf{SA.UpdateDS}(\mathsf{DS}_t^{\mathsf{SA}}, \mathsf{st}_t, S')$ to obtain $\mathsf{DS}_{t+1}^{\mathsf{SA}}, \mathsf{st}_{t+1}^{\mathsf{SA}}, \mathsf{com}_{t+1}, \pi$. It instantiates $\mathsf{DS}_{t+1} = \mathsf{DS}_t$ and for each $(\mathsf{label}_i, \mathsf{val}_i, t_j) \in S$, it replaces the entry $(\mathsf{label}_i, \mathsf{val}_i, t_j)$ of DS_{t+1} with $(\mathsf{label}_i, \text{TOMBSTONE}, t_j)$. It returns this updated DS_{t+1}, com', the commitment to DS_{t+1}, $\mathsf{st}_{\mathsf{com}'}$ the updated internal state of the server and a proof π_S that com' is a commitment to the data store DS_{t+1} such that (1) if $(\mathsf{label}_i, \text{TOMBSTONE}, t_j) \in \mathsf{DS}_{t+1}$, then $(\mathsf{label}_j, \mathsf{val}_j, t_j) \in \mathsf{DS}_t$, for some val_j, with $t_j \leq t_{\mathsf{stale}}$; (2) if $(\mathsf{label}_k, \mathsf{val}_k, t_k) \in \mathsf{DS}_t$ with $t_k > t_{\mathsf{stale}}$, then $(\mathsf{label}_k, \mathsf{val}_k, t_k) \in \mathsf{DS}_{t+1}$ and, (3) $\mathsf{DS}_{t+1}.\mathsf{Labels} = \mathsf{DS}_t.\mathsf{Labels}$.

- $0/1 \leftarrow \mathsf{oZKS.VerifyTombstone}(\mathsf{com}, \mathsf{com}', \pi_S, t, t_{\mathsf{stale}})$: This algorithm parses π_S to get the set S of entries $\{(\mathsf{label}_i, \text{TOMBSTONE}, t_j)\}$ and a proof π of strong accumulator update. It verifies the proof π that $\mathsf{com}, \mathsf{com}'$ include committments $\mathsf{com}^{\mathsf{SA}}, \mathsf{com}'^{\mathsf{SA}}$, such that $\mathsf{SA.VerifyUpd}(\mathsf{com}^{\mathsf{SA}}, \mathsf{com}'^{\mathsf{SA}}, S, \pi)$ outputs 1. It also checks that $\mathsf{com}, \mathsf{com}'$ commit to datastores DS, DS' such that $\mathsf{DS}'.\mathsf{Labels} = \mathsf{DS}.\mathsf{Labels}$, and that for any $\mathsf{label} \in \mathsf{DS}'.\mathsf{Labels}$ such that $(\mathsf{label}, \mathsf{val}, t_j) \in \mathsf{DS}$ and $(\mathsf{label}, \mathsf{val}', t_j) \in \mathsf{DS}'$, with $\mathsf{val} \neq \mathsf{val}'$, then $t_j \leq t_{\mathsf{stale}}$ and $\mathsf{label} \in S$. If all these checks pass, and com is the commitment to the epoch t and $t_{\mathsf{stale}} = t - \mathsf{StaleParam}$, then this algorithm outputs 1, otherwise it outputs 0.
- $(\mathsf{com}', \mathsf{st}_{\mathsf{com}'}, \mathsf{DS}_{t+1}, \pi_S, t + 1) \leftarrow \mathsf{oZKS.DeleteElts}(\mathsf{st}_{\mathsf{com}}, \mathsf{DS}_t, t)$: This algorithm is only run if $t - \mathsf{DeletionParam} \in \mathsf{TombstoneElts}$. If so, it takes in the current datastore DS_t, the internal server state $\mathsf{st}_{\mathsf{com}}$, the current server epoch t and computes the set $S = \{(s\mathsf{VRF}(SK, \mathsf{label}), \cdot, \cdot) | (\mathsf{label}, \text{TOMBSTONE}, \cdot) \in \mathsf{DS}_t\}$. It returns $\mathsf{DS}_{t+1} = \mathsf{DS}_t \setminus S$, it gets com' and $\mathsf{st}_{\mathsf{com}'}$ by computing the subroutine $\mathsf{SA.DeleteElts}(\mathsf{DS}_t^{\mathsf{SA}}, \mathsf{st}_t^{\mathsf{SA}}, S)$, correspondingly updates its state to get DS_{t+1}, $\mathsf{st}_{\mathsf{com}'}$ the updated internal state of the server and a proof π_S that com' is a commitment to DS_{t+1} such that $\mathsf{DS}_{t+1} \subseteq \mathsf{DS}_t$ and each entry of $\mathsf{DS}_t \setminus \mathsf{DS}_{t+1} = \{(\mathsf{label}, \text{TOMBSTONE}, \cdot) | \mathsf{label} \in \mathsf{DS}_t.\mathsf{Labels}\}$.
- $0/1 \leftarrow \mathsf{oZKS.VerifyDel}(\mathsf{com}, \mathsf{com}', \pi_S, t)$: If $t - \mathsf{DeletionParam} \notin \mathsf{TombstoneElts}$, then this algorithm outputs 0. Otherwise, it verifies the proof π_S that $\mathsf{com}, \mathsf{com}'$ commit to some datastores $\mathsf{DS}, \mathsf{DS}'$ respectively, such that $\mathsf{DS}' \subseteq \mathsf{DS}$ and that $\mathsf{DS} \setminus \mathsf{DS}' = \{(\mathsf{label}, \text{TOMBSTONE}, \cdot) | \mathsf{label} \in \mathsf{DS}_t.\mathsf{Labels}\}$ by verifying $\mathsf{SA.VerifyDel}$.

3) *Efficiency improvements due to* oZKS

The major advantage of constructing an oZKS and including the epoch a leaf was inserted in the tree is that if audits are honestly verified, a SA membership proof includes when the leaf was inserted. This means that only the latest commitment of the SA needs to be verified, even in the case of historical queries about some label. This alleviates the need to store all states of an evolving oZKS for the VKD construction, making space complexity of the VKD's state st linear in the number of leaves, rather than in the number of epochs, unlike the construction using the aZKS of [16]. Additionally, this means that the RAM complexity of multiple, simultaneous history queries can be amortized over the number of queries and the number of proofs π_i^j being proven over all the history queries.

D. *Witness API*

The witness API is a set of algorithms ($\mathsf{GetCom}, \mathsf{VerifyCert}, \mathsf{ProposeNewEp}$) and a variable Epoch, initialized to 0.

- $(\mathsf{com}_t, \mathsf{cert}_t)/\perp \leftarrow \mathsf{WitnessAPI.GetCom}(t)$: This algorithm, callable by any party, takes as input an epoch t and if it has a commitment for this epoch, it returns the epoch com_t and the corresponding certificate cert_t. Else, it returns $\perp$.
- $0/1 \leftarrow \mathsf{WitnessAPI.VerifyCert}(\mathsf{com}_t, \mathsf{cert}_t, \mathsf{p})$: This algorithm verifies the certificate cert_t to com_t, with respect to public parameters p.
- $\mathsf{cert}_t/\perp \leftarrow \mathsf{WitnessAPI.ProposeNewEp}(t, \mathsf{com}_t, \pi)$: This algorithm, called by a server, takes as input a proposed commitment com_t, an epoch t, as well as a proof π. It verifies π using the Audit algorithm of the VKD solution implemented by the server, and if this verifies, if $\mathsf{WitnessAPI.Epoch} = t - 1$, WitnessAPI this algorithm sets $\mathsf{Epoch} = t$ and outputs cert_t. Else, it outputs $\perp$ and leaves $\mathsf{Epoch} = t - 1$.

Note that we assume an implicit setup (which includes determining the various cryptographic operations, the identity of the IdP, etc. We also assume that the identity of the parties jointly supplying the WitnessAPI is public and they each have known public keys.

E. oZKS*-based VKD Construction*

Below, we discuss the details of each of the API calls of our VKD construction, using an oZKS. We assume that the server's identity (public key) is public and signs its responses to user/auditor queries, so they cannot be impersonated.

- $(\mathsf{Dir}_t, \mathsf{st}_t, \Pi^{\mathsf{Upd}})/\perp \leftarrow \mathsf{VKD.Publish}(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)$: The server receives as input a set S_t containing $(\mathsf{label}, \mathsf{val})$ pairs. The algorithm instantiates a set $S_t' = \emptyset$. If label does not exist in VKD, it sets $\alpha = 1$, else, label must have a version number $\alpha - 1$. In either case, the algorithm adds $(\mathsf{label}|\alpha, \mathsf{val})$ to S_t'. If $\alpha > 1$, it also adds $(\mathsf{label}|\alpha - 1|'\mathsf{stale}', 0)$ to S_t'. It retrieves $(\mathsf{st}_{t-1}^{\mathsf{oZKS}}, \mathsf{DS}_{t-1})$ from st_{t-1}, and gets the output $(\mathsf{com}_t, \mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t, \pi, t)$ of $\mathsf{oZKS.InsertionToDS}(\mathsf{st}_{t-1}^{\mathsf{oZKS}}, \mathsf{DS}_{t-1}, S_t', t-1)$, updates st_{t-1} with $\mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t$ to get st_t, sets $\Pi_t^{\mathsf{Upd}} \leftarrow \pi$. Dir_t starts off as equal to Dir_{t-1}. For each $\mathsf{label} \in S_t$, if α (set above) is 1, the algorithm initializes an empty list L_{label}, else, it retrieves $(\mathsf{label}, L_{\mathsf{label}})$; it appends $(\alpha, \mathsf{val}, t + 1)$ to the front of L_{label} and includes this updated $(\mathsf{label}, L_{\mathsf{label}})$ in Dir_t. Then, the algorithm calls $\mathsf{WitnessAPI.ProposeNewEp}(t, \mathsf{com}_t, \Pi^{\mathsf{Upd}})$. If WitnessAPI.

$\mathsf{ProposeNewEp}$ outputs $\perp$, the algorithm reverts all its internal changes to st_{t-1}, Dir_{t-1}, and outputs $\perp$. Finally, the algorithm returns $(\mathsf{Dir}_t, \mathsf{st}_t, \Pi^{\mathsf{Upd}})$.

- $(\pi, \mathsf{val}, \alpha) \leftarrow \mathsf{VKD.Query}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: If $\mathsf{label} \notin \mathsf{Dir}_t$, this algorithm outputs $\perp$. Else, it recovers α, the latest version number for label, i.e. the first entry of L_{label}. It sets $\beta = 2^{\lfloor \log(\alpha) \rfloor}$, and parses out $(\mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t)$ out of st_t it computes (1) $(\pi_{\mathsf{mem}}, \mathsf{val}, i) \leftarrow \mathsf{oZKS.QueryMem}(\mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t, \mathsf{label}|\alpha)$, (2) $\pi_{\mathsf{fresh}} \leftarrow \mathsf{oZKS.QueryNonMem}(\mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t, \mathsf{label}|\alpha|'\mathsf{stale}')$ and (3) $(\pi_{\mathsf{hist}}, \mathsf{val}_{\mathsf{hist}}, i_{\mathsf{hist}}) \leftarrow \mathsf{oZKS.QueryMem}(\mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t, \mathsf{label}|\beta)$. The algorithm sets $\pi = (\pi_{\mathsf{mem}}, \pi_{\mathsf{fresh}}, \pi_{\mathsf{hist}}, i, \mathsf{val}_{\mathsf{hist}}, i_{\mathsf{hist}},)$ and returns $(\pi, \mathsf{val}, \alpha)$.
- $0/1 \leftarrow \mathsf{VKD.VerifyQuery}(t, \mathsf{label}, \mathsf{val}, \pi, \alpha)$: The client running this algorithm calls $\mathsf{WitnessAPI.GetCom}(t)$ to obtain $(\mathsf{com}_t, \mathsf{cert}_t)$. If $\mathsf{WitnessAPI.VerifyCert}(\mathsf{com}_t, \mathsf{cert}_t, \mathsf{p})$ outputs 0, then this algorithm outputs 0. Else, it parses π as $(\pi_{\mathsf{mem}}, \pi_{\mathsf{fresh}}, \pi_{\mathsf{hist}}, i, \mathsf{val}_{\mathsf{hist}}, i_{\mathsf{hist}})$, sets $\beta = 2^{\lfloor \log(\alpha) \rfloor}$ and returns 1 if all of the following return 1: (1) $\mathsf{oZKS.VerifyMem}(\mathsf{com}_t, \mathsf{label}|\alpha, \mathsf{val}, i, \pi_{\mathsf{mem}})$, (2) $\mathsf{oZKS.VerifyNonMem}(\mathsf{com}_t, \mathsf{label}|\alpha|'\mathsf{stale}', \pi_{\mathsf{fresh}})$ and (3) $\mathsf{oZKS.VerifyMem}(\mathsf{com}_t, \mathsf{label}|\beta, \mathsf{val}_{\mathsf{hist}}, i_{\mathsf{hist}}, \pi_{\mathsf{hist}})$.
- $((\mathsf{val}_i)_{i=1}^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{VKD.KeyHistory}(\mathsf{st}_t, \mathsf{Dir}_t, t, \mathsf{label})$: If label is not in Dir_t, then the algorithm returns $\perp$. Else, it computes $\mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t$ from st_t and retrieves $(\mathsf{label}, L_{\mathsf{label}})$ from Dir_t. Let $L_{\mathsf{label}} = \{(i, \mathsf{val}_i, t_i)\}_{i=1}^n$. Then, for each $i = 2, ..., n$, the algorithm computes
  - $(\pi_1^i, \mathsf{val}_i, t_i) \leftarrow \mathsf{oZKS.QueryMem}(\mathsf{st}, \mathsf{DS}, \mathsf{label}|i)$,
  - $(\pi_2^i, 0, t_i) \leftarrow \mathsf{oZKS.QueryMem}(\mathsf{st}, \mathsf{DS}, \mathsf{label}|i{-}1|'\mathsf{stale}')$,

  It also computes $(\pi_1^1, \mathsf{val}_1, t_1) \leftarrow \mathsf{oZKS.QueryMem}(\mathsf{st}, \mathsf{DS}, \mathsf{label}|1)$. With $a = \lfloor \log(n) \rfloor + 1$, $b = \lfloor \log(t) \rfloor$, and $\alpha = 2^a - 1$, the algorithm computes $\pi_3^j \leftarrow \mathsf{oZKS.QueryNonMem}(\mathsf{st}, \mathsf{DS}, \mathsf{label}|j)$ for $j = n+1, ..., \alpha$. It also gets $\pi_4^k \leftarrow \mathsf{oZKS.QueryNonMem}(\mathsf{st}, \mathsf{DS}, \mathsf{label}|2^k)$ for $k \in a, ..., b$. Finally, it sets $\Pi^{\mathsf{Ver}} = ((\pi_1^i)_{i=1}^n, (\pi_2^i)_{i=2}^n, (\pi_3^j)_{j=n+1}^{\alpha}, (\pi_4^k)_{k=a}^b)$ and outputs $((\mathsf{val}_i, t_i)_{i=1}^n, \Pi^{\mathsf{Ver}})$.
- $0/1 \leftarrow \mathsf{VKD.VerifyHistory}(t, \mathsf{label}, (\mathsf{val}_i, t_i)_{i=1}^n, \Pi^{\mathsf{Ver}})$: This algorithm calls $\mathsf{WitnessAPI.GetCom}(t)$ to obtain $(\mathsf{com}_t, \mathsf{cert}_t)$. If $\mathsf{WitnessAPI.VerifyCert}(\mathsf{com}_t, \mathsf{cert}_t, \mathsf{p})$ outputs 0, then this algorithm outputs 0. Else, this algorithm parses Π^{Ver} as $((\pi_1^i)_{i=1}^n, (\pi_2^i)_{i=2}^n, (\pi_3^j)_{j=n+1}^{\alpha}, (\pi_4^k)_{k=a}^b)$ where $a = \lfloor \log(n) \rfloor + 1$, $b = \lfloor \log(t) \rfloor$ and $\alpha = 2^a - 1$. It outputs 1 if $t_1 < t_2 < ... < t_n$, and all of the following output 1:
  - $\mathsf{oZKS.VerifyMem}(\mathsf{com}_t, \mathsf{label}|i, \mathsf{val}_i, t_i, \pi_1^i)$ for $i \in [1, n]$.
  - $\mathsf{oZKS.VerifyMem}(\mathsf{com}_t, \mathsf{label}|i-1|'\mathsf{stale}', 0, t_i, \pi_2^i)$ for $i = [2, n]$.
  - $\mathsf{oZKS.VerifyNonMem}(\mathsf{com}_t, \mathsf{label}|j, \pi_3^j)$ for $j \in [n+1, \alpha]$.
  - $\mathsf{oZKS.VerifyNonMem}(\mathsf{com}_t, \mathsf{label}|2^k, \pi_4^k)$ for $k \in [a, b]$.
- $0/1 \leftarrow \mathsf{VKD.VerifyUpd}(t_1, t_n, (\mathsf{com}_t, \Pi_t^{\mathsf{Upd}})_{t=t_1}^{t_n-1}, \mathsf{com}_{t_n})$: This algorithm outputs 1 if $\mathsf{oZKS.VerifyInsertions}(\mathsf{com}_{t_i}, \mathsf{com}_{t_i+1}, \pi_{t_i}, t_i+1)$ outputs 1 for each $i \in [1, n-1]$.
- $0/1 \leftarrow \mathsf{VKD.Audit}(t_1, t_n, (\Pi_t^{\mathsf{Upd}})_{t=t_1}^{t_n-1})$: This algorithm takes as input a starting time t_1 and an end time t_n. It checks that $t_1 < t_n$ and that the tuple $(\Pi_t^{\mathsf{Upd}})_{t=t_1}^{t_n-1})$ parses to exactly $t_n - t_1$ proofs Π_t^{Upd}. If this check fails, it outputs 0, and if it passes, the algorithm obtains $(\mathsf{com}_t, \mathsf{cert}_t) \leftarrow \mathsf{WitnessAPI.GetCom}(t)$ for $t \in [t_1, t_n]$. It verifies $\mathsf{WitnessAPI.VerifyCert}(\mathsf{com}_t, \mathsf{cert}_t, \mathsf{p})$, for $t \in [t_1, t_n]$, and if it outputs 0, the algorithm outputs 0. Else, the algorithm outputs 1 if $\mathsf{VKD.VerifyUpd}(t_i, t_i + 1, (\mathsf{com}_{t_i}, \Pi_{t_i}^{\mathsf{Upd}}), \mathsf{com}_{t_i+1})$ outputs 1 for each $i \in [1, n-1]$.

F. cVKD*: VKD with Secure Compaction*

1) Secure Compaction: Attempt 1

As a first attempt, let us extend the oZKS data structure of [6] to support two additional algorithms:

- $(\mathsf{com}', \mathsf{st}_{\mathsf{com}'}, \mathsf{DS}_{t+1}, \pi_S, t+1) \leftarrow \mathsf{oZKS.DeleteElts}_1(\mathsf{st}_{\mathsf{com}}, \mathsf{DS}_t, S, t, t_{\mathsf{stale}})$: This algorithm takes in the current datastore DS_t, the internal server state $\mathsf{st}_{\mathsf{com}}$, the current server epoch t, a stale epoch parameter t_{stale} and a set S of triples $\{(\mathsf{label}_i, \mathsf{val}_i, t_i)\}$ for an update, such that $(\mathsf{label}_i, \mathsf{val}_i, t_i) \in \mathsf{DS}_t$. The algorithm checks that for each $(\cdot, \cdot, t_j) \in S$, $t_j \leq t_{\mathsf{stale}}$. It returns $\mathsf{DS}_{t+1} = \mathsf{DS}_t \setminus S'$, com', the commitment to DS_{t+1}, $\mathsf{st}_{\mathsf{com}'}$ the updated internal state of the server and a proof π_S that com' is a commitment to DS_{t+1} such that $\mathsf{DS}_{t+1} \subseteq \mathsf{DS}_t$ and each entry of $\mathsf{DS}_t \setminus \mathsf{DS}_{t+1} = \{(\mathsf{label}_j, \mathsf{val}_j, t_j)$ for some $t_j \leq t\}$.
- $0/1 \leftarrow \mathsf{oZKS.VerifyDel}_1(\mathsf{com}, \mathsf{com}', \pi_S, t_{\mathsf{stale}})$: Verifies the proof π_S that $\mathsf{com}, \mathsf{com}'$ commit to some datastores $\mathsf{DS}, \mathsf{DS}'$ respectively, such that $\mathsf{DS}' \subseteq \mathsf{DS}$ and that $\mathsf{DS} \setminus \mathsf{DS}' = \{(\mathsf{label}, \mathsf{val}, t)$ for some $t \leq t_{\mathsf{stale}}\}$.

Based on these two algorithms, we could construct compaction as follows:

- $(\mathsf{Dir}_t, \mathsf{st}_t, \Pi^{\mathsf{Upd}})/\perp \leftarrow \mathsf{VKD.Compact}_1(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, t-1, t_{\mathsf{stale}})$: The algorithm initializes $S_t = \emptyset$. For $(\mathsf{label}, L_{\mathsf{label}}) \in \mathsf{Dir}_{t-1}$, for $(\alpha, \mathsf{val}, t_\alpha) \in L_{\mathsf{label}}$, if $t_\alpha \leq t_{\mathsf{stale}}$, it adds $(\mathsf{label}|\alpha)$ to S_t. If $\alpha > 1$, it also adds $(\mathsf{label}|\alpha - 1|'\mathsf{stale}')$ to S_t. Then, it obtains $(\mathsf{com}_t^{\mathsf{oZKS}}, \mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t^{\mathsf{oZKS}}, \pi_{S_t}, t) \leftarrow \mathsf{oZKS.DeleteElts}_1(\mathsf{st}_{\mathsf{com}}, \mathsf{DS}_{t-1}, S_t, t_{\mathsf{stale}})$, updates the corresponding state $\mathsf{st}_t = (\mathsf{DS}_t^{\mathsf{oZKS}}, \mathsf{st}_t^{\mathsf{oZKS}})$, deletes the corresponding $(\alpha, \mathsf{val}, t_\alpha)$ tuples from L_{label} for each label to get Dir_t, $\mathsf{com}_t = \mathsf{com}_t^{\mathsf{oZKS}}$. Then, it calls $\mathsf{WitnessAPI.ProposeNewEp}(t, \mathsf{com}_t, \Pi^{\mathsf{Upd}})$. If $\mathsf{WitnessAPI.ProposeNewEp}$ outputs $\perp$, the algorithm reverts all its internal changes to st_{t-1}, Dir_{t-1}, and outputs $\perp$. Otherwise it returns $(\mathsf{Dir}_t, \mathsf{st}_t, \Pi^{\mathsf{Upd}})$.

The auditing algorithm $\mathsf{VKD.VerifyCompact}_1$, which verifies the proof output by $\mathsf{VKD.Compact}_1$ would work in the obvious way, calling $\mathsf{oZKS.VerifyDel}_1$ as a subroutine. Correspondingly, $\mathsf{VKD.Audit}$ must be modified, so at epochs which include deletions, it calls $\mathsf{VKD.VerifyCompact}_1$, instead of $\mathsf{VKD.VerifyUpd}$.

This construction, however, creates a problem if a user is not always online. Consider the following attack: suppose the server is at epoch 1000, the label Alice is at version 10, with value val_{10}. Also suppose that the entry val_{10} for Alice was inserted in epoch 100. Now, if $t_{\mathsf{stale}} = 101$ the server could compact the oZKS label Alice|10 and roll back her key to its previous version.

2) Secure Compaction: Attempt 2

Recall that when a client performs a lookup for label label, the IdP internally calls VKD.Query to obtain parameters including a value val, a version α and proofs $\pi = (\pi_{\mathsf{mem}}, \pi_{\mathsf{fresh}}, \pi_{\mathsf{hist}})$. Here π_{fresh} is a non-membership proof for the label label$|\alpha|$'stale' in the underlying oZKS. If compaction simply designed as VKD.Compact$_1$ above, a malicious server may only add label$|\alpha|$'stale' label to S_t for some time-step t and never the label$|\alpha$ label, which makes it possible for a lookup proof to pass with a stale (or compromised key) returned upon a lookup.

We attempt to mitigate this issue by introducing a special public parameter DeletionEpochs, such that calls to oZKS deletion will only verify if they are made at an epoch in the set DeletionEpochs. We propose an updated deletion for the oZKS.

- $(\mathsf{com}', \mathsf{st}_{\mathsf{com}'}, \mathsf{DS}_{t+1}, \pi_S, t+1) \leftarrow$ oZKS.DeleteElts$_2(\mathsf{st}_{\mathsf{com}}, \mathsf{DS}_t, S, t, t_{\mathsf{stale}})$: This algorithm runs if and only if $t \in$ DeletionEpochs. All other operations are the same as oZKS.DeleteElts$_1$.
- $0/1 \leftarrow$ oZKS.VerifyDel$_2(\mathsf{com}, \mathsf{com}', \pi_S, t_{\mathsf{stale}}, t)$: In addition to the checks in oZKS.VerifyDel$_1$, this algorithm also ensures that the version number of DS is t and $t \in$ DeletionEpochs.

Since deletions in the updated oZKS API are only possible at epochs in the set, DeletionEpochs, we can update our VKD to include a corresponding public parameter CompactionEpochs, and introduce the following assumption:

All users come online at deletion epochs to ensure their stale keys are deleted appropriately.

As in the previous construction VKD.Compact$_2$ and VKD.VerifyCompact$_2$ call oZKS.DeleteElts and oZKS.VerifyDel as subroutines. VKD.Compact$_2$ only runs at epochs in CompactionEpochs and VKD.VerifyCompact$_2$ only verifies if the epoch presented is in CompactionEpochs.

Finally, we augment VKD.Audit as follows:

- VKD.Audit$(t_1, t_n, (\Pi_t^{\mathsf{Upd}}, \mathsf{aux}_t)_{t=t_1}^{t_n-1})$: Now, for $t = t_1, ..., t_n - 1$, if $t \in$ CompactionEpochs, this algorithm parses aux $= t_{\mathsf{stale}}$ and calls VKD.VerifyCompact$_2(\mathsf{com}_t, \mathsf{com}_{t+1}, \Pi^{\mathsf{Upd}}, t_{\mathsf{stale}})$. Else, it calls VKD.VerifyUpd$(t, t+1, (\mathsf{com}_t, \Pi_t^{\mathsf{Upd}}), \mathsf{com}_{t+1})$. It outputs 1, if all subroutines output 1, otherwise it outputs 0.

The algorithm VKD.KeyHistory gets modified to the following:

- $0/1 \leftarrow$ VKD.VerifyHistory$_1(t, \mathsf{label}, (\mathsf{val}_i, t_i)_{i=j}^{n}, \Pi^{\mathsf{Ver}})$: This algorithm calls WitnessAPI.GetCom(t) to obtain $(\mathsf{com}_t, \mathsf{cert}_t)$. If WitnessAPI.VerifyCert$(\mathsf{com}_t, \mathsf{cert}_t, \mathsf{p})$ outputs 0, then this algorithm outputs 0. The algorithm parses Π^{Ver} as $(\pi^{\mathsf{deleted}}, \pi^{\mathsf{undeleted}})$.
  It parses π^{deleted} as $((\pi_1^i)_{i=1}^{j}, (\pi_2^i)_{i=2}^{j})$. Then, it verifies
  - oZKS.VerifyNonMem$(\mathsf{com}_t, \mathsf{label}|i, \pi_1^i)$ for $1 \le i < j$.
  - oZKS.VerifyNonMem$(\mathsf{com}_t, \mathsf{label}|i|\mathsf{stale}, \pi_2^i)$ for $1 \le i < j$.

  This algorithm parses $\pi^{\mathsf{undeleted}}$ as $((\pi_1^i)_{i=1j}^{n}, (\pi_2^i)_{i=2}^{n}, (\pi_3^j)_{j=n+1}^{\alpha}, (\pi_4^k)_{k=a}^{b})$ where $a = \lfloor \log(n) \rfloor + 1$, $b = \lfloor \log(t) \rfloor$ and $\alpha = 2^a - 1$. It outputs 1 if $t_1 < t_2 < ... < t_n$, and all of the following output 1:
  - oZKS.VerifyMem$(\mathsf{com}_t, \mathsf{label}|i, \mathsf{val}_i, t_i, \pi_1^i)$ for $i \in [j, n]$.
  - oZKS.VerifyMem$(\mathsf{com}_t, \mathsf{label}|i-1|'\mathsf{stale}', 0, t_i, \pi_2^i)$ for $i = [j+1, n]$.
  - oZKS.VerifyNonMem$(\mathsf{com}_t, \mathsf{label}|j, \pi_3^j)$ for $j \in [n+1, \alpha]$.
  - oZKS.VerifyNonMem$(\mathsf{com}_t, \mathsf{label}|2^k, \pi_4^k)$ for $k \in [a, b]$.

While the above patch mitigates the previous attack of arbitrary mutations, this patch would only work to satisfy soundness if the user audits her own key history at every epoch $t \in$ CompactionEpochs. If not, there is at least one epoch where the server could cheat by deleting her latest version, rolling back to a previous one, then reinserting the correct version. Recall that our original assumption was that a user should be able to check their entire key-history when coming online at any epoch, after any amount of time offline.

Besides, even if each user came online at each epoch in CompactionEpochs, the server losing part of its history in compaction epochs means that if the server deletes a user's latest key (if it was old enough), the audit would pass and the user would have no way to show that its latest key was deleted.

In the following design, we (1) slacken the requirement for when a user needs to come online to check that it's key is going to be correctly deleted, and (2) give the user a way to contest the changes to its own existing keys.

3) Two-phase Compaction

In this section, we present our final construction, which we call a *two-phase compaction*, which allows a VKD to support the algorithm Compact, without additional privacy leakage. As stated before, even if compaction were supported with fixed epochs which are demarcated for compaction, since after compaction, there is no record of the changes made, unless a user is online at the compaction epoch, she cannot ensure that any oZKS entries associated with her label were not modified. We weaken this requirement by allowing a *grace period* for a user to check her key history, before a compaction, by marking oZKS entries slated for deletion as tombstoned for a while, before they are actually deleted.

Now, we use the algorithms (oZKS.TombstoneElts, oZKS.VerifyTombstone, oZKS.DeleteElts, oZKS.VerifyDel), defined in section III-C to construct a VKD with compaction.

Recall that when a client performs a lookup for label label, the IdP internally calls VKD.Query to obtain parameters including a value val, a version α and proofs $\pi = (\pi_{\mathsf{mem}}, \pi_{\mathsf{fresh}}, \pi_{\mathsf{hist}})$. Here π_{fresh} is a non-membership proof for the label $\mathsf{label}|\alpha|'\mathsf{stale}'$ in the underlying oZKS. If compaction simply designed as $\mathsf{VKD.Compact}_1$ above, a malicious server may only add $\mathsf{label}|\alpha|'\mathsf{stale}'$ label to S_t for some time-step t and never the $\mathsf{label}|\alpha$ label, which makes it possible for a lookup proof to pass with a stale (or compromised key) returned upon a lookup.

Finally, we can define compaction as follows:

- $(\mathsf{com}_t, \mathsf{Dir}_t, \mathsf{st}_t, \Pi^{\mathsf{Upd}}, t) \leftarrow \mathsf{VKD.TombstoneElts}(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, t - 1, t_{\mathsf{stale}})$: As in $\mathsf{VKD.Compact}_1$ and $\mathsf{VKD.Compact}_2$, the algorithm initializes $S_t^{\mathsf{oZKS}} = \emptyset$, $S_t^{\mathsf{AVD}} = \emptyset$. For $(\mathsf{label}, L_{\mathsf{label}}) \in \mathsf{Dir}_{t-1}$, for $(\alpha, \mathsf{val}, t_\alpha) \in L_{\mathsf{label}}$:
  - If $t_\alpha \leq t_{\mathsf{stale}}$, it adds $(\mathsf{label}|\alpha)$ to S_t.
  - If $\alpha > 1$, it also adds $(\mathsf{label}|\alpha - 1|'\mathsf{stale}')$ to S_t.

  Then, it obtains the required $\mathsf{st}_t^{\mathsf{oZKS}}$ and DS_t and calls $\mathsf{oZKS.TombstoneElts}(\ \mathsf{st}_{\mathsf{com}}, \mathsf{DS}_t, S_t, t_{\mathsf{stale}})$. It updates the corresponding state, deletes the corresponding $(\alpha, \mathsf{val}, t_\alpha)$ tuples from L_{label} for each label. Then, the algorithm calls $\mathsf{WitnessAPI.ProposeNewEp}(t, \mathsf{com}_t, \Pi^{\mathsf{Upd}})$. If $\mathsf{WitnessAPI.ProposeNewEp}$ outputs $\perp$, the algorithm reverts all its internal changes to st_{t-1}, Dir_{t-1}, and outputs $\perp$. Finally, this algorithm returns the updated commitments and proof output by $\mathsf{oZKS.TombstoneElts}$.
- $0/1 \leftarrow \mathsf{VKD.VerifyTombstone}(\mathsf{com}_t, \mathsf{com}_{t+1}, \Pi^{\mathsf{Upd}}, t_{\mathsf{stale}})$: This algorithm calls $\mathsf{WitnessAPI.GetCom}(t)$ to obtain $(\mathsf{com}_t, \mathsf{cert}_t)$ and $\mathsf{WitnessAPI.GetCom}(t+1)$ to get $(\mathsf{com}_{t+1}, \mathsf{cert}_{t+1})$. Finally, it parses Π^{Upd} to get $\Pi^{\mathsf{Upd}}_{\mathsf{oZKS}}$. If either $\mathsf{WitnessAPI.VerifyCert}(\mathsf{com}_t, \mathsf{cert}_t, \mathsf{p})$ or $\mathsf{WitnessAPI.VerifyCert}(\mathsf{com}_{t+1}, \mathsf{cert}_{t+1}, \mathsf{p})$ outputs 0, the algorithm outputs 0. Otherwise, it returns the output of $\mathsf{oZKS.VerifyDel}(\mathsf{com}_{\mathsf{oZKS}}, \mathsf{com}'_{\mathsf{oZKS}}, \Pi^{\mathsf{Upd}}_{\mathsf{oZKS}}, t_{\mathsf{stale}})$.
- $(\mathsf{com}_{t+1}, \mathsf{Dir}_{t+1}, \mathsf{st}_{t+1}, \Pi^{\mathsf{Upd}}, t{+}1) \leftarrow \mathsf{VKD.Compact}(\mathsf{Dir}_t, \mathsf{st}_t, t, t_{\mathsf{stale}})$: This algorithm parses out $\mathsf{DS}_t^{\mathsf{oZKS}}, \mathsf{st}_t^{\mathsf{oZKS}}$ from st_t and calls $\mathsf{oZKS.DeleteElts}(\mathsf{st}_t^{\mathsf{oZKS}}, \mathsf{DS}_t^{\mathsf{oZKS}}, t, t_{\mathsf{stale}})$, to obtain $(\mathsf{com}_{t+1}^{\mathsf{oZKS}}, \mathsf{st}_{t+1}^{\mathsf{oZKS}}, \mathsf{DS}_{t+1}^{\mathsf{oZKS}}, \pi_S, t{+}1)$, which it uses to update $\mathsf{com}_t, \mathsf{st}_t, \mathsf{Dir}_t$ to get $\mathsf{com}_{t+1}, \mathsf{st}_{t+1}, \mathsf{Dir}_{t+1}$ and sets $\Pi^{\mathsf{Upd}} = \pi_S$ and returns $(\mathsf{com}_{t+1}, \mathsf{Dir}_{t+1}, \mathsf{st}_{t+1}, \Pi^{\mathsf{Upd}}, t{+}1)$.
- $0/1 \leftarrow \mathsf{VKD.VerifyCompact}(\mathsf{com}_t, \mathsf{com}_{t+1}, \Pi^{\mathsf{Upd}}, t_{\mathsf{stale}})$: This algorithm simply parses out $\mathsf{com}_t^{\mathsf{oZKS}}, \mathsf{com}_{t+1}^{\mathsf{oZKS}}$ from $\mathsf{com}_t, \mathsf{com}_{t+1}$, respectively, and outputs the output of $\mathsf{oZKS.VerifyDel}(\mathsf{com}_t^{\mathsf{oZKS}}, \mathsf{com}_{t+1}^{\mathsf{oZKS}}, \Pi^{\mathsf{Upd}}, t_{\mathsf{stale}})$.

The $\mathsf{VerifyHistory}$ algorithm has the following extra checks:

- $0/1 \leftarrow \mathsf{VKD.VerifyHistory}(t, \mathsf{label}, (\mathsf{val}_i, t_i)_{i=\alpha_{\mathsf{min}}}^n, \Pi^{\mathsf{Ver}})$: This algorithm is the same as $\mathsf{VKD.VerifyHistory}_1$, except it also ensures that if $\mathsf{val}_i = \textsc{Tombstone}$, then both the labels $\mathsf{label}|i$ and $\mathsf{label}|i|\mathsf{stale}$ have the value $\textsc{Tombstone}$. It also ensures that if $\mathsf{label}|i$ does not correspond to the value $\textsc{Tombstone}$, then neither does $\mathsf{label}|i|\mathsf{stale}$. Finally, it ensures that $\mathsf{val}_n \neq \textsc{Tombstone}$, i.e. the most recent value of this user is not tombstoned. If all checks pass, this algorithm outputs 1, otherwise it outputs 0.

The updated algorithms mean that $\mathsf{VKD.Audit}$ is defined as follows:

- $\mathsf{VKD.Audit}(t_1, t_n, (\Pi_t^{\mathsf{Upd}})_{t=t_1}^{t_n-1})$: If for any of the epochs, $t \in [t_1, t_n - 1]$, t is in $\mathsf{TombstoneEpochs}$, the auditor calls $\mathsf{VKD.VerifyTombstone}(\mathsf{com}_t, \mathsf{com}_{t+1}, \Pi_t^{\mathsf{Upd}}, t - \mathsf{StaleParam})$, else, if $t - \mathsf{DeletionParam} \in \mathsf{TombstoneEpochs}$, i.e. $t \in \mathsf{CompactionEpochs}$, call $\mathsf{VKD.VerifyCompact}(\mathsf{com}_t, \mathsf{com}_{t+1}, \Pi_t^{\mathsf{Upd}}, t - \mathsf{DeletionParam} - \mathsf{StaleParam})$. For all other $t \in [t_1, t_n - 1]$, this algorithm calls $\mathsf{VKD.VerifyUpd}(t, t{+}1, (\mathsf{com}_t, \Pi_t^{\mathsf{Upd}}), \mathsf{com}_{t+1})$. It outputs 1, if all subroutines output 1, otherwise it outputs 0.

Soundness for VKD with Compaction.

Theorem 1. *The construction for a VKD with compaction in appendix F3 satisfies Definition 4.*

Proof. First, note that for any epoch t, the value com_t received by all pairs of parties must be equal. If not, the adversary of the $\mathsf{WitnessAPI}$ can use a VKD adversary to output inconsistent values, breaking the security of the $\mathsf{WitnessAPI}$.

Suppose for every $t \in \mathsf{TombstoneElts}$ such that $[t, t + \mathsf{DeletionParam}] \cap [t_1, t_{\mathsf{current}}]$ is non-empty, the user receives a verifying key history proof at some epoch $t' \in [t, t + \mathsf{DeletionParam}]$. Also, assume that the system parameter $\mathsf{StaleParam}$ is large enough that no data relating to changes in the label-value set of the VKD made between between two consecutive tombstone epochs is tombstoned.

Now, if the user receives two proofs $(t, \{(\mathsf{val}_k^t, \mathsf{Epoch}_k^t)\}_{k=\mathsf{min}_t}^{\mathsf{max}_t}, \Pi_t^{\mathsf{Ver}})$, $(u, \{(\mathsf{val}_k^u, \mathsf{Epoch}_k^u)\}_{k=\mathsf{min}_u}^{\mathsf{max}_u}, \Pi_u^{\mathsf{Ver}})$ in response to $\mathsf{VKD.KeyHistory}$, with $t_1 \leq t < u \leq t_{\mathsf{current}}$. For any version number $\beta \in [\mathsf{min}_t, \mathsf{max}_t] \cap [\mathsf{min}_u, \mathsf{max}_u]$, we claim that $(\mathsf{val}_\beta^t, \mathsf{Epoch}_\beta^t) = (\mathsf{val}_\beta^u, \mathsf{Epoch}_\beta^u)$. Suppose not: to generate such a pair of proofs, the adversary would have to include membership proofs for diverging views of the same label '$\mathsf{label}|\beta$' in the oZKS at times t and u. Recall that we assume that if any deletion epochs occurred between t and u, the user checked correct tombstoning of any deleted values. This means that if the label $\mathsf{label}|\beta$ is included in both proofs, the minimum version number $\mathsf{min}_u \leq \beta$ and the value committed with the label $\mathsf{label}|\beta$ must have remained unchanged in the epochs between t and u. Thus, for val_β^u and val_β^t to diverge, the adversary must have produced proofs of diverging views in the oZKS.

Fact. For two history proofs from epochs t and u, for any version β, included in both proofs, $(\mathsf{val}_\beta^t, \mathsf{Epoch}_\beta^t) = (\mathsf{val}_\beta^u, \mathsf{Epoch}_\beta^u)$.

We have established that if a user is checking once between any pair of tombstone epochs in $[t_1, t_{\mathsf{current}}]$, any two $\mathsf{VKD.KeyHistory}$ proofs which pass must contain the same $(\mathsf{val}, \mathsf{Epoch})$ for a given version number β.

Now, suppose a call to $\mathsf{VKD.Query}$ returns proof π, which verifies at some epoch $t^* \in [t_1, t_n]$, with version α and the associated value-epoch pair $(\mathsf{val}, \mathsf{Epoch})$. Also, suppose that for some $t \in [t_1, ..., t_n]$, the proof Π_t^{Ver}, with associated values $(\mathsf{val}_k, \mathsf{Epoch}_k)\}_{k=\mathsf{min}_t}^{\mathsf{max}_t}$ passes, such that $t^* \in [\mathsf{Epoch}_\beta, \mathsf{Epoch}_{\beta+1}]$ for some version number $\beta \in [\mathsf{min}_t, \mathsf{max}_t]$.

We only consider the case where $\mathsf{val} \neq$ TOMBSTONE, since users should not accept TOMBSTONE as a value, in response to VKD.Query.

At a high level, our soundness definition requires that with high probability, $\mathsf{val} = \mathsf{val}_\beta$.

Case I $t^*, t \in [\mathsf{TombEp}_k, \mathsf{TombEp}_{k+1})$, for some k. An adversary could cause a disparity between val and val_β in one of the following ways:

- $\alpha = \beta$, $\mathsf{val} \neq \mathsf{val}'$: In this case, the adversary must produce a membership proof for $\mathsf{label}|\alpha$ in the oZKS, with values $\mathsf{val}, \mathsf{val}_\beta$ which are unequal. Recall that except for tombstone and deletion epochs, the oZKS must remain append-only. Further, any oZKS values which are deleted in a deletion epoch must be tombstoned. If $\mathsf{val} \neq$ TOMBSTONE at the epoch TOMBSTONE_k, this breaks the security of the oZKS, since an oZKS adversary can use this VKD adversary as a subroutine to produce proofs for disparate values within epochs which are not tombstone epochs.
- $\alpha < \beta$: For VKD.VerifyHistory to output 1 with $\mathsf{Epoch}_\beta \leq t^*$, for all $\alpha < \beta$, it should either (1) receive a non-membership proof of $\mathsf{label}|\alpha$ in the oZKS or (2) a membership proof of $\mathsf{label}|\alpha|\mathsf{stale}$ that should have been added at epoch $\mathsf{Epoch}_\alpha \leq \mathsf{Epoch}\beta$. On the other hand, for VKD.VerifyQuery to pass, it should have verified a membership proof of $\mathsf{label}|\alpha$ as well as a non-membership proof of $\mathsf{label}|\alpha|\mathsf{stale}$ at epoch t^*. Hence, if a VKD.VerifyQuery returned 1, and either (1) or (2) is true, the adversary violated the requirement that the auditors ensure that the epoch when a node is inserted is committed with its value in the oZKS and that both membership and non-membership checks of non-tombstoned values passed,
- $2^{\lfloor \log(\beta) \rfloor + 1} > \alpha > \beta$: In order for Π^{Ver} to verify with the history at epoch t, with the version number β for epochs $[\mathsf{Epoch}_\beta, \mathsf{Epoch}_{\beta+1})$, it needs to present one of the following: (1) a proof of oZKS membership for the label $\mathsf{label}|\alpha$ with value (val, t^*), with $t^* > t$, or, (2) a non-membership proof in the oZKS of the label $\mathsf{label}|\alpha$ at the epoch t. If either of these oZKS proofs was generated, in order for π to also verify, the adversary would have to also generate either (1) both a membership and non-membership proof for the same label, or (2) generate two membership proofs mapping to distinct value-epoch pairs. This would allow constructing an oZKS adversary.
- $\alpha \geq 2^{\lfloor \beta \rfloor + 1} + 1$: In this case, for the proof π to verify, at epoch t^*, the adversary must have generated a proof of oZKS membership for the label $\mathsf{label}|2^{\lfloor \alpha \rfloor}$ mapping to a tuple of the form $(\cdot, t'')$ such that $t'' < t$. As in the previous case, the adversary must have also generated either (1) a proof of oZKS membership for the label $\mathsf{label}|2^{\lfloor \alpha \rfloor}$ but mapping to a tuple $(\cdot, t''')$ with $t''' > t^*$, or (b) an oZKS non-membership proof for the same label. All of these violate oZKS soundness.

Case II $t^* \in [\mathsf{TombEp}_k, \mathsf{TombEp}_{k+1})$, and $t \in [\mathsf{TombEp}_{k+1}, \mathsf{TombEp}_{k+2})$, for some k. Again, we consider the following cases:

- $\alpha = \beta$, $\mathsf{val} \neq \mathsf{val}_\beta$: Recall that $(\mathsf{val}_\beta, \mathsf{Epoch}_\beta)$ are only included in the output of KeyHistory if $\mathsf{val}_\beta \neq$ TOMBSTONE. Also recall that the oZKS auditors check that any value which is not marked tombstoned is not mutated. Hence, this reduces to the problem of the adversary showing diverging views for the label $\mathsf{label}|\alpha$ in the oZKS – the probability of the adversary succeeding is negligible due to the security of the oZKS.
- $\alpha < \beta$: This reduces to two cases: (1) $\mathsf{min}_t \leq \alpha < \beta \leq \mathsf{max}_t$ and (2) $\alpha < \mathsf{min}_t$. In case (1), if the proof Π^{Ver} verifies, we know that for $i < j$, $\mathsf{Epoch}_i < \mathsf{Epoch}_j$ and the label $\mathsf{label}|i|\mathsf{stale}$ must include a proof of being inserted at the epoch Epoch_{i+1}. Hence, at epoch t^*, the adversary must have shown a non-membership proof for $\mathsf{label}|\alpha|\mathsf{stale}$ as part of π, but a membership proof for $\mathsf{label}|\alpha|\mathsf{stale}$ with epoch $\mathsf{Epoch}_{\alpha+1} \leq t^* \leq \mathsf{Epoch}_\beta$ as part of Π^{Ver}. This violates the oZKS security. For case (2), this means that α was never included with the proof Π^{Ver} and hence we do not need to consider it.
- $\beta < \alpha \leq \mathsf{max}_t$: By the same argument as in the previous case, for VerifyQuery to verify with version number α at epoch t^*, $\mathsf{Epoch} \leq t^*$. However, for VerifyHistory to pass, it must include a membership proof of the label $\mathsf{label}|\beta|\mathsf{stale}$, inserted at an epoch $\mathsf{Epoch}_\beta < e \leq \mathsf{Epoch} \leq t^*$. Thus the adversary either included membership proofs for $\mathsf{label}|\alpha$ with epochs $\mathsf{Epoch}_\alpha \neq \mathsf{Epoch}$ in Π^{Ver}, π, or got an Audit to pass with the wrong epoch committed at a leaf.
- $\mathsf{max}_t < \alpha \leq 2^{\lfloor \log \beta \rfloor + 1} - 1$ or $\alpha \geq 2^{\lfloor \log \beta \rfloor + 1}$: The cases are identical to the corresponding cases in **Case I** above.

Hence, with all but negligible probability, $\alpha = \beta$ and $\mathsf{val} = \mathsf{val}'$ in both cases. Note that we require that for every pair of tombstone epochs which intersect with the interval $[t_1, t_{\mathsf{current}}]$, we require some element of $\{t_1, ..., t_n\}$ to be in that interval. We also showed that the values for the same version number during two key history checks should match, hence covering these two cases is exhaustive.

□

G. *Summary of the aZKS Construction from [16]*

Append-only Zero Knowledge Set (aZKS) was introduced in [16] (Section 3). We summarize the construction below. Part of the text is copied verbatim from [16].

aZKS is a primitive that lets a (malicious) prover commit to an append-only dictionary of (label,value) pairs (where the labels form a set) such that: 1) the commitment is succinct and does not leak any information about the committed dictionary 2) the prover can prove statements about membership and non-membership of (label,value) pairs with respect to the succinct commitment 3) the prover can prove that for two dictionaries, D_1, D_2, $D_1 \subseteq D_2$ with respect to their respective succinct commitments and 4) the proofs are efficient and do not leak any information about the rest of the committed dictionary.

aZKS has two security properties: *Soundness* and *Zero-Knowledge Privacy (with leakage)*. Soundness ensures that a malicious prover 1) will not be able to produce two verifying proofs for two different values for the same label with respect to a commitment or 2) should not be able to modify an existing label. Zero-Knowledge privacy property captures that the query proofs and append-proofs leak no information beyond the query answer (which is a value or $\perp$ for membership queries and a bit indicating validity of an an append operation) and a well-defined leakage function.

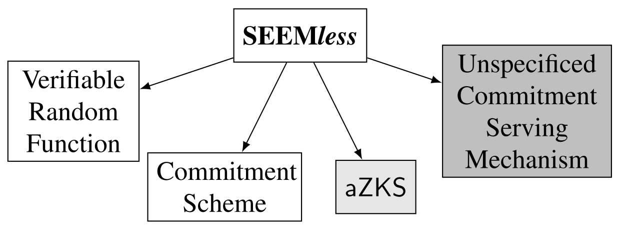
H. Summary of SEEMless construction [16]

In this section, we summarize the construction from [16], Section 4. Part of the text is copied verbatim from [16].

SEEMless assumes that server's identity and public key is known to each user and auditor and all the messages from the server are signed under the server's key, so that the server cannot be impersonated.

SEEMless uses two aZKS: one "all" aZKS to store all versions of all $(\mathsf{label}, \mathsf{val})$ pairs, and a second "old" aZKS that stores all of the out of date label versions. SEEMless also uses a *hash chain*. Hash Chain is a classical authentication data structure that chains multiple data elements by successively applying cryptographic hashes, e.g. a hash chain that hashes elements a, b, c is $H(c, H(b, H(a)))$.

- $\mathsf{VKD.Publish}(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)$: At every epoch, the server gets a set S_t of (label, value) pairs that have to be added to the VKD. The server first checks if the label already exists for some version $\alpha - 1$, else sets $\alpha = 1$. It adds a new entry $(\mathsf{label} \mid \alpha, \mathsf{val})$ to the "all" aZKS and also adds $(\mathsf{label} \mid \alpha - 1, \mathsf{null})$ to the "old" aZKS if $\alpha > 1$. If the new version $\alpha = 2^i$ for some i, then the server adds a marker entry $(\mathsf{label} \mid \mathsf{mark} \mid i,$"marker") to the "all" aZKS. The server computes commitments to both the aZKS, and adds them to the hash chain to obtain a new head com_t. It also produces a proof Π^{Upd} consisting of the previous and new pair of aZKS commitments $\mathsf{com}_{\mathsf{all},t-1}, \mathsf{com}_{\mathsf{all},t}$ and $\mathsf{com}_{\mathsf{old},t-1}, \mathsf{com}_{\mathsf{old},t}$ and the corresponding aZKS update proofs.
- $\mathsf{VKD.Query}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: When a client Bob queries for Alice's label, he should get the val corresponding to the latest version α for Alice's label and a proof of correctness. Bob gets three proofs in total: First is the membership proof of $(\mathsf{label} \mid \alpha, \mathsf{val})$ in the "all" aZKS. Second is the membership proof of the most recent marker entry $(\mathsf{label} \mid \mathsf{mark} \mid a)$ for $\alpha \geq 2^a$. And third is non membership proof of $\mathsf{label} \mid \alpha$ in the "old" aZKS. Proof 2 ensures that Bob is not getting a value higher than Alice's current version and proof 3 ensures that Bob is not getting an old version for Alice's label.
- $\mathsf{VKD.VerifyQuery}(t, \mathsf{label}, \mathsf{val}, \pi, \alpha)$: The client checks each membership or non-membership proof, and the hash chain. Also check that version α as part of proof is less than current epoch t.
- $\mathsf{VKD.KeyHistory}(\mathsf{st}_t, \mathsf{Dir}_t, t, \mathsf{label})$: The server first retrieves all the update epochs $t_1, \ldots, t_\alpha$ for label versions $1, \ldots, \alpha$ from T, the corresponding $\mathsf{com}_{\mathsf{all},t_1-1}, \mathsf{com}_{\mathsf{all},t_1}, \ldots, \mathsf{com}_{\mathsf{all},t_\alpha-1}, \mathsf{com}_{\mathsf{all},t_\alpha}$ and $\mathsf{com}_{\mathsf{old},t_1}, \ldots, \mathsf{com}_{\mathsf{old},t_\alpha}$ and the hashes necessary to verify the hash chain: $H(\mathsf{com}_{\mathsf{all},0}, \mathsf{com}_{\mathsf{old},0}), \ldots, H(\mathsf{com}_{\mathsf{all},t}, \mathsf{com}_{\mathsf{old},t})$. For versions $i = 1$ to n, the server retrieves the val_i for t_i and version i of label from Dir_{t_i}. Let $2^a \leq \alpha < 2^{a+1}$ for some a where α is the current version of the label. The server generates the following proofs (together called as Π):
  1) **Correctness of** com_{t_i} **and** com_{t_i-1}: For each i, output com_{t_i} com_{t_i-1}. Also output the values necessary to verify the hash chain: $H(\mathsf{com}_{\mathsf{all},0}, \mathsf{com}_{\mathsf{old},0}), \ldots, H(\mathsf{com}_{\mathsf{all},t}, \mathsf{com}_{\mathsf{old},t})$.
  2) **Correct version** i **is set at epoch** t_i**:** For each i: Membership proof for $(\mathsf{label} \mid i)$ with value val_i in the "all" aZKS with respect to com_{t_i}.
  3) **Server couldn't have shown version** $i - 1$ **at or after** t_i : For each i: Membership proof in "old" aZKS with respect to com_{t_i} for $(\mathsf{label} \mid i - 1)$.
  4) **Server couldn't have shown version** i **before epoch** t_i: For each i: Non membership proof for $(\mathsf{label} \mid i)$ in "all" aZKS with respect to com_{t_i-1}.
  5) **Server can't show any version from** $\alpha + 1$ **to** 2^{a+1} **at epoch** t **or any earlier epoch:** Non membership proofs in the "all" aZKS with respect to com_t for $(\mathsf{label} \mid i + 1), (\mathsf{label} \mid i + 2), \ldots, (\mathsf{label} \mid 2^{a+1} - 1)$.
  6) **Server can't show any version higher than** 2^{a+1} **at epoch** t **or any earlier epoch:** Non membership proofs in "all" aZKS with respect to com_t for marker nodes $(\mathsf{label} \mid \mathsf{mark} \mid a + 1)$ up to $(\mathsf{label} \mid \mathsf{mark} \mid \log t)$.
- $\mathsf{VKD.VerifyHistory}(t, \mathsf{label}, (\mathsf{val}_i, t_i)_{i=1}^n, \Pi^{\mathsf{Ver}})$: Verify each of the above proofs.
- $\mathsf{VKD.Audit}(t_1, t_n, \left(\Pi_t^{\mathsf{Upd}}\right)_{t=t_1}^{t_{n-1}})$: Auditors will audit the commitments and proofs to make sure that no entries ever get deleted in either aZKS. They do so by verifying the update proofs Π^{Upd} output by the server. They also check that at each epoch both aZKS commitments are added to the hash chain. Note that, while the Audit interface gives a monolithic audit algorithm, our audit is just checking the updates between each adjacent pair of aZKS commitments, so it can be performed by many auditors in parallel.


SEEMless
Verifiable Random Function
Commitment Scheme
aZKS
Unspecificed Commitment Serving Mechanism
Fig. 10: Schematic of the building blocks for SEEM*less*.

I. Storage

In the next two subsections we go over specific storage requirements of SEEMless and our solution Parakeet, respectively. We first provide high-level statistics about a world-scale key transparency solution based on compressed Merkle

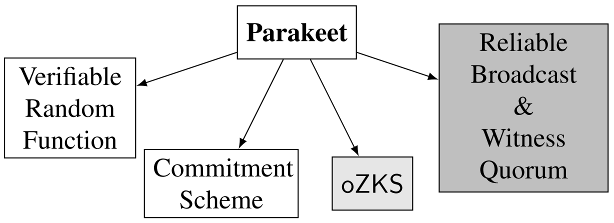

Parakeet
Verifiable Random Function
Commitment Scheme
oZKS
Reliable Broadcast & Witness Quorum

Fig. 11: Schematic of the building blocks for Parakeet. Note that the shaded components are different from the corresponding components of SEEM*less*, which are shaded in Figure 10.

trees and use WhatsApp (the most popular E2EE messaging app) discussed in section I, as an example. We assume that WhatsApp has about two billion existing keys (K_I), and roughly ten million daily key updates (K_D). To simplify our current calculations, we assume that after initially setting up the key transparency solution, the server only receives requests to update keys for existing users. Note that updates to keys for existing users require more values to be inserted into the tree (to mark old version numbers as stale), so our assumptions are closer to a worst case scenario. This means that:

- The total number of keys added to the system in the first year is $K_T = 2 \times 10^9 + 365 \times 10^7 = 5.65$ billion keys.
- In both SEEMless [16] and Parakeet which are based on a (compressed) Patricia Merkle Trie, the number of nodes needed for key transparency is roughly $N_T = 2 \times K_I + 4 \times K_D = 18.6$ billion by the following reasoning
  - Adding a leaf node results in additional one node (i.e., two nodes in total) for the longest common prefix parent.
  - When initially setting up, all keys are treated as a users initial version and hence require only one leaf of the form uname$|1$, for each user name.
  - For all subsequent updates, two leaves must be added for each key: one of the form uname$|i+1$, to add the new key and, the other of the form uname$|i|$stale, to mark the old key as stale.
- Number of nodes initially is $N_I = 2 \times K_I = 4$ billion.
- Number of nodes created daily is $N_D = 4 \times K_D = 40$ million.
- Number of nodes that need to be updated for a new leaf node is upper-bounded by the depth of the tree. The amortized depth of any inserted node is $\log(n)$ where n is the number of leaves in the tree before this insertion.

1) *SEEMless Storage*

SEEMless relies on saving the state of a node every time the node is updated. This makes the storage cost highly dependent on the number of epochs and the number of nodes in the tree.

Let us use the number of leaves in the tree initially as a lower bound on how many node states need to be updated; and we assume there is one epoch per-day. Total number of nodes that SEEMless needs to store states for in this case is N_I nodes initially and $N_D * \log N_I$ daily. In total $2 \times 10^9 + 365 \times 4 \times 10^8 \times \log_2(4 \times 10^9)$ is approximately 470 billion in the first year.

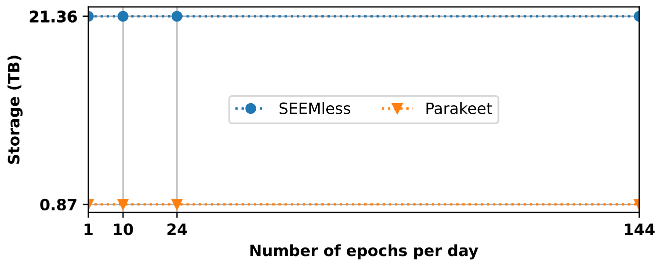


Fig. 12: Comparison of storage costs of Parakeet and SEEMless in the first year with varying number of epochs per day ranging from daily to every 10 minutes.

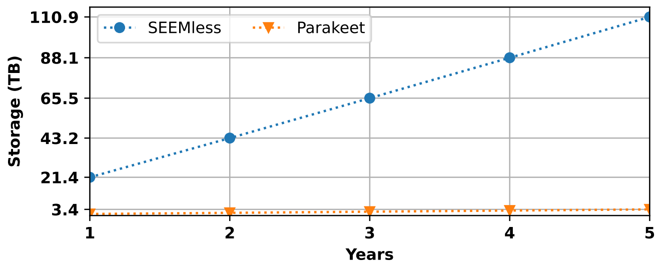


Fig. 13: Comparison of storage costs of Parakeet and SEEMless in the first five years.

With 64-byte node states, for hash function output (32-byte) along with other node info (32-byte) such as the parent and children, the total storage requirement is around 27TBs.

2) *Parakeet Storage*

Parakeet's main advantage is that only the latest state of a node needs to be stored. Essentially the final cost depends on the number of total nodes in the tree at the end of the year.

The total cost considering 64-byte node states – same as SEEMless, $N_T * 64B = 1.1$TBs. Since to allow concurrent proof generation and Parakeet stores a previous node state, the final cost is 2.2TBs – an order of magnitude more efficient than the previous best solution. Furthermore, our compaction mechanism (See Section III) can further reduce the storage requirements for Parakeet.

In comparison, the storage cost of actual keys and their owners' information (e.g., phone number) is around 360GB (considering 64B record sizes). An efficient key transparency solution with Parakeet is highly feasible.