SEEM*less*: Secure End-to-End Encrypted Messaging with *less* Trust

Melissa Chase
Microsoft Research
melissac@microsoft.com

Apoorvaa Deshpande
Brown University
apoorvaa_deshpande@brown.edu

Esha Ghosh
Microsoft Research
esha.ghosh@microsoft.com

Harjasleen Malvai
Cornell University
hm553@cornell.edu

ABSTRACT

End-to-end encrypted messaging (E2E) is only secure if participants have a way to retrieve the correct public key for the desired recipient. However, to make these systems usable, users must be able to replace their keys (e.g. when they lose or reset their devices, or reinstall their app), and we cannot assume any cryptographic means of authenticating the new keys. In the current E2E systems, the service provider manages the directory of public keys of its registered users; this allows a compromised or coerced service provider to introduce their own keys and execute a man in the middle attack.

Building on the approach of CONIKS (Melara et al, USENIX Security '15), we formalize the notion of a *Privacy-Preserving Verifiable Key Directory* (VKD): a system which allows users to monitor the keys that the service is distributing on their behalf. We then propose a new VKD scheme which we call SEEMless, which improves on prior work in terms of privacy and scalability. In particular, our new approach allows key changes to take effect almost immediately; we show experimentally that our scheme easily supports delays less than a minute, in contrast to previous work which proposes a delay of one hour.

KEYWORDS

Privacy-preserving verifiable directory service, Zero knowledge sets, Accumulators, Persistent Patricia Trie, History Tree, PKI, Transparency, Security definitions

1 INTRODUCTION

A number of popular messaging apps such as iMessage, WhatsApp and Signal have recently deployed end-to-end encryption (E2EE) in an attempt to mitigate some of the serious privacy concerns that arise in these services. E2EE is a system of communication where only encrypted messages leave the sender's device. These messages are then downloaded and decrypted on the recipient's device, ensuring that only the communicating users can read the messages. But E2EE relies on a Public Key Infrastructure (PKI); this in practice requires the provider of the messaging service (such as Apple, Facebook, Microsoft etc.) to maintain a centralized directory of the public keys of its registered users. To be accessible to average users, these systems assume the user will store her secret key on her personal device, and do not assume she has any other way to store long term secrets. When a user loses her device (and thus her secret key), she will need to generate a new (secret key, public key) pair and replace her old public key stored in the PKI with the newly generated public key.

Such a system naturally places a lot of trust in the service provider – a malicious service provider (or one who is compelled to act maliciously, possibly because of a compromise) can arbitrarily set and reset users' public keys. It might, for example replace an honest user's public key with one whose secret key it knows, and thus implement a man-in-the-middle attack without the communicating users ever noticing. Ironically, this defeats the purpose of E2EE. Without some way of verifying that the service provider is indeed returning the correct keys, E2E encryption does not provide any protection against malicious (or coerced) service providers. This problem has been well recognized as an important and challenging open problem [15, 21, 26].

Some service providers provide a security setting option to notify a sender when a recipient's public key changes. In WhatsApp, the sender can scan a QR code on the recipient's device to verify the authenticity of the new public key. Skype encrypted messaging provides a similar interface for checking fingerprints. This option, however, is turned off by default to provide a seamless user experience. Moreover, the communicating users will be able to verify each other's QR codes only if their devices are physically close to each other, which is most often not the case. And these features are something few users use, as is evidently uncovered in this attack [1].

To enable E2EE with real security, we need to keep the inherent constraints of the system in mind. To begin with, the primary objective of any E2EE messaging system is to provide a *secure and seamless* communication service between remote users. This problem is made even more challenging by the fact that we must assume that a user can lose her device and along with it all of her secrets. Moreover, in our attempt to reduce trust on the service provider, we must not introduce any new attack surface. For example, if we introduce a mechanism that will enable a user to verify that she is receiving the correct key of the intended recipient, this mechanism should not leak any additional information about the other users and the public keys they have registered with the service provider. Privacy may not be very important in a traditional PKI, where all the players are usually public entities like businesses, but in the context of private messaging, privacy is very important. Hiding usernames may help prevent spam messaging. And the user's update pattern may itself be sensitive information. Users change their keys primarily when they change devices, or detect that their devices/keys have been compromised, either of which may be sensitive. Moreover, if a user rarely changes her key, then compromising her secret key gives the attacker the ability to decrypt large volumes of messages at once, making her a more vulnerable and attractive target. Or if a device is compromised but the user does not update her key, then the attacker knows the compromise has gone undetected.

Recently, Keybase has rolled out an auditable key directory service that aims to reduce trust on the centralized service provider managing the public key distribution [16, 17]. While we are aiming to solve a similar problem, the solution proposed by Keybase is significantly different from ours in the assumptions they make. Keybase assumes that the users have multiple truted devices the and they have access to long-term cryptographic signature keys with which they sign all their encryption key updates. This multiple-device assumption already rules out a good chunk of the end-user population. Moreover, for an end user, it is incredibly difficult to manage cryptographic keys, so we believe this assumption is not very realistic. Our system can also support these *power users* who can sign all their updates, but this is not a requirement. Another significant difference between our goals is the privacy requirement. All the key changes by the end-users in Keybase are publicly available, so their system does not provide any privacy for the end-users. Finally, contrary to our design, the Keybase solution assumes that Alice posts her latest signature verification key on some social media platform [16], which Bob will have to lookup to get assurance that he is seeing the latest encryption key for Alice. This introduces significant overhead for both Alice and Bob.

We design and provide a prototype implementation of SEEMless – a privacy-preserving directory service for end-user key verification. We also initiate the study of the privacy and security guarantees of such a key directory service formally. Our major contributions are the following.

Formalizing Key Directory Services. We formalize the security and privacy requirements of a verifiable key directory service, (such as SEEMless, CONIKS [21], EthIKS [4]), in terms of a new primitive that we call Verifiable Key Directories (VKD) (Section 2). A VKD consists of three types of entities: an identity provider or server, users, and external auditors. The server stores a directory Dir with the names of the users and their corresponding keys. VKD provides different query interfaces to the users to interact with the server: 1. Alice can register her (username, key) to Dir and update it at any point* 2. Bob can query for Alice's key 3. Alice can obtain the history of her key updates. This last query allows Alice to verify that the only key updates that occurred were those that she requested (we assume that Alice roughly remembers when she changed her device or reinstalled her software forcing a key update). Note that we do not assume Alice can remember any cryptographic secret as she can lose her device any time. We discuss the assumptions we make for a VKD system in Section 4.

A key criterion of such a system is that Alice must be able to verify that any user Bob who queried for her key received a value consistent with what the server tells her when she checks her key history. To enable consistency checking, a VKD server periodically signs and publishes a privacy-preserving digest of its latest Dir. We assume an additional set of parties which we call auditors who help verify that the server is showing the same view of Dir to all its users by periodically auditing these digests. These auditors are not trusted for privacy, so any interested party (privacy conscious users, or privacy advocacy organizations) can serve as an auditior. If a VKD server ever tries to equivocate by issuing multiple keys for a single username, this would require publishing invalid digests or conflicting digests which would provide irrefutable proof of the server's equivocation.

Soundness of a VKD requires that Bob sees Alice's latest key as long as Alice has checked her key history and there is at least one honest auditor that audited each of the published server digests. *Privacy* is parameterized by a well specified leakage function $\mathcal{L}$; informally, we require that the published digests and the proofs should not leak any information about Dir beyond the query answer and $\mathcal{L}(\mathsf{Dir})$. For example, for every query performed by the users or auditors, our construction leaks the total number of key registrations and updates performed so far. In addition, when Bob queries for Alice's key, he learns when Alice's key was last updated and how many times she has changed her key so far, but nothing else about her past or future updates, or about any other user.

Specifying a precise leakage function leads to a better understanding of the privacy guarantee of any VKD system. As a concrete example, we were able to identify a *tracing attack* in current implementations [4, 21] through which it might be possible to trace the entire update history of a particular user. This means once Alice queried for Bob's key, she might be able to completely trace when Bob's key changed without ever querying for his key again. Even if Bob has deleted Alice from his contact list (and the server does not give out Alice's key to Bob after he has removed Alice from his contact list), Alice will still be able to trace when Bob's key changed just my looking at the proof of her own key. We explain the technical details of this attack in Appendix D.

Building Blocks. We take a modular approach in designing our VKD system, SEEMless (Section 4). We first define a new primitive, *Append-Only Zero Knowledge Set* (aZKS), that generalizes the definition of a traditional static zero-knowledge set [6, 22] by accounting for updates. We construct SEEMless using aZKS in a blackbox way. Then we construct aZKS modularly from a *strong accumulator* [5] and a *verifiable random function* [7]. By providing modular constructions, we simplify the presentation and analysis and allow for independent study of each of these building blocks: an improvement in any of them would directly translate to improvement in SEEMless.

The high level idea in our VKD construction (Section 4) is to use aZKS to commit to the current directory and an append-only Merkle Hash Tree to efficiently maintain server commitments over time. A zero knowledge set allows a server to commit to a set of (label, value) pairs, respond to lookup queries for different labels, and prove that those responses are correct without revealing any additional information. Append-only ZKS allow for additional pairs to be added to the database; that is the only legal update. Since the aZKS is append only and each label can only appear once, we append each username with a version number showing how many times that key has been updated so far.

We use *two* aZKS; one which consists of all the (username | version, key) pairs corresponding to all the updates made so far. The other

*We assume that the identity provider can verify that the update request comes from Alice through some out-of-band, non-cryptographic check, like verifying security questions, sending a text to an appropriate phone number, and/or requiring voice or in person communication.

aZKS has (username | version, key) pairs with all the versions except the latest, i.e. the versions that are now old. The purpose of maintaining two aZKS is to prevent the server from serving stale keys; now when Alice updates her key, instead of deleting the old entry, the server will add it to the "old" aZKS, and when Bob queries for Alice's key the server will additionally prove that the entry it returns is not "old". We have a more sophisticated approach for preventing the server from showing Bob a higher version number for Alice's key, which involves storing a "marker" every 2^ith update; we defer details to Section 4.

Auditors in the system will verify that (1) the server's published commitments are linearly ordered and (2) the underlying aZKS are indeed append-only. The first goal can be achieved using a hash chain (or for improved efficiency an append-only Merkle Hash Tree as described above). SEEMless does not take a position on how to ensure that users and auditors eventually see the same commitments. This can be achieved either with a gossip protocol [21] or by posting the commitments on a public blockchain like [4, 27].

Persistent Patricia Trie. Recall that SEEMless provides a KeyHistory API which allows Alice to audit all her key changes. Adding this functionality requires the server to maintain the history of the two aZKS starting with the very first epoch; the same is true for CONIKs in order to support users who are not always online. This is obviously very space consuming and wasteful since there might be a significant overlap between the aZKS of consecutive epochs. Motivated by this constraint, we develop a *Persistent Patricia Trie* (PPTr) data structure and use it to build a persistent version of our strong accumulator, i.e. one in which we can also answer queries relative to older versions of the data structure. This directly translates to space-efficient persistent aZKS. PPTr is a space-efficient history-preserving data structure (like history tree [24]) which lets the server generate historical proofs (KeyHistory in our case) efficiently. The details of PPTr are in Section 6. Our PPTr construction presents a way of maintaining any temporal, append-only, authenticated log of data. We believe PPTr will find applications in several contexts, e.g. tamper-evident logging, distributed identity providers and distributed certificate authorities and may be of independent interest.

Epoch length. One key parameter in a VKD is the length of the epochs between server updates. Note that, when Alice updates her key, the change is not reflected in the server's digest until the next update. This presents two options: either the server can continue to respond with the old key until the end of the epoch, or the server can respond with a key which is not consistent with the digest. CONIKS proposes this latter approach; it is unclear how the user can verify that this new key is in fact correct. In our solution, we propose making the epochs short enough that we can take the first approach without significantly affecting security or usability of the system. This requires significant efficiency and scalability improvements over previous work.

Privacy and Efficiency Improvement. SEEMless improves upon the existing VKD constructions [4, 21] on privacy and scalability aspects. Our construction is more space efficient for the server, thanks to our PPTr construction. In particular, the space required depends not on the number of epochs, but only on the number of key updates performed, despite the fact that we must be able to answer questions about historical key values. We also significantly improve on the monitoring cost of [4, 21]. CONIKS [21] required that Alice query the server for every epoch to ensure that her key was correct at that time; this meant both that epochs couldn't be too short without substantial burden on the system, and that a user who was offline for a period would have to perform all the checks for that time on coming back online. ETHIKS [4] improved this, by moving most of the work to the blockchain, but the epoch length is still limited by the blockchain block frequency.[†] In our system, using the KeyHistory API, Alice can monitor her key revisions with cost that depends on the number of times her key has been updated rather than the number of epochs that have passed. Similarly she can be offline for several server epochs and can run KeyHistory when she comes online. This together with the storage improvements mean that we can support much shorter epochs, which in turn leads to a more secure system as discussed above, in that Alice's key updates can be very quickly reflected in the directory. Finally, our construction provides stronger privacy guarantees compared to [4, 21].

Experiments. We give a prototype implementation of our VKD system, SEEMless and experimentally evaluate the cost incurred for the users, the server and the auditors. We also run CONIKS to fairly compare our performance costs. Note that our prototype *only implements a privacy-preserving verifiable key directory* which can be integrated with an E2E Encrypted messaging system. In our experiments, the cost of lookup query in SEEMless is slightly more expensive than in CONIKS (at most 3×) due to our stronger privacy guarantees, but it is still about $10ms$ from request to complete verification, which is reasonably fast. However, SEEMless performs and scales well with frequent server updates, as opposed to CONIKS. For the same update frequency and experimental setup, CONIKS cannot handle more than 4.5M users, owing to the inefficiency of their underlying data structure (our system could handle 10M users seamlessly) and hangs with more frequent epochs due to garbage collection overhead. Moreover, users of SEEMless need not monitor their keys as frequently as the server publishes its digest; they can verify their entire key history by downloading around 2MB (even if they are updating their keys often) when they come back online; in contrast, a CONIKS, user will need to download about 577MB in a similar scenario, which makes the monitoring cost quickly infeasible. The details of our experiments and performance comparison with CONIKS is in Section 7.

Organization of the paper In Section 2, we define the primitive of Verifiable Key Directories (VKD) along with the security properties. In Section 3, we define append-only Zero Knowledge Sets (aZKS) which is a building block for VKD construction. In Section 4, we describe our SEEMless construction starting with an overview of the same. In Section 5, we give concrete instantiations of aZKS. In Section 6 we describe our space efficient Persistent Patricia Trie construction. We present our prototype implementation and performance results in Section 7. Finally, in Appendix A we discuss some design considerations, including assumptions we are making

[†]It also required Alice to remember her last version number which we feel is less natural to remember.

about the users, the underlying infrastructure, and the way that this system would be integrated with a messaging app.

2 VERIFIABLE KEY DIRECTORY (VKD)

In this section, we will define the primitive of a *Verifiable Key Directory* (VKD) and formalize its properties. The goal of a VKD is to capture the functionality and security of a privacy-preserving verifiable key directory system such as CONIKS or EthIKS [4, 21].

A VKD consists of three types of parties: an identity provider or server, clients or users and external auditors. The server stores a directory Dir with the names of the users (which we call labels) and their corresponding public keys (the values corresponding to the labels). For the ease of exposition, let Alice and Bob be two users. VKD provides the following query interface to the users. 1) Alice can add her (username, key), i.e., (label=username, val=key), to Dir. 2) Alice can update her key and request that Dir be updated with the new key value. 3) She can query the server periodically to obtain history of her key updates over time (VKD.KeyHistory). 4) Bob can also query for the key corresponding to username Alice (VKD.Query) at the current time.

The functionality VKD.KeyHistory warrants further discussion since this is not a functionality that one usually expects from a key directory service. In a privacy-preserving verifiable key directory service, intuitively, we expect the server to be able to prove to Bob that he is seeing Alice's latest key without leaking any additional information about the directory. This is trivial to achieve if we assume that Alice can always sign her new public key with her old secret key. But this is a completely unreasonable assumption from an average user who may lose her device or re-install the software, thereby losing her previous secret key; the user will only have access to her latest secret key which is stored on her latest device. It is crucial to *not assume* that Alice or Bob can remember any cryptographic secret. Under this constraint, we need Alice to monitor her key sufficiently often to make sure her latest key is in the server directory. Only then, we can talk about Bob getting Alice's latest key in a meaningful way. Alice could of course check every epoch to make sure that her key is being correctly reported, but this becomes costly, particularly when epochs are short. Instead, we allow Alice to query periodically and retrieve a list of all the times her key has changed and the resulting values. This is precisely the VKD.KeyHistory interface which Alice users to monitor her own key.

The server applies updates from its users (of type 1 and 2 described above) in batches, and publishes a commitment to the current state of the database com and proof Π^{Upd} that a valid update has been performed (VKD.Publish). The batch updates should happen at sufficiently frequent intervals of time, so that the user's keys are not out-of-date for long. The exact interval between these time intervals, or epochs has to be chosen as a system parameter. We use time and epoch interchangeably in our descriptions. The server also publishes a public datastructure which maintains information about all the commitments so far.

The auditors in the system keep checking the update proofs and the public datastructure in order to ensure global consistency (VKD.Audit) of Dir. Our definition captures a general notion of audit where independent auditors can audit arbitrary intervals $[t_1, t_n]$. The audit need not be monolithic and can consist of many independent audit steps. As long as some honest auditor executes each audit step, soundness will be guaranteed.

The server also produces proofs for VKD.Query and VKD.KeyHistory. At a very high level, the users verify the proofs (VKD.QueryVer, VKD.HistoryVer) to ensure that the server is not returning an incorrect key corresponding to a username or an inconsistent key history for the keys corresponding to a username. VKD also requires the proofs to be privacy-preserving, i.e., the proofs should not leak information about any other key (that has not been queried) in Dir. The auditors may not be trusted and hence, the proofs that the server produces as part of VKD.Publish need to be privacy-preserving. Since the auditors do not have to be trusted with any private information, anyone can become an auditor.

Definition 1. A Verifiable Key Directory is comprised of the algorithms (VKD.Publish,VKD.Query, VKD.QueryVer,VKD.KeyHistory,VKD.HistoryVer,VKD.Audit) and all the algorithms have access to the system parameters. We do not make the system parameters explicit. The algorithms are described below.

Periodic Publish:

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

This algorithm takes the previous state of the server and the key directory at previous epoch $t-1$ and also a set S_t of elements to be updated. Whenever a client submits a request to add a new label or update an existing label from epochs $t-1$ to t, the corresponding $(\mathsf{label}, \mathsf{val})$ pair is added to S_t to be added in the VKD at epoch t. The algorithm produces a commitment to the current state of the directory com_t and a proof of valid update Π^{Upd} all of which it broadcasts at epoch t. It also outputs the updated directory Dir_t and an updated internal state st_t. If this is the first epoch, i.e., no previous epoch exists, then the server initializes an empty directory and its internal state first, then updates them as described above. We will denote $(\mathsf{com}_t, \Pi^{\mathsf{Upd}}_t)$ as pub_t.

Querying for a Label:

▹ $(\mathsf{val}, \pi) \leftarrow \mathsf{VKD.Query}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: This algorithm takes the current state of the server for epoch t, the directory Dir_t at that epoch and a query label label and returns the corresponding value if it is present in the current directory, $\perp$ if it is not present, a proof of membership or non-membership respectively.

▹ $1/0 \leftarrow \mathsf{VKD.QueryVer}(\mathsf{com}, \mathsf{label}, (\mathsf{val}, \pi))$:
This algorithm takes a commitment with respect to some epoch, a label, value pair and verifies the above proof.

Checking Consistency of Key Updates:

▹ $(\{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{VKD.KeyHistory}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: This algorithm takes in the server state, the directory at current time t and a label. It outputs $\{(\mathsf{val}_i, t_i)\}_{i=1}^n$ which are all the times at which the value corresponding to label was updated so far, the resulting val's, along with a proof Π^{Ver}.

▹ $1/0 \leftarrow \mathsf{VKD.HistoryVer}(\mathsf{com}_t, \mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}})$: This algorithm takes the commitment published by the server for the current time t, a label, and $\{(\mathsf{val}_i, t_i)\}_{i=1}^n$, and verifies the above proof.

Auditing the VKD:

▹ $1/0 \leftarrow \mathsf{VKD.Audit}(t_1, t_n, \{\mathsf{pub}_t\}_{t=t_1}^{t_n})$: This algorithm takes the epochs t_1 and t_n between which audit is being done, the server's published pub for all the epochs from times t_1 to t_n. It outputs a boolean indicating whether the audit is successful.

Note that for auditing we assume that all auditors see consistent versions of published commitments and the users see the same commitments as the auditors. That is, everyone sees the same broadcast values com_t at any epoch t. This can be enforced in different ways. For a discussion on the different implementation mechanisms, please see the *assumptions on the system* in Section A.

Now we discuss the security properties we require from a VKD. We give the informal descriptions here and defer the formal definitions to Appendix C. The security properties are the following.

- **Completeness:** We want to say that if a VKD is set up properly and if the server behaves honestly at all epochs, then all the following things should happen for any label updated at $t_1, \ldots, t_n$ with $\mathsf{val}_1, \ldots, \mathsf{val}_n$: 1) their history proof with $\{(\mathsf{val}_i, t_i)\}_{i=1}^n$ and Π^{Ver} should verify at t_n 2) the query proof for the label at any $t_j \le t^* < t_{j+1}$ should verify with respect to the value consistent with the versions proof at t_j which is val_j and 3) the audit from epochs t_1 to t_n should verify.

Note that for KeyHistory and HistoryVer, we consider epochs $t_1, t_2, \ldots, t_n$ when the updates have happened for a label. These will be epochs distributed in the range $[t_1, t_n]$. However for Audit, we consider all possible pairwise epochs between t_1 and t_n. For example, for $t_1 = 3$ to $t_n = 10$, there might be updates at 3, 5, 8, 10 but for audit we need to consider all of the epochs 3, 4, 5, 6, 7, 8, 9, 10.

- **Soundness:** VKD soundness guarantees that if Alice has verified the update history of her key till time t_n and if there exists at least one honest auditor whose audits have been successful from the beginning of time till time t_n then, whenever Bob queried before t_n, he would have received Alice's key value that is consistent with the key value reported in Alice's history query. Thus soundness is derived from all of VKD.Publish, VKD.QueryVer, VKD.HistoryVer and VKD.Audit.

Note that the onus is on the user, Alice, to make sure that the server is giving out the most recent and *correct* value for her key. Soundness guarantees that under the circumstances described above, Bob will always see a key consistent with what Alice has audited. But Alice needs to verify that her key as reported in the history query is consistent with the actual key that she chose.

- **Privacy:** The privacy guarantee of a VKD system is that the outputs of Query, HistoryVer or Audit should not reveal anything beyond the answer and a well defined leakage function on the state of the directory. In other words, the proofs for each of these queries should be simulatable given the output of the leakage function and the query answer.

3 APPEND-ONLY ZERO KNOWLEDGE SET (AZKS)

In this section we introduce a new primitive, *Append-Only Zero Knowledge Set* (aZKS) which we will use to build SEEMless. Zero Knowledge Set [6, 22] (ZKS) is a primitive that lets a (potentially malicious) prover commit to a static collection 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 collection 2) the prover can prove statements about membership/non-membership of labels (from the domain of the labels) in the committed collection with respect to the succinct commitment 3) the proofs are efficient and do not leak any information about the rest of the committed collection. Our primitive, *Append-Only Zero Knowledge Set* (aZKS) generalizes the traditional zero-knowledge set primitive by accounting for append-only updates and characterizing the collection with a leakage function.

Here it is worth pointing out that the notion of soundness one would expect from updates in a ZKS is not obvious. For example, if the expectation is that updates leak absolutely no information about the underlying sets or type of updates (inserts/deletes), then there is no reasonable definition of soundness of updates: any set the prover chooses will be the result of some valid set of updates. In [20], Liskov did not define any soundness notion for updates. In our context, we want to be able to define an append-only ZKS, which makes the expectation of update soundness clear: it should ensure for any label, its value never gets modified and in particular, it never gets deleted.

Here we describe the primitive and informally define its security properties. The formal security definition is in Appendix G.

Definition 2. Append-Only Zero Knowledge Set is comprised of the algorithms (ZKS.CommitDS, ZKS.Query, ZKS.Verify, ZKS.UpdateDS, ZKS.VerifyUpd)[‡] described as follows:

▹ $(\mathsf{com}, \mathsf{st}_{\mathsf{com}}) \leftarrow \mathsf{ZKS.CommitDS}(1^\lambda, \mathsf{D})$: This algorithm takes the security parameter and the datastore to commit to as input, and produces a commitment to the data store and an internal state to pass on to the Query algorithm. Datastore D will be a collection of (label, val) pairs.

▹ $(\pi, \mathsf{val}) \leftarrow \mathsf{ZKS.Query}(\mathsf{st}_{\mathsf{com}}, \mathsf{D}, \mathsf{label})$: This algorithm takes the state output by ZKS.CommitDS, the datastore and a query label and returns its value (⊥ if not present) and a proof of membership/non-membership.

▹ $1/0 \leftarrow \mathsf{ZKS.Verify}(\mathsf{com}, \mathsf{label}, \mathsf{val}, \pi)$: This algorithm takes a (label, value) pair, its proof and a commitment by ZKS.CommitDS and verifies the above proof.

▹ $(\mathsf{com}', \mathsf{st}'_{\mathsf{com}}, \mathsf{D}', \pi_S) \leftarrow \mathsf{ZKS.UpdateDS}(\mathsf{st}_{\mathsf{com}}, \mathsf{D}, S)$: This algorithm takes in the current server state $\mathsf{st}_{\mathsf{com}}$, the current state of the datastore and a set $S = \{(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_k, \mathsf{val}_k)\}$ of new (label, value) pairs for update. It outputs an updated commitment to the datastore, an updated internal state and an updated version of the datastore and proof π_S that the update has been done correctly.

▹ $0/1 \leftarrow \mathsf{ZKS.VerifyUpd}(\mathsf{com}, \mathsf{com}', \pi_S)$: This algorithm takes in two commitments to the datastore before and after an update and verifies the above proof.

We require the following security properties of an *append-only* ZKS:
Soundness: For soundness we want to capture two things: First, a malicious prover $\mathcal{A}^*$ algorithm should not be able to produce

[‡]The original ZKS definition also included a setup algorithm run by a trusted party to generate public parameters used in all the algorithms. Our construction does not need such a set up (we show security in the random oracle model), so we omit it here.

two verifying proofs for two different values for the same label with respect to a com. Second, since the aZKS is append-only, a malicious server should not be able to change or delete an existing label.

Zero-Knowledge with Leakage: We generalize the definition ZKS [6, 22] by introducing leakage functions in the classical definition. The goal of our privacy definition is to capture the following: we want the query proofs and update proofs to leak no information beyond the query answer (which is a val/$\perp$ in case of query and a bit indicating validity of the update operation). But often, it is reasonable to tolerate a small leakage to gain more efficiency. To capture this sort of leakage formally, we parameterize our definition with a leakage function. If the leakage function is set to null, then our definition reduces to the classical ZKS definition.

Append-Only Strong Accumulator: Finally, we remark that the primitive *strong accumulator* (SA) [5] can be extended to a new primitive of *append-only strong accumulator* (aSA) trivially from the definition of aZKS. An aSA is essentially a aZKS with completeness and soundness and *without* the privacy requirement. In Section 5, we will first construct an efficient aSA and then construct aZKS using aSA in a blackbox way.

4 SEEMLESS CONSTRUCTION

In this section, we will describe SEEMless: our construction of VKD from *append-only* zero-knowledge sets (aZKS). We first give an informal overview of the construction and then describe the construction more formally.

System Setup: The high level idea is to have two aZKS that are updated every epoch: one is what we call the "all" aZKS which consists of the entries for all the updates of a label so far. Here label is the username and the value is the user's public key. The other aZKS that we will refer to as "old" aZKS has entries with all the versions of the label except the latest, versions that are now old. The purpose of maintaining two aZKS is that when a user Alice updates her key, instead of deleting the old entry, it will be added to the "old" aZKS. This means we can use an append operation to capture key updates, while relying on the append-only property to guarantee that the server maintains an accurate record of past updates. In both of these aZKS, the server stores each (label, value) pair along with its version number, so when an item is initially added, we add (label| 1, val) indicating that this label is on version 1. The "all" aZKS will also include *marker entries i* for version 2^i of each label. Intuitively, the markers will help limit the checks that Alice needs to make when verifying her key history. It will help her make sure that the server is not responding to queries with version numbers much higher than the correct version. The server also stores an internal table T with a list for each label of all of the epochs when it has been updated. The server also maintains a hash chain with each pair of "old" and "all" aZKS commitments. We refer to the head of this hashchain at time t as com_t.

Remark 1. *We describe our construction with a hash chain to avoid confusion with the Merkle tree used in our aSA construction in Section 5.1. However, we note that this could instead be replaced with a Merkle tree built over the list of all commitments to date. This would result in slightly higher update and audit costs (adding a new entry to the end of this list would require up to a logarithmic number of hashes), but would significantly reduce the cost of history queries (from linear in the number of epochs to logarithmic). We discuss this in Section 7 (Update, Gethistory and Audit experiments).*

Periodic Publish: 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 (label | α, val) to the "all" aZKS and also adds (label | $\alpha - 1$, $\mathsf{val_{old}}$) to the "old" aZKS for the label's previous value $\mathsf{val_{old}}$ if it exists i.e. for $\alpha > 1$. If the new version $\alpha = 2^i$ for some i, then the server adds a marker entry (label | mark | 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. However, the auditors can get the update proofs from the server by explicitly querying it, i.e., the server *need not broadcast* them along with the new hashchain head com_t. We don't require any non-tamperability or privacy guarantees of the distribution mechanism of the update proofs.

Querying for a Label Value: 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 (label | α, val) in the "all" aZKS. Second is the membership proof of the most recent marker entry (label |mark |a) for $\alpha \geq 2^a$. And third is non membership proof of label | α 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.

Querying for the History of Key Updates: Now we are explicitly maintaining versions in the construction and hence one label will have multiple entries associated with it depending on the number of times it was updated. Perhaps a malicious server can give out an older version of the label or it could create an entry corresponding to a later version on its own and give that whenever someone queries. In order to prevent these behaviors we incorporate the some checks in the key history queries. The purpose of each of these checks are described in the algorithm description.

Auditing: 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. For security, it is sufficient to have at least one honest auditor perform audits over each adjacent pair.

Privacy: The main insight for the gain in privacy is that in our construction we always add new entries since we are using aZKS, we never delete or update existing entries. This hides which entries are being updated and hides the lifespan of each key. However, our construction does leak some benign information that we make explicit in each algorithm description (Query, History and Audit).

VKD CONSTRUCTION: In the construction we assume that the server's identity and public key is known to each user and auditor

Notation	Description
T	A table containing usernames (labels) and all epochs at which their values were updated until the current epochs.
$D_{\mathsf{all},t}$	A datastore containing all the label-value pairs until epoch t.
$D_{\mathsf{old},t}$	A datastore containing all the *stale* label-value pairs until epoch t.
Dir_t	The ordered pair $(D_{\mathsf{all},t}, D_{\mathsf{old},t})$.
$\mathsf{st}_{\mathsf{all},t}$	The internal state of the aZKS built on $D_{\mathsf{all},t}$.
$\mathsf{st}_{\mathsf{old},t}$	The internal state of the aZKS built on $D_{\mathsf{old},t}$.
st_t	The internal state of the VKD at epoch t.
S_t	The set of elements (updates to existing usernames and new registrations) added between epochs $t-1$ and t.
α_i	The version number of the value corresponding to label_i. Sometimes written as $\alpha_{\mathsf{label}_i}$

Table 1: Notation for our VKD construction.

T	$D_{\mathsf{all},t-1}$		$D_{\mathsf{old},t-1}$
(bob, $\{10, 100\}$)	(bob|1, $\mathsf{PK}_{\mathsf{b},1}$) (bob|2, $\mathsf{PK}_{\mathsf{b},2}$)	(bob|mark|0, . . .) (bob|mark|1, . . .)	(bob|1, null)

Entries at epoch t − 1

S_t (requested updates)	M_t	S'_t (for $D_{all,t}$)	S_t^{old} (updates to existing labels)	$S_t^{\mathsf{old}'}$ (for $D_{old,t}$)
(bob, $\mathsf{PK}_{\mathsf{b},3}$) (alice,$\mathsf{PK}_{\mathsf{a},1}$)	alice|mark|0	(bob|3, $\mathsf{PK}_{\mathsf{b},3}$) (alice|1, $\mathsf{PK}_{\mathsf{a},1}$)	(bob,$\mathsf{PK}_{\mathsf{b},3}$)	(bob|2, null)

Sets for new entries to be added at epoch t

T	$D_{\mathsf{all},t}$		$D_{\mathsf{old},t}$
(bob, $\{10, 100, t\}$) (alice, $\{t\}$)	(bob|1, $\mathsf{PK}_{\mathsf{b},1}$) (bob|2, $\mathsf{PK}_{\mathsf{b},2}$) (bob|3, $\mathsf{PK}_{\mathsf{b},3}$) (alice|1, $\mathsf{PK}_{\mathsf{a},1}$)	(bob|mark|0,. . .) (bob|mark|1,. . .) (alice|mark|0, …)	(bob|1, null) (bob|2, null)

Entries at epoch t

Table 2: An example with two users, with views of data structures at epochs $t-1$ and t.

and all the messages from the server are signed under the server's key, so that the server cannot be impersonated.

Along with the steps of our construction, we will provide a running example for expositional clarity.

> Consider a VKD with 2 chat client users, in which the labels are usernames and the values are the corresponding public keys. Suppose at some point in time between server epochs $t-1$ and t, alice requested registration with her first ever public key $\mathsf{PK}_{\mathsf{a},1}$ and an existing user bob requested to update his public key for the 2nd time to $\mathsf{PK}_{\mathsf{b},3}$. These values will reflect in the VKD at epoch t. Previously, bob registered his first key $\mathsf{PK}_{\mathsf{bob},1}$ at server epoch 10 and updated it to $\mathsf{PK}_{\mathsf{bob},2}$ at server epoch 100.

▹ $\mathsf{VKD.Publish}(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)$:
$\underline{t = 0}$: This is when we are setting up the directory for the first time and let $\mathsf{Dir}_0 = (D_{\mathsf{all},0}, D_{\mathsf{old},0})$ as in Table 1 with the $D_{\mathsf{all},0}, D_{\mathsf{old},0}$ are initialized to empty. Compute $(\mathsf{com}_{\mathsf{all},0}, \mathsf{st}_{\mathsf{all},0}) \leftarrow \mathsf{ZKS.CommitDS}(\mathsf{st}_{\mathsf{all}}, D_{\mathsf{all},0})$ and $(\mathsf{com}_{\mathsf{old},0}, \mathsf{st}_{\mathsf{old},0}) \leftarrow \mathsf{ZKS.CommitDS}(\mathsf{st}_{\mathsf{old}}, D_{\mathsf{old},0})$. Set the hash chain head to $\mathsf{com}_0 = H(H(\mathsf{com}_{\mathsf{all},0}, \mathsf{com}_{\mathsf{old},0}), 0)$. Also, initialize T as in Table 1.
Output com_0 , $\mathsf{st}_0 = (\mathsf{st}_{\mathsf{all},0}, \mathsf{st}_{\mathsf{old},0}, T, \mathsf{Dir}_0)$.
$\underline{t > 0}$: Let $S_t = \{(\mathsf{label}_1, v_1), \ldots, (\mathsf{label}_k, v_k)\}$ be the (label, value) pairs to be added to the VKD at epoch t. These could be new additions or updates to existing labels.

- For each $\mathsf{label}_i \in S_t$, such that label_i is present in T, append t to the list of epochs corresponding to label_i. For a new label_i, add $(\mathsf{label}_i, \{t\})$ to T. Let β_i be the number of entries in T for label_i (not including the newest entry, t). $\beta_i = 0$ if label_i has no previous entries associated with it. $\alpha_i = \beta_i + 1$ is the version number of the new key. In our example, since the label alice is being added to the VKD for the first time, $\alpha_{\mathtt{alice}}$ would be 1. For bob a third value is being added, so α_{bob} would be 3. Based on the version number, we will create the (label, value) pairs to be added in the "all" aZKS.
- If for any label_i, $\alpha_i = 2^y$ for some $y \geq 0$ add the following (label, value) pair to the marker set M_t: $(\mathsf{label}_i | \text{ mark } | y, \text{“marker entry } 2^y \text{ for } \mathsf{label}_i\text{”})$.
- Compute new update set to update $D_{\mathsf{all},t-1}$ to $D_{\mathsf{all},t}$: $S'_t = \{(\mathsf{label}'_i, \mathsf{val}_i) \mid (\mathsf{label}_i, \mathsf{val}_i) \in S_t \wedge \mathsf{label}'_i = \mathsf{label} | \alpha_i\}$. See Table 2 for concrete examples. Compute the update on the "all" aZKS for the set $S'_t \cup M_t$: $(\mathsf{com}_{\mathsf{all},t}, \mathsf{st}_{\mathsf{all},t}, D_{\mathsf{all},t}, \pi_{S_t}) \leftarrow \mathsf{ZKS.UpdateDS}(\mathsf{st}_{\mathsf{all},t-1}, D_{\mathsf{all},t-1}, S'_t \cup M_t)$.
- Form a new set of (label, value) pairs to be added to the "old" aZKS. Let S_t^{old} be the list of entries in S_t that are updates to existing labels. For each $\mathsf{label}_i \in S_t^{\mathsf{old}}$, concatenate it with its version $\alpha_i - 1$ before the update. Hence, let $S_t^{\mathsf{old}'} = \{(\mathsf{label}'_i, \mathsf{null}) | (\mathsf{label}_i, \mathsf{val}_i) \in S_t^{\mathsf{old}} \wedge \mathsf{label}'_i = \mathsf{label}_i | \alpha_i - 1\}$. For the "old" tree, compute $(\mathsf{com}_{\mathsf{old},t}, \mathsf{st}_{\mathsf{old},t}, D_{\mathsf{old},t}, \pi_{S_t^{\mathsf{old}}}) \leftarrow \mathsf{ZKS.UpdateDS}(\mathsf{st}_{\mathsf{old},t-1}, D_{\mathsf{old},t-1}, S_t^{\mathsf{old}'})$. See examples of these sets in Table 2.
- Update the hash chain: $\mathsf{com}_t = H(H(\mathsf{com}_{\mathsf{all},t}, \mathsf{com}_{\mathsf{old},t}), \mathsf{com}_{t-1})$.

Output com_t, $\mathsf{st}_t = (\mathsf{st}_{\mathsf{all},t}, \mathsf{st}_{\mathsf{old},t}, T, \mathsf{Dir}_t)$ and $\Pi_t^{\mathsf{Upd}} = (\pi_{S_t}, \pi_{S_t^{\mathsf{old}}}, \mathsf{com}_{\mathsf{all},t}, \mathsf{com}_{\mathsf{old},t}, \mathsf{com}_{\mathsf{all},t-1}, \mathsf{com}_{\mathsf{old},t-1}, \mathsf{com}_{t-2})$.

▹ $\mathsf{VKD.Query}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: Retrieve latest version number α for queried label from table T (by counting the number of epoch entries for label). Let β be the largest power of 2 less than α such that $\beta = 2^b$. Compute the following proofs:

- $(\pi_1, \mathsf{val}_1) \leftarrow \mathsf{ZKS.Query}(\mathsf{st}_{\mathsf{all},t}, D_{\mathsf{all},t}, \mathsf{label} | \alpha)$: This gives a proof of membership of the latest version of label in the "all" aZKS and its corresponding value.
- $(\pi_2, \mathsf{val}_2) \leftarrow \mathsf{ZKS.Query}(\mathsf{st}_{\mathsf{all},t}, D_{\mathsf{all},t}, \mathsf{label} | \text{ mark } | b)$: This gives a proof of membership of the marker entry right before the current version α.
- $(\pi_3, \mathsf{val}_3) \leftarrow \mathsf{ZKS.Query}(\mathsf{st}_{\mathsf{old},t}, D_{\mathsf{old},t}, \mathsf{label} | \alpha)$: This gives a proof of non membership of the latest version in the "old" aZKS making sure that the claimed "latest" version is not outdated.

Output $\Pi = (\pi_1, \pi_2, \pi_3, \mathsf{com}_{\mathsf{all},t}, \mathsf{com}_{\mathsf{old},t}, \mathsf{com}_{t-1}))$ and $\mathsf{val} = (\mathsf{val}_1, \mathsf{val}_2, \bot)$ and α.

In our example, if alice requested to see bob's public key at epoch t, the server would count the length of the list corresponding to bob in T. Then, alice would receive proofs for $\text{bob}|3 \in \text{Dir}_{\text{all},t}$ with value $\text{PK}_{\text{b},3}$ and $\text{bob}|\text{mark}|1 \in \text{Dir}_{\text{all},t}$ and lastly, $\text{bob}|3 \notin \text{Dir}_{\text{old},t}$. Additionally, alice will receive $\text{com}_{\text{all},t}, \text{com}_{\text{old},t}, \text{com}_{t-1}$.

▹ $\text{VKD.QueryVer}(\text{com}_t, \text{label}, \text{val}_t, \pi_t, \alpha)$: 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.

▹ $\text{VKD.KeyHistory}(\text{st}_t, \text{Dir}_t, \text{label})$: The server first retrieves all the update epochs $t_1, \ldots, t_\alpha$ for label versions $1, \ldots, \alpha$ from T, the corresponding $\text{com}_{\text{all},t_1-1}, \text{com}_{\text{all},t_1}, \ldots, \text{com}_{\text{all},t_\alpha-1}, \text{com}_{\text{all},t_\alpha}$ and $\text{com}_{\text{old},t_1}, \ldots, \text{com}_{\text{old},t_\alpha}$ and the hashes necessary to verify the hash chain: $H(\text{com}_{\text{all},0}, \text{com}_{\text{old},0}), \ldots, H(\text{com}_{\text{all},t}, \text{com}_{\text{old},t})$. 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 tis 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: $H(\text{com}_{\text{all},0}, \text{com}_{\text{old},0}), \ldots, H(\text{com}_{\text{all},t}, \text{com}_{\text{old},t})$.
(2) **Correct version** i **is set at epoch** t_i**:** For each i: Membership proof for (label| 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 (label| $i-1$).
(4) **Server couldn't have shown version** i **before epoch** t_i: For each i: Non membership proof for (label| 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 (label| $i+1$), (label| $i+2$), $\ldots$, (label| $2^{a+1}-1$).
(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 (label| mark| $a+1$) up to (label| mark| $\log t$).

▹ $\text{VKD.HistoryVer}(\text{com}_t, \text{label}, \{(\text{val}_i, t_i)\}_{i=1}^n, \Pi^{\text{Ver}})$: Verify each of the above proofs.

In our example, if bob queried for his key history at epoch t, he would check the following,

(1) $\text{com}_{10}, \text{com}_9, \text{com}_{100}, \text{com}_{99}, \text{com}_t$ and com_{t-1} and the hashes necessary to verify the the hashchain $H(\text{com}_{\text{all},0}, \text{com}_{\text{old},0}), \ldots, H(\text{com}_{\text{all},t}, \text{com}_{\text{old},t})$.
(2) bob|1 exists and has value $\text{PK}_{\text{b},1}$ in $\text{Dir}_{\text{all},10}$, bob|2 exists and has value $\text{PK}_{\text{b},2}$ in $\text{Dir}_{\text{all},100}$ and bob|3 exists and has value $\text{PK}_{\text{b},3}$ in $\text{Dir}_{\text{all},t}$.
(3) $\text{bob}|1 \in \text{Dir}_{\text{old},100}$ and $\text{bob}|2 \in \text{Dir}_{\text{old},t}$.
(4) $\text{bob}|1 \notin \text{Dir}_{\text{all},9}$, $\text{bob}|2 \notin \text{Dir}_{\text{all},99}$ and $\text{bob}|3 \notin \text{Dir}_{\text{all},t-1}$.
(5) Since bob's version is $3 < 2^2 = 4$, nothing to check here.
(6) $\text{bob}|\text{mark}|2 \ldots, \text{bob}|\text{mark}|\log t \notin \text{Dir}_{\text{all},t}$

Remark 2. *The client software runs* VKD.HistoryVer *to monitor the history of the user keys. This software either downloads the entire proof from the server each epoch it runs* VKD.HistoryVer *(when the software is re-installed or the user installs it on a new device) or cache parts of the proof to from the first run of* VKD.HistoryVer *to use in the subsequent verifications. We experimentally evaluate the performance for* VKD.HistoryVer *with and without caching in Section 7.*

▹ $\text{VKD.Audit}(t_1, t_n, \{\text{pub}_t\}_{t=t_1}^{t_n})$: Recall $\text{pub}_t = (\text{com}_t, \Pi_t^{\text{Upd}})$ and parse each $\Pi_t^{\text{Upd}} = (\pi_{S_t}, \pi_{S_t^{\text{old}}}, \text{com}_{\text{all},t}, \text{com}_{\text{old},t}, \text{com}_{\text{all},t-1}, \text{com}_{\text{old},t-1}, \text{com}_{t-2})$. For each t, verify that $\text{com}_t = H(H(\text{com}_{\text{all},t}, \text{com}_{\text{old},t}), \text{com}_{t-1})$. Do the checks $\text{ZKS.VerifyUpd}(\text{com}_{\text{all},t}, \text{com}_{\text{all},t-1}, \Pi_{\text{all},t}^{\text{Upd}})$ and $\text{ZKS.VerifyUpd}(\text{com}_{\text{old},t}, \text{com}_{\text{old},t-1}, \Pi_{\text{old},t}^{\text{Upd}})$ for all consecutive epochs from t_1 to t_n.

Remark 3. *In our implementation, the marker entry doubles up as the normal key entry for that version, where the version number is* $\log(k)$, k *being the count of number of updates for a specific user. We use a special symbol* mark *for the marker entries. For example, the* 3*rd key for* bob *will be saved as* bob|3 *and the* 1*st key for* alice *will be saved as* alice|mark|0, *(since* $2^0 = 1$*), hence only two new entries are made for any user update request.*

Leakage. A party who only acts as an auditor learns only the numbers of keys added and keys updated each epoch. If that party additionallly acts as a user (Alice) performing KeyHistory queries, the combined leakage may reveal when her keys are updated (even if she does not perform more KeyHistory queries), but that is expected to be something Alice knows since she is the one requesting the updates. If Alice additionally queries for Bob's key, the leakage reveals the version number of Bob's current key and the epoch when it was last updated, and may reveal when that key is no longer valid (because Bob performed an update), but will not reveal anything about subsequent or previous updates. For a more detailed specification, see Appendix C.

Discussion We discuss some of the design considerations and assumptions that SEEMless makes about the users and the deployment infrastructure in Appendix A. We also discuss how we envision our system to be integrated on top of current E2E apps to provide a seamless user experience. Here we give a brief summary of the discussion.

Having a smooth user-experience while interacting with the functionality of SEEMless is crucial for successful adoption of this system. We envision a user interface where the client software would periodically run KeyHistory in the background and only notify a user if the verification fails.

In designing SEEMless, we make the following assumptions: 1) a user device can store a cryptographic key and the server has a non-cryptographic means of authenticating its users 2) the participating parties have approximately synchronized clocks 3) the infrastructure provides a mechanism that ensures all parties have consistent views of the root commitments published by the server.

Finally, even though we present SEEMless as a single *logical* server, it can be implemented using a distributed network of servers. Our system also supports multiple devices per user with keys for each device being stored on the server.

5 aZKS INSTANTIATIONS

In this section we will give a concrete instantiation for the aZKS used for SEEMless. We refer the reader to Appendix B for definitions of the standard cryptographic primitives used in the construction of the aZKS.

5.1 Append-Only Strong Accumulator (aSA) Construction

Here, we give a construction of an *append-only* strong accumulator over a data collection of (label,value) pairs. The high level idea is to build a Patricia Trie (PTr) [18] over the labels. PTr is a succinct representation of the labels such that each child has a unique suffix string associated with it and the leaf nodes constitute the actual label values. See Fig 1 for an illustrative example.Our aSA construction is built on a PTr. We use a collision-resistant hash function $H : \{0,1\}^* \mapsto \{0,1\}^m$ in our construction.

▹ SA.CommitDS(1^λ, D) : Datastore D $= \{(l_1, v_1), \ldots, (l_n, v_n)\}$, a collection of label-value pairs. Choose a constant $k_D \xleftarrow{\$} \{0,1\}^\lambda$. Let $\{y_1, \ldots, y_n\}$ be a lexicographic ordering corresponding to $\{l_1, \ldots, l_n\}$. Build a Patricia trie on $\{y_i\}$ and output com $= (h_{\text{root}}, k_D)$. For the nodes in the tree, hashes are computed as follows.

Leaf nodes: For a node y_i, compute: $h_{y_i} = H(k_D|y_i|v_i)$. For example, in the tree in Figure 1, $h_2 = H(k_D|0100|v_2)$.

Interior nodes: For an interior node x, let $x.s_0$ and $x.s_1$ be the labels of its children. Compute: $h_x = H(k_D|x|h_{x.s_0}|h_{x.s_1}|s_0|s_1)$. For example in Figure 1, $h_6 = H(k_D|0|h_1|h_7|010|1)$

▹ SA.Query(D, l) : If $l \in$ D, output value v associated to it. Let h_l be the hash value of the node. Give the sibling path for h_l in the Patricia trie along with the common prefix x at each sibling node and the suffixes s_0, s_1 which form the nodes $x.s_0$ and $x.s_1$. For example, proof for 0100 will be its value and $[h_2, (h_3, 01, 00, 11), (h_1, 0, 010, 1), (h_8, \epsilon, 0, 1)]$

If $l \notin$ D, let z be the longest prefix of l such that z is a node in the Patricia tree. Let zu_0, zu_1 be its children. Output $z, h_z, u_0, u_1, h_{zu_0}, h_{zu_1}$ along with sibling path of z. For example, proof for 1010 will be $\perp$ and $[1, 1000, 1100, h_8, h_4, h_5, (h_6, \epsilon, 0, 1)]$.

▹ SA.Verify(com, l, v, π) : Parse π as the hash values and the auxiliary prefix information at each node. Compute h_l according to leaf node calculation and verify the given value. Compute the hash values upto the root with help of the proof Let this hash value be h. Verify that $h = h_{\text{root}}$. In case of a non-membership proof, additional verification is required. Let z_l and z_r be the labels of the left and right child (respectively) of the returned node z. Verify that 1) z is the longest common prefix of z_l and z_r 2) z_l and z_r are distinct.

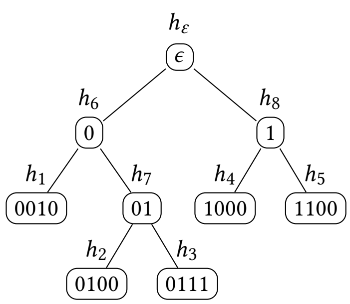

Figure 1: For the universe of be all 4 bit binary strings, a Patricia trie built on subset $P = \{0010, 0100, 0111, 1000, 1100\}$

▹ SA.UpdateDS(D, S): First we check that S is a valid set of updates which means that for all $(\text{label}_i, \text{val}_i) \in S$, $\text{label}_i \notin$ D. Initialize sets $Z_{\text{new}}, Z_{\text{old}}, Z_{\text{const}}$ to empty. For all $\text{label}_j \in S$, compute: $h_{\text{label}_j} = H(k_D|l_j|\text{val}_j)$ and add h_{label_j} to the appropriate position in the trie and change the appropriate hash values. Add the old hash values to Z_{old} and the updated to Z_{new} and those that remain unchanged after all updates to Z_{const}. Output the final updated root hash h'_{root} as com′, output the final st′ and D′ $=$ D$\cup\{(\text{label}_j, \text{val}_j)\}$ and output $\pi_S = (Z_{\text{old}}, Z_{\text{new}}, Z_{\text{const}})$ and set S.

▹ SA.VerifyUpd(com, com′, S, π_S): Parse $\pi_S = (Z_{\text{old}}, Z_{\text{new}}, Z_{\text{const}})$. Compute the root hash from all the values in Z_{old} and Z_{const} and check that it equals com. Similarly, compute the root hash from all the values in Z_{new} and Z_{const} and check that it equals com′. Let Z_S be the set of roots of the subtrees formed by labels in S. Check that $Z_{\text{old}}, Z_{\text{new}}$ are exactly the hash values of all nodes in Z_S. Output 1 if all checks go through.

Security This aSA construction is secure if H is a collision-resistant hash function. The security of this primitive is proven in [24].

5.2 aZKS Instantiations

Our construction builds upon append-only strong accumulators which directly give us completeness and soundness. To get zero-knowledge we will use a *simulatable Verifiable Random Function* (hitherto denoted as an sVRF) and a *simulatable Commitment Scheme* (hitherto denoted as an sCS). For definitions of these primitives, see Appendix B. At a high level, the sVRF helps map an element in the datastructure to a random position and the commitment helps in hiding the exact value that is being accumulated.

Let (sVRF.KeyGen, sVRF.Eval, sVRF.Prove, sVRF.Verify) be a simulatable VRF and let (CS.Commit, CS.Open, CS.VerifyOpen) be a simulatable commitment scheme as described before. Given D $= \{(\text{label}_1, \text{val}_1), \ldots, (\text{label}_n, \text{val}_n)\}$, ZKS.Commit will first generate an sVRF key pair SK, PK $\leftarrow$ sVRF.KeyGen(1^λ), build a new D′ $= \{(l_1, v_1), \ldots, (l_n, v_n)\}$ where $l_i =$ sVRF.Eval(SK, label_i), $v_i =$ CS.Commit(($\text{label}_i, \text{val}_i$) ; r_i) for random r_i. It will then build an append-only SA on D′ and output that together with PK. ZKS.Query will return the appropriate l_i along with the sVRF proof, the opening to the commitment, and the strong accumulator membership/non-membership proof for (l_i, v_i). ZKS.UpdateDS will compute the l_i, v_i pairs for the new datastore entries and then run SA.UpdateDS. The formal proof of security is in Appendix G.

Leakage In our aZKS construction, ZKS.Query leaks the size of the datastore, and when the queried element was added (assuming it is a member). ZKS.UpdateDS leaks the size of the datastore before and after an update. For each element that was added, it also leaks whether and when the adversary previously queried for that element.

6 PERSISTENT PATRICIA TRIE

In our construction, SEEMless, the server maintains two aZKS at every epoch. In Section 4, we described how to construct a aZKS using a aSA, which in turn, we implemented using Patricia Trie.

Recall that, in SEEMless, the server needs to store the entire history of the key directory and the two corresponding aSA (used in the aZKS construction) to be able to answer key history queries. Naively, the server can maintain all the aSA from the beginning of time, but this will blow up the server's storage significantly and hinder scalability of the system. This is particularly wasteful, when in fact, there might be a significant overlap between the aSA of consecutive epochs. To address this problem, we build a persistent data structure that retains information about every epoch, while being space efficient. We call this persistent data structure *Persistent Patricia Trie* (PPTr). We believe that PPTrs will find a wide number of applications (such as the tamper-evident logging in [24]).

Challenges: The idea is similar to that of history tree (HT) [24], built on Sparse Merkle Tree. However, we can not naively use the same technique to build a Persistent Patricia Trie (PPTr). Recall that Patricia Trie (PTr) is compressed for efficiency — there is no empty node in the tree. Compression introduces several subtle challenges in building its persistent version, i.e., PPTr. For example, unlike in HT [24], nodes in a Patricia Trie do not have a fixed position. A node with a certain label may be at lower depth at an earlier epoch (root being at depth 0) and fall to a higher depth at a later epoch. A node with a given label may change depth several times from its birth epoch until the latest epoch. Please see Fig 2 for an illustrative example. Therefore, a parent to child pointer does not remain fixed throughout the life time of a PTr. In the example, the node 00 pointed to 0000 as its left child at time t_2, and to node 000 at time t_4. The HT construction in [24] crucially relies on parent to children pointers being fixed throughout the lifetime of the tree. We will build our final construction of PPTr gradually.

Attempt 1: First, let us observe some invariants in a append-only PTr (aPTr): (1) The depth of a PTr is defined with respect to the root node of the tree, therefore the root node is always at depth 0 (2) Any node in a append-only PTr can only fall to higher depths over epochs, it never goes to lower depths. (3) Any node with k bit label can be at depth at most k in a PTr.

Now let us attempt to build a PPTr. Every node in a PTr is identified by its label with a binary string. For every node u, we store some information corresponding to every epoch when the hash value at node u changed. Let us denote this information as nodestate. These epochs and the corresponding nodestates are stored in a hashtable. Corresponding to every epoch t when the hash value at node u changed, $\mathsf{nodestate}_t$ stores the following.

(1) h_t: hash value of node u at epoch t
(2) leftskip: a binary string indicating how many levels have to be skipped to get the left child of u at epoch t. Let the left child label at epoch t is l
(3) leftepoch: the key to lookup l's hashtable and get the corresponding nodestate
(4) rightskip: a binary string indicating how many levels have to be skipped to get the left child of u at epoch t. Let the right child label at epoch t is r
(5) rightepoch: the key to lookup r's hashtable and get the corresponding nodestate

Since we are constructing a aPTr, we need to support only two operations efficiently on the data structure:

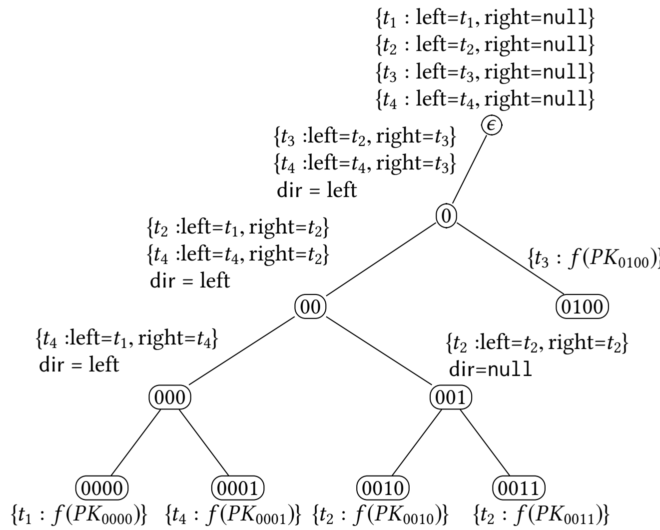

Figure 2: Append only Patricia Trie: final view at epoch t_4

Generate proof for a historical query: Recall (non-) membership proof for certain node u at epoch t consists of the siblings of the nodes on the path from the leaf (or an internal node with both children's labels non-prefixes of label u) to the root. For any previous epoch t, this information can be easily extracted traversing the tree from the root and following at each node u the (leftskip, leftepoch) and (rightskip, rightepoch) pointers from hashtable entry of epoch t. Thus, the cost of generating membership and non-membership proofs for any epoch in the past is proportional to the tree height, and is independent of the number of epochs.

Insert new node in aPTr: When a new node gets inserted to a aPT, all nodes on its path from the root gets updated. But note that, each node on this path has a hash table (of (epoch, nodestate)) associated with it. For each of these entries, the skip entries might need to get updated while updating the path. Hence, the cost of updating each node on the path is not constant, it is proportional to the size of the hashtable at that node. Consequently, the cost of inserting a new node is not proportional to the tree height.

Attempt 2: Our next attempt is to retain the cost for generating historical proofs while bringing down the cost of insertion. Notice that The skip entry at the parent of node u gets updated if and only if a new node with label v is inserted to the tree where v is a proper prefix of u. v becomes the new parent of u. We keep a direction flag with every node v that indicates whether u became the left or the right child of the newly inserted node v. Let us denote this flag as dir. Along with this, at every node we store the following additional information: birthepoch: the epoch at which the label u was inserted.

Now we do not need to keep the skip fields any more, because dir will help us get to the correct node. More specifically, for a historical query (u, t), we will go down from the root following as long as birthepoch $\geq t$ guided by the dir bit. For example, in Figure 2, at t_1, 0000 was the leftchild of the root but at time t_2 00 replaced 0000 and 0000 became the left child of 00. So the dir flag at t_2 is set to left. The birthepoch of 00 is t_2 which equals the queried epoch t_2. Now, if we query (0000, t_2), we will get to node 00 whose node state entry t_2 will have $\mathsf{leftepoch} = t_1$ and dir is set to Left. So we go down the leftchild of 00 and find the entry 0000 at epoch t_2.

Now insertion cost is proportional to the height of the tree since cost of insertion on every node on the path is constant. But we have introduced yet another problem that arises at the time of historical query. The proof needs the sibling path and we may need to go down several levels from a parent node (following dir) to get the correct sibling for the queried epoch.

Final construction: To overcome this problem, we keep at the nodestate of every node, the label and the hashes of its current children. This way, we get the required sibling directly from the parent node. Nodestate now contains the following.

(1) h_t: hash value of node u at epoch t
(2) leftlabel: the label of the left child of node u at time t. Let l denote the label
(3) lefthash: the hash value of node l at epoch t
(4) leftepoch: the key to lookup l's hashtable and get the corresponding nodestate
(5) rightlabel: the label of the right child of node u at time t. Let r denote the label
(6) righthash: the hash value of node r at epoch t
(7) rightepoch: the key to lookup r's hashtable and get the corresponding nodestate

With this new nodestate, combined with the additional information of dir, birthepoch, we have a persistent Patricia Trie data structure where both historical query and insertion works in time proportional to the height of the Trie.

7 EXPERIMENTS

7.1 Experimental Setup

Our experiments were conducted on a Linux VM running on a 2.30GHz Intel Xeon E5-2673 CPU with 125GB of memory and a 64GB heap allocated to the JVM for the server-side experiments for both SEEMless and CONIKS. All client experiments are done on a 2.8 GHz Intel Core i7 laptop. We used Google's protocol buffers [28] to implement communication between the client and the server, due to their efficiency and compression. All sizes below are computed using the `getSerializedSize()` API call for protocol buffer messages, which returns the number of bytes used to encode a message.

We implemented the VRF using the *Secp384r1* curve using the Bouncy Castle library, Icart's function [12] for hashing to the curve and SHA384. We use the technique of [11] of hashing the input twice (using SHA384(0||*input*) and SHA384(1||*input*)) and applying Icart on both these hashes, since applying Icart once does not provide a distribution indistinguishable from random.

7.2 Performance Evaluation

Server updates: The graph in Fig. 3 shows the time taken for server update in the following experiment: we increased the total users by 100k at each epoch and made 1k new user registrations and 1k key updates by existing users. For a server with 10M registered users, it takes $0.28s$ on average to update the authentication structure (the two PPTrs) and its directory. As expected, the update time is proportional to the log of total number of registrations and updates in the system. The server adds the aZKS commitments in a Merkle Tree (instead of a hashchain) as discussed in Section 4. We estimate this cost using the sparse Merkle Tree data structure used in CONIKS (which is an over-estimate). The time to insert a aZKS commitment in the Mekle Tree of aZKS commitments is $61\mu s$ after 10M server epochs have passed.

Both CONIKS and SEEMless are implemented such that the VRF for an update or registration are computed between epochs, when a client message arrives. This means, the online update phase only needs to update the authentication data structure and does not incur the VRF cost. For SEEMless, registering a new user requires a single VRF computation (for inserting into the "all" aZKS) and an update requires 2 VRF computations – one for the new key entering the "all" aZKS and the other for the old key entering the "old" aZKS. Therefore the offline VRF computation cost at the server (per client request) is at least $1.3ms$ and at most $2.5ms$, which is relatively low computation between epochs. For CONIKS, the cost is a single VRF computation (for registration, no VRF is computed for update), hence the cost is at most $1.3ms$.[§]

Scalability comparison We tried to run the same update experiment with CONIKS, on the same machine and saw that the heap filled up by the time we had around 4.5M users. Hence, we decided to run a limited experiment with CONIKS which only measured the time for 1k new users and 1k updates at selected epochs. This illustrates the ability of SEEMless to scale in contrast to the limitation of a CONIKS server to less frequent epochs. CONIKS performs even worse (and hangs) with more frequent epochs due to garbage collection overhead.

Query: To evaluate the cost of Query, we measure the computation and verification times at the server and the client respectively, and the size of a proof. We vary the number of users registered with the server up to 10M. We report the costs averaged over a 100 users with 10 trials for each user.

Time The time spent in computing and verifying a Query proof has two components: The VRF proof and the authentication path in the authentication structure (which is Patrica Trie in SEEMless and sparse Merkle Tree in CONIKS). The numbers in Table 3 show the amount of time taken to compute and verify Query proofs including the VRF cost, which is the most expensive operation and dominates the Query cost. We computed the VRF costs as follows. The compute, prove and verify functions for our VRF implemenation, each on average takes 1.3, 1.9 and 3.4 *ms* respectively over 5k trials. Since the time for each of these computations is independent of the input, we used them as constants to simulate the VRF costs for the Query experiments (recall that each Query response has 1 VRF proof in CONIKS and 3 in SEEMless).

In our experiments, we saw the cost of authentication path generation and verification is negligible compared to the VRF – it varied between $0.2 - 0.3ms$ for SEEMless and $0.05 - 0.06ms$ for CONIKS[¶].Since SEEMless requires 3 VRF proofs, its VRF cost is triple that of CONIKS. However, performance of both systems remains comparable and on the order of 10 milliseconds, which is not

[§]In the current implementation, CONIKS does not cache the VRF, so it has to be re-computed for every update and Query. Keeping a hash map for the VRFs would be a simple optimization. Note that CONIKS did not implement the VRF for their evaluation.

[¶]Note that the CONIKS Java implementation did not store the commitments to the public keys, instead they stored it in plaintext (which obviously is a crucial privacy omission). This slightly reduces their cost

	Prove Query	Verify Query
SEEMless	6.03 ms	10.51 ms
CONIKS	2.01 ms	3.47 ms

Table 3: Simulated time taken for Query **proof generation at the server and verification at the client in the above experiments at** 10**M users.**

a noticeable delay for a user. Also, note that the various authentication paths and VRF computations in SEEMless proof generation and verification in our current implementation are done sequentially. We can gain further speedup by parallelizing these operations.

Size of Proofs In our implementation, the proof contains the username and public key (which are sent in addition to the proofs as part of the Querys) along with a commitment to the public key, its opening, the VRF proof and the authentication path in the Patricia Trie. The average size of the proof is about 8600B when the number of users is 10M. For scale, the length of a Twitter post is famously 140 characters (1,120B)[8], so, the bandwidth consumed by the proof is less that of reading 8 Twitter posts in plaintext.

The proof size is proportional to the logarithm of the total number of users registered, as expected (Fig 7 in Appendix E). We also measure the size of the CONIKS proofs. Since we have 3 authentication paths per Query proof in our system, whereas CONIKS has one, the size is at most 3× the proof of CONIKS, as expected.

KeyHistory: Here we measure the computation and verification times of KeyHistory query and the size of the proofs. We note that our implementation of KeyHistory functionality currently implements some redundant checks (for verification without caching) which we optimized in the protocol description in Section 4.

The cost of KeyHistory is determined by 3 parameters: 1) the number of server side epochs , 2) the latest key version or number of key updates made by the user in question and 3) the total number of updates on the server side (i.e. total number of entries in the history trees). We have already seen how the Query proofs grow with respect to parameter 3, so it is not very instructive to explore this again. We conducted two sets of experiments to measure how the first two parameters affect the cost of the history proofs.

Both our experiments include a set S of 10 'special' users, each of whom not only register, but also update their keys as per the experiments. The set of other users, let us call it the set R of 'regular' users, serves as a control: these users register and then update their keys only once. In both experiments, we try to simulate *special* users updating their keys at regular intervals. This means that if there are 200k epochs and the *special* users update their keys 10 times: they register initially and then update every 20k epochs. As a control, the users in R register at a fixed rate. For instance, if $|R| = 1$M, with a total of 200k epochs, 5 users in R register and 5 update their keys at every epoch. We first consider the experiments sans VRF. We do not include the cost of verifying the aZKS commitments in the Merkle Hash Tree (as described in Section 4) in these experiments. We simulate the cost of the Merkle Tree by using the sparse Merkle Tree of CONIKS (which is an over-estimate). The verification time for one leaf is about $49\mu s$ even for 10M server epochs in this sparse Merkle Tree, which would be close to running the server for several years with half minute epochs. Hence, if a user updated her key on 10 occasions, she would have less than an additional $0.5ms$ of verification time.

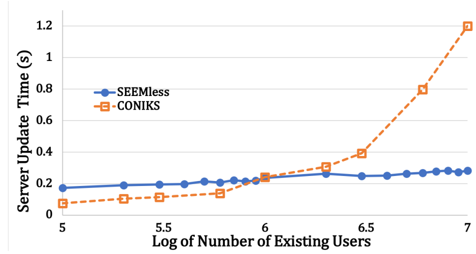

Figure 3: Mean times to update SEEMless and CONIKS for a new epoch with 1**k new users and** 1**k updates. The** x**-axis is logarithmic in base** 10 **and each data point is the mean of 10 trials.**

Dependence on epochs To measure dependence of KeyHistory response sizes and verification times on epochs, we fixed the size of R at 1M, the number of updates by the users in S at 10 and varied the number of epochs (adding an equal number of elements in R at each epoch). Note that the number of minutes in a year is 525, 600, so 1M is close to the number of epochs which would pass if the length of 1 epoch were half a minute.

In our experiments, the average bandwidth consumed by a user who updates 10 times and checks her history after every update, with a server which has 1M epochs is ≈ 1.09MB and not significantly more than at 100k epochs which is ≈ 1.03MB (Fig 4). The slight growth in size is due to the increasing number of marker entries to be checked as the number of epochs increases. The verification of the history proofs took the client a total computation time of 0.02s for 1M epochs excluding the cost for the VRF. In contrast, CONIKS requires a user to check her key at each epoch, making the total cost of monitoring her key grow rapidly, as shown in Fig 4. From these experiments, we conclude that of SEEMless is scalable with very short epochs as opposed to CONIKS.

Dependence on key updates The graph in Fig 8 (in Appendix E) shows the total bandwidth consumed by the proofs for various numbers of updates by users (running verification with caching) in S, where the epochs are fixed at $200k$ and $|R| = 1$M. In the case of our system, even a user who updates her keys almost every week for about five months (20 updates), has to download only an average of 2.01MB to verify her entire key history and the downloaded proof can be verified in an average of less than 0.05s excluding the VRF cost. This includes sending the keys and usernames themselves (at least 1000 bytes per key update), since the user may want to verify the used values. If a user changes her key somewhat less frequently, say every two weeks, then she can monitor the entire history of her keys by downloading about 1.05MB. This does not change significantly as the number of epochs grows, as demonstrated by the previous experiment. On the other hand, a CONIKS user would still need to download a proof every epoch to monitor her key binding. This amounts to almost 576.8MB over the course of as many epochs, hence severely constraining the frequency of epochs.

VRF verification with caching From the experiments above, we see that the size of the authentication paths in the history proofs are small in our case and so is the time to verify these paths. The dominating cost is the cost of VRF verification, but once a VRF has been verified, it can be cached at the client (without the need to verify the VRF proofs again). More specifically, a user may cache the values

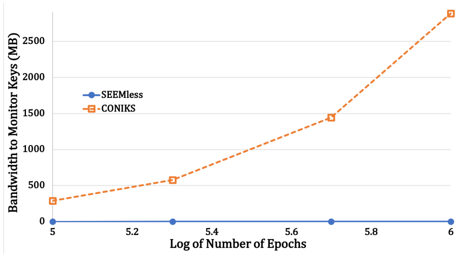

Figure 4: Mean bandwidth consumed in downloading the history proofs over 10 users (running verification with caching) in S with 10 updates each and 1M users in R as the number of total epochs varies. Note that the growth for SEEMless bandwidth is slow: it is ≈ 1.03MB with 10^5 epochs and ≈ 1.09MB with 10^7 epochs.

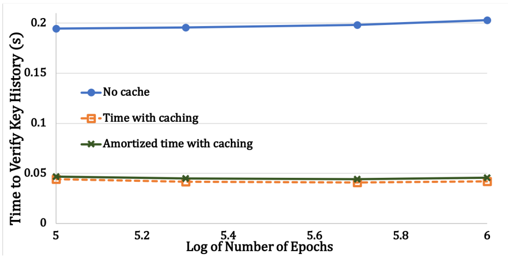

Figure 5: Mean computation time to verify SEEMless history proofs over 10 users in S and 1M users in R and 10 key updates as the number of epochs varies including the time VRF verifications. Note that in CONIKS, the total computation time for a non-caching user depends on the number of epochs and equals over 349s, even at 10^5 epochs.

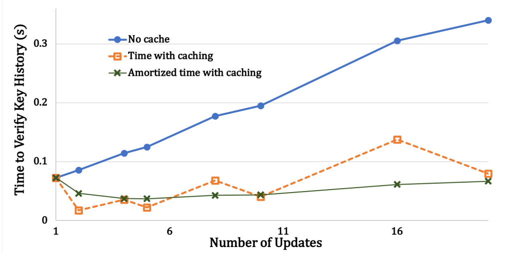

Figure 6: Mean computation time to verify SEEMless history proofs over 10 users in S and 1M users in R and 200k epochs as the number of key updates by users in S varies including the time for VRF verifications. In CONIKS, the time would only depend on the number of epochs and equal over 699s for a non-caching user and over 14s with caching at 200k epochs.

taken by the SHA256 hashes of the VRF values at various versions as she verifies them. Figures 5 and 6 show the total computation times for verifying the history proofs including simulated costs for the VRF verifications with or without caching as well as amortized costs. We defer detailed discussion to Appendix E.

Auditing: The auditors of SEEMless verify 1) the aZKS commitments are correctly added to the hash-chain or Merkle Tree 2) the aZKS are growing in an append-only manner. The cost for checking 1) is $49\mu s$ and requires downloading about 3KB for a tree on 10M epochs (as already discussed in KeyHistory experiments). To monitor 2), i.e., that a Patricia trie is growing in a append-only manner, its auditors receive parts of the tree which are unchanged and the changed leaves, from which they reconstruct the new version of the tree. We simulate this cost by considering the number of new hashes our aZKS trees have to perform for an update (which we experimentally evaluate). We simulate cost for the following: at the start of the epoch, the SEEMless server has 10M registered users and a total of 100k keys that have been updated by those users. Over the course of the epoch, 1k new registrations and 1k updates are made and the server publishes a new digest and proof. For an auditor, the time to verify the append only property of the updated authentication structure is 0.22s, requires computing under 64k SHA256 hashes and the downloaded proof size is upper bounded by 4.24MB. Our simulated proof size estimates a generous upper-bound; in practice, this cost will be much lower. The details of our simulation is in Appendix E.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1617676.

REFERENCES

[1] 2017. WhatsApp Security Vulnerability. https://www.schneier.com/blog/archives/2017/01/whatsapp_securi.html. (2017). Accessed: 2019-01-25.
[2] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J. Freedman. 2016. Blockstack: A Global Naming and Storage System Secured by Blockchains. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 181–194. https://www.usenix.org/conference/atc16/technical-sessions/presentation/ali
[3] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. 2017. Accumulators with Applications to Anonymity-Preserving Revocation. *IACR Cryptology ePrint Archive* 2017 (2017), 43.
[4] Joseph Bonneau. 2016. EthIKS: Using Ethereum to audit a CONIKS key transparency log. In *International Conference on Financial Cryptography and Data Security*. Springer, 95–105.
[5] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. 2008. Strong accumulators from collision-resistant hashing. In *International Conference on Information Security*. Springer, 471–486.
[6] Melissa Chase, Alexander Healy, Anna Lysyanskaya, Tal Malkin, and Leonid Reyzin. 2005. Mercurial commitments with applications to zero-knowledge sets. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 422–439.
[7] Melissa Chase and Anna Lysyanskaya. 2007. Simulatable VRFs with applications to multi-theorem NIZK. In *Annual International Cryptology Conference*. Springer, 303–322.
[8] developer.twitter.com. 2010. Counting Characters. https://developer.twitter.com/en/docs/basics/counting-characters.html. (2010). Accessed: 2018-12-25.
[9] Saba Eskandarian, Eran Messeri, Joe Bonneau, and Dan Boneh. 2017. Certificate Transparency with Privacy. *arXiv preprint arXiv:1703.02209* (2017).
[10] Mohammad Etemad and Alptekin Kupcu. 2015. Efficient Key Authentication Service for Secure End-to-end Communications. Cryptology ePrint Archive, Report 2015/833. (2015). https://eprint.iacr.org/2015/833.
[11] Reza R Farashahi, Pierre-Alain Fouque, Igor Shparlinski, Mehdi Tibouchi, and J Voloch. 2013. Indifferentiable deterministic hashing to elliptic and hyperelliptic curves. *Math. Comp.* 82, 281 (2013), 491–512.
[12] Thomas Icart. 2009. How to hash into elliptic curves. In *Advances in Cryptology-CRYPTO 2009*. Springer, 303–316.
[13] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 177–194.
[14] Keybase.io. 2014. Keybase is now writing to the Bitcoin blockchain. https://keybase.io/docs/server_security/merkle_root_in_bitcoin_blockchain. (2014). Accessed: 2018-10-05.
[15] Keybase.io. 2019. Keybase is not softer than TOFU. https://keybase.io/blog/chat-apps-softer-than-tofu. (2019). Accessed: 2019-05-05.
[16] Keybase.io. 2019. Managing Teams and Keys with Keybase. (2019).
[17] Keybase.io. 2019. Protocol Security Review. (2019).
[18] Donald Ervin Knuth. 1998. *The art of computer programming: sorting and searching*. Vol. 3. Pearson Education.

[19] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. *Certificate transparency*. Technical Report.
[20] Moses Liskov. 2005. Updatable zero-knowledge databases. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 174–198.
[21] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. 2015. CONIKS: Bringing Key Transparency to End Users.. In *Usenix Security*. 383–398.
[22] Silvio Micali, Michael Rabin, and Joe Kilian. 2003. Zero-knowledge sets. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium*. IEEE, 80–91.
[23] Namecoin. 2014. https://namecoin.org. (2014). Accessed: 2018-10-05.
[24] Alina Oprea and Kevin D Bowers. 2009. Authentic time-stamps for archival storage. In *European Symposium on Research in Computer Security*. Springer, 136–151.
[25] LEAP Encryption Access Project. 2012. Nicknym. https://leap.se/en/docs/design/nicknym. (2012). Accessed: 2018-10-05.
[26] signal.org. 2016. Identity binding. https://www.signal.org/docs/specifications/x3dh. (2016). Accessed: 2019-05-05.
[27] Alin Tomescu and Srinivas Devadas. 2017. Catena: Efficient non-equivocation via Bitcoin. In *IEEE Symp. on Security and Privacy*.
[28] Kenton Varda. 2008. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul* 72 (2008).

A DISCUSSION

User Experience Having an end-user seamlessly interact with the functionality of SEEMless (or any VKD system) is crucial for successful deployment of this system. Designing a smooth user interface that exposes the VKD functionality to end users without drastically changing their experience of using existing messaging software requires usability studies, which we leave for future work. However, here we discuss a blueprint of what we envision the user interface of SEEMless to be.

The high level goal is to have an end-user's software run the queries and verification the background *in a timely manner* and alert an end-user only when some verification fails. When Alice first registers for the service, the client software installed on her device generates keys in the background, and makes a KeyHistory query to the server to verify that this is the first key that was issued for her, and a Query to verify that the key is stored correctly. This happens in the background and is invisible to Alice unless verification of one of these fails, in which case she will be alerted of possible misbehavior. Similarly, when Alice requests Bob's key, the client software will run verification of the proof received from the server in the background and will only notify Alice if the verification fails. In addition to that, we want Alice's client to check her key history *sufficiently often*. This entails running KeyHistory query in the background periodically and notifying Alice if it is not consistent with the updates she has made.[‖] *keyver* must also be run when Alice changes her device or reinstalls the client software (thereby forcing a key update) in which case cases the software would display to Alice a list of the times when her key was updated.

Multiple devices We have described the system as storing only a single key per user, but in reality Alice might use her account on multiple devices, and the system would be used to store a list of public keys, one for each of her devices. Bob would then encrypt under all of these keys, and the service would determine which to deliver. This works without any modification of the VKD system. The only change in user experience is that when Alice's device runs periodic updates, it might find updates made by other devices: the times for these updates should be displayed to Alice for verification.

What we assume from users In this work our goal is to make the system usable by average users who cannot be assumed to remember or correctly store long term cryptographic secrets. We do assume that the user's device can store a secret (the user's current key), although that device might be lost/re-imaged, etc in which case the secret would be lost. We assume that the user has some way of authenticating to the service provider (this could be through a second factor text message, or a phone call to answer security questions, etc); this is already a requirement in existing end-to-end encryption systems or other messaging services so we are not increasing the attack surface. Finally, we assume that the user can recognize the approximate times when she updated her key (enough to be able to identify if extra updates have been included in the list). To help with this we could also have the server store a note from the user about each update, e.g. "bought new iphone" or "re-installed app".

Power users Our system could of course support options for power users to make KeyHistory queries whenever they want, to explicitly compare public keys with their friends (as in WhatsApp), or to sign their key updates.

Assumptions on system We do assume that Alice, Bob, and the server have clocks that are approximately in sync. We also assume some way of ensuring that all parties have consistent views of the current root commitment or at least that they periodically compare these values to make sure that their views have not forked. A simple way of doing this would be for the server to publish the head periodically to a blockchain (as in Catena-CONIKs [27]), but we could also implement it with a gossip protocol as in CONIKS [21]. If we implement this by having the server post the commitment on the blockchain, then this means we assume the client software periodically interact with the blockchain. (Because of the hash chain, even if clients do not check the root on the blockchain every epoch, if two different client's views of the root diverge, they will eventually detect that when they do contact the blockchain.) Similarly, our epoch length need not be limited by the blockchain block time – it is enough if the server periodically posts the current commitment on the blockchain; a misbehaving server will be caught as soon as the next blockchain post happens. This vulnerability window is a tunable parameter. The bandwidth requirements (for interacting with the blockchain) of log auditors of the system and of thin clients (like mobile phones) can be significantly reduced by using an efficiently-verifiable blockchain witnessing service, like Catena [27].

Distributing the service We have described the service in terms of a single server, but in practice, the "server" can be implemented using a distributed network of servers, for reliability and redundancy. Our model captures the server as a *single logical entity*, so it can accommodate a distributed implementation fairly easily. Constructing proofs just requires reading the shared data structures for previous time epochs, so that can easily be done by many processes in parallel. Once all of the updates for a given epoch have been collected, then the authentication data structure needs to be updated, but even that can be parallelized fairly easily because of the tree structure. We can support queries even during the data structure update by keeping a snapshot of the last authentication

[‖]Determining how often the client software needs to run KeyHistory depends depends how quickly users want to be notified of misbehavior and requires user studies which we defer for future work.

data structure until the update epoch completes. Once the epoch completes, the snapshot can be discarded (so this does not blow up memory).

B CRYPTOGRAPHIC PRIMITIVES FOR AZKS

Collision Resistant Hash Function (CRHF): A hash function H is collision resistant if it is hard to find two inputs that hash to the same output; that is, two inputs x and y such that $H(x) = H(y)$, and $x \neq y$. We will in some of our building blocks require a stronger hash function, which we will treat as a random oracle. The random oracle heuristic proves security when the hash function is replaced with a random function. This roughly provides security against attackers who use the hash function as a black box.

Instantiation. For our experiments we use SHA384 in our VRF construction and SHA256 everywhere else.

Simulatable Commitment Scheme (sCS) : A simulatable commitment scheme consists of algorithms (CS.Commit, CS.Open, CS.VerifyOpen): (1) $\mathsf{com}_\sigma \leftarrow \mathsf{CS.Commit}(1^\lambda, m; r)$: takes security parameter λ and produces commitment com_σ to message m using randomness r. (2) $\tau \leftarrow \mathsf{CS.Open}(\sigma, m, r, \mathsf{com}_\sigma)$: outputs a decommitment value corresponding to commitment com_σ for message m and randomness r. (3) $1/0 \leftarrow \mathsf{CS.VerifyOpen}(\sigma, \mathsf{com}_\sigma, m, \tau)$: accepts or rejects the decommitment of com_σ to message m in terms of the decommitment value τ.

An sCS satisfies the standard requirements of commitment schemes with respect to hiding and binding: commitments hide the committed message, and it is hard to open a commitment to two different messages. In addition, sCS requires that there exist a hypothetical "simulator" with special powers (either a trapdoor or in our case control of the random oracle) who could form commitments that can later be opened to any value m. This then shows that the commitment and opening give the adversary no additional information.

Instantiation. We can construct this commitment scheme from a hash function which we model as a random oracle: $\mathsf{Commit}(\sigma, m; r)$ samples random string $r \leftarrow \{0,1\}^\lambda$ and outputs $H(m, r)$.

Simulatable Verifiable Random Function (sVRF) [7]: A Verifiable Random Function (VRF) is similar to a pseudorandom function, with the additional property of verifiability: corresponding to each secret key SK, there is a public key PK, such that, for any $y = \mathsf{VRF.Eval}(\mathsf{SK}, x)$, it is possible to verify that y is indeed the value of the VRF seeded by SK evaluated on input x. A simulatable VRF (sVRF) is a VRF for which this proof can be simulated, so a hypothetical simulator with special powers (e.g. controlling the random oracle) can fake a proof that the value of $\mathsf{VRF.Gen}(\mathsf{SK}, x)$ is any y. This guarantees that the proof does not compromise the pseudorandomness of this or any other output. An sVRF is comprised of the following algorithms: 1) $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{sVRF.KeyGen}(1^\lambda)$: takes security parameter λ and outputs the public key PK and secret key SK. 2) $y \leftarrow \mathsf{sVRF.Eval}(\mathsf{SK}, x)$: takes the secret key SK and x and outputs the sVRF evaluation of x as y 3) $\pi \leftarrow \mathsf{sVRF.Prove}(\mathsf{SK}, x)$: takes the secret key SK and x and outputs a proof for the sVRF evaluation of x 4) $1/0 \leftarrow \mathsf{sVRF.Verify}(\mathsf{PK}, x, y, \pi)$: verifies the proof.

Instantiation. We describe an efficient constructions of sVRFs based on the DDH assumption as proposed in [21]. The proof for pseudorandomness follows from DDH as observed in [21]. Let $\mathbb{G}$ be a DDH group of order q and let g be a generator. Let $H_1 : \{0,1\}^* \mapsto \mathbb{G}$ and $H_2 : \{0,1\}^* \mapsto \mathbb{Z}_q^*$ be two hash functions. The sVRF construction is the following. (1) sVRF.KeyGen: Choose $k \xleftarrow{\$} \mathbb{Z}_q^*$ and output $\mathsf{SK} = k$ and $\mathsf{PK} = g^k$ (2) sVRF.Eval: Output $y = (H_1(x))^{\mathsf{SK}}$ (4) sVRF.Prove: The proof is the Fiat-Shamir transformation of Schnorr protocol for proving common exponent; proving that prover knows $\mathsf{SK} = k$ such that $\mathsf{PK} = g^k$ and $y = h^k$ for $h = H_1(x)$. Prover chooses $r \xleftarrow{\$} \mathbb{Z}_q^*$ and outputs proof as $\pi = (s, t)$ for $s = H_2(x, g^r, h^r)$ and $t = r - sk$. (5) sVRF.Verify: Parse π as (s, t). Check that $s = H_2(x, g^t(\mathsf{PK})^s, h^t y^s)$ for $h = H_1(x)$

C VERIFIABLE KEY DIRECTORY

Notation: A function $\nu : \mathbb{Z}^+ \mapsto \mathbb{R}^+$ is a negligible function if for all $c \in \mathbb{Z}^+$, there exists k_0 such that for all $k \geq k_0$, $\nu(k) < k^{-c}$. An algorithm $\mathcal{A}$ is said to have oracle access to machine O if $\mathcal{A}$ can write an input for O on a special tape, and tell the oracle to execute on that input and then write its output to the tape. We denote this oracle access by $\mathcal{A}^O$. For the rest of the sections, we will use (label, val) and (username,public key) interchangeably.

Security Properties: Here we give the formal definitions that we informally described in Section 2. Note that in the security definitions, t_1 denotes the first server epoch at which the label in the definitions gets registered for the first time with the directory Dir.

- **Completeness:** Let $\mathsf{Dir}_0 = \{\}$, $\mathsf{st}_0 = \perp$.
  For all possible labels, for all t_{current}, n, for all sets $\{\mathsf{val}_i\}_{i=1}^n$ for all update sets $S_1, \ldots, S_{t_{\mathsf{current}}}$ such that $(\mathsf{label}, \mathsf{val}_i) \in \{S_{t_i}\}_{i=1}^n$ is the set of all occurrences of label in $S_1, \ldots, S_{t_{\mathsf{current}}}$, and $\forall t^* < t_{\mathsf{current}}$:

$$\begin{aligned}\Pr[(\{(\mathsf{pub}_t, \mathsf{st}_t, \mathsf{Dir}_t) \leftarrow \mathsf{VKD.Publish}(\mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)\}_{t=1}^{t_{\mathsf{current}}} \wedge \\ \{(\mathsf{com}_t, \Pi_t^{\mathsf{Upd}}) = \mathsf{pub}_t\}_{t=1}^{t_{\mathsf{current}}} \wedge \\ (\{(\mathsf{val}'_i, t'_i)\}_{i=1}^{n'}, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{VKD.KeyHistory}(\mathsf{st}_{t_{\mathsf{current}}}, \mathsf{Dir}_{t_{\mathsf{current}}}, \mathsf{label}) \wedge \\ n = n' \wedge \forall i \in [1, n](t'_i = t_i \wedge \mathsf{val}'_i = \mathsf{val}_i) \wedge \\ \mathsf{VKD.HistoryVer}(\mathsf{com}_{t_{\mathsf{current}}}, \mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}) \wedge \\ \mathsf{VKD.Audit}(t_1, t_{\mathsf{current}}, \{\mathsf{com}_k, \Pi_k^{\mathsf{Upd}}\}_{k=t_1}^{t_{\mathsf{current}}}) \wedge \\ (\pi, \mathsf{val}) \leftarrow \mathsf{VKD.Query}(\mathsf{st}_{t^*}, \mathsf{Dir}_{t^*}, \mathsf{label}) \wedge \\ \mathsf{VKD.QueryVer}(\mathsf{com}_{t^*}, \mathsf{label}, \mathsf{val}, \pi) \wedge \\ \exists j \in [1, n] s.t. (t_j \leq t^* < t_{j+1} \wedge (\mathsf{val} = \mathsf{val}_j))] = 1\end{aligned}$$

- **Soundness:** Formally, we want to capture that for any label label if versions proofs verifies with respect to $\{(\mathsf{val}_i, t_i)\}_{i=1}^n$ and if the audit verifies from t_1 to t_n then at any time t^* between an interval $[t_j, t_{j+1}]$ for some $j \in [n]$, a malicious server cannot give out a proof for label with a value which is inconsistent with the corresponding versions proof at t_j that is, $\mathsf{val} \neq \mathsf{val}_j$ for t_j. Checking this is enough because if the server has given an incorrect key at time t^*, he will have to introduce an additional update sometime later to potentially fix it and will hence be caught with high probability. Hence a malicious server S^* should not be able to come up with a label, $\{(\mathsf{val}_i, t_i)\}_{i=1}^n$ with versions proof Π^{Ver}, commitments and update proofs Π^{Upd} for all times between t_1 to t_n and query proof (π, val) for some t^* for $\mathsf{val} \neq \mathsf{val}_j$.

For all PPT S^*, there exists a negligible function $\nu()$ such that for all $\lambda \in \mathbb{N}$:

$$\begin{aligned}\Pr[(\mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}, \{\mathsf{com}_k, \Pi_k^{\mathsf{Upd}}\}_{k=t_1}^{t_{\mathsf{current}}},\\ t^*, j, (\pi, \mathsf{val})) \leftarrow S^*(1^\lambda) :\\ \mathsf{VKD.QueryVer}(\mathsf{com}_{t^*}, \mathsf{label}, \mathsf{val}, \pi)\\ \wedge\ \mathsf{VKD.Audit}(t_1, t_{\mathsf{current}}, \{\mathsf{com}_k, \Pi_k^{\mathsf{Upd}}\}_{k=t_1}^{t_{\mathsf{current}}})\\ \wedge\ \mathsf{VKD.HistoryVer}(\mathsf{com}_{t_{\mathsf{current}}}, \mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}})\\ \wedge\ (\mathsf{val} \neq \mathsf{val}_j)\ \wedge\ (t_j \leq t^* < t_{j+1})\\ \wedge\ j \in [1, n] \wedge t_1 \leq \ldots \leq t_n \leq t_{\mathsf{current}}] \leq \nu(\lambda)\end{aligned}$$

Theorem C.1. *The construction in Section 4 satisfies VKD soundness as defined above.*

Proof. First, we claim that every pair of proofs $\Pi_k^{\mathsf{Upd}}, \Pi_{k+1}^{\mathsf{Upd}}$ must contain consistent values for $\mathsf{com}_{\mathsf{all},k}, \mathsf{com}_{\mathsf{old},k}$. If not, we could directly build a reduction breaking collision resistance of H.

Now, we observe that we have a chain of aZKS commitments and associated aZKS update proofs, all of which verify.

Similarly, we claim that the aZKS commitments $\mathsf{com}_{\mathsf{all},t^*}, \mathsf{com}_{\mathsf{old},t^*}$ and $\mathsf{com}_{\mathsf{all},t_i}, \mathsf{com}_{\mathsf{old},t_i}$ contained in π and Π^{Ver} must also be consistent with those given in the Π^{Upd} proofs. If not, again we can break collision resistance of H.

Next, say that π and Π^{Ver} are aZKS-inconsistent if they contain a pair of proofs w.r.t. either the "old" or "all" aZKS such that one is a membership proof at time x and the other is a nonmembership proof for the same label at some time $y \geq x$, or a pair of proofs with contradictory vals for the same label (even at different times). Note that an adversary who with non-negligible probability wins the above soundness game with π, Π^{Ver} that are aZKS-inconsistent can be directly used to build an adversary attacking the aZKS soundness property. Thus, we have only to show that any adversary who successfully breaks VKD soundness must produce aZKS-inconsistent proofs. We argue that as follows. Let α be the version number contained in proof π produced by the adversary. Now we consider the following cases:

$\alpha < j$**:** In this case Π^{Ver} (step (3) of KeyHistory) contains a membership proof for $(\mathsf{label}|\alpha)$ w.r.t. $\mathsf{com}_{\mathsf{old},t_{\alpha+1}}$ and π contains a non-membership proof for $(\mathsf{label}|\alpha)$ w.r.t. $\mathsf{com}_{\mathsf{old},t^*}$, where $t^* > t_j \geq t_{\alpha+1}$. (The first inequality follows from the definition, the second inequality follows from $\alpha < j$.)

$\alpha = j$**:** In this case Π^{Ver} (step (2) of KeyHistory) contains a membership proof for $(\mathsf{label}|\alpha)$ with value val_j in $\mathsf{com}_{\mathsf{all},t_j}$, while π contains a membership proof for $(\mathsf{label}|\alpha)$ with value $\mathsf{val} \neq \mathsf{val}_j$ in $\mathsf{com}_{\mathsf{all},t^*}$.

$j < \alpha \leq n$**:** In this case Π^{Ver} (step (4) of KeyHistory) contains a nonmembership proof for $(\mathsf{label}|\alpha)$ w.r.t. $\mathsf{com}_{\mathsf{all},t_\alpha - 1}$, while π contains a membership proof for $(\mathsf{label}|\alpha)$ w.r.t. $\mathsf{com}_{\mathsf{all},t^*}$. Note that $t_\alpha - 1 \geq t_{j+1} - 1 \geq t^*$, where the first inequality follow from $j < \alpha$ and the second follows from the definition.

$n < \alpha < 2^{a+1}$ **where a is the largest integer s.t.** $2^a \leq n < 2^{a+1}$: In this case Π^{Ver} (step (5) of KeyHistory) contains a nonmembership proof for $(\mathsf{label}|\alpha)$ w.r.t. $\mathsf{com}_{\mathsf{all},t_{\mathsf{current}}}$, while π contains a membership proof for $(\mathsf{label}|\alpha)$ w.r.t. $\mathsf{com}_{\mathsf{all},t^*}$. Note that $t_{\mathsf{current}} \geq t^*$, from the definition.

$\alpha \geq 2^{a+1}$ **where a is the largest integer s.t.** $2^a \leq n < 2^{a+1}$: Let b be the largest integer s.t. $2^b \leq \alpha$ In this case Π^{Ver} (step (6) of KeyHistory) contains a nonmembership proof for $(\mathsf{label}|\mathsf{mark}|b)$ w.r.t. $\mathsf{com}_{\mathsf{all},t_{\mathsf{current}}}$, while π contains a membership proof for $(\mathsf{label}|\mathsf{mark}|b)$ w.r.t. $\mathsf{com}_{\mathsf{all},t^*}$. Note that $t_{\mathsf{current}} \geq t^*$, from the definition.

□

- **$\mathcal{L}$-Privacy:**
  We will say that a VKD is private for leakage function $\mathcal{L} = (L_{\mathsf{Publish}}, L_{\mathsf{Query}}, L_{\mathsf{KeyHistory}})$ if there exists a simulator $\mathcal{S} = (\mathcal{S}_{\mathsf{Publish}}, \mathcal{S}_{\mathsf{Query}}, \mathcal{S}_{\mathsf{KeyHistory}})$ such that for any PPT client C^*, the outputs of the following two experiments are computationally indistinguishable:
  In the real game, C^* is given access to $O_P, O_\pi, O_{\Pi^{\mathsf{Ver}}}$, three stateful oracles that share state. O_π is the proofs oracle which on query a label label, will output π generated by $(\pi, \mathsf{val}) \leftarrow \mathsf{VKD.Query}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$. $O_{\Pi^{\mathsf{Ver}}}$ is the key history oracle, which on query, label outputs Π^{Ver} generated by $(\{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{VKD.KeyHistory}(\mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$. O_P is the publish oracle which on input update set S, updates directory Dir_t and outputs $(\mathsf{com}_t, \Pi_t^{\mathsf{Upd}})$ as computed by $\mathsf{VKD.Publish}()$.
  In the simulated game C^* is given access to an oracle which maintain Dir_t, but calls the simulators to produce commitments and proofs. On a Publish query, the oracle $\mathcal{S}_{\mathsf{Publish}}$ is given leakage on the update set S given by $L_{\mathsf{Publish}}(S)$, and emulates the publish oracle O_P. On a Query query, the oracle looks up val for label in the current directory Dir_t, and calls $\mathcal{S}_{\mathsf{Query}}(L_{\mathsf{Query}}(\mathsf{label}, \mathsf{val}), \mathsf{label}, \mathsf{val})$[**] to emulate the proof oracle O_π. On a KeyHistory query, the oracle looks up the history for val_i in its directories $\mathsf{Dir}_1, \ldots, \mathsf{Dir}_t$, and calls $\mathcal{S}_{\mathsf{KeyHistory}}(L_{\mathsf{KeyHistory}}(\mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n), \mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n)$[††] to emulate the key history oracle $O_{\Pi^{\mathsf{Ver}}}$.

Leakage The leakage for our construction is as described below For discussion see section 4.

Leakage for Publish For each label that was updated, if this is the first update since the adversary queried for it via Query then add it to set Q_{Query}, and if it was previously queried to KeyHistory then add it to set $Q_{\mathsf{KeyHistory}}$. The leakage from

[**] Recall that the leakage functions share state, so this also implicitly gets access to the leakage from all previous Publish, Query, or KeyHistory queries.

[††] see **

this query is the number of new registrations and of key updates in the latest epoch, and the sets $Q_{\mathsf{Query}}, Q_{\mathsf{KeyHistory}}$.

Leakage for Query The leakage from this query is the version number of the queried key and the epoch at which it was last updated.

Leakage for KeyHistory There is no additional leakage from this query.

THEOREM C.2. *The construction in Section 4 when implemented with an* aZKS *with the leakage profile described in appendix G satisfies VKD privacy as defined above, with the above leakage functions.*

PROOF. We first describe the necessary simulators. Recall that the simulators share state. In this case that state will include three tables: T_{current}, which stores the most recent version for labels which are in $Q_{\mathsf{KeyHistory}}$, T_{Query}, which stores the version for labels which are in Q_{Query} at the time of their latest Query query, and $T_{\mathsf{KeyHistory}}$, which stores the version and epoch for the latest KeyHistory query. Our simulators will run two instances of the aZKS simulator, which we will denote $\mathsf{aZKS.Sim}^{\mathsf{all}}$ and $\mathsf{aZKS.Sim}^{\mathsf{old}}$.

$\mathsf{Sim}_{\mathsf{Publish}}$ is given as input the number of new registration and number of key updates, and the sets $Q_{\mathsf{Query}}, Q_{\mathsf{KeyHistory}}$. On the first query, $\mathsf{Sim}_{\mathsf{Publish}}$ calls $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{CommitDS}}$ and $\mathsf{aZKS.Sim}^{\mathsf{old}}_{\mathsf{CommitDS}}$ on an empty directory D_0 to generate $\mathsf{com}_{\mathsf{all},0}, \mathsf{com}_{\mathsf{old},0}$. On subsequent queries it behaves as follows:

First, note that the number of additions to $\mathsf{ZKS._{all}}$ will be the total number of new registrations and updates, while the number of additions to $\mathsf{ZKS._{old}}$ will be the number of updates. Next the VKD simulator must construct the sets of labels Q_{all} and Q_{old} in the current ZKS update sets for which there has been a previous non-membership query. We do this as follows: for each label $\in Q_{\mathsf{Query}}$, look up the version number α in T_{Query} and add $\mathsf{label}|\alpha$ to Q_{old} (unless $\alpha = 2^a$ in which case we add $\mathsf{label}|\mathsf{mark}|a$). For each query label $\in Q_{\mathsf{KeyHistory}}$, lookup the current version $\alpha_{\mathsf{current}}$ in T_{current} and the version $\alpha_{\mathsf{KeyHistory}}$ and query epoch $t_{\mathsf{KeyHistory}}$ from the most recent key history query in $T_{\mathsf{KeyHistory}}$. Let $\alpha_{\mathsf{current}} = \alpha_{\mathsf{current}} + 1$, and update the value in T_{current}. Let a be the maximum value such that $2^a \leq \alpha_{\mathsf{KeyHistory}}$. If $\alpha_{\mathsf{current}} < 2^{a+1}$ add $(\mathsf{label}|\alpha_{\mathsf{current}})$ to Q_{all}. If $\alpha_{\mathsf{current}} = 2^b$ for $b \geq a+1$ and $\alpha_{\mathsf{current}} \leq t_{\mathsf{KeyHistory}}$, add $(\mathsf{label}|\mathsf{mark}|b)$ to Q_{all}. Finally, it calls $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{CommitDS}}$ and $\mathsf{aZKS.Sim}^{\mathsf{old}}_{\mathsf{CommitDS}}$ with these leakage values to obtain updated commitments and proofs.

$\mathsf{Sim}_{\mathsf{Query}}$ is given as input the (label, val), and the leakage L_{Query}, which is the version number α of label in the current directory, and the epoch t_{prev} at which it was last updated.

- It will call the simulator $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{Query}}(t_{\mathsf{current}}, (\mathsf{label}|\alpha), \mathsf{val}, t_{\mathsf{prev}})$ where t_{current} is the current epoch to generate the membership proof in $\mathsf{ZKS._{all}}$ (or if $\alpha = 2^a$, it will query with $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{Query}}(t_{\mathsf{current}}, (\mathsf{label}|\mathsf{mark}|a), \mathsf{val}, t_{\mathsf{prev}})$.
- It will call the simulator $\mathsf{aZKS.Sim}^{\mathsf{old}}_{\mathsf{Query}}(t_{\mathsf{current}}, (\mathsf{label}|\alpha), \perp, \perp)$ to generate the non-membership proof in $\mathsf{ZKS._{old}}$.

Finally, it updates the entry for label in T_{Query} to store version α (or adds it if it does not exist).

$\mathsf{Sim}_{\mathsf{KeyHistory}}$ is given label, $\{(\mathsf{val}_i, t_i)\}_{i=1}^{\alpha_{\mathsf{current}}}$ and generates the simulated proof as follows:

(1) For each i it outputs the commitments $\mathsf{com}_{t_i}, \mathsf{com}_{t_i - 1}$ produced by $\mathsf{Sim}_{\mathsf{Publish}}$, and similarly for the hash values.
(2) For each i it will call the simulator $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{Query}}(t_i, (\mathsf{label}|i), \mathsf{val}_i, t_i)$ to generate the membership proof in $\mathsf{ZKS._{all}}$ (or if $i = 2^a$, it will query with $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{Query}}(t_i, (\mathsf{label}|\mathsf{mark}|a), \mathsf{val}_i, t_i)$.
(3) For each i it will call the simulator $\mathsf{aZKS.Sim}^{\mathsf{old}}_{\mathsf{Query}}(t_i, (\mathsf{label}|i - 1), \mathsf{null}, \perp)$ to generate the membership proof in $\mathsf{ZKS._{old}}$.
(4) For each i it will call the simulator $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{Query}}(t_i - 1, (\mathsf{label}|i), \mathsf{null}, \perp)$ to generate the nonmembership proof in $\mathsf{ZKS._{all}}$.
(5) For each j from $\alpha_{\mathsf{current}} + 1$ to $2^{a+1} - 1$ it will run $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{Query}}(t_{\mathsf{current}}, (\mathsf{label}|j), \mathsf{null}, \perp)$ to generate the nonmembership proof in $\mathsf{ZKS._{all}}$.
(6) For each j from 2^{a+1} to $\log(t_{\mathsf{current}})$ it will run $\mathsf{aZKS.Sim}^{\mathsf{all}}_{\mathsf{Query}}(t_{\mathsf{current}}, (\mathsf{label}|j), \mathsf{null}, \perp)$ to generate the nonmembership proof in $\mathsf{ZKS._{all}}$.

Finally, it updates the entry for label in $T_{\mathsf{KeyHistory}}$ to store $(\alpha_{\mathsf{current}}, t_{\mathsf{current}})$ (or adds it if it does not exist). If there is no entry in T_{current} for label, it adds it.

Now that we have defined our simulator in terms of the aZKS simulators, the proof is very straightforward. We introduce one hybrid game which proceeds as in the simulated game except that the aZKS simulator for old is replaced by the real aZKS algorithms. Then we can argue that the real game is indistinguishable from the hybrid game by the aZKS privacy property, and similarly that the hybrid game is indistinguishable from the simulated game by the aZKS privacy property. □

D TRACING VULNERABILITY IN CONIKS

In [21], we discovered a privacy leakage that can be damaging for targeted user attacks. We call this leakage *tracing vulnerability*. The leakage is the following: In CONIKS, when a user queries for the same label several times, she gets values in the proof which depend on the position of her label in the authentication tree. These values can be positions of other labels and give information about a label that was not queried. Hence if Alice gets the proofs for the same label over time, she can infer about other labels, whether they were updated or deleted. For example, consider a system with 4 users: Alice, Bob, Charlie, Mary with P(Alice) = 010, P(Bob) = 011 P(Charlie) = 101, P(Mary) = 110, P() denotes the position of the label in the tree. The proof for Alice's key will contain 011, being its sibling, which is P(Bob). Since the position is fixed for the entire lifetime of the directory, now Alice can trace when Bob's key changes just by querying for her own key and observing when sibling node changes. While the username is not directly leaked, once Alice queried for Bob's key, she will be able to completely trace when the key changed without ever querying for his key again. This means, even if Bob has deleted Alice from his contact list, Alice will still be able to trace when Bob's key changed just my looking at the proof of her own key.

E EXPERIMENTS

Here we discuss the details of caching VRF proofs/labels both at a client and at the server.

Caching VRF labels at the client Recall, that to verify and key version, i, such that $2^a \leq i < 2^{a+1}$ for some non-negative integer a, her verifications include:

- $i - 1$: She just needs to check for any version that the previous entry is in the "old" aZKS. One additional version is moved to the "old" aZKS for each new update.
- $i + 1, \ldots, 2^{a+1} - 1$: This means that unless she updates to a new version i such that $i = 2^k$ for some $k \in \mathbb{Z}$, the VRF values for versions in the set $i + 1, \ldots, 2^{a+1} - 1$ have already been verified and are cached and do not need to be verified, just checked for equality. If indeed $i = 2^k$ for some $k \in \mathbb{Z}$, she will have to download the VRF values $i + 1, \ldots, 2^{k+1}$ which are 2^k in number. This amortizes her cost to 1 VRF verification per update.
- $2^{a+1}, \ldots, 2^{\lfloor \log t \rfloor}$: This means that only when verifying after some new 2^kth epoch will a user need to verify an additional VRF. For 10 updates at regular intervals, once 1M epochs pass, she will have to verify a total of 20 VRFs, getting an additional amortized cost of 2 VRF verifications, per update.

Once verified, she can just save the 32B Patricia trie labels for the "all" aZKS and the "old" aZKS which, on the client side, results in the following formula

$$\text{client cache} \approx [2i + 2^{\lfloor \log i \rfloor + 1} + \log t - (\lfloor \log i \rfloor + 1)] \times 32\text{B}$$

where i is the number of updates by the user and t is the number of server epochs which have passed. In practice, this means that even a user who has 20 updates over 1M epochs only needs to cache 2.8KB. Note that this is much smaller than even a low resolution image take on a flip phone. Figures 5 and 6 show the total computation times for verifying the history proofs including simulated costs for the VRF verifications with or without caching as well as amortized costs. Compare the numbers in figure 5 with a total of over 349s in CONIKS for a user needing to verifying her key every epoch regardless of number of updates when she does not save the VRF locally at just 100k epochs. Alternatively, if she were to store the tree label corresponding to her name, it would require constant monitoring and take a total of over 7s to ensure the server never shows an incorrect key for her. The numbers for CONIKS would increase significantly when the number of epochs is higher to 3495s when 1M epochs pass if she doesn't save the VRF, and 71s when she does. In the experiment in Figure 6 in Appendix E, the epochs are fixed at 200k so even as the number of updates in CONIKS increases, its computation time would remain fixed at over 699s for verifier without caching and more than 14s for a verifier with caching.

Caching VRF proofs at the server If the server were to cache the VRF proofs for each user, where it had 10M users and each user had 20 updates, it would correspondingly have to store 28GB of data. However, updates and KeyHistorys are relatively infrequent, (such as when a user reinstalls or updates her app), it is reasonable to assume that even about a second of startup time for an app to verify key history is not a barrier to usability. To respond to Query without the need to compute a VRF will require caching the VRF value and proof for the current key of each user. This amounts to 3×48B (the VRF value is 48B) for each registered user, which is 1.44GB over 10M users. Further optimizations can be made, such as saving values for inactive users to disk. We leave such optimizations as future work.

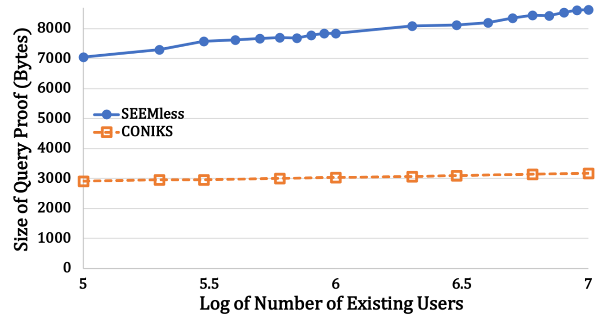

Figure 7: Mean size of Query **proofs for** 100 **nodes as the number of users in the VKD varies. The** x**-axis is logarithmic in base** 10 **and each data point is the mean of** 10 **trials.**

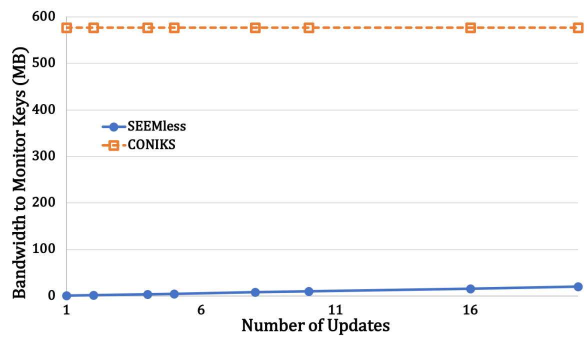

Figure 8: Mean bandwidth to verify the history proofs over 10 **users (running verification with caching) in** S **and 1M users in** R **and** 200**k epochs as the number of key updates by users in** S **varies and each user checks her key history after each update.**

Auditor Cost To monitor that the Patricia tries are growing in a append-only manner, its auditors receive parts of the tree which are unchanged and the changed leaves, from which they reconstruct the new version of the tree. We simulate this cost by considering the number of new hashes our aZKS trees have to perform, since our implementation only computes the hashes that changed. For the "all" aZKS, less than 47k new hashes had to be computed and this took 0.17s, and the "old" aZKS computed less than 17k new hashes and it took 0.05s. So the total cost of hashing is 0.22s.

The auditors need to download the respective new leaves and the nodes whose subtrees did not change. For downloading the new leaves, the auditors need to download 1k leaves for the "old" aZKS, 2k leaves for the "all" aZKS, which are 2×256 bits each (leaf label and commitment), totaling 192KB for the two aZKS. To estimate an upper-bound on the number of roots of the unchanged aZKS subtrees, consider the following. Any node which is the root of an unchanged subtree must have a sibling which has changed. If not, then the node as well as its sibling are unchanged, the parent of this node must be unchanged and thus, the parent or some other ancestor must be the root of the unchanged subtree. Therefore, contenders for roots of unchanged subtrees in an aZKS tree are only siblings of nodes along the path from the root to new leaves. Recall that the average depth of a node in a compressed Patricia Trie is $\log n$ (we experimentally confirmed this) where n is the total number of leaves. Hence, the number of unchanged roots of unchanged subtrees is at most $k \times \log n$ where k is the number of new leaves. Corresponding to our experiment in Section 7, this amounts to about 1000 log 100k unchanged nodes for the "old" aZKS and 2000 log 10.1M for the "all" aZKS, totaling an upper bound of 1.06MB+2.98MB= 4.04MB (including hash values and labels). Thus, the total proof size that the auditors need to download is less than 4.24MB. Note that, sizes of the labels becomes smaller at each level, but this estimate counts them as 256 bits at every level. This estimate also ignores the double counting of nodes which lie at the intersection of paths to multiple new leaves. So, in practice, this proof size will be much lower.

F RELATED WORK

Our work broadly falls in the category of building provably secure and efficient privacy-preserving key directory service, particularly relevant in the context of end-to-end secure messaging services. In the recent past, this problem has received significant attention both in the academic community and industry [2, 4, 10, 14, 21, 23, 25, 27]. This line of work is related to transparency logs [9, 19], but here we focus on the works that are most relevant to us.

CONIKS [21], a directory service that lets users of an end-to-end encrypted communication system verify that their keys are being correctly reported to all other users. The system is built on ideas similar to *transparency logs* [19] which are public authenticated datastructures for valid SSL/TLS certificates. EthIKS [4] implements CONIKS using the Ethereum for auditing. Catena [27] provides a more generalized infrastructure for managing application specific logs of *append only* statements using the `OP_RETURN` function of the Bitcoin blockchain and implements CONIKS using Catena. Both EthIKS [4] and Catena-CONIKS [27] use a global transaction ledger to allow all clients to agree on the same history of the directory. The original CONIKS proposal was to use a different mechanism (gossip) to achieve the same functionality. Apart from that, the core functionality and implementation of CONIKS remains unchanged in [4, 27]. But all these works provide weaker privacy than our construction and they lack any rigorous security model. In our SEEMless construction we use append-only strong accumulators (SA) in a way which is somewhat similar to the constructions in [3]. Baldmitsi et al. [3] maintain two accumulators and have an index associated with each element through which they keep track of non membership which is a similar idea to ours. However, they assume that the accumulator manager is trusted, which does not hold for us, since our goal is to detect misbehavior by the server. Considering this stronger setting of an untrusted accumulator adds additional challenges. We added several new ideas to prevent a malicious server from showing arbitrary values on query while still maintaining efficiency. We define and implement a new primitive of *append-only zero-knowledge sets* which generalizes zero-knowledge sets [6, 22] by allowing updates and parameterizing the privacy property with a leakage function. While there have been some attempts to generalize the notion of zero-knowledge sets (e.g., [13, 20]), this is the first attempt that combines both updates and leakage. Our implementation of append-only zero-knowledge sets uses a persistent Patricia Tree data structure which is reminiscent of history trees built on Merkle Trees [24]. The construction of [24] does not provide any privacy and it is still less efficient than our construction.

G AZKS DEFINITIONS

Soundness. We allow $\mathcal{A}^*$ to win the soundness game if it is able to do either of the following: Output $\mathsf{com}, \mathsf{label}, \mathsf{val}_1, \mathsf{val}_2, \pi_1, \pi_2$ such that both proofs verify for $\mathsf{val}_1 \neq \mathsf{val}_2$. Or output $\mathsf{com}_1, \ldots \mathsf{com}_n, \mathsf{label}, \mathsf{val}_1, \mathsf{val}_2, \pi_1, \pi_2, S, \pi_S$ such that π_1 verifies for $\mathsf{com}_1, \mathsf{val}_1$ for $\mathsf{val}_1 \neq \perp$ and π_2 verifies for $\mathsf{com}_n, \mathsf{val}_2$ for $\mathsf{val}_2 \neq \mathsf{val}_1$ and the update verifies for $\mathsf{com}_1, \ldots \ldots, \mathsf{com}_n$,.
In general, we want that for all PPT $\mathcal{A}^*$ algorithm there exists a negligible function $\nu()$ such that for all n, λ:

$$\begin{aligned}
\Pr[(\mathsf{com}_1, \{(\mathsf{com}_i, \pi^i)\}_{i=2}^n, \mathsf{label}, \mathsf{val}_1, \mathsf{val}_2, \pi_1, \pi_2) \\
\leftarrow \mathcal{A}^*(1^\lambda, \mathsf{pp}) : \\
\{\mathsf{ZKS.VerifyUpd}(\mathsf{com}_{i-1}, \mathsf{com}_i, \pi^i) = 1\}_{i=2}^n \\
\wedge (\mathsf{val}_1 \neq \perp) \ \wedge \ (\mathsf{val}_1 \neq \mathsf{val}_2) \\
\wedge \ \mathsf{ZKS.Verify}(\mathsf{com}_1, \mathsf{label}, \mathsf{val}_1, \pi_1) = 1 \\
\wedge \ \mathsf{ZKS.Verify}(\mathsf{com}_n, \mathsf{label}, \mathsf{val}_2, \pi_2) = 1] \leq \nu(\lambda)
\end{aligned}$$

Theorem G.1. *The construction in Section 5.2 satisfies* aZKS *soundness as defined above.*

Proof. Here we consider three cases

- The sVRF values presented for label in π_1 and π_2 are different. In this case we can directly reduce to the verifiability property of the sVRF.
- $\mathsf{val}_2 \neq \perp$ and the leaf commitments presented for label in π_1, π_2 are the same. In this case we can directly reduce to the bindind property of the commitment scheme.
- The sVRF values are the same, and either the commitments are different, or π_2 is a non-membership proof. In this case we can directly reduce to soundness of the strong accumulator.

□

Privacy. In our definition, we have an initial commitment leakage function L_{CommitDS}, a query leakage function L_{Query}, and a leakage function on the updates L_{UpdateDS}. L_{CommitDS} captures what we leak about the collection/datastore when initializing the system, L_{Query} captures what is leaked by the proofs for each query, and L_{UpdateDS} captures what we leak during an update. Note that all of these are stateful functions that share state, so e.g. L_{Query} may depend on previous CommitDS and UpdateDS queries.

We will say that an updatable ZKS is zero knowledge for leakage function $L = (L_{\mathsf{CommitDS}}, L_{\mathsf{Query}}, L_{\mathsf{UpdateDS}})$ if there exists a simulator $\mathsf{Sim} = (\mathsf{Sim}_{\mathsf{CommitDS}}, \mathsf{Sim}_{\mathsf{Query}}, \mathsf{Sim}_{\mathsf{UpdateDS}})$ such that for any PPT malicious client algorithms C^*, the outputs of the following two experiments are computationally indistinguishable:

In the real game, the adversary C^* produces an initial directory D_0, and receives the output $(\mathsf{com}, \mathsf{st}_{\mathsf{com}}) \leftarrow \mathsf{ZKS.CommitDS}(1^\lambda, \mathsf{D}_0)$. It then gets oracle access to two oracles, O_Q, and O_U. O_U is the update oracle which on input a set of updates $S = \{(\mathsf{label}_i, \mathsf{val}_i)\}$ will output $(\mathsf{com}', \mathsf{st}_{\mathsf{com},t+1}, \pi_S) \leftarrow \mathsf{ZKS.UpdateDS}(\mathsf{st}_{\mathsf{com},t}, \mathsf{D}_t, S)$. The game keeps track of the state of the datastore D_t after every update. The second oracle is O_Q, the query oracle which on query datastore version t and label_i, will output $(\pi_i, \mathsf{val}_i) \leftarrow \mathsf{ZKS.Query}(\mathsf{st}_{\mathsf{com},t}, \mathsf{D}_t, \mathsf{label}_i)$.

In the simulated game, the C^* produces an initial datastore D_0, and receives the output $\mathsf{com} \leftarrow \mathsf{Sim}_{\mathsf{CommitDS}}(1^\lambda, L_{\mathsf{CommitDS}}(\mathsf{D}_0))$. C^* then gets access to simulated versions of the two oracles. On UpdateDS queries, $\mathsf{Sim}_{\mathsf{UpdateDS}}$ emulates the update oracle O_U: it gets a leakage on the set to be updated outputs $(\mathsf{com}', \pi_S) \leftarrow \mathsf{Sim}_{\mathsf{UpdateDS}}(Ł_{\mathsf{UpdateDS}}(S))$. On Query queries, the oracle outputs val_i and $\pi_i \leftarrow \mathsf{Sim}_{\mathsf{Query}}(t, \mathsf{label}_i, \mathsf{val}_i, L_{\mathsf{Query}}(t, \mathsf{label}_i, \mathsf{val}_i))$.

Leakage. The concrete leakage for our ZKS construction is as follows:

L_{CommitDS} reveals the size of the datastore.

L_{Query} reveals when each queried item was added to the datastore.

L_{UpdateDS} reveals the number of items added. It also reveals the set Q of labels of items in this update for which there had been a previous non-membership query.

Theorem G.2. *The construction in Section 5.2 satisfies* aZKS *privacy as defined above with the specified leakage functions.*

Proof. We first define the simulator:

$\mathsf{Sim}_{\mathsf{CommitDS}}$ takes as input the size N of the data store. It uses the sVRF simulator to generate an sVRF public key. It chooses N random strings as the output of the sVRF (we will refer to them below as leaf strings), and uses the sCS simulator to form the commitments. Then it builds a tree as in the real protocol.

$\mathsf{Sim}_{\mathsf{Query}}$ takes as input the zks version t, the label label, the corresponding value val (or $\perp$ if label is not in D_t), and the update t_{prev} when label was added to the datastore. If $\mathsf{val} \neq \perp$ it chooses an unused leaf string from the t_{prev}th update, simulates the sVRF proof to show that that is the correct value for label, and simulates an opening of sCS to val. If $\mathsf{val} = \perp$, it chooses a random leaf string, simulates the sVRF proof to show that that is the correct value for label, and constructs the rest of the proof as in the real protocol. Finally, it records that that leaf string has been assigned to label. (Future Query's for label will use the same leaf string.)

$\mathsf{Sim}_{\mathsf{UpdateDS}}$ takes as input the number of items added and the set Q of new items for which there had been a previous non-membership query. For each label in Q, it looks up the leaf string that was assigned to label by $\mathsf{Sim}_{\mathsf{Query}}$, for the remaining number of items it chooses random leaf strings. For each item it also uses the sCS simultator to form the corresponding commitment. Then it performs the rest of the update algorithm as in the real protocol.

The proof that this simulator satisfies the privacy definition follows from a fairly straightforward series of games:

Game 1 : Real game

Game 2 : As in game 1, but commitments and openings are simulated. *Indistinguishable by the hiding property of the sCS.*

Game 3 : As in game 2, but the sVRF public key and sVRF proofs are generated by the sVRF simulator, and the leaf strings are chosen at random. *Indistinguishable by the simulatability property of the sVRF.*

Game 4 : Simulated game. *Identical to game 3.*

□