Privacy Preserving Verifiable Key Directories

Melissa Chase
Microsoft Research
melissac@microsoft.com

Apoorvaa Deshpande
Brown University
acdeshpa@cs.brown.edu

Esha Ghosh
Microsoft Research
esha.ghosh@microsoft.com

Abstract

In recent years, some of the most popular online chat services such as iMessage and WhatsApp have deployed end-to-end encryption to mitigate some of the privacy risks to the transmitted messages. But facilitating end-to-end encryption requires a Public Key Infrastructure (PKI), so these services still require the service provider to maintain a centralized directory of public keys. A downside of this design is placing a lot of trust in the service provider; a malicious or compromised service provider can still intercept and read users' communication just by replacing the user's public key with one for which they know the corresponding secret. A recent work by Melara et al. builds a system called CONIKS where the service provider is required to prove that it is returning a consistent for each user. This allows each user to monitor his own key and reduces some of the risks of placing a lot of trust in the service provider. New systems [EthIKS,Catena] are already being built on CONIKS. While these systems are extremely relevant in practice, the security and privacy guarantees of these systems are still based on some ad-hoc analysis rather than on a rigorous foundation. In addition, without modular treatment, improving on the efficiency of these systems is challenging. In this work, we formalize the security and privacy requirements of a verifiable key service for end-to-end communication in terms of the primitive called *Verifiable Key Directories* (VKD). Our abstraction captures the functionality of all three existing systems: CONIKS, EthIKS and Catena. We quantify the leakage from these systems giving us a better understanding of their privacy in concrete terms. Finally, we give a VKD construction (with concrete efficiency analysis) which improves significantly on the existing ones in terms of privacy and efficiency. Our design modularly builds from another primitive that we define as *append-only zero knowledge sets* (aZKS) and from append-only Strong Accumulators. By providing modular constructions, we allow for the independent study of each of these building blocks: an improvement in any of them would directly result in an improved VKD construction. Our definition of aZKS generalizes the definition of the zero knowledge set for updates, which is a secondary contribution of this work, and can be of independent interest.

1 Introduction

In recent years, the use of online chat services for communication has seen an exponential rise. However, concerns have also arisen about the security of messages sent over these chat services. A number of popular services such as Apple iMessage, WhatsApp and Signal have recently deployed end-to-end encryption [FB14, Sch15] which mitigates some of the previous security threats. But end-to-end encryption most often relies on a Public Key Infrastructure (PKI) and these services still require the service provider to maintain a centralized directory of the public keys of its registered users. To be accessible to average users, these systems need the user to store her secret key on her personal device, and cannot 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 with the PKI, with the newly generated public key. We only assume that the service provider can somehow verify that the update request comes from her.[*]

Such a system 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 some way of verifying that the service provider is indeed returning the correct keys, end-to-end encryption does not provide any protection against malicious (or coerced) service providers. 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.

Recently, the work by Melara et al. has built a system called CONIKS [MBB$^+$15] which reduces the trust on the service provider. CONIKS requires the service provider to return cryptographic proofs to prove to a user Alice that it is returning the correct value for her public key to her and other users (without leaking information about other public keys in the directory). The work of CONIKS [MBB$^+$15] has already inspired a Ethereum based implementation EthIKS [Bon16] and a Bitcoin based implementation Catena [TD17].

Privacy may not be very important in traditional PKI, where all the players are usually public entities like businesses. But in the context of private messaging, privacy is important for various reasons. Hiding users' identities may help with preventing spam messaging. Also, users change their keys primarily when they change devices, or detect that their devices/keys have been compromised, either of which may be sensitive information. The ability to track a users change in public key (which we refer to as tracing attack) can have serious security implications. 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.

While services such as CONIKS are extremely relevant in practice, the security and privacy guarantees of these systems are still based on some ad-hoc analysis rather than on a rigorous foundation. Without a formal treatment, it is hard to precisely understand the security and privacy guarantees of the system. In addition, without modular treatment, improving on the efficiency of these systems becomes extremely challenging.

1.1 Our Contributions and Techniques

We initiate the study of the primitive of *Verifiable Key Directories* (VKD), analyze its building blocks and give more efficient constructions for the building blocks and in turn, for VKD. More concretely, we summarize our contributions as follows:

Formalizing VKD We formalize the security and privacy of a key verification service in terms of the primitive 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

[*]This could be achieved 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.

Dir with the names of the users and their corresponding keys. The VKD provides different query interfaces to the users to interact with the server: 1. Alice can add her (username, key) to Dir and update it at any point. 2. Bob can query for Alice's key and verify its consistency 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. Finally, the auditors are untrusted entities that will help verify that updates have been correctly performed; as long as at least one honest auditor verifies each update, we will be able to guarantee that a misbehaving server will get caught.

Security and Privacy Definition: At a very high level, we define the soundness of a VKD to ensure that the server's responses to users' queries for Alice's key must be consistent with its responses to her key history queries. We give a simulation based definition for the privacy of a VKD system where the simulator is parameterized with a well specified leakage function $\mathcal{L}(.)$ on the state of the key directory. Informally, we require that the proofs should not leak information about Dir beyond the query answer and the updates should leak no information except $\mathcal{L}(\mathsf{Dir})$. Our framework is also general enough to capture the current implementations [MBB+15, Bon16, TD17]. We briefly describe this in Section 6.1.

The precise leakage function leads to a better understanding of the privacy guarantee of the VKD system. As a concrete example, we were able to identify a *tracing attack* in current implementations [MBB+15, Bon16, TD17] through which it might be possible to trace update times of a particular user. We explain this attack and the corresponding leakage in Section 6.1.

Building Blocks We take a modular approach in designing our VKD (Section 4). We show how a VKD can be built using (our newly defined primitive) *Append-Only Zero Knowledge Sets* (aZKS) and Append-Only Strong Accumulators (SA) in a blackbox manner.

aZKS generalizes the definition of a traditional zero-knowledge set [MRK03, CHL+05] by accounting for updates and parameterizing the zero-knowledge simulator with a well specified leakage function. The definition of aZKS (Section 3) and its *efficient* modular instantiation is a secondary contribution of our work that can be of independent interest. 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 result in an improved VKD construction.

Our VKD Construction The high level idea in our construction (Section 4) is to use aZKS along with an append-only SA that efficiently maintains 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 also allows for additional pairs to be added to the database. In a naïve construction for VKD, the server could commit to (username, public key) pairs. 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.

Note that this naïve solution would allow the server to store many different (username | version, public key) pairs for the same username, and then return one version to Alice and a different version when Bob queries for Alice's key. We address this as follows: We use *two* aZKS; one which consists of all the (username | version, key) pairs corresponding to all the updates Alice has made so far. The other 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 that, now when a user Alice updates her key, instead of deleting the old entry, server will add it to the "old" aZKS, and when Bob queries Alice's key the server will additionally prove that the entry it returns is not "old". Auditors in the system will verify the server's proofs that updates have been correctly performed, thus guaranteeing that the datastructures are indeed append-only. We have a more complex approach for preventing the server from showing Bob a higher version number, which involves storing a "marker" every 2^ith update; we defer details to the body.

Finally, the server will maintain a SA storing the two aZKS commitments at every epoch, so that as long as users and auditors eventually see the same SA value[†], we can be guaranteed that any misbehavior will be caught.

Privacy and Efficiency Improvement Our VKD construction improves upon the existing VKD constructions [MBB$^+$15, Bon16] on the efficiency and privacy aspects. We give concrete efficiency and count the cryptographic operations for our VKD construction in Section 6. In our construction we rely only on hashes and a couple of exponentiations. Our proof sizes are logarithmic in the total number of users. Also we only leak the number of users who update their keys each epoch; beyond that all parties just learn the responses to their queries. Finally, the work that Alice must do to verify that the server has correctly returned all of the times her key was updated is now proportional only to the number of updates she has made, rather than the total number of epochs elapsed.

1.2 Related Work

Our work is inspired by the recent line of work of started by CONIKS [MBB$^+$15], 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 that of *transparency logs* [LLK13] which are public authenticated datastructures for valid SSL/TLS certificates. Recently, Eskandarian et al. [EMBB17] address some of the privacy challenges in transparency logs.

The directory structure in CONIKS has already seen an Ethereum and Bitcoin based implementation [Bon16, TD17]. These implementations 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 [Bon16, TD17]. These works do not formalize or analyze the security of their constructions. Our formalization of VKD model is generic enough to capture these implementions [MBB$^+$15, Bon16, TD17] and also quantify their leakage in terms of privacy.

In our VKD construction we use append-only strong accumulators (SA) in a way which is somewhat similar to the constructions in [BCD$^+$17]. Baldmitsi et al. [BCD$^+$17] 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 also define a new primitive of *append-only zero-knowledge sets* which generalizes zero-knowledge sets [MRK03, CHL$^+$05] by allowing updates and parameterizing the privacy property

[†]As in CONIKS, we can achieve this either with a gossip protocol or by posting the SA on a public blockchain.

with a leakage function. While there have been some attempts to generalize the notion of zero-knowledge sets (e.g., [KZG10, Lis05a]), this is the first attempt that combines both updates and leakage.

A related line of work is persistent storage at an untrusted server [CW09] in which servers store documents in their memory with timestamps. Oprea *et. al.* [OB09] formalize the security of a time-stamping scheme for archiving documents which has efficient proofs for existence or non-existence of documents at specific times. Crosby *et. al.* [CW09] also construct a system of timestamping documents that is secure against an untrusted logger, i.e., server. But these systems do not support a full-fledged key directory service. Moreover, they do not have any privacy guarantees against the verifiers or the auditors in the system.

1.3 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 construction of VKD starting with an overview of the same. In Section 5, we give concrete instantiations of aZKS along with description of other primitives used in the construction. Finally in Section 6 we describe how existing implementations [MBB$^+$15, Bon16] can be expressed as instantiations of VKD, we analyze the concrete costs of them against our construction.

2 Verifiable Key Directory (VKD)

In this section, we will define the primitive *Verifiable Key Directories* (VKD) and formalize its properties. The goal of a VKD is to capture the functionality and security of a privacy-preserving verifiable key service such as CONIKS or EthIKS [MBB$^+$15, Bon16].

A VKD consists of three 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). The VKD provides the following query interface to the users. For the ease of exposition, let Alice and Bob be two users:

1. Alice can add her (username, key), i.e., (label=username, val=key) to Dir.
2. She can also 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 updates over time (VKD.KeyHist).
4. Bob can also query for the key corresponding to username Alice (VKD.Query) at the current time.

The server applies the updates from its users (of type 1 and 2 described above) in batches, and publishes the latest commitment 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. It 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 monotonic and can consist of many independent audit steps. As long as some honest auditor executes each audit step, security will be guaranteed.

The server also produces proofs for $\mathsf{VKD.Query}$ and $\mathsf{VKD.KeyHist}$. At a very high level, the users verify the proofs ($\mathsf{VKD.QueryVer}, \mathsf{VKD.HistVer}$) 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 $\mathsf{VKD.Publish}$ need to be privacy-preserving. Since the auditors do not have to be trusted with any private information, anyone can become an auditor.

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 $k \geq k_0$, $\nu(k) < k^{-c}$. We denote by $r \xleftarrow{\$} S$ the operation that r is uniformly at random drawn from the set S. An algorithm $\mathcal{A}$ is said to have oracle access to machine $\mathcal{O}$ if $\mathcal{A}$ can write an input for $\mathcal{O}$ 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 $\mathcal{A}^{\mathcal{O}}$. For the rest of the sections, we will use ($\mathsf{label}, \mathsf{val}$) and (username,public key) interchangeably.

Definition 1. A Verifiable Key Directory comprises of the algorithms ($\mathsf{VKD.Setup}, \mathsf{VKD.Gen}, \mathsf{VKD.Publish}, \mathsf{VKD.Query}, \mathsf{VKD.QueryVer}, \mathsf{VKD.KeyHist}, \mathsf{VKD.HistVer}, \mathsf{VKD.Audit}$) described as follows:

System Setup:

$\triangleright$ $\hat{\mathsf{pp}} \leftarrow \mathsf{VKD.Setup}(1^\lambda)$: This algorithm takes the security parameter as input and outputs public parameters pp for the scheme and must be run by a *trusted party*[‡].

$\triangleright$ $(\mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_0, \mathsf{Dir}_0) \leftarrow \mathsf{VKD.Gen}(\mathsf{pp})$: This algorithm takes the public parameters and outputs a public key, an initial state which is private to the server and an initial directory Dir_0. For simplifying notation we let $\mathsf{pp} = (\hat{\mathsf{pp}}, \mathsf{PK}_{\mathsf{Gen}})$

Periodic Publish:

$\triangleright$ $(\mathsf{com}_t, \mathsf{acc}_t, \Pi_t^{\mathsf{Upd}}, \mathsf{st}_t, \mathsf{Dir}_t) \leftarrow \mathsf{VKD.Publish}(\mathsf{pp}, \mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)$: This algorithm takes in the public parameters, 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, an updated datastructure acc_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. For simplifying notation, we let $\mathsf{pub}_t = (\mathsf{com}_t, \mathsf{acc}_t, \Pi_t^{\mathsf{Upd}})$.

Querying for a Label:

$\triangleright$ $(\mathsf{val}, \pi, \alpha) \leftarrow \mathsf{VKD.Query}(\mathsf{pp}, \mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: This algorithm takes the public parameters, 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 and version number α of the value. By version number we mean the number of times label has been updated.

[‡]In the Random Oracle model, this algorithm might be as simple as choosing some hash functions

$\triangleright$ $1/0 \leftarrow \mathsf{VKD.QueryVer}(\mathsf{pp}, \mathsf{com}, \mathsf{label}, (\mathsf{val}, \pi, \alpha))$: This algorithm takes the public parameters, the commitment with respect to some epoch, a label, value pair and the corresponding proof and version for the label. It verifies the proof and the correctness of the label, value pair with respect to com and returns a boolean value indicating success or failure.

Checking Consistency of Versions:

$\triangleright$ $(\{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{VKD.KeyHist}(\mathsf{pp}, \mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: This algorithm takes in the public parameters, its internal state, the directory at current time t and the 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 and the resulting val's, along with a proof Π^{Ver}.

$\triangleright$ $1/0 \leftarrow \mathsf{VKD.HistVer}(\mathsf{pp}, \mathsf{com}_t, \mathsf{acc}_t, \mathsf{label}, \Pi^{\mathsf{Ver}}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n)$: This algorithm takes the public parameters, the commitment and chain information output by server for current time t, a label, $\{(\mathsf{val}_i, t_i)\}_{i=1}^n$ which are the values and times corresponding to all the updates of the label and its versions proof. It returns a boolean value indicating success or failure.

Auditing the VKD:

$\triangleright$ $1/0 \leftarrow \mathsf{VKD.Audit}(\mathsf{pp}, t_1, t_n, \{\mathsf{pub}_t\}_{t=t_1}^{t_n})$: This algorithm takes the public parameters, the epochs t_1 and t_n between which audit is being done, the server's publish 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 Remark 2.

We require the following properties from a Verifiable Key Directory:

– **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 $\mathsf{val}_1, \ldots, \mathsf{val}_n$: their version proof with $\{(\mathsf{val}_i, t_i)\}_{i=1}^n$ and Π^{Ver} should verify at t_n, the query proof for the label at any $t_j \leq t^* < t_{j+1}$ should verify with respect to the value consistent with the versions proof at t_j which is val_j and the audit from epochs t_1 to t_n should verify.

  Hence for all possible labels, for all update sets $S_1, \ldots, S_T$ such that $(\mathsf{label}, \mathsf{val}_i) \in \{S_{t_i}\}_{i=1}^n$ and $\forall t^*$,

$$\Pr[\hat{\mathsf{pp}} \leftarrow \mathsf{VKD.Setup}(1^\lambda)\ ;\ (\mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_0, \mathsf{Dir}_0) \leftarrow \mathsf{VKD.Gen}(\mathsf{pp})\ ;\ \{(\mathsf{pub}_t, \mathsf{st}_t, \mathsf{Dir}_t) \leftarrow \mathsf{VKD.Publish}(\mathsf{pp}, \mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)\}_{t=t_1}^{t_n}\ ;\ (\{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{VKD.KeyHist}(\mathsf{pp}, \mathsf{st}_{t_n}, \mathsf{Dir}_{t_n}, \mathsf{label})\ ;\ (\pi, \mathsf{val}, \alpha) \leftarrow \mathsf{VKD.Query}(\mathsf{pp}, \mathsf{st}_{t^*}, \mathsf{Dir}_{t^*}, \mathsf{label}) : \mathsf{VKD.HistVer}(\mathsf{pp}, \mathsf{com}_t, \mathsf{acc}_t, \mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}) = 1 \text{ for } t = t_n\ \wedge\ \mathsf{VKD.QueryVer}(\mathsf{pp}, \mathsf{com}_{t^*}, \mathsf{label}, \mathsf{val}, \pi, \alpha) = 1\ \wedge\ t_j \leq t^* < t_{j+1}\ \wedge\ (\mathsf{val} = \mathsf{val}_j)\ \wedge\ (\alpha = j)\ \wedge\ \mathsf{VKD.Audit}(\mathsf{pp}, t_1, t_n, \{\mathsf{pub}_t\}_{t=t_1}^{t_n}) = 1] = 1$$

  Note that for $\mathsf{KeyHist}$ and $\mathsf{HistVer}$, 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 that Alice verified. Thus soundness is derived from all of $\mathsf{VKD.Publish}$, $\mathsf{VKD.QueryVer}$, $\mathsf{VKD.HistVer}$ and $\mathsf{VKD.Audit}$.

  More 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, chain information and update proofs Π^{Upd} for all times between t_1 to t_n and query proof $(\pi, \mathsf{val}, \alpha)$ for some t^* for $\mathsf{val} \neq \mathsf{val}_j$.

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

$$\begin{aligned} \Pr[\hat{\mathsf{pp}} \leftarrow \mathsf{VKD.Setup}(1^\lambda)\ ;\ (\mathsf{label}, (\{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}), (\mathsf{PK}_{\mathsf{Gen}}, \{\mathsf{pub}_t\}_{t=t_1}^{t_n}, t^*, j, \\ (\pi, \mathsf{val}, \alpha)) \leftarrow S^*(\mathsf{pp})\ :\ \mathsf{VKD.QueryVer}(\mathsf{pp}, \mathsf{com}_{t^*}, \mathsf{label}, \mathsf{val}, \pi, \alpha) = 1 \\ \mathsf{VKD.Audit}(\mathsf{pp}, t_1, t_n, \{\mathsf{pub}_t\}_{t=t_1}^{t_n}) = 1\ \wedge\ \mathsf{VKD.HistVer}(\mathsf{pp}, \mathsf{com}_t, \mathsf{acc}_t, \mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n, \\ \Pi^{\mathsf{Ver}}) = 1 \text{ for } t = t_n\ \wedge\ (\mathsf{val} \neq \mathsf{val}_j)\ \wedge\ (t_j \leq t^* < t_{j+1})] \leq \nu(\lambda) \end{aligned}$$

  Remark 1. 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 her key history that the server maintains to make sure it is consistent with the the actual key that she stored.

– **$\mathcal{L}$-Privacy:** The privacy guarantee of a VKD system is that the outputs of Query, $\mathsf{HistVer}$ or Audit should not reveal anything beyond the answer and a well defined leakage on the state of the directory. The leakage is characterized by a leakage function $L(.)$. Also, the proofs for each of these queries should be simulatable given the output of $L(.)$ and the query answer.

  We will say that a VKD is private if there exists a simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4)$ and a leakage function $L(.)$ such that for any PPT client C^*, the outputs of the following two experiments are computationally indistinguishable:

  Real:

$$\mathsf{pp} \leftarrow \mathsf{VKD.Setup}(1^\lambda)\ ;\ (\mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_0, \mathsf{Dir}_0) \leftarrow \mathsf{VKD.Gen}(\mathsf{pp})\ :\ C^{*O_P, O_\pi, O_{\Pi^{\mathsf{Ver}}}(\mathsf{st}_t, -)}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}) = 1$$

  Simulated:

$$(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_1) \leftarrow \mathcal{S}_1(1^\lambda)\ :\ C^{*\mathcal{S}_2(\mathsf{st}'_t, L(S_t), -), \mathcal{S}_3, \mathcal{S}_4(\mathsf{st}'_t, -)}(\mathsf{st}_C, \mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}) = 1$$

O_P is the publish oracle which on input update set S_t, outputs $(\mathsf{com}_t, \mathsf{acc}_t, \Pi_t^{\mathsf{Upd}})$ as computed by $\mathsf{VKD.Publish}()$. O_π is the proofs oracle which on query a label label_i, will output $(\pi_i, \mathsf{val}_i, \alpha_i) \leftarrow \mathsf{VKD.Query}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_t, \mathsf{Dir}, \mathsf{label}, t)$. $O_{\Pi^{\mathsf{Ver}}}$ is the key versions oracle, which on query, label, $(\{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}) \leftarrow \mathsf{VKD.KeyHist}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$.

For the simulated game, $\mathcal{S}_2$ emulates the publish oracle O_P and it gets some leakage on the update set S_t given by $L(S_t)$. $\mathcal{S}_3$ emulates the proofs oracle O_π. It gets the $(\mathsf{val}_i, \mathsf{label}_i, \alpha_i)$ for the label_i queried and answers the membership queries by producing the proof. $\mathcal{S}_4$ emulates the key versions oracle O_V and outputs proof given $(\mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^n)$.

3 Building Blocks for VKD

In this section we introduce the primitive of *Append-Only Zero Knowledge Set* (aZKS) and *Append-Only Strong Accumulator* (SA) which will be used as building blocks for the construction of a Verifiable Key Directory (VKD). By append-only we mean that the only updates we allow are adding new elements.

3.1 Append-Only Zero Knowledge Sets

An aZKS generalizes the privacy properties of a traditional zero-knowledge set [MRK03] by accounting for append-only updates and characterizing the set with a leakage function. In our definition, we have a setup leakage function $L_1()$ and a leakage function on the updates $L_2()$. 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 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. In [MRK03], 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.

Definition 2. Append-Only Zero Knowledge Set comprises of the algorithms ($\mathsf{ZKS.Setup}$, $\mathsf{ZKS.Gen}$, $\mathsf{ZKS.CommitDS}$, $\mathsf{ZKS.Query}$, $\mathsf{ZKS.Verify}$, $\mathsf{ZKS.UpdateDS}$, $\mathsf{ZKS.VerifyUpd}$) described as follows:

$\triangleright$ $\hat{\mathsf{pp}} \leftarrow \mathsf{ZKS.Setup}(1^\lambda)$: This algorithm takes the security parameter and outputs public parameters pp.

$\triangleright$ $(\mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_{\mathsf{Gen}}) \leftarrow \mathsf{ZKS.Gen}(\hat{\mathsf{pp}})$: This algorithm takes the public parameters and outputs a public key and a state that the prover can pass on to subsequent algorithms. Here prover generates a public key primitive and passes on the trapdoor information as $\mathsf{st}_{\mathsf{Gen}}$. The previous definitions of zero knowledge sets [MRK03, CHL+05] can accommodate Gen inside of $\mathsf{CommitDS}$, but separating the two allows for a stronger adversary who can pick the datastore after seeing the public key. For simplifying notation, let $\mathsf{pp} = (\hat{\mathsf{pp}}, \mathsf{PK}_{\mathsf{Gen}})$.

$\triangleright$ $(\mathsf{com}, \mathsf{st}_{\mathsf{com}}) \leftarrow \mathsf{ZKS.CommitDS}(\mathsf{pp}, \mathsf{st}_{\mathsf{Gen}}, \mathsf{D})$: This algorithm takes the public parameters, private state and the datastore to commit to, 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 $(\mathsf{label}, \mathsf{val})$ pairs.

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

▷ $1/0 \leftarrow \mathsf{ZKS.Verify}(\mathsf{pp}, \mathsf{com}, \mathsf{label}, \mathsf{val}, \pi)$: This algorithm takes the public key, its proof, a (label, value) pair and a commitment by $\mathsf{ZKS.CommitDS}$ and verifies the proof and the consistency of the (label, value) pair with the commitment using the proof; returns a boolean indicating success or failure.

▷ $(\mathsf{com}', \mathsf{st}'_{\mathsf{com}}, \mathsf{D}', \pi_S) \leftarrow \mathsf{ZKS.UpdateDS}(\mathsf{pp}, \mathsf{st}_{\mathsf{com}}, \mathsf{D}, S)$: This algorithm takes in the public parameters, server public key, 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.

▷ $0/1 \leftarrow \mathsf{ZKS.VerifyUpd}(\mathsf{pp}, \mathsf{com}, \mathsf{com}', \pi_S)$: This algorithm takes in the public parameters, the server public key, two commitments to the datastore before and after an update and a proof π_S proving correctness of the update. It outputs a boolean to indicate the success or failure of the verification.

We require the following security properties from an *append-only* ZKS:

Completeness: For all security parameters λ, for all D_0 whose size is polynomial in λ, for all n and for all update sets $(S_1, \ldots, S_n)$ and for every label,

$$\begin{aligned}
\Pr[\hat{\mathsf{pp}} \leftarrow \mathsf{ZKS.Setup}(1^\lambda); (\mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_{\mathsf{Gen}}) \leftarrow \mathsf{ZKS.Gen}(\hat{\mathsf{pp}})\\
; (\mathsf{com}_0, \mathsf{st}^0) \leftarrow \mathsf{ZKS.CommitDS}(\mathsf{pp}, \mathsf{st}_{\mathsf{Gen}}, \mathsf{D}_0); \{(\mathsf{com}_i, \mathsf{st}^i,\\
\mathsf{D}_i, \mathsf{com}_S^i, \pi_S^i) \leftarrow \mathsf{ZKS.UpdateDS}(\mathsf{pp}, \mathsf{st}^{i-1}, \mathsf{D}_{i-1}, S_i)\}_{i=1}^n\\
; \ \{(\pi_i, \mathsf{val}_i) \leftarrow \mathsf{ZKS.Query}(\mathsf{pp}, \mathsf{st}^i_{\mathsf{com}}, \mathsf{D}_i, \mathsf{label})\}_{i=1}^n\\
: \ \{\mathsf{ZKS.VerifyUpd}(\mathsf{pp}, \mathsf{com}_{i-1}, \mathsf{com}_i, \mathsf{com}_S^i, \pi_S^i) = 1\}_{i=1}^n\\
\wedge \ \{\mathsf{ZKS.Verify}(\mathsf{pp}, \mathsf{com}_i, \mathsf{label}, \mathsf{val}, \pi_i) = 1\}_{i=0}^n] = 1
\end{aligned}$$

Soundness: For soundness we want to capture two things: First, a malicious server $\mathcal{A}^*$ algorithm should not be able to produce 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. 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 $\mathsf{com}_1, \mathsf{com}_2, \mathsf{label}, \mathsf{val}_1, \mathsf{val}_2, \pi_1, \pi_2, S, \pi_S$ such that π_1 verifies for $\mathsf{com}_1, \mathsf{val}_1$ for $\mathsf{val}_1 \neq \perp$ and π_2 verifies for $\mathsf{com}_2, \mathsf{val}_2$ for $\mathsf{val}_2 \neq \mathsf{val}_2$ and the update verifies for $\mathsf{com}_1, \mathsf{com}_2, S, \pi_S$.
In general, we want that for all PPT $\mathcal{A}^*$ algorithm there exists a negligible function $\nu()$ such that for all n, λ:

$$\begin{aligned}
\Pr[\hat{\mathsf{pp}} \leftarrow \mathsf{ZKS.Setup}(1^\lambda) \ ; \ (\{\mathsf{PK}_{\mathsf{Gen}}, \mathsf{D}_0, \mathsf{com}_0, (\mathsf{com}_i, \mathsf{D}_i, \mathsf{com}_S^i, \pi_S^i)\}_{i=1}^n, j^*, \mathsf{label},\\
\mathsf{val}_1, \mathsf{val}_2, \pi_1, \pi_2) \leftarrow \mathcal{A}^*(1^\lambda, \mathsf{pp}) : \{\mathsf{ZKS.VerifyUpd}(\mathsf{pp}', \mathsf{com}_{i-1}, \mathsf{com}_i, \mathsf{com}_S^i, \pi_S^i) = 1\}_{i=1}^n\\
\wedge \ j^* \in [n] \ \wedge \ \mathsf{val}_1 \neq \perp \ \wedge \ \mathsf{val}_1 \neq \mathsf{val}_2 \ \wedge \ \mathsf{ZKS.Verify}(\mathsf{pp}', \mathsf{com}_{j^*}, \mathsf{label}, \mathsf{val}_1, \pi_1) = 1 \ \wedge\\
\Big(\mathsf{ZKS.Verify}(\mathsf{pp}', \mathsf{com}_{j^*}, \mathsf{label}, \mathsf{val}_2, \pi_2) = 1 \ \vee \ \mathsf{ZKS.Verify}(\mathsf{pp}', \mathsf{com}_{j^*+1}, \mathsf{label}, \mathsf{val}_2, \pi_2) = 1\Big)] \leq \nu(\lambda)
\end{aligned}$$

where $\mathsf{pp} = (\hat{\mathsf{pp}}, \mathsf{PK_{Gen}})$ for notational convenience.

Zero-Knowledge with Leakage: We will say that a set is zero knowledge with respect to updates if there exists a simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Sim}_3, \mathsf{Sim}_4)$ and a leakage function $L = (L_1, L_2)$ such that for any PPT malicious client algorithms $\mathcal{C}^*$, the outputs of the following two experiments are computationally indistinguishable:
Real:

$$\mathsf{pp} \leftarrow \mathsf{ZKS.Setup}(1^\lambda)\ ;\ (\mathsf{PK_{Gen}}, \mathsf{st_{Gen}}) \leftarrow \mathsf{ZKS.Gen}(\mathsf{pp}); (\mathsf{D}, \mathsf{st}_C) \leftarrow \mathcal{C}^*(1^\lambda, \mathsf{pp}, \mathsf{PK_{Gen}})\ ;$$
$$(\mathsf{com}, \mathsf{st}_P) \leftarrow \mathsf{ZKS.CommitDS}(1^\lambda, \mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{st_{Gen}}, \mathsf{D})\ :\ \mathcal{C}^{*O_Q(\mathsf{pp},\mathsf{PK_{Gen}},\mathsf{st}_P,-),O_U(\mathsf{pp},\mathsf{PK_{Gen}},\mathsf{st_{com}},\mathsf{D},-)}(\mathsf{st}_C) = 1$$

Simulated:

$$(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{st}_1) \leftarrow \mathsf{Sim}_1(1^\lambda)\ ;\ (\mathsf{D}, \mathsf{st}_C) \leftarrow \mathcal{C}^*(1^\lambda, \mathsf{pp}, \mathsf{PK_{Gen}})\ ;$$
$$(\mathsf{com}, \mathsf{st}_2) \leftarrow \mathsf{Sim}_2(1^\lambda, L_1(D), \mathsf{st}_1)\ :\ \mathcal{C}^{*\mathsf{Sim}_3(\mathsf{st}_2,-),\mathsf{Sim}_4(\mathsf{st}_2,-)}(\mathsf{st}_C) = 1$$

We have oracle O_Q, the query oracle which on query label_i, will output $(\pi_i, \mathsf{val}_i) \leftarrow \mathsf{ZKS.Query}(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{st_{com}}, \mathsf{D}, \mathsf{label}_i)$ and O_U is the update oracle which on getting a set of updates $S = \{(\mathsf{label}_i, \mathsf{val}_i)\}$ as input will first check that S is an allowed update set and output $(\mathsf{com}', \mathsf{st}'_{\mathsf{com}}, \pi_S) \leftarrow \mathsf{ZKS.UpdateDS}$ $(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{st_{Gen}}, \mathsf{D}, S)$.

For the simulated game, Sim_3 answers to membership queries and outputs $(\pi_i, \mathsf{val}_i) \leftarrow \mathsf{Sim}_3(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{label}_i, \mathsf{val}_i, \mathsf{st}_2)$ and Sim_4 emulates the update oracle O_U. Sim_4 gets a leakage on the set to be updated if it is an allowed update set and outputs $(\mathsf{com}', \mathsf{st}'_S, \pi_S) \leftarrow \mathsf{Sim}_4(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{st}_2, L_2(S))$

3.2 Append-Only Strong Accumulator:

We can think of append-only SA as an append-only aZKS with completeness and soundness and *without* privacy requirement.

Definition 3. An Append-Only Strong Accumulator [CHKO08] [§] comprises of the algorithms $(\mathsf{SA.Setup}, \mathsf{SA.CommitDS}, \mathsf{SA.Query}, \mathsf{SA.Verify}, \mathsf{SA.UpdateDS}, \mathsf{SA.VerifyUpd})$ described as follows:

$\mathsf{pp} \leftarrow \mathsf{SA.Setup}(1^\lambda)$: This algorithm takes the security parameter as input and outputs public parameters pp for the scheme.

$\mathsf{com} \leftarrow \mathsf{SA.CommitDS}(\mathsf{pp}, \mathsf{D})$: This algorithm takes in the public parameters and a datastore and produces a commitment to the datastore. We will generally think of the datastore D to be a set of $(\mathsf{label}, \mathsf{val})$ pairs.

$(\pi, \mathsf{val}) \leftarrow \mathsf{SA.Query}(\mathsf{pp}, \mathsf{D}, \mathsf{label})$: This algorithm takes the public parameters, the datastore and a query label and returns the corresponding value, if it is present in the directory and a proof of membership/non-membership.

$1/0 \leftarrow \mathsf{SA.Verify}(\mathsf{pp}, \mathsf{com}, \mathsf{label}, \mathsf{val}, \pi)$: This algorithm takes the public parameters, a (label, value) pair, its proof and verifies the proof and the consistency of the (label, value) pair with the commitment using the proof. This algorithm returns a boolean value indicating success or failure.

[§][CHKO08] define a static version of strong accumulator. We add algorithms to account for updates and to verify the correctness of updates

$(\mathsf{com}', \mathsf{D}', S, \pi_S) \leftarrow \mathsf{SA.UpdateDS}(\mathsf{pp}, \mathsf{D}, S)$: This algorithm takes in the public parameters, the current datastore and a set $S = \{(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_k, \mathsf{val}_k)\}$ describing the updates. It outputs an updated commitment to the datastore, an updated version of the datastore and proof π_S about the update.

$0/1 \leftarrow \mathsf{SA.VerifyUpd}(\mathsf{pp}, \mathsf{com}, \mathsf{com}', S, \pi_S)$: This algorithm takes in the public parameters, two commitments to the datastore before and after an update, the update proof π_S and outputs a boolean to indicate the success or failure of the verification.

We require the following security properties from a Strong Accumulator:

- **Completeness:** For all security parameters λ, for all D_0 whose size is polynomial in λ, for all n and for all update sets $(S_1, \ldots, S_n)$ and for every label,

$$\begin{aligned} \Pr[&\mathsf{pp} \leftarrow \mathsf{SA.Setup}(1^\lambda); \mathsf{com}_0 \leftarrow \mathsf{SA.CommitDS}(\mathsf{pp}, \mathsf{D}_0) \\ &;\ \{(\mathsf{com}_i, \mathsf{D}_i, S_i, \pi_S^i) \leftarrow \mathsf{SA.UpdateDS}(\mathsf{pp}, \mathsf{D}_{i-1}, S_i)\}_{i=1}^n \\ &;\ \{(\pi_i, \mathsf{val}_i) \leftarrow \mathsf{SA.Query}(\mathsf{pp}, \mathsf{D}_i, \mathsf{label})\}_{i=1}^n \\ &;\ \{\mathsf{SA.VerifyUpd}(\mathsf{pp}, \mathsf{com}_{i-1}, \mathsf{com}_i, S_i, \pi_S^i) = 1\}_{i=1}^n \\ &\wedge\ \{\mathsf{SA.Verify}(\mathsf{pp}, \mathsf{com}_i, \mathsf{label}, \mathsf{val}, \pi_i) = 1\}_{i=0}^n] = 1 \end{aligned}$$

- **Soundness:** For soundness we want to capture two things: First, a malicious server S^* algorithm should not be able to produce two verifying proofs for two different values for the same label with respect to a com. Second, since the SA is append-only, a malicious server should not be able to change or delete an existing label. We allow S^* 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 $\mathsf{com}_1, \mathsf{com}_2, \mathsf{label}, \mathsf{val}_1, \mathsf{val}_2, \pi_1, \pi_2, S, \pi_S$ such that π_1 verifies for $\mathsf{com}_1, \mathsf{val}_1$ for $\mathsf{val}_1 \neq \perp$ and π_2 verifies for $\mathsf{com}_2, \mathsf{val}_2$ for $\mathsf{val}_2 \neq \mathsf{val}_2$ and the update verifies for $\mathsf{com}_1, \mathsf{com}_2, S, \pi_S$.

  More generally, we want that for all PPT S^* algorithm there exists a negligible function $\nu()$ such that for all n, λ:

$$\begin{aligned} &\Pr[\mathsf{pp} \leftarrow \mathsf{SA.Setup}(1^\lambda)\ ;\ (\mathsf{D}_0, \mathsf{com}_0, \{(\mathsf{com}_i, \mathsf{D}_i, S_i, \pi_S^i)\}_{i=1}^n, j^*, \mathsf{label}, \\ &\mathsf{val}_1, \mathsf{val}_2, \pi_1, \pi_2) \leftarrow S^*(1^\lambda, \mathsf{pp})\ :\ \{\mathsf{SA.VerifyUpd}(\mathsf{pp}, \mathsf{com}_{i-1}, \mathsf{com}_i, S_i, \pi_S^i) = 1\}_{i=1}^n\ \wedge \\ &j^* \in [n]\ \wedge\ \mathsf{val}_1 \neq \perp\ \wedge\ \mathsf{val}_1 \neq \mathsf{val}_2\ \wedge\ \mathsf{SA.Verify}(\mathsf{pp}, \mathsf{com}_{j^*}, \mathsf{label}, \mathsf{val}_1, \pi_1) = 1\ \wedge \\ &\mathsf{SA.Verify}(\mathsf{pp}, \mathsf{com}_{j^*}, \mathsf{label}, \mathsf{val}_2, \pi_2) = 1\ \vee\ \mathsf{SA.Verify}(\mathsf{pp}, \mathsf{com}_{j^*+1}, \mathsf{label}, \mathsf{val}_2, \pi_2) = 1)] \leq \nu(\lambda) \end{aligned}$$

4 VKD Construction

In this section, we will describe a construction of VKD from *append-only* zero-knowledge sets (aZKS) (as in Definition 3.1) and an *append-only* Strong Accumulator. 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

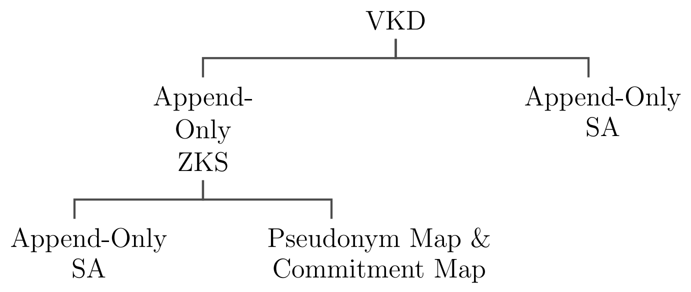
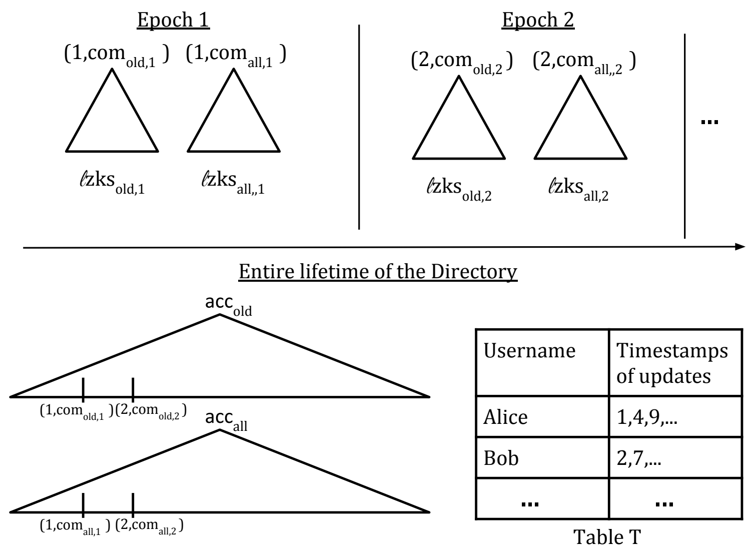


Figure 1: High-level Description of VKD with Building Blocks


Epoch 1
$(1, com_{old,1})$ $(1, com_{all,1})$
$\ell zks_{old,1}$ $\ell zks_{all,,1}$
Epoch 2
$(2, com_{old,2})$ $(2, com_{all,,2})$
$\ell zks_{old,2}$ $\ell zks_{all,2}$
...
Entire lifetime of the Directory
acc_{old}
$(1, com_{old,1})(2, com_{old,2})$
acc_{all}
$(1, com_{all,1})$ $(2, com_{all,2})$
Table T

Username	Timestamps of updates
Alice	1,4,9,...
Bob	2,7,...
...	...

Figure 2: Internal state of the Server over Lifetime of the Directory

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. 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 $(\mathsf{label}|\ 1, \mathsf{val})$ 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 all the times of update of each label.

The server also maintains and updates two public append-only SA: "all" SA on the "all" aZKS com's and "old" SA on the "old" aZKS com's. We denote the commitments corresponding to the append-only SA as $\mathsf{acc} = (\mathsf{acc}_{\mathsf{all}}, \mathsf{acc}_{\mathsf{old}})$. In the setup phase, the parameters of the aZKS, SA. and T are initialized. In Figure 2 we give a diagrammatic representation the server's internal storage.

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 α, else sets

$\alpha = 0$. It adds a new entry ($\mathsf{label} \mid \alpha+1, \mathsf{val}$) to the "all" aZKS and also adds ($\mathsf{label} \mid \alpha, \mathsf{val}_{\mathsf{old}}$) to the "old" aZKS for the label's previous value $\mathsf{val}_{\mathsf{old}}$ if it exists i.e. for $\alpha > 0$. If the new version $\alpha + 1 = 2^i$ for some i, then the server adds a marker entry ($\mathsf{label} \mid \mathsf{mark} \mid i$,"marker") to the "all" aZKS. The server publishes commitments to both the aZKS, also updates both "all" and "old" SA with $(t, \mathsf{com}_{\mathsf{all},t})$ and $(t, \mathsf{com}_{\mathsf{old},t})$ to get updated $(\mathsf{acc}_{\mathsf{all}}, \mathsf{acc}_{\mathsf{old}})$ respectively. It also publishes a proof Π^{Upd} which contains the update proofs with respect to $\mathsf{com}_{\mathsf{all},t-1}, \mathsf{com}_{\mathsf{all},t}$ and $\mathsf{com}_{\mathsf{old},t-1}, \mathsf{com}_{\mathsf{old},t}$

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 ($\mathsf{label} \mid \alpha, \mathsf{val}$) in the "all" aZKS. Second is the membership proof of the most recent marker entry ($\mathsf{label} \mid \mathsf{mark} \mid a$) for $\alpha \geq 2^a$. And third is non membership proof of $\mathsf{label} \mid \alpha$ in the "old" aZKS. Proof 2 ensures that Bob is not getting a value higher than Alice's current version and proof 3 ensures that Bob is not getting an old version for Alice's label.

Checking Consistency of Versions: 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 and to make sure that the server only gives the latest version, we incorporate several proofs and checks in the construction. In particular, we have a way for clients to verify that the server is not giving out an incorrect version of its key.

Each time a label is updated a new entry is created for the new version and possibly new marker entry in the "all" aZKS. Server also keeps adding older versions to the "old" aZKS. All these three types of entries make it possible to check that the server has been giving the correct version value all along. If at an epoch t, Alice's label is on version α for some val, she needs to check that the previous version $\alpha - 1$ is in the "old" aZKS. Let $2^a \leq \alpha < 2^{a+1}$ for some a. Alice needs to check that no marker entries from $a+1$ until $\log t$ are present, so that the server cannot give a later version after 2^{a+1}. Alice should also check non membership of all versions from $\alpha + 1$ upto $2^{a+1} - 1$ in the "all" aZKS. Recall that Bob checks that marker entry ($\mathsf{label} \mid \mathsf{mark} \mid a$) for $\alpha \geq 2^a$ has been added to the 'all" aZKS before accepting version α. If Alice verifies her key history every time she updates, then it is sufficient for her to remember the last time she updated her key (which is natural and easy to remember). But even if she doesn't, Alice will still be able to lazily check that the server has always been giving consistent values for her key for all times until now as long as she remembers the timestamps at which she updated her key.

Auditing: Auditors will audit the commitments and proofs broadcasted by the server to make sure that no entries ever get deleted in either aZKS. They do so by verifying all the update proofs Π^{Upd} output by the server. They also check that at each timestamp the appropriate (t, com_t) is added to both "all" and "old" SA. Note that, while the Audit interface gives a monotonic audit algorithm, our audit is just checking the updates between each adjacent pair of $(\mathsf{acc}, \mathsf{com}_t)$, so it can be performed by many auditors in parallel. For the security of the system, it is sufficient to have at least one honest auditor perform audits over all the adjacent pairs until the current epoch.

Remark 2 (Implementing Audits)**.** The mechanism that checks that all the auditors see consistent versions of published acc and the users see the same acc as the auditors at epoch t can be implemented through gossip between the auditors as in CONIKS [MBB+15] or through using a distributed global append-only log such as Ethereum blockchain [Bon16] or Bitcoin blockchain [TD17]. We do not enforce either of the mechanisms and leave it on the implementation.

Security and Privacy Guarantees: At any epoch, as long as Alice has checked up to her latest key versions and as long as at least one honest auditor has audited each the updates published by the server, Bob will get the latest version of Alice's key. Moreover, the query proofs and the key history proofs do not leak any information beyond the answer to the queries because of the guarantees of the aZKS. The digests published by the server do not leak anything beyond $L(S) = (n, u)$ where n is the number of new labels to be added in the VKD plus the number of existing labels that are being updated and u is the the number of existing labels that have reached a version 2^i for some i. 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 the old pseudonyms is being updated, avoids timing links between new and old pseudonyms, and hides the lifespan of each key.

VKD CONSTRUCTION:

▷ $\mathsf{VKD.Setup}(1^\lambda)$: Let $\mathsf{pp_{ZKS.}} \leftarrow \mathsf{ZKS.Setup}(1^\lambda)$ and $\mathsf{pp_{SA.}} \leftarrow \mathsf{SA.Setup}(1^\lambda)$. We will denote $\mathsf{pp} = (\mathsf{pp_{ZKS.}} \cup \mathsf{pp_{SA.}})$ and be using the appropriate pp.

▷ $\mathsf{VKD.Gen}(\mathsf{pp})$: Run $\mathsf{ZKS.Gen}(\mathsf{pp})$ twice to get the public and private parameters for two aZKS. Hence we have $(\mathsf{PK_{all}}, \mathsf{st_{all}}) \leftarrow \mathsf{ZKS.Gen}(\mathsf{pp})$ and $(\mathsf{PK_{old}}, \mathsf{st_{old}}) \leftarrow \mathsf{ZKS.Gen}(\mathsf{pp})$. Output $\mathsf{PK_{Gen}} = (\mathsf{PK_{all}}, \mathsf{PK_{old}})$ and $\mathsf{st}_0 = \mathsf{st_{all}} \cup \mathsf{st_{old}}$. Also initialize and output a pair of empty directories $\mathsf{Dir}_0 = (D_{\mathsf{all},0}, D_{\mathsf{old},0})$. Output $(\mathsf{PK_{Gen}}, \mathsf{st}_0, \mathsf{Dir}_0)$

▷ $\mathsf{VKD.Publish}(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)$:

Parse $\mathsf{Dir}_{t-1} = (D_{\mathsf{all},t-1}, D_{\mathsf{old},t-1})$. The directory Dir_{t-1} includes $(\mathsf{label}|\ i, \mathsf{val})$ pairs and marker entries that are already part of it before time t and S_t is the set of updates to be added to the directory at epoch 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 initialized before. Compute $(\mathsf{com}_{\mathsf{all},0}, \mathsf{st}_{\mathsf{all},0}) \leftarrow \mathsf{ZKS.CommitDS}(\mathsf{pp}, \mathsf{PK_{all}}, \mathsf{st_{all}}, D_{\mathsf{all},0})$ for the "all" aZKS. Similarly, compute $(\mathsf{com}_{\mathsf{old},0}, \mathsf{st}_{\mathsf{old},0}) \leftarrow \mathsf{ZKS.CommitDS}(\mathsf{pp}, \mathsf{PK_{old}}, \mathsf{st_{old}}, D_{\mathsf{old},0})$. The server will maintain two append-only strong accumulators "all" and "old" SA built over the commitments of both aZKS. More concretely, at $t = 0$, $\mathsf{acc}_{\mathsf{all},0} \leftarrow \mathsf{SA.CommitDS}(\mathsf{pp}, C_{\mathsf{all}} = (t_0, \mathsf{com}_{\mathsf{all},0}))$ and $\mathsf{acc}_{\mathsf{old},0} \leftarrow \mathsf{SA.CommitDS}(\mathsf{pp}, C_{\mathsf{old}} = (t_0, \mathsf{com}_{\mathsf{old},0}))$ respectively.

Initialize an update set and a marker set for next epoch $S_0 = \phi$ and $M_0 = \phi$. Also, initialize a table T which will later store the usernames (labels) and their times of updates for all epochs until the current.

Output $(\mathsf{com}_0 = (\mathsf{com}_{\mathsf{all},0}, \mathsf{com}_{\mathsf{old},0}), \mathsf{acc}_0 = (\mathsf{acc}_{\mathsf{all},0}, \mathsf{acc}_{\mathsf{old},0}), \Pi_0^{\mathsf{Upd}} = \perp, \mathsf{st}_0 = (\mathsf{st}_{\mathsf{all},0}, \mathsf{st}_{\mathsf{old},0}, T, S_0, M_0)$ and $\mathsf{Dir}_0 = (D_{\mathsf{all},0}, D_{\mathsf{old},0})$

$\underline{t > 0}$: Let $S_t = \{(\mathsf{label}_1, v_1), \dots, (\mathsf{label}_k, v_k)\}$ 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 (label_i, v_i), retrieve the times of update for label_i from table T if it exists. For each $\mathsf{label}_i \in S_t$, append t to the list of times (if it exists) or add (label_i, t) for new labels in the table T.

  Let $S_t^o = \{(\mathsf{label}_i, v_i^o) \mid \mathsf{label}_i \in S_t, v_i^o$ is the value for label_i at epoch t-1$\}$. If label_i is a new addition, v_i^0 will not exist. Let α_i be the version corresponding to each label_i which is the number of entries in T for label_i, 0 if no entry exists. Based on the version number, we will create the (label, value) pairs to be added in the "all" aZKS.

- If for any label_i, $\alpha_i + 1 = 2^y$ for some $y \geq 0$ then add the following (label, value) pair to the marker set M_t: $(\mathsf{label}_i | \ \mathsf{mark} \mid y,$ "marker entry 2^y for label_i" $)$
- Compute new update set to do ZKS.UpdateDS: $S'_t = \{(\mathsf{label}'_i, \mathsf{val}_i) \mid (\mathsf{label}_i, \mathsf{val}_i) \in S_t \wedge \mathsf{label}'_i = \mathsf{label} | \ \alpha_i + 1\}$ Compute the update on the "all" aZKS for the set $S'_t \cup M_t$: $(\mathsf{com}_{\mathsf{all},t}, \mathsf{st}_{\mathsf{all},t}, D_{\mathsf{all},t}, \pi_{S_t}) \leftarrow \mathsf{ZKS.UpdateDS}(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{st}_{\mathsf{all},t-1}, D_{\mathsf{all},t-1}, S'_t \cup M_t)$.
- Form a new set of (label, value) pairs to be added to the "old" aZKS. For each $\mathsf{label} \in S_t$ that is not a new addition, concatenate it with its version before the update. Hence, let $S_t^{o'} = \{(\mathsf{label}'_i, \mathsf{val}_i) \mid (\mathsf{label}_i, \mathsf{val}_i) \in S_t^o \wedge \mathsf{label}'_i = \mathsf{label} | \ \alpha_i\}$. For the "old" tree, compute $(\mathsf{com}_{\mathsf{old},t}, \mathsf{st}_{\mathsf{old},t}, D_{\mathsf{old},t}, \pi_{S_t^o}) \leftarrow \mathsf{ZKS.UpdateDS}(\mathsf{pp}, \mathsf{PK}_{\mathsf{old}}, \mathsf{st}_{\mathsf{old},t-1}, D_{\mathsf{old},t-1}, S_t^{o'})$
- Update both the chains to $\mathsf{acc}_{\mathsf{all},t}$ and $\mathsf{acc}_{\mathsf{old},t}$ in the following way: $(\mathsf{acc}_{\mathsf{all},t}, C_{\mathsf{all}} \cup (t, \mathsf{com}_{\mathsf{all},t}), \pi_{\mathsf{acc},\mathsf{all}}) \leftarrow \mathsf{SA.UpdateDS}(\mathsf{pp}, C_{\mathsf{all}}, (t, \mathsf{com}_{\mathsf{all},t}))$ and $(\mathsf{acc}_{\mathsf{old},t}, C_{\mathsf{old}} \cup (t, \mathsf{com}_{\mathsf{old},t}), \pi_{\mathsf{acc},\mathsf{old}}) \leftarrow \mathsf{SA.UpdateDS}(\mathsf{pp}, C_{\mathsf{old}}, (t, \mathsf{com}_{\mathsf{old},t}))$

Output $\mathsf{com}_t = (\mathsf{com}_{\mathsf{all},t}, \mathsf{com}_{\mathsf{old},t})$, $\mathsf{acc}_t = (\mathsf{acc}_{\mathsf{all},t}, \pi_{\mathsf{acc},\mathsf{all}}, \mathsf{acc}_{\mathsf{old},t}, \pi_{\mathsf{acc},\mathsf{old}})$, $\Pi_t^{\mathsf{Upd}} = (\pi_{S_t}, \pi_{S_t^o})$ and $\mathsf{st}_t = (\mathsf{st}_{\mathsf{all},t}, \mathsf{st}_{\mathsf{old},t}, T, S_{t+1}, M_{t+1}, \mathsf{Dir}_{t-1})$ and $\mathsf{Dir}_t = (D_{\mathsf{all},t}, D_{\mathsf{old},t})$. Figure 2 shows the internal storage of the server over time.

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

- $(\pi_1, \mathsf{val}_1) \leftarrow \mathsf{ZKS.Query}(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{st}_{\mathsf{all},t}, D_{\mathsf{all},t}, \mathsf{label} | \ \alpha)$: This gives a proof of membership of the latest version of label in the "all" aZKS and its corresponding value.
- $(\pi_2, \mathsf{val}_2) \leftarrow \mathsf{ZKS.Query}(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{st}_{\mathsf{all},t}, D_{\mathsf{all},t}, \mathsf{label} |$
$\mathsf{mark} \mid b)$: This gives a proof of membership of the marker entry right before the current version α.
- $(\pi_3, \mathsf{val}_3) \leftarrow \mathsf{ZKS.Query}(\mathsf{pp}, \mathsf{PK}_{\mathsf{old}}, \mathsf{st}_{\mathsf{old},t}, D_{\mathsf{old},t}, \mathsf{label} | \ \alpha)$: 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 $\Pi = (\pi_1, \pi_2, \pi_3)$ and $\mathsf{val} = (\mathsf{val}_1, \mathsf{val}_2, \perp)$ and α

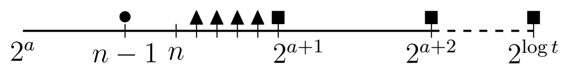
▷ $\mathsf{VKD.QueryVer}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{com}_t, \mathsf{label}, \mathsf{val}_t, \pi_t, \alpha)$: The client checks each membership or non-membership proof. Also check that version α as part of proof is less than current epoch t. More specifically, parse $\Pi = (\pi_1, \pi_2, \pi_3)$. Compute $\mathsf{ZKS.Verify}(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{com}_{\mathsf{all},t}, \mathsf{label} | \ \alpha, \mathsf{val}_1, \pi_1)$ and $\mathsf{ZKS.Verify}(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{com}_{\mathsf{all},t}, \mathsf{label} | \ \mathsf{mark} | \ b, \mathsf{val}_2, \pi_2)$ and $\mathsf{ZKS.Verify}(\mathsf{pp}, \mathsf{PK}_{\mathsf{old}}, \mathsf{com}_{\mathsf{old},t}, \mathsf{label} | \ \alpha, \perp, \pi_3)$. Output 1 if all the proofs verify and $\alpha < t$.

▷ $\mathsf{VKD.KeyHist}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_t, \mathsf{Dir}_t, \mathsf{label})$: The server first retrieves all the update times $t_1, \ldots, t_n$ for label versions $1, \ldots, n$ from T. It also retrieves corresponding $\mathsf{com}_{t_1}, \ldots, \mathsf{com}_{t_n}$ for "all" and "old" aZKS and also the roots $\mathsf{acc}_{t,\mathsf{all}}, \mathsf{acc}_{t,\mathsf{old}}$. For versions $i = 1$ to n, the server does the following:

Retrieve the val_i for t_i and version i of label from Dir_{t_i}. Let $2^a \leq i < 2^{a+1}$ for some a where i is the current version of the label. The server will generate five kinds of proofs (together called as Π_i) as follows:

1. **Correctness of com_{t_i}:** Output com_{t_i} and membership proof of com_{t_i} with respect to acc_t for both of the "all" and "old".
2. **Current Version in "all":** Membership proof for $(\mathsf{label}|\ i)$ in the "all" aZKS with respect to com_{t_i} corresponding to val_i which is $(\pi_i, \mathsf{val}_i) \leftarrow \mathsf{ZKS.Query}(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{st}_{\mathsf{all},t_i}, D_{\mathsf{all},t_i}, \mathsf{label}|\ i)$
3. **Older Version in "old"**: Membership proof in "old" aZKS with respect to com_{t_i} for $(\mathsf{label}|\ i-1)$. $(\pi_o, \mathsf{val}_o) \leftarrow \mathsf{ZKS.Query}(\mathsf{pp}, \mathsf{PK}_{\mathsf{old}}, \mathsf{st}_{\mathsf{old},t}, \mathsf{label}|i-1)$.
4. **No Higher Versions in "all":** Non membership proofs in the "all" aZKS with respect to com_{t_i} for $(\mathsf{label}|\ i+1)$ for $i = 1$ to $n-1$. For $i = n$, give proofs for $(\mathsf{label}|\ i+1), (\mathsf{label}|\ i+2), \ldots, (\mathsf{label}|\ 2^{a+1}-1)$.
5. **No Higher Marker Entries:** A non membership proof in "all" aZKS with respect to com_{t_i} for marker nodes $(\mathsf{label}|\ \mathsf{mark}|\ a+1)$ upto $(\mathsf{label}|\ \mathsf{mark}|\ \log t)$. If any of the marker proofs were already part of version $i-1$, do not include those.

Let Π_i contain all the five types of proofs described above for $i = 1$ to n. The proof types 3,4,5 for $i = n$ are shown in Figure 3. Finally output $(\{(\mathsf{val}_i, t_i)\}_{i=1}^{n}, \Pi^{\mathsf{Ver}} = (\Pi_1, \ldots, \Pi_{n-1}))$.



■ Non-membership proofs for markers $2^{a+1}, \ldots, 2^{\log t}$ in "all" lZKS.
▲ Non-membership proofs for markers $n+1, \ldots, 2^{a+1}-1$ in "all" lZKS.
● Membership proof for $n-1$ in "old" lZKS.

Figure 3: Versions Proofs in Π_n with respect to com_{t_n}

▷ $\mathsf{VKD.HistVer}(\mathsf{pp}, \mathsf{com}_t, \mathsf{acc}_t, \mathsf{label}, \{(\mathsf{val}_i, t_i)\}_{i=1}^{n}, \Pi^{\mathsf{Ver}})$: Parse $\Pi^{\mathsf{Ver}} = (\Pi_1, \ldots, \Pi_{n-1})$. For each i, first verify that com_{t_i} is correct with respect to acc_t. Then using the correct com_{t_i}, verify all of the proofs in each Π_i with respect to the appropriate com_t and output 1 if all the proofs verify.

▷ $\mathsf{VKD.Audit}(\mathsf{pp}, t_1, t_n, \{\mathsf{pub}_t\}_{t=t_1}^{t_n})$: Recall $\mathsf{pub}_t = (\mathsf{com}_t, \mathsf{acc}_t, \Pi_t^{\mathsf{Upd}})$. Do the checks $\mathsf{ZKS.VerifyUpd}$ $(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{com}_{\mathsf{all},t}, \mathsf{com}_{\mathsf{all},t-1}, \Pi_{\mathsf{all},t}^{\mathsf{Upd}})$ and $\mathsf{ZKS.VerifyUpd}(\mathsf{pp}, \mathsf{PK}_{\mathsf{old}}, \mathsf{com}_{\mathsf{old},t}, \mathsf{com}_{\mathsf{old},t-1}, \Pi_{\mathsf{old},t}^{\mathsf{Upd}})$ for all consecutive epochs from t_1 to t_n. Also check that both "all" and "old" SA are appropriately updated with $(t_i, \mathsf{com}_{\mathsf{all},t_i})$ and $(t_i, \mathsf{com}_{\mathsf{old},t_i})$ respectively for $1 \leq i \leq n$. Output 1 iff. all the checks verify.

Theorem 1. *Let* $(\mathsf{ZKS.Setup}, \mathsf{ZKS.Gen}, \mathsf{ZKS.CommitDS}, \mathsf{ZKS.Query}, \mathsf{ZKS.Verify}, \mathsf{ZKS.UpdateDS}, \mathsf{ZKS.VerifyUpd})$ *be an* aZKS *as in Definition 3.1 and let H be a collision resistant hash function. Then the construction described above is a VKD. It is $\mathcal{L}$-private with respect to the following leakage function: $L(S) = n_{\mathsf{upd}}, (n_{\mathsf{new}} + n_{\mathsf{mark}})$ where n_{new} is the number of new labels to be added* *in the VKD, n_{upd} is the number of existing labels that are being updated and n_{mark} is the number of existing labels that have reached a version 2^i for some i.*[¶]

Proof: We defer the formal proof to Appendix C.

5 aZKS Instantiations

In this section we will give concrete instantiation of aZKS used for our VKD and also of the primitives used in the construction of the aZKS.

5.1 Cryptographic Primitives

Here we briefly describe the cryptographic primitives that we will use:

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 $x \neq y$.

Definition 4 (Collision-resistant Hash Function [KL08])**.** We say that a family of functions $\{\mathcal{H}\}_\lambda$ which consists of functions of the form $H : \{0,1\}^n \mapsto \{0,1\}^m$ for $m < n$ is a family of collision-resistant hash functions parametrized by the security parameter $\lambda \in \mathbb{N}$, if for any PPT $\mathcal{A}$, there exists a negligible function $\mu()$ such that,

$$\Pr[H \leftarrow \{\mathcal{H}\}_\lambda \ ; \ (x_1, x_2) \leftarrow \mathcal{A}(1^\lambda, H) \ : \ H(x_1) = H(x_2)] \leq \mu(\lambda)$$

Simulatable Commitment Scheme (sCS) [CHL$^+$05]: A simulatable commitment [CHL$^+$05] scheme [‖] consists of four algorithms ($\mathsf{CS.Setup}, \mathsf{CS.Commit}, \mathsf{CS.Open}, \mathsf{CS.VerifyOpen}$) with description as follows:

$\sigma \leftarrow \mathsf{CS.Setup}(1^\lambda)$: The setup outputs global parameters σ for commitment scheme.

$\mathsf{com}_\sigma \leftarrow \mathsf{CS.Commit}(\sigma, m; r)$: Using σ, the commit algorithm produces commitment com_σ to message m using randomness r.

$\tau \leftarrow \mathsf{CS.Open}(\sigma, m, r, \mathsf{com}_\sigma)$: This outputs a decommitment value corresponding to commitment com_σ for message m and randomness r.

$1/0 \leftarrow \mathsf{CS.VerifyOpen}(\sigma, \mathsf{com}_\sigma, m, \tau)$: This accepts or rejects the decommitment of com_σ to message m in terms of the decommitment value τ.

A sCS satisfies the standard requirements of commitment schemes with respect to hiding and binding. In addition, there is an efficient proof to show that a given string com is a legitimate commitment on a given value m, and this proof is efficiently simulatable given any proper pair (com, m). We defer the readers to Appendix A for detailed description of security properties of sCS.

[¶]We can hide even this by adding a fixed number of entries each update by augmenting with dummy entries

[‖]In [CHL$^+$05], Chase et. al. define the primitive of *Mercurial Commitments* which also has additional algorithms to be able to do soft commitments. This definition is an adaptation of their definition with just the usual commitment functionality.

Simulatable Verifiable Random Function (sVRF) [CL07]: A Verifiable Random Function (VRF) is similar to pseudorandom functions, with the additional property of verifiability: corresponding to each secret key SK, there is a public key PK, such that, for any $y = \mathsf{VRF.Gen}(\mathsf{SK}, x)$, it is possible to verify that y is indeed the value of the VRF evaluated on input x seeded by SK. A simulatable VRF (sVRF) is a VRF for which this proof can be simulated, so a simulator can pretend that the value of $\mathsf{VRF.Gen}(\mathsf{SK}, x)$ is any y. A sVRF comprises of the algorithms $(\mathsf{sVRF.Setup}, \mathsf{sVRF.KeyGen}, \mathsf{sVRF.Eval}, \mathsf{sVRF.Prove}, \mathsf{sVRF.Verify})$ with descriptions as follows:

$\mathsf{params} \leftarrow \mathsf{sVRF.Setup}(1^\lambda)$: The setup takes in the security parameter and outputs params

$(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{sVRF.KeyGen}(\mathsf{params})$: The key generation takes in the global params and outputs the public key PK and secret key SK.

$y \leftarrow \mathsf{sVRF.Eval}(\mathsf{params}, \mathsf{SK}, x)$: The evaluation takes in params, the secret key SK and x and outputs the sVRF evaluation of x as y

$\pi \leftarrow \mathsf{sVRF.Prove}(\mathsf{params}, \mathsf{SK}, x)$: The prove takes in params, the secret key SK and x and outputs a proof for the sVRF evaluation of x

$1/0 \leftarrow \mathsf{sVRF.Verify}(\mathsf{params}, \mathsf{PK}, x, y, \pi)$: The verification takes in params, the public key PK, input x, evaluation y and proof of evaluation π and it outputs a bit whether the verification passed.

We defer the readers to Appendix A for the security properties of an sVRF. Chase-Lysyanskaya [CL07] give a construction of sVRF using bilinear maps and prove its security based on composite order subgroup decision assumption and an extension of the Q-BDHI assumption, but it is not efficient. We describe an efficient constructions of sVRFs in Appendix B based on DDH assumption as proposed in [MBB+15].

5.2 Strong Accumulator Construction

In this subsection, we give a new construction of an *append-only* strong accumulator. For our SA, the expected depth of the tree and the proofs is logarithmic in the size of the set [OB09, Knu98]. Our construction achieves efficiency improvement over the previous constructions [MBB+15, CHKO08], where the depth is order of the set size. The high level idea is to build a Patricia tree [Knu98] over the labels. **Patricia tree** 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. It has a wide range of applications due to the efficient insertion and deletion operations. Here we illustrate with an example. For the universe of be all 4 bit binary strings, a Patricia tree built on subset $P = \{0010, 0100, 0111, 1000, 1100\}$ is represented in Figure 4. We now describe our SA construction:

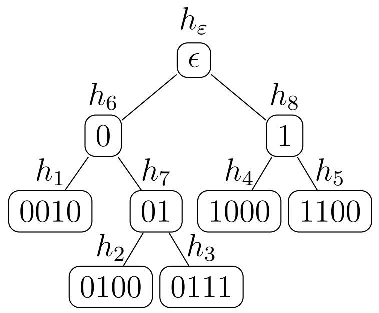


Figure 4: Patricia Tree on subset P

▷ $\mathsf{SA.Setup}(1^\lambda)$: Choose a collision-resistant hash function $H : \{0,1\}^* \mapsto \{0,1\}^m$. Output $\mathsf{pp} = [H]$

▷ $\mathsf{SA.CommitDS}(1^\lambda, \mathsf{pp}, \mathsf{D})$: Datastore $\mathsf{D} = \{(l_1, v_1), \ldots, (l_n, v_n)\}$, 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 tree on $\{y_i\}$. For different types of nodes, we will associate the following hash values:

– **Leaf nodes:** For a node y_i at depth d_i, compute: $h_{y_i} = H(k_D|y_i|d_i|v_i)$. For example, in the tree in Figure 4, $h_2 = H(k_D|0100|3|v_2)$.
– **Interior nodes:** For an interior node x at depth d_x, let $x.s_0$ and $x.s_1$ be the labels of its children. Compute: $h_x = H(k_D|x|d_x|h_{x.s_0}|h_{x.s_1}|s_0|s_1)$. For example in Figure 4, $h_6 = H(k_D|0|1|h_1|h_7|010|1)$

$\mathsf{SA.CommitDS}(1^\lambda, \mathsf{pp}, \mathsf{D})$ outputs $\mathsf{com} = (h_{\mathsf{root}}, k_D)$.

▷ $\mathsf{SA.Query}(\mathsf{pp}, \mathsf{D}, l)$: If $l \in \mathsf{D}$, 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 tree 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 $l \notin \mathsf{D}$, 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 $[1, 1000, 1100, h_8, h_4, h_5, (h_6, \epsilon, 0, 1)]$

▷ $\mathsf{SA.Verify}(\mathsf{pp}, \mathsf{com}, l, v, \pi)$: 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 and output 1 iff. you get h_{root}.

▷ $\mathsf{SA.UpdateDS}(\mathsf{pp}, \mathsf{D}, S)$: First we check that S is a valid set of updates which means that for all $(\mathsf{label}_i, \mathsf{val}_i) \in S$, $\mathsf{label}_i \notin \mathsf{D}$. Initialize sets $Z_{\mathsf{new}}, Z_{\mathsf{old}}, Z_{\mathsf{const}}$ to empty. For all $\mathsf{label}_j \in S$, compute: $h_{\mathsf{label}_j} = H(k_D|l_j|d_j|\mathsf{val}_j)$ and add h_{label_j} to the appropriate position in the tree 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 $\mathsf{D}' = \mathsf{D} \cup \{(\mathsf{label}_j, \mathsf{val}_j)\}$ and output $\pi_S = (Z_{\mathsf{old}}, Z_{\mathsf{new}}, Z_{\mathsf{const}})$ and set S.

▷ $\mathsf{SA.VerifyUpd}(\mathsf{pp}, \mathsf{com}, \mathsf{com}', S, \pi_S)$: Parse $\pi_S = (Z_{\mathsf{old}}, Z_{\mathsf{new}}, Z_{\mathsf{const}})$. 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 $Z_{\mathsf{old}}, Z_{\mathsf{new}}$ are exactly the hash values of all nodes in Z_S. Output 1 if all checks go through.

Theorem 2. *Assuming H is collision-resistant hash function, the construction as described above is an append-only SA with completeness and soundness.*

Proof: At a very high level, if an adversary is able to forge a proof (i.e., prove contradictory statements about the presence of some element in the set), then, the forgery can be leveraged to break the collision resistance of H (at some node in the Patricia tree). We present the details of the proof in Appendix D.

5.3 aZKS Instantiations

In this section, we give concrete instantiations for aZKS. The trivial way of implementing a fully private aZKS is to compute a fresh commitment to the set each time an update happens [Lis05b]. This does not leak anything except that an update happened. Such a construction can be instantiated from a static zero-knowledge set in a black-box way but is very inefficient.

Here we focus on a more efficient instantiation with a reasonably small leakage. Our construction is a combination of a *append-only strong accumulator* and a *pseudonym map* and *commitment scheme*. Strong accumulators directly give us completeness and soundness. To get zero-knowledge for a $\mathsf{D} = \{(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)\}$, we will use a pseudonym map $P(\cdot)$ and a commitment map $C(\cdot, \cdot)$ with some specific properties.

We need the pseudonym map to be pseudorandom and verifiable so that the position l_i is pseudorandom in the SA, but given the corresponding label_i, it is verifiable. We need the commitment map to be binding and hiding. Moreover, for satisfying the privacy property for aZKS, we need these primitives to be simulatable. Therefore, we implement the pseudonym map with a sVRF and the commitment scheme with a sCS.

Let $(\mathsf{sVRF.Setup}, \mathsf{sVRF.KeyGen}, \mathsf{sVRF.Eval}, \mathsf{sVRF.Prove}, \mathsf{sVRF.Verify})$ be a simulatable VRF and let $(\mathsf{CS.Setup}, \mathsf{CS.Commit}, \mathsf{CS.Open})$ be a simulatable commitment scheme as described before. Let $P_{\mathsf{sVRF.}}(\mathsf{label}) = \mathsf{sVRF.Eval}(\mathsf{params}, \mathsf{SK}, \mathsf{label})$ for $\mathsf{params} \leftarrow \mathsf{sVRF.Setup}(1^\lambda)$ and let $C_{\mathsf{Commit}}(\mathsf{label}_i, \mathsf{val}_i)$ $= \mathsf{CS.Commit}(\sigma, (\mathsf{label}_{j_i}, \mathsf{val}_{j_i}); r_i)$ for $\sigma \leftarrow \mathsf{CS.Setup}(1^\lambda)$ and some randomness r_i. Given $\mathsf{D} =$ $\{(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)\}$, build a new $\mathsf{D}' = \{(l_1, v_1), \ldots, (l_n, v_n)\}$ where $l_i = P(\mathsf{label}_i)$ and $v_i = C_{\mathsf{Commit}}(\mathsf{label}_i, \mathsf{val}_i\ ;\ r_i)$ for $P()$ and $C()$ as described above. Build an append-only SA on D' and that gives us an aZKS.

More concretely, we now describe the construction of aZKS from an SA, sVRF and sCS. Let $(\mathsf{SA.Setup}, \mathsf{SA.Gen}, \mathsf{SA.CommitDS}, \mathsf{SA.Query}, \mathsf{SA.Verify},$
$\mathsf{SA.UpdateDS}, \mathsf{SA.VerifyUpd})$ be an append-only SA and let $P(\cdot)$ and $C(\cdot, \cdot)$ be the $P_{\mathsf{sVRF.}}(\cdot)$ and $C_{\mathsf{Commit}}(\cdot, \cdot)$ maps respectively, as described above.

CONSTRUCTION:

$\triangleright$ $\mathsf{ZKS.Setup}(1^\lambda)$: Output $\mathsf{pp} \leftarrow \mathsf{SA.Setup}(1^\lambda)$.

$\triangleright$ $\mathsf{ZKS.Gen}(1^\lambda, \mathsf{pp})$: Run $\mathsf{params} \leftarrow \mathsf{sVRF.Setup}(1^\lambda)$ and key generation to get $(\mathsf{PK}, \mathsf{SK}) \leftarrow$ $\mathsf{sVRF.KeyGen}(\mathsf{params})$. Also do $\sigma \leftarrow \mathsf{CS.Setup}(1^\lambda)$. Output $\mathsf{PK}_{\mathsf{Gen}} = [\mathsf{PK}, \sigma, \mathsf{params}]$ and $\mathsf{st}_{\mathsf{Gen}} = \mathsf{SK}$.

$\triangleright$ $\mathsf{ZKS.CommitDS}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_{\mathsf{Gen}}, \mathsf{D})$: Let $\mathsf{D} = \{(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)\}$. Build a new $\mathsf{D}' = \{(l_1, v_1), \ldots, (l_n, v_n)\}$ where $l_i = P(\mathsf{label}_i)$ and $v_i = C_{\mathsf{Commit}}(\mathsf{label}_i, \mathsf{val}_i\ ;\ r_i)$ for $P()$ and $C()$ as described above. Let $\mathsf{com} \leftarrow \mathsf{SA.CommitDS}(\mathsf{pp}, \mathsf{D}')$ and $\mathsf{st}_{\mathsf{com}} = (\mathsf{SK}, \mathsf{D}, \mathsf{D}', r_1, \ldots, r_n)$. Output $(\mathsf{com}, \mathsf{st}_{\mathsf{com}})$

$\triangleright$ $\mathsf{ZKS.Query}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_{\mathsf{com}}, \mathsf{D}, \mathsf{label})$: Compute $(\pi_{\mathsf{SA.}}, \mathsf{val}) \leftarrow \mathsf{SA.Query}(\mathsf{pp}, \mathsf{D}, P(\mathsf{label}))$. We also need a proof that $P()$ was correctly computed. Compute $\pi_{\mathsf{label}} = \mathsf{sVRF.Prove}(\mathsf{params}, \mathsf{SK}, \mathsf{label})$. Retrieve the com, r corresponding to label in $\mathsf{st}_{\mathsf{com}}$ and compute $\tau \leftarrow \mathsf{CS.Open}(\sigma, (\mathsf{label}, \mathsf{val}), r, \mathsf{com})$. Output $\Pi = ((P(\mathsf{label}), \pi_{\mathsf{label}}, \mathsf{com}, \tau, \pi_{\mathsf{SA.}}), \mathsf{val})$.

$\triangleright$ $\mathsf{ZKS.Verify}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{com}, \mathsf{label}, \mathsf{val}, \Pi)$: Parse proof as $\Pi = ((y = P(\mathsf{label}), \pi_{\mathsf{label}}, \mathsf{com}, \tau, \pi_{\mathsf{SA.}}), \mathsf{val})$. First check that $\mathsf{sVRF.Verify}(\mathsf{params}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{label}, y, \pi_{\mathsf{label}}) = 1$. Also verify the commitment by checking that $\mathsf{CS.VerifyOpen}(\sigma, \mathsf{com}, (\mathsf{label}, \mathsf{val}), \tau) = 1$. Finally output $\mathsf{SA.Verify}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{com},$ $P(\mathsf{label}), C(\mathsf{label}, \mathsf{val}), \pi)$.

▷ $\mathsf{ZKS.UpdateDS}(\mathsf{pp}, \mathsf{st_{com}}, \mathsf{D}, S)$: Compute $S' = \{(l_j, v_j) \mid l_i = P(\mathsf{label}_j) \wedge v_j = C(\mathsf{label}_j, \mathsf{val}_j; r_j) \wedge (\mathsf{label}_j, \mathsf{val}_j) \in S\}$. Let $(\mathsf{com}'', \mathsf{D}'', \pi'_S, S') \leftarrow \mathsf{SA.UpdateDS}(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{D}', S')$. Output $(\mathsf{com}'', \mathsf{st_{com}}, \mathsf{D}'', \pi'_S = (\pi_S, S'))$

▷ $\mathsf{ZKS.VerifyUpd}(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{com}, \mathsf{com}', \pi'_S)$: Output $\mathsf{SA.VerifyUpd}(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{com}, \mathsf{com}', \pi'_S = (\pi_S, S'))$

Theorem 3. *Assuming* (sVRF.Setup, sVRF.KeyGen, sVRF.Eval, sVRF.Prove, sVRF.Verify) *is a simulatable VRF and* (CS.Setup, CS.Commit, CS.Open) *is a simulatable commitment scheme and* (SA.Setup, SA.Gen, SA.CommitDS, SA.Query, SA.Verify, SA.UpdateDS, SA.VerifyUpd) *is an* **append-only** *strong accumulator, the construction as described above is an* **append-only** *zero-knowledge set for leakage functions* $L_1(\mathsf{D}) = |\mathsf{D}| = N$ *and* $L_2(S) = |S|$.

Proof: At a high level, the soundness follows from 1) the verifiability of sVRF 2) the binding property of sCS and 3) the soundness of SA. The zero-knowledge property with the specified leakage profile follows from the simulatability of sVRF and sCS. The detailed proof is in Appendix E.

6 Efficiency Analysis and Comparison with other VKDs

In this section we will discuss how our VKD construction compares to the other VKDs [MBB+15, Bon16, TD17] in terms of efficiency as well as privacy.

6.1 Current Implementations as VKD

First we describe how CONIKS/EthIKS/Catena can be expressed as an instantiation of VKD. Corresponding VKD algorithm descriptions will be as follows:

$\mathsf{VKD.Setup}(1^\lambda)$: Choose a collision-resistant hash function $H : \{0,1\}^* \mapsto \{0,1\}^m$. Choose constants $k_l, k_e \xleftarrow{\$} \{0,1\}^\lambda$. Output $\mathsf{pp} = [H, k_l, k_e]$

$\mathsf{VKD.Gen}(\mathsf{pp})$: Output $(\mathsf{PK}, \mathsf{st}) \leftarrow \mathsf{sVRF.KeyGen}(\mathsf{pp})$.

$\mathsf{VKD.Publish}(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{Dir}_{t-1}, \mathsf{st}_{t-1}, S_t)$: Our directory is a set of (label, val) pairs corresponding to usernames and their keys.

$t = 1$: This is when we are setting up the directory for the first time and let Dir_0 be the initial set of (label, val) pairs. Build $\mathsf{Dir}'_0 = \{(l_1, v_1), \ldots, (l_n, v_n)\}$ where $l_i = \mathsf{sVRF.Eval}(\mathsf{PK_{Gen}}, \mathsf{label}_i)$ and $v_i = \mathsf{Commit}(\mathsf{label}_i, \mathsf{val}_i\ ;\ r_i)$. Choose a constant $k_D \xleftarrow{\$} \{0,1\}^\lambda$. Let $\{y_1, \ldots, y_n\}$ be a lexicographic ordering of $\{l_1, \ldots, l_n\}$ and build a Merkle hash tree over $\{y_i\}$. For different nodes, do the following:

- **Leaf nodes:** Compute: $h_{y_i} = H(k_l | k_D | y_i | m | v_i)$
- **Empty nodes:** For a string s which corresponds to an empty node as described above, compute: $h_s = H(k_e | k_D | s | x = |s|)$
- **Interior nodes:** For an interior node int, compute: $h_{\mathsf{int}} = H(h_{\mathsf{ch},0} | h_{\mathsf{ch},1})$

Initialize a linear acc structure in form of a hash chain: $\mathsf{hash}_1 = H(1 \mid \mathsf{com}_1)$. Output $(\mathsf{pub}_1 = (h_{\mathsf{root}}, k_D), \mathsf{hash}_1, \mathsf{st}_1 = \{r_i\})$ where $\{r_i\}$ is randomness for commitments and $\mathsf{Dir}_1 = S_1$

For any $t > 1$, let $S_t = \{(\mathsf{label}_1, \mathsf{val}_1), .., (\mathsf{label}_k, \mathsf{val}_k)\}$ be the updates to be done at epoch t which include new ($S_{t,\mathsf{new}}$) as well as existing labels to be updated $S_{t,\mathsf{update}}$. For new labels in $S_{t,\mathsf{new}}$, compute $l_i = \mathsf{sVRF.Eval}(\mathsf{PK_{Gen}}, \mathsf{label}_i)$, $v_i = \mathsf{Commit}(\mathsf{label}_i, \mathsf{val}_i\ ;\ r_i)$ and compute the leaf node as described above updating hash values upto the root. If l_i already exists in $S_{t,\mathsf{update}}$, update the v_i as commitment to new val_i and again update hash values along the path to root. Update the hash chain by computing: $\mathsf{hash}_t = H(t-1 \mid t \mid \mathsf{com}_t \mid \Pi_t^{\mathsf{Upd}} \mid \mathsf{hash}_{t-1})$. Output corresponding pub_t, updated $\mathsf{Dir}_t, \mathsf{st}_t$

$\mathsf{VKD.Query}(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{st}_t, \mathsf{Dir}, \mathsf{label}, t)$: Compute $l = \mathsf{sVRF.Eval}(\mathsf{PK_{Gen}}, \mathsf{label})$ and output the hash values for the path of l, commitment v and also the opening for the commitment

$\mathsf{VKD.QueryVer}(\mathsf{pp}, \mathsf{PK_{Gen}}, \mathsf{com}_t, \mathsf{label}, \mathsf{val}, \pi, t)$: Verify the sVRF value, the commitment opening and check that the hash values are consistent with com_t

$\mathsf{VKD.Audit}(\mathsf{pp}, \mathsf{PK_{Gen}}, t_1, t_n, \{\mathsf{pub}_t\}_{t=t_1}^{t_n})$: Check that hash chain has grown appropriately. Output 1 iff. both the checks output 1. In CONIKS, this is achieved via gossip protocols between the auditors and in EthIKS, Catena auditors can directly audit the public blockchain.

These constructions map each label to a fixed location and an update involves updating the value at the fixed location. So at any point of time, only one version of a value is present per label. Hence these implementations do not support $\mathsf{VKD.KeyHist}()$ or $\mathsf{VKD.HistVer}()$.

Theorem 4. *The construction described above is a VKD with leakage function as follows: $L(S = S_{\mathsf{new}} \cup S_{\mathsf{update}}) = (|S_{\mathsf{new}}|, \{P(\mathsf{label}_i) \mid \mathsf{label}_i \in S_{\mathsf{update}}\})$ where S_{new} is the set of new (label, value) pairs to be added and S_{update} is the existing set of labels whose value has to be updated.*

Proof: We give a proof overview in Appendix F.

Tracing Attack: The leakage in Theorem 4 causes a "tracing attack" as follows: When you query for the same label several times, you get values in the proof which depend on the pseudonym of the label. These values can be pseudonyms of other labels and give information about a label that you did not query. Hence if you get the proofs for the same label over time, you can infer about other pseudonyms, 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. The proof for Alice's key will contain 011 being its sibling which is P(Bob). Since the pseudonym 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 (i.e., which username the pseudonym belonged to), she will be able to completely trace when the key changed.

6.2 Improvements in our VKD

The major improvements of our VKD implementation over that of CONIKS, EthIKS, Catena [MBB+15, Bon16, TD17] are the following:

Privacy: The only natural way to extend current VKDs to achieve the same level of privacy as ours, would be to compute fresh pseudonyms (requiring expensive public key operations) for all usernames and rebuild the directory every epoch. We instead, add version number of each key and treat each deletion as a new insertion under a different pseudonym (which is cheap to

	CONIKS [MBB$^+$15]	EthIKS [Bon16] CONIKS on Catena log [TD17]	This paper
Publish (per epoch cost)	$n\mathsf{ZKS.UpdateDS}$	$n\mathsf{ZKS.UpdateDS}$	$3n\mathsf{ZKS.UpdateDS}$ $+ \log N\mathsf{hash}$
Query	ZKS.Query	ZKS.Query	3ZKS.Query
QueryVer	ZKS.Verify	ZKS.Verify	3ZKS.Verify
KeyHist			$(m + \log t)\mathsf{ZKS.Query}$ $+m \log N\mathsf{hash}$
HistVer			$(m + \log t)\mathsf{ZKS.Verify}$ $+m \log N\mathsf{hash}$
Audit (per epoch cost)	hash	hash	$n\mathsf{ZKS.VerifyUpd}$ $+ \log N\mathsf{hash}$
Update frequency	Provider-chosen	Ethereum block frequency Bitcoin block frequency	Provider-chosen
Monitoring frequency	every epoch	latest epoch (Alice remembers last version number of her key)	latest epoch (Alice remembers last time she updated her key)
Privacy	Vulnerable to tracing attack	Pseudonym map of updated keys	$n_{\mathsf{upd}}, (n_{\mathsf{new}} + n_{\mathsf{mark}})$

Table 1: Let $N = |\mathsf{Dir}_t|$ be the size of the current directory, n be the total number of updates at epoch t, t be the current time and m be the number of versions of a key for which KeyHist, HistVer are run. Let n_{new}= #(new labels added at t), n_{upd} = #(labels updated at t) and n_{mark} = #(labels whose update version is 2^i for $i > 0$). Hence $n = (n_{\mathsf{upd}} + n_{\mathsf{new}} + n_{\mathsf{mark}})$

compute). This hides which old pseudonym is being updated, avoids timing links between new and old pseudonyms, and hides the lifespan of that key.

Auditing updates: The work of verifying that the server is behaving correctly is now reduced to verifying that nothing is ever deleted or updated; this can be done by an untrusted auditor as it involves no secret/private information.

Monitoring frequency: Using the KeyHist interface, Alice can monitor her key revisions after every update (she only has to remember the last time she did an update) which is a significant improvement over [MBB$^+$15] which required Alice to monitor her key at every epoch. [Bon16, TD17] improved the monitoring frequency to be any epoch for a Ethereum/Bitcoin-aware client, but it required Alice to remember her last version number which is less natural to remember.

We summarize the differences along with concrete cost in terms of number of calls to the zero knowledge set primitive and hash function in Table 1. Then we count the number of cryptographic operations in out aZKS instantiation in Section 6.3. Here we compare with respect to the same underlying aZKS for all implementations. The actual aZKS instantiations may not be directly comparable. We describe our aZKS instantiation and the count the number of cryptographic operations in Section 6.3.

6.3 aZKS costs

The size of the membership or non-membership proofs in aZKS in all the constructions comes from underlying datastructures. In CONIKS, it is proportional to the security parameter λ since underlying structure is a Merkle tree on $O(2^\lambda)$ nodes. In practice this is something like 256 bits. Our underlying datastructure is a Patricia Tree (Section 5.2) in which the size of query proofs is proportional to the log of (number of users * the number of times they updated their key) in the system. This cost is typically around 30 bits for a single update for a huge user base such as WhatsApp. We count the exact number of operations required for each algorithm of aZKS built using our SA (Section 5.2), the sVRF implementation from [MBB+15] and a sCS implemented as hash of message with randomness i.e. $\mathsf{com}_\sigma = H(m, r)$. (Full construction in Appendix B) For completeness we give efficient sVRF construction based on DDH assumption in Appendix B.

aZKS *costs:* Let N = the size of the set, hash = the cost of computing one cryptographic hash, exp = the cost of one group exponentiation.

$\mathsf{ZKS.Setup}$: $4N\mathsf{hash} + N\mathsf{exp}$

$\mathsf{ZKS.Query}$: $\mathsf{hash} + 2\mathsf{exp}$

$\mathsf{ZKS.Verify}$: $(3 + 2\log N)\mathsf{hash} + 4\mathsf{exp}$

$\mathsf{ZKS.UpdateDS}$ **(updating 1 label):** $(2 + \log N)\mathsf{hash}$

$\mathsf{ZKS.VerifyUpd}$ **(verifying 1 update):** $(2 + \log N)\mathsf{hash}$

7 Acknowledgements

The work by Esha Ghosh was supported in part by NSF grant CNS-1228485. The work by Apoorvaa Deshpande was supported in part by NSF grant CNS-1422361.

References

[BCD+17] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. *IACR Cryptology ePrint Archive*, 2017:43, 2017.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, pages 416–432, 2003.

[Bon16] Joseph Bonneau. Ethiks: Using ethereum to audit a coniks key transparency log. In *International Conference on Financial Cryptography and Data Security*, pages 95–105. Springer, 2016.

[CHKO08] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In *International Conference on Information Security*, pages 471–486. Springer, 2008.

[CHL+05] Melissa Chase, Alexander Healy, Anna Lysyanskaya, Tal Malkin, and Leonid Reyzin. Mercurial commitments with applications to zero-knowledge sets. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 422–439. Springer, 2005.

[CL07] Melissa Chase and Anna Lysyanskaya. Simulatable vrfs with applications to multi-theorem nizk. In *Annual International Cryptology Conference*, pages 303–322. Springer, 2007.

[CW09] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.

[EMBB17] Saba Eskandarian, Eran Messeri, Joe Bonneau, and Dan Boneh. Certificate transparency with privacy. *arXiv preprint arXiv:1703.02209*, 2017.

[FB14] T Fox-Brewster. Whatsapp adds end-to-end encryption using textsecure. *The Guardian, Nov*, 2014.

[KL08] Jonathan Katz and Yehuda Lindell. Introduction to modern cryptography, 2008.

[Knu98] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[KZG10] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 177–194. Springer, 2010.

[Lis05a] Moses Liskov. Updatable zero-knowledge databases. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 174–198. Springer, 2005.

[Lis05b] Moses Liskov. Updatable zero-knowledge databases. In Bimal K. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, volume 3788 of *Lecture Notes in Computer Science*, pages 174–198. Springer, 2005.

[LLK13] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. Technical report, 2013.

[MBB$^+$15] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. Coniks: Bringing key transparency to end users. In *Usenix Security*, pages 383–398, 2015.

[MRK03] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-knowledge sets. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium*, pages 80–91. IEEE, 2003.

[OB09] Alina Oprea and Kevin D Bowers. Authentic time-stamps for archival storage. In *European Symposium on Research in Computer Security*, pages 136–151. Springer, 2009.

[Sch15] B Schneier. Apples imessage encryption seems to be pretty good, 2015.

[TD17] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via bitcoin. In *IEEE Symp. on Security and Privacy*, 2017.

A Definitions of Some Cryptographic Primitives

Definition 5 (Simulatable Commitments [CHL+05])**.** A simulatable commitment [CHL+05] scheme consists of four algorithms ($\mathsf{CS.Setup}, \mathsf{CS.Commit}, \mathsf{CS.Open}, \mathsf{CS.VerifyOpen}$) with the following descriptions and properties:

$\sigma \leftarrow \mathsf{CS.Setup}(1^\lambda)$: The setup algorithm outputs global parameters σ for the commitment scheme.

$\mathsf{com}_\sigma \leftarrow \mathsf{CS.Commit}(\sigma, m; r)$: Using the global parameters σ, the commit algorithm produces commitment com_σ to message m using randomness r.

$\tau \leftarrow \mathsf{CS.Open}(\sigma, m, r, \mathsf{com}_\sigma)$: This algorithm outputs a decommitment value corresponding to commitment com_σ for message m and randomness r.

$1/0 \leftarrow \mathsf{CS.VerifyOpen}(\sigma, \mathsf{com}_\sigma, m, \tau)$: This algorithm accepts or rejects the decommitment of com_σ to message m in terms of the decommitment value τ.

Correctness: For all security parameters $\lambda \in \mathbb{N}$ and for all $m \in$ Domain,

$$\begin{aligned}\Pr[\sigma \leftarrow \mathsf{CS.Setup}(1^\lambda)\ ;\ r \xleftarrow{\$} \{0,1\}^\lambda\ ;\ \mathsf{com}_\sigma \leftarrow \\ \mathsf{CS.Commit}(\sigma, m; r)\ ;\ \tau \leftarrow \mathsf{CS.Open}(\sigma, m, r, \mathsf{com}_\sigma)\ :\\ \mathsf{CS.VerifyOpen}(\sigma, \mathsf{com}_\sigma, m, \tau) = 1] = 1\end{aligned}$$

Binding: For all security parameters $\lambda \in \mathbb{N}$ and for all m_1, m_2, for all PPT adversaries $\mathcal{A}$, there exists a negligible function $\nu()$ such that,

$$\begin{aligned}\Pr[\sigma \leftarrow \mathsf{CS.Setup}(1^\lambda); (\mathsf{com}_\sigma, m_1, m_2, \pi_1, \pi_2) \leftarrow \mathcal{A}(\sigma) :\\ m_1 \neq m_2 \in D(\sigma) \wedge \mathsf{CS.VerifyOpen}(\sigma, \mathsf{com}_\sigma, m_1, \tau_1) = 1 \wedge \\ \mathsf{CS.VerifyOpen}(\sigma, \mathsf{com}_\sigma, m_2, \tau_2) = 1] \leq \nu(\lambda)\end{aligned}$$

Hiding (Simulatability): Before we describe this security property, let us define the following two oracles:

- $\mathcal{O}_\sigma$: This is initialized with input $\sigma \leftarrow \mathsf{CS.Setup}(1^\lambda)$ and a list of commitments L is initialized to empty. On input m, $\mathcal{O}_\sigma$ chooses a random string r and creates $\mathsf{com}_\sigma \leftarrow \mathsf{CS.Commit}(\sigma, m; r)$ and stores the pair (com_σ, m) in L. It outputs com_σ.

  On input (com_σ, m), $\mathcal{O}_\sigma$ first checks if it exists in L. If yes, it outputs $\tau = \mathsf{CS.Open}(\sigma, m, r, \mathsf{com}_\sigma)$. Else it outputs $\perp$.

- $\mathcal{O}_{\sigma,\mathsf{td}}$: This oracle works with the assistance of a simulator $\mathsf{Sim} = (\mathsf{SimSetup}, \mathsf{SimCommit}, \mathsf{SimOpen})$ that we describe below.

  - $(\sigma, \mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda)$: The $\mathsf{SimSetup}$ algorithm takes in the security parameter and outputs global parameters σ and also a trapdoor td with it.
  - $\mathsf{com}_\sigma \leftarrow \mathsf{SimCommit}(\sigma, \mathsf{td}, r)$: The $\mathsf{SimCommit}$ algorithm takes in the global parameters σ and the trapdoor td generated by $\mathsf{SimSetup}()$ and also randomness r and outputs a commitment string com_σ. Note that $\mathsf{SimCommit}$ does not take a message to commit to.

– $\tau \leftarrow \mathsf{SimOpen}(\sigma, \mathsf{td}, m, \mathsf{com}_\sigma)$: The $\mathsf{SimOpen}$ algorithm takes in the global parameters σ and the trapdoor td generated by $\mathsf{SimSetup}()$, commitment string com_σ output by $\mathsf{SimCommit}()$ and also a message m that it wants to decommit to and outputs an opening τ.

The oracle $\mathcal{O}_{\sigma,\mathsf{td}}$ is initialized with input $(\sigma, \mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda)$ and a list of commitments L which is initialized to empty. On input m, $\mathcal{O}_{\sigma,\mathsf{td}}$ chooses a random string r and calls $\mathsf{com}_\sigma \leftarrow \mathsf{SimCommit}(\sigma, \mathsf{td}, r)$ and stores the pair (com_σ, m) in L. It outputs com_σ. Note that $\mathsf{SimCommit}$ does not get to see the message that it is committing to.

On input (com_σ, m), $\mathcal{O}_{\sigma,\mathsf{td}}$ first checks if it exists in L. If yes, it outputs $\tau = \mathsf{SimOpen}(\sigma, \mathsf{td}, m, \mathsf{com}_\sigma)$. Else it outputs $\perp$. Hence, given message m, $\mathsf{SimOpen}$ should be able to decommit com_σ to m even though com_σ was formed without knowing m.

A commitment scheme is said to be *simulatable* if there exists simulator $\mathsf{Sim} = (\mathsf{SimSetup}, \mathsf{SimCommit}, \mathsf{SimOpen})$ such that for all adversaries $\mathcal{A}$, there exists a negligible function $\nu()$ such that,

$$\begin{aligned}\Pr[\sigma_0 \leftarrow \mathsf{Setup}(1^\lambda)\ ;\ (\sigma_1, \mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda)\ ;\\ \mathcal{O}_0 = \mathcal{O}_{\sigma_0}\ ;\ \mathcal{O}_1 = \mathcal{O}_{\sigma_1,\mathsf{td}}\ ;\ b \xleftarrow{\$} \{0,1\}\ ;\\ b' \leftarrow \mathcal{A}^{\mathcal{O}_b}(\sigma_b)\ :\ b' = b] = 1/2 + \nu(\lambda)\end{aligned}$$

Definition 6 (Simulatable VRF [CL07])**.** A simulatable verifiable random function (sVRF) is like a regular VRF with the additional requirement that the VRF proofs can be simulated. sVRF comprises of the algorithms $(\mathsf{sVRF.Setup}, \mathsf{sVRF.KeyGen}, \mathsf{sVRF.Eval}, \mathsf{sVRF.Prove}, \mathsf{sVRF.Verify})$ described as follows:

$\mathsf{params} \leftarrow \mathsf{sVRF.Setup}(1^\lambda)$: The setup algorithm takes in the security parameter and outputs a set of global parameters params

$(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{sVRF.KeyGen}(\mathsf{params})$: The key generation algorithm takes in the global params and outputs the public key PK and secret key SK for the sVRF.

$y \leftarrow \mathsf{sVRF.Eval}(\mathsf{params}, \mathsf{SK}, x)$: The evaluation algorithm takes in params, the secret key SK and x and outputs the sVRF evaluation of x as y

$\pi \leftarrow \mathsf{sVRF.Prove}(\mathsf{params}, \mathsf{SK}, x)$: The prove algorithm again takes in params, the secret key SK and x and outputs a proof for the sVRF evaluation of x

$1/0 \leftarrow \mathsf{sVRF.Verify}(\mathsf{params}, \mathsf{PK}, x, y, \pi)$: The verification algorithm takes in params, the public key PK, input x, evaluation y and proof of evaluation π and it outputs a bit to indicate whether the verification passed.

A simulatable VRF for input domain $D(.)$ and output range $R(.)$, has the following properties:
Correctness: For all security parameters $\lambda \in \mathbb{N}$, for all $\mathsf{params} \leftarrow \mathsf{sVRF.Setup}(1^\lambda)$ and for all inputs $x \in D(\mathsf{params})$,

$$\Pr[(\mathsf{PK},\mathsf{SK}) \leftarrow \mathsf{sVRF.KeyGen}(\mathsf{params}); y \leftarrow \mathsf{sVRF}.\\ \mathsf{Eval}(\mathsf{params},\mathsf{SK},x); \pi \leftarrow \mathsf{sVRF.Prove}(\mathsf{params},\mathsf{SK},x) :\\ \mathsf{sVRF.Verify}(\mathsf{params},\mathsf{PK},x,y,\pi) = 1] = 1$$

Pseudorandomness: Informally, this property captures the following: given $(\mathsf{params},\mathsf{PK})$ and oracle access to $\mathsf{sVRF.Eval}(\mathsf{params},\mathsf{SK},-)$ and to $\mathsf{sVRF.Prove}(\mathsf{params},\mathsf{SK},-)$ for $x \in D(\lambda)$, the outputs of the sVRF should be indistinguishable from a distribution of truly random strings of the same length. More formally, For all parameters $\lambda \in \mathbb{N}$, for all $x \in D(\lambda)$ and for all PPT $\mathcal{A}$, there exists a negligible function $\nu()$ such that,

$$\Pr[\mathsf{params} \leftarrow \mathsf{sVRF.Setup}(1^\lambda); (\mathsf{PK},\mathsf{SK}) \leftarrow \mathsf{sVRF.KeyGen}\\ (\mathsf{params}); (Q_e, Q_p, x, \mathsf{st}_A) \leftarrow \mathcal{A}^{\mathsf{sVRF.Eval},\mathsf{sVRF.Prove}(\mathsf{params},\mathsf{SK},-)}\\ (\mathsf{params},\mathsf{PK}); y_0 = \mathsf{sVRF.Eval}(\mathsf{params},\mathsf{SK},x)\ ;\\ y_1 \leftarrow R(\mathsf{params})\ ;\ b \xleftarrow{\$} \{0,1\}\ ;\\ (Q'_e, Q'_p, b') \leftarrow \mathcal{A}^{\mathsf{sVRF.Eval},\mathsf{sVRF.Prove}(\mathsf{params},\mathsf{SK},-)}(\mathsf{st}_A, y_b)\ :\\ b' = b \wedge x \notin (Q_e \cup Q_p \cup Q'_e \cup Q'_p)] \leq 1/2 + \nu(\lambda)$$

where Q_e and Q_p denote the contents of the query tape that records $\mathcal{A}$'s queries to $\mathsf{sVRF.Eval}$ and $\mathsf{sVRF.Prove}$ oracles respectively in the first query phase and Q'_e and Q'_p denote the contents of the query tape in second query phase.

Verifiability: For all security parameters $\lambda \in \mathbb{N}$, for all $\mathsf{params} \leftarrow \mathsf{sVRF.Setup}(1^\lambda)$ and for all PPT $\mathcal{A}$ there exists a negligible function $\nu()$ such that,

$$\Pr[(\mathsf{PK}, x, y_1, \pi_1, y_2, \pi_2) \leftarrow \mathcal{A}(\mathsf{params})\ :\\ y_1 \neq y_2\ \wedge\ \mathsf{sVRF.Verify}(\mathsf{params},\mathsf{PK},x,y_1,\pi_1) = 1 \wedge\\ \mathsf{sVRF.Verify}(\mathsf{params},\mathsf{PK},x,y_2,\pi_2) = 1] \leq \nu(\lambda)$$

Simulatability: $(\mathsf{sVRF.Setup},\mathsf{sVRF.KeyGen},\mathsf{sVRF.Eval},\mathsf{sVRF.Prove},\mathsf{sVRF.Verify})$ is a simulatable VRF if there exist algorithms $(\mathsf{SimSetup},\mathsf{SimKeyGen},\mathsf{SimProve})$ described as follows:

– $(\mathsf{params},\mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda)$: The $\mathsf{SimSetup}$ algorithm takes in the security parameter and outputs global parameters params and also a trapdoor td with it.
– $(\mathsf{PK},\mathsf{SK}) \leftarrow \mathsf{SimKeyGen}(\mathsf{params},\mathsf{td})$: The $\mathsf{SimKeyGen}$ algorithm takes in the global parameters params and the trapdoor td generated by $\mathsf{SimSetup}()$ and outputs a public key-secret key pair.
– $\pi \leftarrow \mathsf{SimProve}(\mathsf{params},\mathsf{SK},x,y,\mathsf{td})$: The $\mathsf{SimProve}$ algorithm takes in $\mathsf{params},\mathsf{SK},\mathsf{td}$, a point x and its sVRF evaluation y for which you want to generate the proof and it outputs the sVRF proof π.

These algorithms should be such that the distribution $\mathsf{Setup}(1^\lambda)$ is computationally indistinguishable from the distribution $\mathsf{SimSetup}(1^\lambda)$ and for all PPT $\mathcal{A}$, views of $\mathcal{A}$ in the following two games are indistinguishable:

Real: $(\mathsf{params}) \leftarrow \mathsf{sVRF.Setup}(1^\lambda)$; $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{sVRF.KeyGen}(\mathsf{params})$; $\mathcal{A}(\mathsf{params}, \mathsf{PK})$ gets access to the following oracle $\mathcal{O}$: On query x, $\mathcal{O}$ returns $y = \mathsf{sVRF.Eval}(\mathsf{params}, \mathsf{SK}, x)$ and $\pi = \mathsf{sVRF.Prove}(\mathsf{params}, \mathsf{SK}, x)$

Simulated: $(\mathsf{params}, \mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda)$; $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{SimKeyGen}(\mathsf{params}, \mathsf{td})$. $\mathcal{A}(\mathsf{params}, \mathsf{PK})$ gets access to the following oracle $\tilde{\mathcal{O}}$: On query x, $\tilde{\mathcal{O}}$ does the following: checks if x has previously been queried. If yes, then returns the stored answer.Otherwise, obtains $y \leftarrow R(\mathsf{params})$ and $\pi \leftarrow \mathsf{SimProve}(\mathsf{params}, \mathsf{SK}, x, y, \mathsf{td})$, and returns and stores (y, π).

B Simulatable VRF and Commitment Constructions

In this section we give two efficient instantiations of Simulatable VRF (sVRF) in the Random oracle model and a construction of simulatable commitments also in Random oracle model. The sVRF constructions are not new but they were not proven to be sVRFs. We give the proof (sketch) here.

B.1 sVRF based on DDH

We describe the construction of the VRF from [MBB+15] and prove that it is simulatable. The proof for pseudorandomness follows from DDH as observed in [MBB+15] and . The construction is as follows:

$\mathsf{sVRF.Setup}(1^\lambda)$: Choose a DDH group $\mathbb{G}$ of order q and let g be a generator. Choose hash functions $H_1 : \{0,1\}^* \mapsto \mathbb{G}$ and $H_2 : \{0,1\}^* \mapsto \mathbb{Z}_q^*$ and output $\mathsf{params} = (\mathbb{G}, g, H_1(), H_2())$

$\mathsf{sVRF.KeyGen}(\mathsf{params})$: Choose $k \xleftarrow{\$} \mathbb{Z}_q^*$ and output $\mathsf{SK} = k$ and $\mathsf{PK} = g^k$

$\mathsf{sVRF.Eval}(\mathsf{params}, \mathsf{SK}, x)$: Output $y = (H_1(x))^{\mathsf{SK}}$

$\mathsf{sVRF.Prove}(\mathsf{params}, \mathsf{SK}, x)$: 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 $t = r - sk$.

$\mathsf{sVRF.Verify}(\mathsf{params}, \mathsf{PK}, x, y, \pi)$: Parse π as (s, t). Check that $s = H_2(x, g^t(\mathsf{PK})^s, h^t y^s)$ for $h = H_1(x)$

Proof of Simulatability: We describe the simulator:

- $\mathsf{SimSetup}(1^\lambda)$ is same as $\mathsf{sVRF.Setup}(1^\lambda)$ with $H_1(), H_2()$ modeled as random oracles. It outputs the same params

- $\mathsf{SimKeyGen}(\mathsf{params})$ is same as $\mathsf{sVRF.KeyGen}(\mathsf{params})$ and outputs $(\mathsf{SK}, \mathsf{PK})$ similarly.

- $\mathsf{SimProve}(\mathsf{params}, \mathsf{PK}, \mathsf{SK}, x, y)$: No given y which is the sVRF evaluation of x, the simulator has to output the corresponding proof. Simulator sets $H_1(x) = y^(1/k)$ and chooses $r \xleftarrow{\$} \mathbb{Z}_q^*$ and outputs proof as $\pi = (s, t)$ for $s = H_2(x, g^r, h^r)$ and $t = r - sk$ for $h = H_1(x) = y^{1/k}$

In the random oracle model, it is easy to see that the proofs output by the simulator have exactly the same distribution as in the real world.

B.2 sVRF based on co-GDH

We also prove that the signature scheme in [BGLS03] is a sVRF. For completeness, we recall the scheme here.

$\mathsf{sVRF.Setup}(1^\lambda)$: Output $(\mathbb{G}, \mathbb{G}_{\mathbb{1}}, e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_{\mathbb{1}}), H_1 : \{0,1\}^* \mapsto \mathbb{G}$ where,

- $\mathbb{G}, \mathbb{G}_1$ are multiplicative cyclic groups of prime order q where $(\mathbb{G}, \mathbb{G})$ are co-GDH.
- g is a generator of $\mathbb{G}$
- e is computable bilinear nondegenerate map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_{\mathbb{1}}$
- $H_1 : \{0,1\}^* \mapsto \mathbb{G}$ is a full domain hash function viewed as a random oracle that can be instantiated with a cryptographic hash function.

$\mathsf{sVRF.KeyGen}(\mathsf{params})$: Choose $k \xleftarrow{\$} \mathbb{Z}_q^*$ and output $\mathsf{SK} = k$ and $\mathsf{PK} = g^k$

$\mathsf{sVRF.Eval}(\mathsf{params}, \mathsf{SK}, x)$: Output $y = (H_1(x))^{\mathsf{SK}}$

$\mathsf{sVRF.Prove}(\mathsf{params}, \mathsf{SK}, x)$: The proof is empty.

$\mathsf{sVRF.Verify}(\mathsf{params}, \mathsf{PK}, x, y, \pi)$: Given the user's public key PK, a input x and its VRF value y, accept if $e(y, g) = e(H_1(x), \mathsf{PK})$ holds.

Proof of Simulatability: Similar to the simulatability proof of th previous sVRF.

B.3 Simulatable Commitments

$\mathsf{CS.Setup}(1^\lambda)$: Pick domain message space D, randomness space R and a CRHF H and output $\sigma = (D, R, H)$

$\mathsf{CS.Commit}(\sigma, m; r)$: Output $\mathsf{com}_\sigma = H(m, r)$

$\mathsf{CS.Open}(\sigma, m, r, \mathsf{com}_\sigma)$: Output decommitment value corresponding to commitment com_σ for message m and randomness r.

$\mathsf{CS.VerifyOpen}(\sigma, \mathsf{com}_\sigma, m, \tau)$: Check if $H(m, r) = \mathsf{com}_\sigma$.

Proof Sketch: The hiding and binding follows from collision-resistance of H. The simulatability follows from H being modeled as a random oracle.

C Proof of Theorem 1

The completeness of the VKD is straightforward. Let us now prove soundness: Suppose there exists S^* which outputs $(\mathsf{label}, (\{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}), \{(\mathsf{com}_t, \mathsf{acc}_t, \Pi_t^{\mathsf{Upd}})\}_{t=t_1}^{t_n}, t^*, j, (\pi, \mathsf{val}, \alpha))$ such that all the versions proofs verify, the query proof for label verifies at t^* for $t_j \le t^* < t_{j+1}$ and the audit from t_1 to t_n also verifies. We also verified that all the com_t given by the server in Π^{Ver} with respect to $\mathsf{com}_{\mathsf{acc}}$ are correct which is guaranteed by SA soundness.
First we will prove that $\alpha \neq j$. If it was, it means that the aZKS verification for "all" went through for $(\mathsf{label}|\ \alpha, \mathsf{val})$ i.e. $\mathsf{VKD.QueryVer}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{com}_{t^*}, \mathsf{label}|\ \alpha, \mathsf{val}, \pi, \alpha)$ which is part of the query proof for label. Also verification for "all" went through for $(\mathsf{label}|\ j, \mathsf{val}_j)$ i.e. $\mathsf{ZKS.QueryVer}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{com}_{t_j}, \mathsf{label}|\ j, \mathsf{val}_j, \pi^v)$ where π^v, val_j and com_{t_j} is part of the versions proof for label, in particular part of $\Pi_i \in \Pi^{\mathsf{Ver}}$. If $\alpha = j$, this directly contradicts the soundness of the "all" aZKS between times t_j and t^*.

Now we will prove that α cannot even be greater than j. If $\mathsf{VKD.HistVer}(\mathsf{pp}, \mathsf{PK}_{\mathsf{Gen}}, \mathsf{com}_{t_n}, \mathsf{acc}_{t_n}, \mathsf{label}, (\{(\mathsf{val}_i, t_i)\}_{i=1}^n, \Pi^{\mathsf{Ver}}) = 1$, in particular it means that none of the nodes $(\mathsf{label}|\ \gamma)$ are present in the "all"aZKS for any $\gamma > \alpha$ since the corresponding non-membership proofs have verified. Hence $\alpha < j$.

Hence $\alpha \in [1, j-1]$ such that version 1 was added at time t_1. But the proofs in $\Pi_i \in \Pi^{\mathsf{Ver}}$ have verified for all $i \in \{1, \dots, j\}$ which means that the membership of all $(\mathsf{label}|\ \beta)$ in the "old"-aZKS for $1 \leq \beta \leq j-1$ with com_β has been verified. Hence the proof $(\pi, \mathsf{val}, \alpha)$ which has to contain non membership proof for $\mathsf{label}|\ \alpha$ in the "old"-aZKS for t^* will again contradict the update soundness of the "old"-aZKS from t_j to t^*.

Finally all the soundness contradictions hold if none of nodes in either of the aZKS are deleted. This is ensured by the fact that the audit from t_1 to t_n has been successful. Hence if the server is able to break the soundness of the VKD, it will contradict the soundness of one of the underlying aZKS. Hence proved. □

Proof of Privacy: Our simulator $(\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4)$ will work as follows:

– $\mathcal{S}_1$ calls the Sim_1 of an append-only aZKS to twice output $(\mathsf{pp}', \mathsf{PK}_{\mathsf{all}}, \mathsf{st}_{\mathsf{all}}) \leftarrow \mathsf{Sim}_1(1^\lambda)$ and $(\mathsf{pp}'', \mathsf{PK}_{\mathsf{old}}, \mathsf{st}_{\mathsf{old}}) \leftarrow \mathsf{Sim}_1(1^\lambda)$ and output $\mathsf{pp} = (\mathsf{pp}', \mathsf{pp}'')$, $\mathsf{PK}_{\mathsf{Gen}} = (\mathsf{PK}_{\mathsf{all}}, \mathsf{PK}_{\mathsf{old}})$, $\mathsf{st}_1 = \mathsf{st}_{\mathsf{all}} \cup \mathsf{st}_{\mathsf{old}}$

– In order to simulate the publish oracle O_P, $\mathcal{S}_2$ calls Sim_2 with empty datastores to get $(\mathsf{com}_{\mathsf{all}}, \mathsf{st}_2) \leftarrow \mathsf{Sim}_2(1^\lambda, D_0, \mathsf{st}_1)$ and also sets up the "old" aZKS with $(\mathsf{com}_{\mathsf{old}}, \mathsf{st}_2') \leftarrow \mathsf{Sim}_2(1^\lambda, D_0', \mathsf{st}_1)$. It also builds SA on both $\mathsf{com}_{\mathsf{all},1}, \mathsf{com}_{\mathsf{old},1}$. It outputs $(\mathsf{com}_1 = (\mathsf{com}_{\mathsf{all},1}, \mathsf{com}_{\mathsf{old},1}), \mathsf{acc}_1 = (\mathsf{acc}_{\mathsf{all},1}, \mathsf{acc}_{\mathsf{old},1}), \mathsf{st}_S = (\mathsf{st}_2 \cup \mathsf{st}_2')$

– For $t > 0$, $\mathcal{S}_2$ gets $L(S_t) = (n = n_{\mathsf{new}}, u = n_{\mathsf{upd}} + n_{\mathsf{mark}})$. It calls Sim_4 of both the append-only aZKS with $(\mathsf{com}_{\mathsf{all},t}, \mathsf{st}_3, \pi_S) \leftarrow \mathsf{Sim}_4(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{st}_2, L_{\mathsf{all}}(S) = n + u)$ and $(\mathsf{com}_{\mathsf{old},t}, \mathsf{st}_3', \pi_S^o) \leftarrow \mathsf{Sim}_4(\mathsf{pp}, \mathsf{PK}_{\mathsf{old}},$
$\mathsf{st}_2', L_{\mathsf{old}}(S) = n)$. Each of the Sim_4 gets the number of new nodes to be added to that aZKS. Output $\mathsf{st}_t = (\mathsf{st}_3 \cup \mathsf{st}_3')$

  $\mathcal{S}_2$ also updates both the SA with the updated commitment values. Output $\mathsf{com}_t = (\mathsf{com}_{\mathsf{all},t},$ $\mathsf{com}_{\mathsf{old},t}), \mathsf{acc}_t = (\mathsf{acc}_{\mathsf{all},t}, \mathsf{acc}_{\mathsf{old},t}$ and $\Pi_t^{\mathsf{Upd}} = \pi_{S_t}, \pi_{S_t^o})$.

– $\mathcal{S}_3$ simulates the proofs oracle O_π by using the Sim_3 of aZKS. Once it gets the corresponding $(\mathsf{label}, \alpha, \mathsf{val})$, it computes $(\pi_1, \mathsf{val}_1) \leftarrow \mathsf{Sim}_3(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{label} \mid \alpha, \mathsf{val}, \mathsf{st}_S)$ and $(\pi_2, \mathsf{val}_2) \leftarrow \mathsf{Sim}_3(\mathsf{pp}, \mathsf{PK}_{\mathsf{all}}, \mathsf{label}|\ \mathsf{mark}|\ a, \text{"marker node"}, \mathsf{st}_S)$ where 2^a is the largest power of 2 less than α. Finally it computes $(\pi_3, \mathsf{val}_3) \leftarrow \mathsf{Sim}_3(\mathsf{pp}, \mathsf{PK}_{\mathsf{old}}, \mathsf{label} \mid \alpha, \perp, \mathsf{st}_S)$. Output $\Pi = (\pi_1, \pi_2, \pi_3, \alpha)$ and $\mathsf{val} = (\mathsf{val}_1, \mathsf{val}_2, \perp)$.

– Similarly for $\mathcal{S}_4$ simulating the versions oracle, once there is a query, $\mathcal{S}_4$ gets all the $\{(\mathsf{val}_i, t_i)\}$ pairs to be output, it can call Sim_3 multiple times to simulate all the proofs for $\alpha = n$ where n is the number of timestamps and values in $\{(\mathsf{val}_i, t_i)\}$.

We will prove privacy through the following hybrids:

Hybrid 0: This is the real privacy game defined in the VKD construction. 2.

Hybrid 1: This is the same as the previous hybrid except now uses simulated proofs for "all" aZKS

Hybrid 2: This is same as the previous hybrid except now simulated values for the "old" aZKS as well. Note that this hybrid is completely simulated according to the simulator described before.

We will now prove the indistinguishability of these hybrids:

Claim 1. *Hybrid* 0 $\approx$ *Hybrid* 1

Suppose there exists $\mathcal{A}$ who can distinguish between hybrids 0 and 1, then we can use him to break the zero-knowledge of the "all" aZKS. Let $\mathcal{B}$ be an adversary of the aZKS game. It will forward the public parameters that it receives to $\mathcal{A}$ and also setup the "old" aZKS honestly and the acc SA to give all the public parameters to $\mathcal{A}$. It will maintain a table T for labels and their update times. Whenever $\mathcal{A}$ queries to the publish oracle with a set S, $\mathcal{B}$ checks the corresponding version and computes a set S' with the marker nodes to query its update oracle on. It also honestly computes set S_{old} for the older versions to do the update on "old" aZKS. Whenever $\mathcal{B}$ queries for a proof, it queries the corresponding "all" aZKS proofs and computes the "old" proofs on its own to give to $\mathcal{B}$.

Hence if $\mathcal{A}$ has a non-negligible advantage of distinguishing between hybrids 1 and 2 then $\mathcal{B}$ has a non-negligible advantage in winning the aZKS zero-knowledge game, which gives us a contradiction.

Claim 2. *Hybrid* 1 $\approx$ *Hybrid* 2

It is similar to the previous proof such that now $\mathcal{B}$ is trying to use his game for the "old" aZKS and not the "all". The way of simulating the proofs is similar. Again, $\mathcal{B}$ maintains a table T of versions to monitor the exact sets to be given to its oracle.

D Proof of Theorem 2

The completeness of the strong accumulator is straightforward. We will now prove soundness: There are two ways for S^* to win the soundness game; either by producing two verifying proofs for two different values for some label wrt. the same com_{j^*} (Case 1) or by producing two verifying proofs for two different values for some label wrt. an update which violates the append-only property (Case 2). Suppose there was such a S^* then we can use it to break either the collision resistance of the underlying hash function.

Case 1: Let $\pi_1 = (h_l^1, \mathsf{sib}_1^1, \ldots, \mathsf{sib}_{m-1}^1)$ for $\mathsf{sib}_i = (x_i, h_{s_i}, t_{i,0}, t_{i,1})$ for membership of (y_1, val_1) and let $\pi_2 = (h_l^2, \mathsf{sib}_1^2, \ldots, \mathsf{sib}_{m-1}^2)$ for membership of (y_2, val_2) so that both are membership proofs for label. Let us assume towards contradiction and suppose there exists some $\mathcal{A}$ that can produce two verifying proofs with non negligible probability. We can use this $\mathcal{A}$ to break the collision resistance of hash function H.

Let $\mathcal{B}$ be the adversary in the collision resistance game of $\mathcal{H}$. $\mathcal{B}$ will give the hash function H that he receives to $\mathcal{A}$ and all other parameters generated honestly. Now suppose $\mathcal{A}$ comes up with π_1, π_2 with sibling paths as described above. $\mathcal{A}$ also has to come up with com_{j^*} such that both the sibling paths hash to this value. We have that $\mathsf{val}_1 \neq \mathsf{val}_2$ and of $h_{y_1} = h_{y_2}$ then there is already a collision for that value. Otherwise, there is collision at least one position till the root. $\mathcal{B}$ can use this collision to win the collision resistance game with non negligible probability which gives a contradiction. We can argue about the case of one membership proof and one non membership proof similarly.

Case 2: We want to prove that a malicious S^* cannot come up with an update S_{j^*} and proof $\pi_S^{j^*}$ such that an element present before in the database has its value modified after the update. If such an S^* exists then we can use it to break the collision resistance of H. Since the update proof verified it means that $Z_{\mathsf{old}}, Z_{\mathsf{new}}$ in $\pi_S^{j^*}$ exactly represent the labels in update set S_{j^*}, S_{j^*} does not contain label and hence the hash value for label is part of Z_{const}. However both π_1, π_2 verified with respect to $\mathsf{com}_{j^*}, \mathsf{val}_1$ and $\mathsf{com}_{j^*+1}, \mathsf{val}_2$ respectively for $\mathsf{val}_1 \neq \mathsf{val}_2$ and hence the hash values from the path of label upto the root have to differ somewhere. But since $\mathsf{com}_{j^*}, \mathsf{com}_{j^*+1}$ both are consistent with the same Z_{const}, there has to be a collision somewhere on the path of label upto the root contradicting collision resistance of H.

E Proof of Theorem 3

The completeness and soundness follows from the completeness and soundness of the strong accumulator and that of the underlying commitment scheme and sVRF. We will now prove zero-knowledge. Let ($\mathsf{SA.Setup}, \mathsf{SA.Gen}, \mathsf{SA.CommitDS}, \mathsf{SA.Query}, \mathsf{SA.Verify}, \mathsf{SA.UpdateDS}, \mathsf{SA.VerifyUpd}$) be a compete and sound SA. Our simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2, \mathsf{Sim}_3, \mathsf{Sim}_4)$ will work as follows:

Sim_1 works as follows: Run $(\mathsf{params}, \mathsf{td}_S) \leftarrow \mathsf{sVRF.SimSetup}(1^\lambda)$ and then run $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{sVRF.SimKeyGen}(\mathsf{params}, \mathsf{td}_S)$. Also run $(\sigma, \mathsf{td}_C) \leftarrow \mathsf{CS.SimSetup}(1^\lambda)$. Do $\mathsf{pp}_{\mathsf{SA.}} \leftarrow \mathsf{SA.Setup}(1^\lambda)$ Output $\mathsf{pp} = \mathsf{pp}_{\mathsf{SA.}}$, $\mathsf{PK}_{\mathsf{Gen}} = [\mathsf{PK}, \mathsf{params}, \sigma]$ and $\mathsf{st}_1 = (\mathsf{td}_S, \mathsf{td}_C, \mathsf{SK})$

Sim_2 works as follows: Given the size of datastore n, the simulator choose n values at random $r_1, r_2, \ldots, r_n \xleftarrow{\$} \{0,1\}^\lambda$ which will be a substitute for the n sVRF evaluations of the labels in the datastore. For $1 \leq i \leq n$, compute $v_i = \mathsf{CS.SimCommit}(\sigma, \mathsf{td}_C)$. Now Sim_2 has $\mathsf{D}' = \{(r_1, v_1), \ldots, (r_n, v_n)\}$ which it will commit to.

Compute $(\mathsf{com}_{\mathsf{SA.}}) \leftarrow \mathsf{SA.CommitDS}(\mathsf{pp}, \mathsf{D})$

Output $\mathsf{com} = \mathsf{com}_{\mathsf{SA.}}$ and $\mathsf{st}_2 = (R = \{r_1, \ldots, r_N\}, v_1, \ldots, v_n, \mathsf{D}')$

Sim_3 works as follows: Initialize a list L of queries as empty. On query for element label, learn the corresponding val from oracle access to the datastore.

Check if label exists in L and if yes output sVRF evaluation y as the corresponding r_i value stored with label. Else, choose $r_i \xleftarrow{\$} R$ and store (label, r_i) in L. Also generate $\pi_{r_i} = \mathsf{sVRF.SimProve}(\mathsf{params}, \mathsf{SK}, \mathsf{label}, r, \mathsf{td}_S)$. Retrieve the v_i associated to r_i and run $\mathsf{SimOpen}(\sigma, v_i, \mathsf{td}_C, (\mathsf{label}, \mathsf{val}))$ to get a decommitment value τ to $(\mathsf{label}, \mathsf{val})$. Now compute $(\pi_{\mathsf{SA.}}, v_i) \leftarrow \mathsf{SA.Query}(\mathsf{pp}_{\mathsf{SA.}}, \mathsf{D}', r_i)$.

Output $\Pi = [(r_i, \pi_{r_i}, \tau, v_i, \pi_{\mathsf{SA.}}), \mathsf{val}]$

Sim_4 works as follows: Suppose that there is an update request for set S such that $|S| = k$. It initializes a set $S_{\mathsf{SA.}}$ to empty. Create k dummy (l, v) pairs, by choosing k random values and computing corresponding simulated commitments $v = \mathsf{CS.SimCommit}(\sigma, \mathsf{td}_C)$. Add all of them to $S_{\mathsf{SA.}}$. Do $(\mathsf{com}', \mathsf{D}'', \pi_S) \leftarrow \mathsf{SA.UpdateDS}(\mathsf{pp}, \mathsf{D}', S_{\mathsf{SA.}})$. Update the simulator state to include the new (r, v) values.

We will now prove that for any malicious verification algorithm $\mathcal{A}$, the interaction with the real algorithms in indistinguishable from the interaction with the simulator described above. The proof will go through following hybrids:

Hybrid 0: This is the zero knowledge game defined for the actual aZKS definition. 3.1.

Hybrid 1: This is the same as the previous hybrid except now use the simulator of the sVRF for $(\mathsf{params}, \mathsf{td}_S) \leftarrow \mathsf{SimSetup}(1^\lambda)$. Use $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{SimKeyGen}(\mathsf{params}, \mathsf{td}_S)$ and output these as $\mathsf{PK}_{\mathsf{Gen}}, \mathsf{st}_{\mathsf{Gen}}$.

Instead of computing $z_i = \mathsf{sVRF.Eval}(\mathsf{params}, \mathsf{SK}, \mathsf{label})$, choose N random values corresponding to each label and use $\pi_{r_i} \leftarrow \mathsf{SimProve}(\mathsf{params}, \mathsf{SK}, \mathsf{label}, r_i, \mathsf{td}_S)$ to create the corresponding proof. Use the datastore as $\{(r_i, \mathsf{CS.Commit}(\sigma, (\mathsf{label}_i, \mathsf{val}_i); s_i))\}$ pairs.

On query for some label, if $\mathsf{val} \neq \perp$, there exists r_i value corresponding to it and the simulated proof π_{r_i}. If $\mathsf{val} = \perp$, choose a random r such that $r \neq r_i$ for all $1 \leq i \leq N$. Correspondingly generate $\pi_r \leftarrow \mathsf{SimProve}(\mathsf{params}, \mathsf{SK}, \mathsf{label}, r_i, \mathsf{td}_S)$. Use these (r_i, π_{r_i}) values in the proofs.

Hybrid 2: This is same as the previous hybrid except now $(\sigma, \mathsf{td}_C) \leftarrow \mathsf{CS.SimSetup}(1^\lambda)$ and use that as part of pp. While committing to $(\mathsf{label}, \mathsf{val})$ pairs, initially as well as in updates, choose randomness s_i and create commitments $\mathsf{com}_i \leftarrow \mathsf{CS.SimCommit}(\sigma, \mathsf{td}_C; s_i)$.

On query label_i, which exists in the database, compute decommitments using $\tau = \mathsf{CS.SimOpen}$ $(\sigma, \mathsf{td}_C, (\mathsf{label}, \mathsf{val}), \mathsf{com}_i)$. This hybrid is in fact completely simulated.

We will now prove the indistinguishability of these hybrids:

Claim 3. *Hybrid* 0 $\approx$ *Hybrid* 1

Let us assume towards contradiction and suppose that $\mathcal{A}$ can distinguish between the two hybrids with non negligible probability. We can use this $\mathcal{A}$ to break the simulatability of the sVRF.

Let $\mathcal{B}$ be an adversary in the simulatability game of the sVRF ($\mathsf{sVRF.Setup}, \mathsf{sVRF.KeyGen}$, $\mathsf{sVRF.Eval}, \mathsf{sVRF.Prove}, \mathsf{sVRF.Verify}$). $\mathcal{B}$ gets real or simulated params from the simulatability game which it provides to $\mathcal{A}$. $\mathcal{B}$ also runs the setup to get $\sigma \leftarrow \mathsf{CS.Setup}(1^\lambda)$ for a computationally hiding perfectly binding commitment scheme ($\mathsf{CS.Setup}, \mathsf{CS.Commit}, \mathsf{CS.Open}$) and the SA setup to give all the parameters to $\mathcal{A}$. On receiving the database from $\mathcal{A}$, it uses $\mathsf{SA.CommitDS}()$ on the modified datastore with the pairs $\{(r_i, \mathsf{CS.Commit}(\sigma, (\mathsf{label}_i, \mathsf{val}_i); s_i))\}$ and sends that commitment to $\mathcal{A}$. Whenever $\mathcal{A}$ queries for proofs for updates it will give query its oracle with respect to that (label_i, r_i) to get a real or simulated proof. It computes the remaining proof according to $\mathsf{SA.Query}()$ and gives it to $\mathcal{A}$.

Hence if $\mathcal{A}$ has a non-negligible advantage of distinguishing between hybrids 0 and 1 then $\mathcal{B}$ has a non-negligible advantage in winning the simulatability game of the underlying sVRF since it will be able to distinguish between real and simulated proofs, which gives us a contradiction.

Claim 4. *Hybrid* 1 $\approx$ *Hybrid* 2

Let us assume towards contradiction and suppose $\mathcal{A}$ can distinguish between the two hybrids with non negligible probability. We can use this $\mathcal{A}$ to break the simulatability of the commitment scheme.

Let $\mathcal{B}$ be the adversary in the simulatability game of the commitment scheme. It gets σ which is either the an output of the $\mathsf{CS.Setup}(1^\lambda)$ or $\mathsf{CS.SimSetup}(1^\lambda)$ and it forwards that to $\mathcal{A}$ along with the sVRF simulated parameters. It then gives the the (label, pairs from $\mathcal{A}'$s database to its commitment oracle and gets commitments $v_1, \ldots, v_n$. $\mathcal{B}$ now uses $\mathsf{SA.Setup}()$ on the (r_i, v_i) pairs and gives the commitment to $\mathcal{A}$. When $\mathcal{A}$ queries for proofs, it will use its commitment oracle to get either $\mathsf{CS.Open}$ value or $\mathsf{CS.SimOpen}$ value and give the corresponding τ_i as part of the proof. Whenever there is an update, $\mathcal{B}$ asks for that many new commitment values to its oracle.

Hence if $\mathcal{A}$ has a non-negligible advantage of distinguishing between hybrids 1 and 2 then $\mathcal{B}$ has a non-negligible advantage in winning the simulatability game of the commitment scheme since it will be able to distinguish between real and simulated commitments and openings, which gives us a contradiction.

Hence proved that there exists a Sim that can produce an indistinguishable distribution from that of a real interaction for any adversary $\mathcal{A}$. $\square$

F Proof Overview of Theorem 4

Completeness is straightforward. For soundness we want to make sure that if a label is updated at times $t_1, \ldots, t_n$ with values $v_1, \ldots, v_n$, if audit is successful between $[t_1, t_n]$ then for some t^* such that $t_j \leq t^* < t_{j+1}$, a malicious server cannot give a verifying proof π^* for some $v^* \neq v_j$. Since a client monitors her key at every epoch, it means that the client verified proof π_j for (label, v_j) at epoch t_j. Hence at t^* there will be two verifying proofs for label with two different values v_j and v^*. This means if the root is the same value h_{root}, there has to be a collision somewhere along the path from label to the root for both the proofs to verify. This contradicts the collision resistance of H.

Now we give an overview of L-privacy: Simulator first sets up simulated parameters of sVRF and simulatable commitment scheme. For $t = 1$, simulator gets the size of the set $|S_1| = n$. It chooses n random values $Y = \{y_1, \ldots, y_n\}$ and builds a hash tree the same way as an honest server. Instead of real commitments, it uses simulated commitments for values v_i. It outputs corresponding h_{root} and maintains the hash chain honestly. On receiving a query for some label, it gets the corresponding value. If the label is queried for the first time, simulator chooses a random position $y \in Y$ for the label and simulates the sVRF proof and commitment opening accordingly. At later points of time, simulator also gets $P(\mathsf{label}_i)$ for the labels that are already present and being updated. It will simulate new commitments for those labels and update the hash values upto the root accordingly. So if some label_i is queried, updated and queried again, the simulator can give consistent proofs for the same just as the server does in reality. Hence the leakage in terms pseudonyms of updated labels is inevitable for this construction causing a tracing attack. $\square$