Oblivious Accumulators

Foteini Baldimtsi[1], Ioanna Karantaidou[1*], and Srinivasan Raghuraman[2]

[1] George Mason University {foteini,ikaranta}@gmu.edu
[2] Visa Research and MIT srraghur@visa.com

Abstract. A cryptographic accumulator is a succinct set commitment scheme with efficient (non-)membership proofs that typically supports updates (additions and deletions) on the accumulated set. When elements are added to or deleted from the set, an update message is issued. The collection of all the update messages essentially leaks the underlying accumulated set which in certain applications is not desirable.
In this work, we define *oblivious accumulators*, a set commitment with concise membership proofs that *hides the elements and the set size* from every entity: an outsider, a verifier or other element holders. We formalize this notion of privacy via two properties: *element hiding* and *add-delete indistinguishability*. We also define *almost-oblivious accumulators*, that only achieve a weaker notion of privacy called *add-delete unlinkability*. Such accumulators hide the elements but not the set size. We consider the trapdoorless, decentralized setting where different users can add and delete elements from the accumulator and compute membership proofs. We then give a generic construction of an oblivious accumulator based on key-value commitments (KVC). We also show a generic way to construct KVCs from an accumulator and a vector commitment scheme. Finally, we give lower bounds on the communication (size of update messages) required for oblivious accumulators and almost-oblivious accumulators.[3]

1 Introduction

A cryptographic *accumulator* [6] is a set commitment, i.e., a compact representation of a set of elements, as a short digest C. It allows a *prover* to generate a short proof of membership w_x, often called a witness, for any element x that has been accumulated, or non-membership proof for any element in the accumulator domain that has not been accumulated. A *verifier* can efficiently verify such proofs using the digest alone without the need to access the entire set. Since their inception, accumulators have been considered in various settings with varying capabilities, leading to a rich taxonomy. Accumulators that only support membership proofs are called *positive*, while those supporting only non-membership proofs are called *negative*. An accumulator that supports both membership and non-membership proofs is called *universal*. Regarding updating the accumulated

[*] Part of this work was done while the second author was an intern at Visa Research.
[3] The authors grant IACR a non-exclusive and irrevocable license to distribute the article under the https://creativecommons.org/licenses/by-nc/3.0/

set, accumulators are called *additive* if they only allow for additions of new elements, *negative* or *subtractive* if they only allow for deletions, and *dynamic* if they support both operations. There exist a variety of accumulator constructions proposed in the literature with different properties and under different computational assumptions [11,28,31,19,3,10,45].

Accumulators have also been classified into two main categories based on the entity who is responsible for updating the accumulated set: *trapdoor-based* accumulators managed by a trusted party, and *trapdoorless* or *strong* accumulators. In a trapdoor-based accumulator, a trusted entity known as the *accumulator manager*, holds some secret information/trapdoor and has the ability to efficiently add or delete elements and create witnesses. Whenever a new element is added to the accumulator, the accumulator manager issues the corresponding membership proof w_x. On the other hand, trapdoorless accumulators allow for public additions or deletions of elements without relying on a trusted entity. Users adding new elements to the accumulator can compute (and later update) their corresponding witnesses themselves. Finally, some accumulator constructions have an interesting property called proof batching [8,37]. In this case, the prover can further compact proofs for multiple elements into one proof of size sublinear in the number of elements, that verifies faster than when compared to verifying each individual proof.

Accumulators have found numerous applications, with the most popular being anonymous credentials [11,1,4,10,20], group signatures [32,29,12], cloud storage [40,45], and more recently, stateless [8,17] and privacy-preserving cryptocurrencies such as ZCash [30] and RingCT 2.0 [38] proposed for Monero. Revocation of anonymous credentials is one of the most prominent applications of trapdoor-based accumulators. The credential issuing authority, that is responsible for granting credentials, also serves as the accumulator manager and maintains a list of valid credentials in the form of an accumulator. When a new credential is issued, it is added to the accumulated set by the issuing authority and the corresponding witness is sent to the user along with the credential. To use the credential, the user will have to prove membership in the accumulator in order to demonstrate to the verifier that the credential is still valid. The issuing authority/accumulator manager is responsible for removing revoked credentials from the accumulated set. Trapdoorless accumulators are mostly used in applications where set updates and witness creation are performed by an untrusted party, for example a cloud storage provider. Recently, trapdoorless accumulators have been proposed for data compression in decentralized settings. An accumulator can be used to construct a stateless blockchain, where anyone can add elements, as long as they can prove that their update is consistent with the previous state of the chain.

Privacy in accumulators. The classic definition of a cryptographic accumulator does not offer any privacy preserving properties: an accumulator is not a hiding commitment and can leak information about the set. Information can be leaked from the accumulator digest itself, a membership proof, and most importantly from the update messages, that usually describe explicitly which element was added or deleted. This information can be used by any entity (proof holders and verifiers) in order to update their proof and/or the accumulator value. However, accumulators are often used in applications where privacy is needed for specific operations carried out on the accumulated set. A common example is in the context of anonymous credentials, where users may wish to hide the specific credential for which they are proving membership. To achieve this, a user can present a commitment to the credential and subsequently provide a zero-knowledge (ZK) proof, demonstrating that the committed value is indeed present in the accumulator. ZK proofs of (non-)membership for accumulators have been previously studied in the literature in the form of individual proofs [4,7,11] and batched proofs [14,37]. In the context of anonymous credentials, the notion of join-revoke unlinkability was previously introduced [5] which guarantees that the addition and revocation of the same credential should not be linkable. This idea could be extended to direct accumulator property of add-delete unlinikability and it is achieved by modular constructions as explained below. Finally, another flavor of privacy for accumulators is that of zero-knowledge accumulators for set operations [21,45,24,16]. In that setting, the guarantees provided by zero-knowledge assures that an external entity, such as a verifier, that gets to see *only* (non-)membership proofs and the accumulator digest, learns nothing else about the accumulated set.

Beyond the privacy-preserving accumulator application of anonymous credentials mentioned above, there are other scenarios where there is a need to conceal the elements of the accumulated set from *all participants*, including witness holders (and the accumulator manager in the case of trapdoor-based constructions), and additionally hide the size of the accumulated set. Consider for example a smart contract that is executed on a public blockchain and is using an accumulator to hold the credentials for all the customers of an organization in order to efficiently check (non-)membership. In such a scenario, it is crucial to hide the accumulated elements, since having access to the credentials of a user might allow for unauthorized access. At the same time, it would also be useful to hide the total number of the elements accumulated in order to conceal the size of the "customer base" of the organization. This brings us to the following interesting question:

Is it possible to construct a dynamic accumulator that hides the accumulated set (both its elements and its size)?

In this work we answer this question affirmatively. For the first time, we define the notion of *oblivious accumulators*, that achieve all the above properties and we provide a construction that achieves our definition. We also show that our construction meets the standard information-theoretic lower bounds that prove for oblivious accumulators, along the lines of similar bounds that have been shown for dynamic accumulators [9] and revocable proof systems [18]. More specifically, we show that the total communication cost in the form of updates in our construction is equal to what must necessarily be stored as a state for an oblivious accumulator with the privacy properties that we propose.

1.1 Our contributions

In Section 4 we define a dynamic positive trapdoorless oblivious accumulator. At a high level, an oblivious accumulator supports the typical accumulator operations. Setup generates the parameters and the initial accumulator C. Add inserts a new element x into the accumulator and computes its membership proof w_x and the new accumulator C'. Similarly, Del deletes an element from the accumulator and computes the new accumulator C'. The addition and deletion processes also release an update message u. The rest of the proof holders run $\mathsf{MemProofUpdate}$ with u as input and update their proofs. The digest U of all update messages can be used to update membership proofs at any point in time, thus alleviating the need for parties to always be online and process update messages as they come in. Anyone can check whether x has been accumulated by running MemVer, given the proof w_x and the accumulator C. A feature specific to our oblivious accumulator is the following: in order to add x, Add also generates auxiliary information aux (only known to the user that holds x), which is necessary in order to later construct a membership proof for x or to delete x.[4]

On top of the typical accumulator security properties, we define new privacy-related properties. The first property, *element hiding*, guarantees that one cannot learn x by looking at the corresponding update message u. Then we define *add-delete indistinguishability* which guarantees that one cannot even tell what type of operation, i.e., an add or a delete, took place, even with the knowledge of the accumulated set before the update. This property is equivalent to hiding the size of the set since, if one could track the number of additions and deletions, they could infer the current size of the set. We also formalize the notion of *add-delete unlinkability* as an intermediate privacy goal (this property was previously discussed in [5] under the term "join-revoke" unlinkability for a revocation system and not directly for an accumulator). Add-delete unlinkability states that despite having access to addition update messages, that may not hide the element, and with the knowledge that an update u corresponds to a deletion, one can still not link which updates refer to the same element. We note that add-delete indistinguishability implies add-delete unlinkability.

An almost-oblivious accumulator. After defining the privacy properties of an oblivious accumulator, we focus on constructions that satisfy these properties. In order to build some intuition, we start by describing a construction, called *almost-oblivious* accumulator, that supports element hiding and add-delete unlinkability but not add-deleted indistinguishability. Aa intuitive way to achieve element hiding: is to accumulate hiding commitments of the element instead of adding them in the clear. Achieving add-delete unlinkability though is a bit a more complex. An implicit solution for unlinkable additions and deletions was presented as Construction A in [5] and describes a modular construction of two accumulators $\mathsf{Acc} = (\mathsf{Acc}_1, \mathsf{Acc}_2)$, Acc_1 is additive and used for added elements and Acc_2 is additive and used for storing deleted elements. Acc_1 supports mem-

[4] In a sense, aux is a summary of how x was hidden in order to achieve the privacy properties that we discuss ahead.

bership proofs and Acc_2 supports non-membership proofs, in order for the overall accumulator Acc to support membership proofs. To prove membership of x in Acc, one has to present a membership proof for x in Acc_1 and a non-membership proof for x in Acc_2. To get add-delete unlinkability in this modular accumulator, one can, instead of adding x in Acc_1, add c_1, a hiding commitment to x. We can delete x by adding a different commitment c_2 in Acc_2. A membership proof w_x in Acc now consists of a membership proof of c_1 in Acc_1 and a non-membership proof of c_2 in Acc_2, together with openings of c_1, c_2 to the same element x. An observer is not be able to link c_1, c_2 unless they see w_x.

A more generic way to describe the modular structure is with two sets, without specifying compression techniques: the set of added elements (previously described as Acc_1) and the set of deleted elements (previously Acc_2). The anonymous cryptocurrency ZCash initially emerged as Zerocoin [30], an accumulator-based system for compressing the set of valid coins. ZCash follows the same modular structure. Random elements (serial numbers) are added as hiding commitments in the fist set and then added in the clear in the second set. The first set is further compressed using a Merkle tree and proofs of membership. The second set is kept as a list and in order to ensure that an element has not been deleted, a lookup operation is performed. The trade-off compared to using compression and non-membership proofs in order to save in space is that storing the whole list, allows for concurrency. Concurrency of operations is a special property that comes up in cryptocurrencies. In this case, a transaction for spending a valid coin x is a membership proof w_x together with a deletion for x. If deletions are compressed, then w_x needs to be updated to reflect the new digest and as a result, transactions cannot be submitted and validated in parallel.

These constructions, which we call *almost-oblivious* accumulators achieve element hiding and add-delete unlinkability. However, since the two sets of added and deleted elements are distinct, such constructions inherently fail to satisfy add-delete indistinguishability (and thus these constructions reveal the size of the accumulated set).

Our construction. Our goal is to build an oblivious accumulator that achieves element hiding and the stronger property of add-delete indistinguishability which will allow for hiding the size of the accumulated set. In order to achieve add-delete indistinguishability, we will use a single data structure for both additions and deletions, as opposed to the modular constructions from above. Both addition and deletion operations will result in new elements being added in the data structure in a way that is indistinguishable yet sound when it comes to proving (non-)membership. A first idea is to insert flags in random-looking positions of a vector commitment (VC) scheme. The positions are derived from the inserted element x and some randomness. However, a VC has to commit to a specified number of positions when initialized. Even if the VC supported a procedure that allows to extend the length of the vector (which some VCs do), this would not be enough, as we would need the random-looking positions to have high entropy, i.e., come from a very large space, which would mean that the length of the vector would have to be exponentially large. For VCs that we know today, this would be grossly inefficient if not impossible. Instead, our approach is to use Key-Value Commitments (KVC) [2,42]. A KVC is a dynamic length, sparse vector commitment with elements $(k,\, v)$, k being the key and v being the inserted value. Its security property is key binding, meaning that proof of opening for $(k,\, v)$ guarantees that there is a tuple of the form $(k,\, \cdot)$ and it is impossible to find a different proof for $(k,\, v')$, and $v \neq v'$. At the same time, KVC can support key non-membership proofs (i.e., proofs that a key k' was never inserted), a property that will be used by our construction in order to prove that an element has not been deleted. Looking ahead, the keys that we will use are essentially the random-looking positions from our prior discussion.

In Section 5, we present a generic trapdoorless, positive, dynamic oblivious accumulator from any KVC scheme that supports non-membership. Our construction is proven secure in the Random Oracle Model. Our construction briefly works as follows. The position of addition is decided by the output of a commitment using a hash function H_1 and the position of a deletion is decided by H_2. More specifically, the user who wishes to add x, generates randomness r and adds a value $v = 1$ in position $H_1(x, r)$. It also sets auxiliary information $\mathsf{aux} = r$ that allows them to delete x and compute a proof of membership on x. A proof of membership consists of a proof of opening for key-value pair $(H_1(x, r),\, 1)$ and a proof that an element in position $H_2(x, r)$ has not been added. We complete our oblivious accumulator construction with a non-membership proof for key $H_2(x, r)$, used by the prover to prove that x has not been deleted. Finally, a deletion for x happens with adding the key-value pair $(H_2(x, r),\, 1)$. This makes a membership proof invalid. In Section 5.5 we briefly discuss an extension of our constructions that allows for the accumulation of unique elements.

In Section 3, we show, as a side result, how to construct a KVC scheme with non-membership, using a universal accumulator and vector commitment with extendable length. We add keys k in the accumulator and elements (k,v) in the vector sequentially. Our construction inherits the position binding properties and additive updates from the vector commitment and the non-membership proofs from the accumulator. Moreover, it preserves efficiency properties of the underlying schemes such as constant commitment and proof size, efficient updates, etc., or features such as proof batching and aggregation or cross-aggregation. This construction is of independent interest as it gives a generic way to build a KVC and can allow for constructions under different assumptions than the ones currently known.

Finally, we investigate the lower bounds for almost-oblivious and oblivious accumulators with constant proof size and constant digest. In more detail, in Section 6, we follow the analysis of [9,18] to prove lower bounds on the communication costs of deletions, and then show that the obliviousness properties of oblivious accumulators translate these bounds for arbitrary operations, not just deletions. In light of this result, our construction has optimal communication cost. In the case of almost-oblivious accumulators, we leverage add-delete unlinkability to show the lower bound, which implies that constructions such as ZCash are essentially optimal.

1.2 Future Directions

Our work is on positive oblivious accumulators, meaning accumulators that only support membership. It is an open problem whether oblivious accumulators with non-membership exist. The challenge for the prover is how to prove non-membership in a set they are unaware of. Solutions so far refer to uncompressed sets that can be processed by the verifier. An indicative way to implement oblivious non-membership in a set of elements is to perform encrypted search. The prover converts their element into an encrypted keyword and hands it to the verifier with a proof of correct computation. The verifier can encrypt the set and ensure the keyword is not found. It is unclear whether a short proof exists in a setting where the verifier only holds a short set digest.

2 Preliminaries

2.1 Notation

For $n \in \mathbb{N}$, let $[n] = \{1, 2, \ldots, n\}$. Let $\lambda \in \mathbb{N}$ denote the security parameter. Symbols in boldface such as $\mathbf{a}$ denote vectors. By a_i we denote the i-th element of the vector $\mathbf{a}$. For a vector $\mathbf{a}$ of length $n \in \mathbb{N}$ and an index set $I \subseteq [n]$, we denote by $\mathbf{a}|_I$ the sub-vector of elements a_i for $i \in I$ induced by I. By $\text{poly}(\cdot)$, we denote any function which is bounded by a polynomial in its argument. An algorithm $\mathcal{A}$ is said to be PPT if it is modeled as a probabilistic Turing machine that runs in time polynomial in λ. Informally, we say that a function is negligible, denoted by negl, if it vanishes faster than the inverse of any polynomial. If S is a set, then $x \leftarrow_\$ S$ indicates the process of selecting x uniformly at random from S (which in particular assumes that S can be sampled efficiently). Similarly, $x \leftarrow_\$ \mathcal{A}(\cdot)$ denotes the random variable that is the output of a randomized algorithm $\mathcal{A}$.

2.2 Compressing Primitives

In this section, we briefly recall definitions of compressing primitives for sets (*accumulators*), vectors (*vector commitments*), and key-value maps (*key-value commitments*). We present the various algorithms underlying the primitives, along with their corresponding correctness and security properties. We include related work for each primitive in Appendix A.

2.2.1 Accumulators

An accumulator (Acc) allows one to commit to a set in such a way that it is later possible to prove or disprove that elements were are in the set. We require an accumulator to be *concise* in the sense that the size of the accumulator string C is independent of the size of the set. We describe the primitive in the universal (supports membership and non-membership proofs) dynamic (supports additions and deletions) setting. We also assume that we are in the trapdoorless setting, i.e., updates can be performed with publicly available information.

We set up the following notation for a set: A set $S \subseteq \mathcal{D}$ is a collection of elements $x \in \mathcal{D}$ where $\mathcal{D}$ is the accumulator domain. We define an accumulator Acc as a non-interactive primitive that can be formally described via the following algorithms:

- $(\mathsf{pp}, C) \leftarrow_{\$} \mathsf{Setup}(1^\lambda)$: On input the security parameter λ, the setup algorithm outputs some public parameters pp (which implicitly define the accumulator domain $\mathcal{D}$) and the initial accumulator string C to the empty set. All other algorithms have access to the public parameters.
- $(C, w_x, u) \leftarrow \mathsf{Add}(C, x)$: On input an accumulator string C and an element $x \in \mathcal{D}$, the addition algorithm outputs a new accumulator string C, a membership proof w_x (that $x \in S$), and update information u.
- $(C, u) \leftarrow \mathsf{Del}(C, x, U)$: On input an accumulator string C, an element $x \in \mathcal{D}$, and the digest of all update information U produced until the current point in time, the deletion algorithm outputs a new accumulator string C and update information u.
- $w_x \leftarrow \mathsf{MemProofUpdate}(w_x, u)$: On input a membership proof w_x and update information u, the membership proof update algorithm outputs an updated membership proof w_x.
- $\overline{w_x} \leftarrow \mathsf{NonMemProofCreate}(x, U)$: On input an element $x \in \mathcal{D}$ and the digest of all updated information U produced until the current point in time, the non-membership proof creation algorithm outputs a non-membership proof $\overline{w_x}$ (that $x \notin S$).
- $\overline{w_x} \leftarrow \mathsf{NonMemProofUpdate}(\overline{w_x}, u)$: On input a non-membership proof $\overline{w_x}$ and update information u, the non-membership proof update algorithm outputs an updated non-membership proof $\overline{w_x}$.
- $0/1 \leftarrow \mathsf{MemVer}(C, x, w_x)$: On input an accumulator string C, an element $x \in \mathcal{D}$, and a membership proof w_x, the membership verification algorithm either outputs 1 (denoting accept) or 0 (denoting reject).
- $0/1 \leftarrow \mathsf{NonMemVer}(C, x, \overline{w_x})$: On input an accumulator string C, an element $x \in \mathcal{D}$, and a non-membership proof $\overline{w_x}$, the non-membership verification algorithm either outputs 1 (denoting accept) or 0 (denoting reject).

For correctness, we require that for all $\lambda \in \mathbb{N}$, for all honestly generated public parameters $\mathsf{pp} \leftarrow_{\$} \mathsf{Setup}(1^\lambda)$, if C is an accumulator to a set S, obtained by running a sequence of calls to Add and Del, w_x is a membership proof corresponding to an element $x \in \mathcal{D}$ for any $x \in S$, generated during the call to Add and updated by appropriate calls to $\mathsf{MemProofUpdate}$, then $\mathsf{MemVer}(C, x, w_x)$ outputs 1 with probability 1. Similarly, if $\overline{w_x}$ is a non-membership proof corresponding to an element $x \in \mathcal{D}$ for any $x \notin S$, generated by a call to $\mathsf{NonMemProofCreate}$ and updated by appropriate calls to $\mathsf{NonMemProofUpdate}$, then $\mathsf{NonMemVer}(C, x, \overline{w_x})$ outputs 1 with probability 1.

The security requirement for accumulators is that of *soundness*. We consider two notions of soundness, i.e., *weak* and *strong soundness*. To satisfy weak soundness, it must be computationally infeasible for any polynomially bounded adversary (with knowledge of pp) to come up with an *honestly generated*[5] accumulator and either a membership proof that certifies membership of an element that has not been added, or a non-membership proof that certifies non-membership of an element that has been added. To satisfy strong soundness, it must be computationally infeasible for any polynomially bounded adversary (with knowledge of pp) to come up with a *potentially adversarially generated* accumulator and a pair of membership and non-membership proofs that certify membership and non-membership, respectively, of the same element.

Alternative formalization. Some works alternatively formalize the algorithms Del and $\mathsf{NonMemProofCreate}$ to take S as input instead of U, but in most cases, the two hold similar information.

2.2.2 Vector Commitments

A vector commitment (VC) allows one to commit to a vector in such a way that it is later possible to open the commitment with respect to any specific index. We require a vector commitment to be *concise* in the sense that the size of the vector commitment string C is independent of the size of the vector. Furthermore, it must be possible to update the vector by updating the value of the vector at a specific position. We also assume that we are in the trapdoorless setting, i.e., updates can be performed with publicly available information.

We set up the following notation for a vector: A vector $\mathbf{v} \in \mathcal{D}^q$ of length q is a collection of q elements $v_i \in \mathcal{D}$[6] for $i \in [q]$. We define a vector commitment VC as a non-interactive primitive that can be formally described via the following algorithms:

- $(\mathsf{pp}, C) \leftarrow_\$ \mathsf{Setup}(1^\lambda, q)$: On input the security parameter λ and a length q, the setup algorithm outputs some public parameters pp (which implicitly define the vector commitment domain $\mathcal{D}$ and the vector length q) and the initial vector commitment string C to the vector of all 0s. All other algorithms have access to the public parameters. All other algorithms also have access to the initial proofs Λ_i for all $i \in [q]$ (that $v_i = 0$).
- $(C, u) \leftarrow \mathsf{Update}(C, (i, \delta))$: On input a vector commitment string C and an *additive* update value $\delta \in \mathcal{D}$, the update algorithm outputs a new vector commitment string C and update information u.
- $\Lambda_i \leftarrow \mathsf{ProofUpdate}(\Lambda_i, u)$: On input a proof Λ_i and update information u, the proof update algorithm outputs an updated proof Λ_i.
- $0/1 \leftarrow \mathsf{ProofVer}(C, (i, v), \Lambda_i)$: On input a vector commitment string C, a position $i \in [q]$, an element $v \in \mathcal{D}$, and a proof Λ_i, the proof verification algorithm either outputs 1 (denoting accept) or 0 (denoting reject).

[5] In the experiment defining security, we also assume that elements that have not yet been added are never requested to be deleted by the adversary.

[6] We assume that $0 \in \mathcal{D}$.

For correctness, we require that for all $\lambda \in \mathbb{N}$, for all honestly generated public parameters $\mathsf{pp} \leftarrow_{\$} \mathsf{Setup}(1^{\lambda})$, if C is the vector commitment to a vector $\mathbf{v}$, obtained by running a sequence of calls to Update, Λ_i is a proof corresponding to a position $i \in [q]$, updated by appropriate calls to $\mathsf{ProofUpdate}$, then $\mathsf{ProofVer}(C, i, v_i, \Lambda_i)$ outputs 1 with probability 1.

The security requirement for vector commitments is that of *position binding*. We consider two notions of soundness, i.e., *weak* and *strong position binding*. To satisfy weak position binding, it must be computationally infeasible for any polynomially bounded adversary (with knowledge of pp) to come up with an *honestly generated* vector commitment and a proof that certifies a value at any position different from the one in the vector that has been committed. To satisfy strong soundness, it must be computationally infeasible for any polynomially bounded adversary (with knowledge of pp) to come up with a *potentially adversarially generated* vector commitment and a pair of proofs that certify different values at the same position.

Alternative formalization. Some works alternatively formalize a vector commitment to not generate an initial vector commitment to the vector of all 0s, but rather to have an initial Commit procedure that takes a vector and generates a vector commitment to it. This formalization would usually be paired with a $\mathsf{ProofCreate}$ algorithm that takes the initial committed vector and a position and outputs the initial proof for that position. Some works also assume that Update takes as input the old and new values at a position, as opposed to an additive update–we say that such an Update algorithm is *non-oblivious*.

Positive length. Occasionally, we will also consider vector commitments which support a dynamic increase of the length of the vector that is being committed to. In this case, the vector commitment has an additional algorithm:

- $(\mathsf{pp}, C, u) \leftarrow_{\$} \mathsf{Extend}(1^{\lambda}, C)$: On input the security parameter λ and a vector commitment string C for a vector $\mathbf{v}$ of length q, the extend algorithm outputs new public parameters pp (corresponding to vectors of length $q+1$), the new vector commitment string C to the vector $\mathbf{v}'$ of length $q+1$, where $\mathbf{v}'|_{[q]} = \mathbf{v}$ and $v'_{q+1} = 0$, and update information u. All other algorithms have access to the initial proof Λ_{q+1} (that $v'_{q+1} = 0$).

2.2.3 Key-Value Commitments

A key-value commitment (KVC) allows one to commit to a key-value map in such a way that it is later possible to open the commitment with respect to any specific key. We require a key-value commitment to be *concise* in the sense that the size of the commitment string C is independent of the size of the map. We describe the primitive in the universal (supports membership and non-membership proofs) setting. Furthermore, it must be possible to update the map, by either adding new key-value pairs or updating the value corresponding to an existing key

We also assume that we are in the trapdoorless setting, i.e., updates can be performed with publicly available information.

We set up the following notation for a key-value map: A key-value map $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{V}$ is a collection of key-value pairs $(k, v) \in \mathcal{K} \times \mathcal{V}$. Let $\mathcal{K}_{\mathcal{M}} \subseteq \mathcal{K}$ denote the set of keys for which values have been stored in the map $\mathcal{M}$. We define a key-value commitment KVC as a non-interactive primitive that can be formally described via the following algorithms:

- $(\mathsf{pp}, C) \leftarrow_{\$} \mathsf{Setup}(1^{\lambda})$: On input the security parameter λ, the setup generation algorithm outputs some public parameters pp (which implicitly define the key space $\mathcal{K}$ and value space $\mathcal{V}$) and the initial commitment C to the empty key-value map. All other algorithms have access to the public parameters.
- $(C, \Lambda_k, u) \leftarrow \mathsf{Insert}(C, (k, v))$: On input a key-value commitment string C and a key-value pair $(k, v) \in \mathcal{K} \times \mathcal{V}$, the insertion algorithm outputs a new key-value commitment string C, a proof Λ_k (that $(k, v) \in \mathcal{M}$), and update information u.
- $(C, u) \leftarrow \mathsf{Update}(C, (k, \delta))$: On input a key-value commitment string C, a key $k \in \mathcal{K}$, and an *additive* update value $\delta \in \mathcal{V}$, the update algorithm outputs a new key-value commitment string C and update information u.
- $\Lambda_k \leftarrow \mathsf{ProofUpdate}(\Lambda_k, u)$: On input a proof Λ_k for some value corresponding to the key k and update information u, the proof update algorithm outputs an updated proof Λ_k.
- $\overline{\Lambda_k} \leftarrow \mathsf{NonMemProofCreate}(k, U)$: On input a key $k \in \mathcal{K}$ and the digest of all updated information U produced until the current point in time, the non-membership proof creation algorithm outputs a non-membership proof $\overline{\Lambda_k}$ (that $k \notin \mathcal{K}_{\mathcal{M}}$).
- $\overline{\Lambda_k} \leftarrow \mathsf{NonMemProofUpdate}(\overline{\Lambda_k}, u)$: On input a non-membership proof $\overline{\Lambda_k}$ and update information u, the non-membership proof update algorithm outputs an updated non-membership proof $\overline{\Lambda_k}$.
- $1/0 \leftarrow \mathsf{Ver}(C, (k, v), \Lambda_k)$: On input a key-value commitment string C, a key-value pair $(k, v) \in \mathcal{K} \times \mathcal{V}$, and a proof Λ_k, the verification algorithm either outputs 1 (denoting accept) or 0 (denoting reject).
- $0/1 \leftarrow \mathsf{NonMemVer}(C, k, \overline{\Lambda_k})$: On input a key-value commitment string C, a key $x \in \mathcal{K}$, and a non-membership proof $\overline{\Lambda_k}$, the non-membership verification algorithm either outputs 1 (denoting accept) or 0 (denoting reject).

For correctness, we require that for all $\lambda \in \mathbb{N}$, for all honestly generated public parameters $\mathsf{pp} \leftarrow_{\$} \mathsf{KeyGen}(1^{\lambda})$, if C is the key-value commitment to a key-value map $\mathcal{M}$, obtained by running a sequence of calls to Insert and Update, Λ_k is a proof corresponding to a key $k \in \mathcal{K}$ for any $k \in \mathcal{K}_{\mathcal{M}}$, generated during the call to Insert and updated by appropriate calls to $\mathsf{ProofUpdate}$, then $\mathsf{Ver}(C, (k, v), \Lambda_k)$ outputs 1 with probability 1 if $(k, v) \in \mathcal{M}$. Similarly, if $\overline{\Lambda_k}$ is a non-membership proof corresponding to a key $k \in \mathcal{K}$ for any $k \notin \mathcal{K}_{\mathcal{M}}$, generated by a call to $\mathsf{NonMemProofCreate}$ and updated by appropriate calls to $\mathsf{NonMemProofUpdate}$, then $\mathsf{NonMemVer}(C, k, \overline{\Lambda_k})$ outputs 1 with probability 1.

The security requirement for key-value commitments is that of *key binding*. We consider two notions of soundness, i.e., *weak* and *strong key binding*. To satisfy weak key binding, it must be computationally infeasible for any polynomially bounded adversary (with knowledge of pp) to come up with an *honestly generated* key-value commitment and either a proof that certifies membership of a key or a key-value pair that has not been inserted, or a non-membership proof that certifies non-membership of a key that has been inserted. To satisfy strong key-binding, it must be computationally infeasible for any polynomially bounded adversary (with knowledge of pp) to come up with a *potentially adversarially generated* key-value commitment and a either a pair of membership and non-membership proofs that certify membership and non-membership, respectively, of the same key, or a pair of proofs that certify different values for the same key.

Alternative formalization. Some works alternatively formalize the algorithm $\mathsf{NonMemProofCreate}$ to take $\mathcal{M}$ as input instead of U, but in most cases, two hold similar information. Some works also assume that Update takes as input the old and new values corresponding to a key, as opposed to an additive update–we say that such an Update algorithm is *non-oblivious*.

3 KVC based on Acc and VC

In Section 3.1, we show how to generically construct a key-value commitment using an accumulator and a vector commitment. The idea is to maintain an accumulator of the keys and a vector commitment of the values, tied by the positions. In realizing this, we will need the property that the vector commitment supports the procedure Extend that can dynamically increase the length of the vector that is being committed to, as described in Section 2.2.2. The efficiency of the final key-value commitment crucially depends on how efficient Extend is. We highlight that our generic construction allows us to achieve all desired properties of a KVC, including non-membership proofs.[7] It also provides a holistic way to look at existing constructions of KVCs, as we describe in Section 3.2.

3.1 Construction

Let Acc be an accumulator as described in Section 2.2.1, and let VC be a vector commitment as described in Section 2.2.2 that supports the procedure Extend. We will be designing a key-value commitment for the space of keys $\mathcal{K}$ which is the same as the space of elements $\mathcal{D}$ of Acc, and the space of values $\mathcal{V}$, where the space of elements $\mathcal{D}$ of VC is $\mathcal{K} \times \mathcal{V}$. We construct our key-value commitment KVC as follows:

– $(\mathsf{pp}, C) \leftarrow_\$ \mathsf{Setup}(1^\lambda)$: On input the security parameter λ, the setup generation algorithm:

[7] One can also readily support *key-deletion*, but we ignore this in our presentation.

  - runs $(\mathsf{pp}_{\mathsf{Acc}}, C_{\mathsf{Acc}}) \leftarrow_\$ \mathsf{Acc.Setup}(1^\lambda)$
  - runs $(\mathsf{pp}_{\mathsf{VC}}, C_{\mathsf{VC}}) \leftarrow_\$ \mathsf{VC.Setup}(1^\lambda, 0)$

  and finally outputs the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{VC}})$ and the initial commitment $C = (C_{\mathsf{Acc}}, C_{\mathsf{VC}}, 0)$ to the empty key-value map. All other algorithms have access to the public parameters.
- $(C, \Lambda_k, u) \leftarrow \mathsf{Insert}(C, (k, v))$: On input a key-value commitment string C and a key-value pair $(k, v) \in \mathcal{K} \times \mathcal{V}$, the insertion algorithm:
  - parses $C = (C_{\mathsf{Acc}}, C_{\mathsf{VC}}, q)$
  - runs $(C_{\mathsf{Acc}}, w_k, u_{\mathsf{Acc}}) \leftarrow \mathsf{Acc.Add}(C_{\mathsf{Acc}}, k)$
  - runs $(\mathsf{pp}_{\mathsf{VC}}, C_{\mathsf{VC}}, u_{\mathsf{VC},1}) \leftarrow_\$ \mathsf{VC.Extend}(1^\lambda, C_{\mathsf{VC}})$
  - runs $(C_{\mathsf{VC}}, u_{\mathsf{VC},2}) \leftarrow \mathsf{VC.Update}(C_{\mathsf{VC}}, (q+1, (k, v)))$
  - runs $\Lambda_{q+1} \leftarrow \mathsf{ProofUpdate}(\Lambda_{q+1}, u_{\mathsf{VC},2})$

  and finally outputs a new key-value commitment string $C = (C_{\mathsf{Acc}}, C_{\mathsf{VC}}, q+1)$, a proof $\Lambda_k = \Lambda_{q+1}$, and update information $u = (u_{\mathsf{Acc}}, u_{\mathsf{VC},1}, u_{\mathsf{VC},2})$.
- $(C, u) \leftarrow \mathsf{Update}(C, (k, \delta))$: On input a key-value commitment string C, a key $k \in \mathcal{K}$ along with a position q_k[8], and an *additive* update value $\delta \in \mathcal{V}$, the update algorithm:
  - parses $C = (C_{\mathsf{Acc}}, C_{\mathsf{VC}}, q)$
  - runs $(C_{\mathsf{VC}}, u_{\mathsf{VC}}) \leftarrow \mathsf{VC.Update}(C_{\mathsf{VC}}, (q_k, \delta))$[9]

  and finally outputs a new key-value commitment string $C = (C_{\mathsf{Acc}}, C_{\mathsf{VC}}, q)$ and update information $u = u_{\mathsf{VC}}$.
- $\Lambda_k \leftarrow \mathsf{ProofUpdate}(\Lambda_k, u)$: On input a proof Λ_k for some value corresponding to the key k and update information u, the proof update algorithm:
  - parses u as either $(\cdot, u_{\mathsf{VC},1}, u_{\mathsf{VC},2})$ or u_{VC}
  - runs $\Lambda_k \leftarrow \mathsf{VC.ProofUpdate}(\mathsf{VC.ProofUpdate}(\Lambda_k, u_{\mathsf{VC},1}), u_{\mathsf{VC},2})$ or $\Lambda_k = \mathsf{VC.ProofUpdate}(\Lambda_k, u_{\mathsf{VC}})$

  and finally outputs an updated proof Λ_k.
- $\overline{\Lambda_k} \leftarrow \mathsf{NonMemProofCreate}(k, U)$: On input a key $k \in \mathcal{K}$ and the digest of all updated information U produced until the current point in time, the non-membership proof creation algorithm:
  - parses $U = \{u\}$, where either $u = (u_{\mathsf{Acc}}, \cdot, \cdot)$ or $u = \cdot$
  - defines $U_{\mathsf{Acc}} = \{u_{\mathsf{Acc}}\}$, the set of all update information released for Acc
  - runs $\overline{w_k} \leftarrow \mathsf{Acc.NonMemProofCreate}(k, U_{\mathsf{Acc}})$

  and finally outputs a non-membership proof $\overline{\Lambda_k} = \overline{w_k}$.
- $\overline{\Lambda_k} \leftarrow \mathsf{NonMemProofUpdate}(\overline{\Lambda_k}, u)$: On input a non-membership proof $\overline{\Lambda_k}$ and update information u, the non-membership proof update algorithm:

[8] This is an implementation detail and can be assumed to be available in practice.

[9] We are slightly cheating here as we have stored the key-value pair as the element in the vector and we only wish to add δ to the value component of this pair. This can be realized in practice by carefully handling the sizes of $\mathcal{K}$ and $\mathcal{V}$ to simulate addition to the value component by performing regular addition and avoiding overflows. The alternative is to store just the value in VC, but then Acc would have to store the keys with the positions where their values are stored in VC, which would mean that a non-membership proof for our KVC would now have to be a batched non-membership proof of Acc, which is also a viable solution, but may be less efficient depending on how large $|\mathcal{K}_{\mathcal{M}}|$ becomes.

    - parses u as either $(u_{\mathsf{Acc}}, \cdot, \cdot)$ or $\cdot$, in the latter case, the algorithm makes no changes to $\overline{\Lambda_k}$
    - runs $\overline{\Lambda_k} \leftarrow \mathsf{Acc.NonMemProofUpdate}(\overline{\Lambda_k}, u_{\mathsf{Acc}})$

  outputs an updated non-membership proof $\overline{\Lambda_k}$.
- $1/0 \leftarrow \mathsf{Ver}(C, (k, v), \Lambda_k)$: On input a key-value commitment string C, a key-value pair $(k, v) \in \mathcal{K} \times \mathcal{V}$ along with a position q_k, and a proof Λ_k, the verification algorithm:
    - parses $C = (C_{\mathsf{Acc}}, C_{\mathsf{VC}}, q)$
    - checks that $q_k \leq q$
    - runs $b \leftarrow \mathsf{VC.ProofVer}(C_{\mathsf{VC}}, (q_k, (k, v)), \Lambda_k)$

  and finally outputs b.
- $0/1 \leftarrow \mathsf{NonMemVer}(C, k, \overline{\Lambda_k})$: On input a key-value commitment string C, a key $x \in \mathcal{K}$, and a non-membership proof $\overline{\Lambda_k}$, the non-membership verification algorithm:
    - parses $C = (C_{\mathsf{Acc}}, C_{\mathsf{VC}}, \cdot)$
    - runs $b \leftarrow \mathsf{Acc.NonMemVer}(C_{\mathsf{Acc}}, k, \overline{\Lambda_k})$

  and finally outputs b.

The correctness of the above scheme follows directly from the construction and the correctness of Acc and VC. Additionally, we have the following lemma with regard to key binding.

Lemma 1. *If* Acc *and* VC *have weak (strong) soundness and position binding, then* KVC *has weak (strong) key binding.*

Proof. This is a fairly simple reduction. Indeed, suppose we have a PPT adversary $\mathcal{A}$ that can break the weak (strong) key binding of KVC. By definition, this means that $\mathcal{A}$, with knowledge of pp, can come up with an *honestly generated* (*potentially adversarially generated*) key-value commitment and either a proof that certifies membership of a key or key-value pair that has not been inserted, or a non-membership proof that certifies non-membership of a key that has been inserted. Suppose it is the former. Recall that in our scheme, a membership proof Λ_k is simply a proof of VC. Therefore, if a membership proof breaks key binding of KVC, it can be used to break the weak (strong) position binding of VC. In the latter case, note that a non-membership proof $\overline{\Lambda_k}$ is simply a non-membership proof of Acc. Therefore, if a non-membership proof breaks key binding of KVC, it can be used to break the weak (strong) soundness of Acc. □

We thus have the following theorem.

Theorem 1. *Assuming the existence of an accumulator and vector commitment (supporting the procedure* Extend*) that satisfy correctness, and weak (strong) soundness and position binding respectively, then there exists a key-value commitment that satisfies correctness and weak (strong) key binding.*

3.2 Relation to existing constructions

We now describe how existing KVC constructions relate to our generic construction. We focus on schemes that support key non-membership and updates. Aardvark [27] is also a generic construction that uses a VC scheme with sequential insertions. It differs in the way it implements key non-membership. In order to support non-membership, it stores elements $(k,\ v,\ succ(k))$, where $succ(k)$ is the smallest key in the list larger than k. To prove non-membership for k', one gives a proof of opening for an element with the same successor as k'. This approach complicates updates, because when a new key is inserted, multiple positions need to be updated. Beyond Aardvark [27], there exist a number of KVC constructions with key non-membership and updates which are based on specific RSA-related assumptions [2,43,42]. Our generic construction could give rise to concrete instantiations under different assumptions.

We now review how the RSA based KVC constructions would fit under our generic paradigm. In KVaC [2,43], the commitment consists of two hidden-order group elements $(C_1,\ C_2)$. C_2 is an RSA accumulator for the multiset of all keys included in updates u. Let w_k be the membership proof of k in C_2. Then C_1 is the product of terms $w_k^{vH(k)}$. Non-membership for a key k corresponds in non-membership in C_2. Given an update $u = (k, \delta)$, KVaC treats the update as a new insertion for element (k, δ). Instead of $H(k)$, C_1, C_2 now accumulate $H(k)^{u_k}$ where u_k is the number of updates on k and is part of the opening proof. The KVC instantiation of [42] builds on top of the RSA VC [15,26,13] which has length that is efficiently and publicly extendable. It follows a similar format, the commitment consists of two hidden order group elements $(S,\ c)$. S is essentially an RSA accumulator that commits to the primes e_k. Each prime e_k corresponds to key k. The commitment c is the product of terms $(S_k)^v$, where S_k can be seen as the membership proof of e_k in S. Non-membership for a key k corresponds in non-membership in S. Given an update $u = (k, \delta)$, in order to update opening proofs for keys $k' \neq k$, the membership proof S_k for k is needed. There is a space-time trade-off. The proofs can be published as parameters pp or can be publicly computed, the computation is linear in the number of inserted keys. This scheme also offers key deletion.

4 Oblivious Accumulators

In this section, we provide our definition of *oblivious accumulators*. The overarching goal of an oblivious accumulator is for its updates to be *completely oblivious*, i.e., hide the details of the underlying operation being performed on the accumulator. In particular, from the definition of the accumulator from Section 2.2.1, we would like for the public parameters pp, accumulator string C, and update any information u that is released by calls to Add or Del (and hence the digest of all update information at any point in time) to hide as much information about the underlying accumulated set S as possible[10].

[10] Indeed, note that if only one operation has been performed, we know that it must be an Add, but we don't necessarily know the element that has been added.

Looking forward, we will formulate three properties that an oblivious accumulator must satisfy. These three properties combined will provide the guarantee that any publicly available information will hide as much information about S as possible. The three properties are:

1. *Element hiding.* Informally, this will mean that any publicly available information does not reveal anything about the elements in S.
2. Add-Del *unlinkability.* Informally, this will mean that any publicly available information does not reveal if two operations correspond to an add and a delete of the same element.
3. Add-Del *indistinguishability.* Informally, this will mean that any publicly available information does not reveal if an operation corresponds to an add or a delete, more than can be deduced given no update information.[11]

We will define the primitive in the positive (supports membership proofs) dynamic (supports additions and deletions) setting. We also assume that we are in the trapdoorless setting, i.e., updates can be performed with publicly available information. Much of our definition from Section 2.2.1 can be used to define oblivious accumulators, but crucially some changes are required. We now discuss those changes, and then present the definition of an oblivious accumulator. Essentially, any operation that is performed, in order to hide particulars of the operation such as the nature of the operation itself (Add or Del) and the associated element, must make use of some *secret* or *auxiliary information*, that we denote by aux. This auxiliary information must at the very least be used in the generation and verification of membership proofs.[12] Furthermore, since Adds and Dels are indistinguishable, it may become necessary for an Add to now take as input all past updates, just as Del does. Based on these observations, we modify our definition from 2.2.1 and define oblivious accumulators below.

4.1 Definition

Recall that we will define the primitive in the positive (supports membership proofs) dynamic (supports additions and deletions) setting. We also assume that we are in the trapdoorless setting, i.e., updates can be performed with publicly available information. We define an *oblivious accumulator* $\mathsf{OblvAcc}$ as a non-interactive primitive that can be formally described via the following algorithms:

– $(\mathsf{pp}, C) \leftarrow_{\$} \mathsf{Setup}(1^\lambda)$: On input the security parameter λ, the setup algorithm outputs some public parameters pp (which implicitly define the accumulator domain $\mathcal{D}$) and the initial accumulator string C to the empty set. All other algorithms have access to the public parameters.

[11] For example, if we have a sequence of four operations, they cannot be one Add and three Dels.

[12] One could imagine that they are also required for updating membership proofs, but we will not need this and so opt for the stronger definition where aux is only needed to generate membership proofs.

- $(C, w_x, u, \mathsf{aux}) \leftarrow_\$ \mathsf{Add}(C, x, U)$: On input an accumulator string C, an element $x \in \mathcal{D}$, and the digest of all update information U produced until the current point in time, the addition algorithm outputs a new accumulator string C, a membership proof w_x (that $x \in S$), update information u, and auxiliary information aux.
- $(C, u) \leftarrow \mathsf{Del}(C, x, U, \mathsf{aux})$: On input an accumulator string C, an element $x \in \mathcal{D}$, the digest of all update information U produced until the current point in time, and auxiliary information aux, the deletion algorithm outputs a new accumulator string C and update information u.
- $w_x \leftarrow \mathsf{MemProofUpdate}(w_x, u)$: On input a membership proof w_x and update information u, the membership proof update algorithm outputs an updated membership proof w_x.
- $0/1 \leftarrow \mathsf{MemVer}(C, x, w_x, \mathsf{aux})$: On input an accumulator string C, an element $x \in \mathcal{D}$, a membership proof w_x, and auxiliary information aux, the membership verification algorithm either outputs 1 (denoting accept) or 0 (denoting reject).

The correctness and soundness properties of an oblivious accumulator are identical to those of a regular accumulator, as defined in Section 2.2.1. We will define the three properties underlying the obliviousness of the accumulator in the next section.

4.2 Obliviousness Properties

In this section, we will define the three properties underlying the obliviousness of the accumulator.

4.2.1 Element hiding

The property of element hiding is meant to provide the guarantee that an adversary that observes the publicly available information does not learn about the elements in the underlying accumulated set S. We define this property as a game between a challenger and an adversary. In the game, the adversary gets to see honestly generated public parameters and then pick two elements $x_0, x_1 \in \mathcal{D}$. The challenger then picks $b \leftarrow_\$ \{0, 1\}$ and performs an Add of x_b, followed by a Del of x_b. The adversary is then given the update information generated by each of the operations and has to guess b. The oblivious accumulator is said to be element hiding if the adversary cannot guess b with non-negligible advantage over $\frac{1}{2}$. We define this formally below.

Definition 1 (***Element hiding***)**.** *An oblivious accumulator* $\mathsf{OblvAcc}$ *is said to be element hiding if for any* PPT *adversary* $\mathcal{A}$*, the following holds:*

$$\Pr\left[b' = b \;\middle|\; \begin{array}{c} (\mathsf{pp}, C_0) \leftarrow_\$ \mathsf{Setup}(1^\lambda) \\ x_0, x_1 \leftarrow_\$ \mathcal{A}(\mathsf{pp}, C_0) \\ b \leftarrow_\$ \{0,1\} \\ (C_1, w_{x_b}, u_1, \mathsf{aux}) \leftarrow_\$ \mathsf{Add}(C_0, x_b, \emptyset) \\ (C_2, u_2) \leftarrow \mathsf{Del}(C_1, x_b, \{u_1\}, \mathsf{aux}) \\ b' \leftarrow_\$ \mathcal{A}(C_1, C_2, u_1, u_2) \end{array} \right] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

We note that while there are other games one could think of to define this property, they would not offer any advantages over our proposed game.

4.2.2 Add-Del unlinkability

We first state a weaker flavor of privacy: Add-Del unlinkability, introduced by Baldimtsi et al.[5] in the context of manager-based anonymous revocation component (ARC) systems. We re-define this property in the context of a trapdoorless accumulator. Add-Del unlinkability is meant to provide the guarantee that an adversary that observes the publicly available information does not learn if two operations correspond to an Add and a Del of the same element. We define this property as a game between a challenger and an adversary. In the game, the adversary gets to see honestly generated public parameters and then pick two elements $x_0, x_1 \in \mathcal{D}$. The challenger first performs an Add of x_0 and x_1, in order. Then, the challenger picks $b \leftarrow_\$ \{0,1\}$ and performs a Del of x_b, followed by a Del of x_{1-b}. The adversary is then given the update information generated by each of the operations and has to guess b. The oblivious accumulator is said to be Add-Del unlinkable if the adversary cannot guess b with non-negligible advantage over $\frac{1}{2}$. We define this formally below.

Definition 2 (Add-Del ***unlinkability***)**.** *An oblivious accumulator* $\mathsf{OblvAcc}$ *is said to be* Add-Del *unlinkable if for any* PPT *adversary* $\mathcal{A}$*, the following holds:*

$$\Pr\left[b' = b \;\middle|\; \begin{array}{c} (\mathsf{pp}, C_0) \leftarrow_\$ \mathsf{Setup}(1^\lambda) \\ x_0, x_1 \leftarrow_\$ \mathcal{A}(\mathsf{pp}, C_0) \\ (C_1, w_{x_0}, u_1, \mathsf{aux}_0) \leftarrow_\$ \mathsf{Add}(C_0, x_0, \emptyset) \\ (C_2, w_{x_1}, u_2, \mathsf{aux}_1) \leftarrow_\$ \mathsf{Add}(C_1, x_1, \{u_1\}) \\ b \leftarrow_\$ \{0,1\} \\ (C_3, u_3) \leftarrow \mathsf{Del}(C_2, x_b, \{u_1, u_2\}, \mathsf{aux}_b) \\ (C_4, u_4) \leftarrow \mathsf{Del}(C_3, x_{1-b}, \{u_1, u_2, u_3\}, \mathsf{aux}_{1-b}) \\ b' \leftarrow \mathcal{A}(C_1, C_2, C_3, C_4, u_1, u_2, u_3, u_4) \end{array} \right] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

We note that while there are other games one could think of to define this property, they would not offer any advantages over our proposed game.

4.2.3 Add-Del indistinguishability

We now define Add-Del indistinguishability, a stronger privacy property which implies Add-Del unlinkability as defined above. The property of Add-Del indistinguishability is meant to provide the guarantee that an adversary that observes the publicly available information does not learn if an operation is an Add or a Del, beyond what it can learn without even observing any update information. We define this property as a game between a challenger and an adversary. In the game, the adversary gets to see honestly generated public parameters and then pick an element $x_0 \in \mathcal{D}$. The challenger first performs an Add of x_0. Then, the challenger picks $b \leftarrow_\$ \{0,1\}$. If $b = 0$, the challenger picks a random element $x_1 \in \mathcal{D}$ and performs an Add of x_1. Otherwise, the challenger performs a Del of x_0. The adversary is then given the update information generated by each of the operations and has to guess b. The oblivious accumulator is said to be Add-Del indistinguishable if the adversary cannot guess b with non-negligible advantage over $\frac{1}{2}$. We define this formally below.

Definition 3 (Add-Del ***indistinguishability***)**.** *An oblivious accumulator* OblvAcc *is said to be* Add-Del *indistinguishable if for any* PPT *adversary $\mathcal{A}$, the following holds:*

$$\Pr\left[b' = b \;\middle|\; \begin{array}{c} (\mathsf{pp}, C_0) \leftarrow_\$ \mathsf{Setup}(1^\lambda) \\ x_0 \leftarrow_\$ \mathcal{A}(\mathsf{pp}, C_0) \\ (C_1, w_{x_0}, u_1, \mathsf{aux}_0) \leftarrow_\$ \mathsf{Add}(C_0, x_0, \emptyset) \\ b \leftarrow_\$ \{0,1\}, x_1 \leftarrow_\$ \mathcal{D} \\ \textbf{if } b = 0\text{: } (C_2, w_{x_1}, u_2, \mathsf{aux}_1) \leftarrow_\$ \mathsf{Add}(C_1, x_1, \{u_1\}) \\ \textbf{if } b = 1\text{: } (C_2, u_2) \leftarrow \mathsf{Del}(C_1, x_0, \{u_1\}, \mathsf{aux}_0) \\ b' \leftarrow \mathcal{A}(C_1, C_2, u_1, u_2) \end{array} \right] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

We note that while there are other games one could think of to define this property, they would not offer any advantages over our proposed game.

Note that Add-Del indistinguishability implies Add-Del unlinkability. Intuitively, if an adversary cannot even tell Adds from Dels, then they certainly cannot identify a pair of updates that correspond to an Add and a Del, let along identifying that they are with respect to the same element. Formally, in the Add-Del unlinkability game, the two Dels can we be swapped with Adds, assuming Add-Del indistinguishability, and then back to Dels, but in the reverse order, again assuming Add-Del indistinguishability, and this would prove Add-Del unlinkability. We call accumulators that satisfy Add-Del unlinkability but not Add-Del indistinguishability as *almost-oblivious accumulators*, and ones that satisfy Add-Del indistinguishability as *oblivious accumulators*.

5 OblvAcc based on KVC

In this section, we show how to generically construct an oblivious accumulator using a key-value commitment. The idea is to maintain an *indicator map* that reflects which elements are in the underlying accumulated set, but where the keys associated with each element are kept secret and hence not publicly known. This helps in the first step of achieving element hiding. To achieve $\mathsf{Add\text{-}Del}$ indistinguishability, we perform both Adds and Dels as Inserts of the key-value commitment, but with different keys. Finally, the fact that the correspondence between elements and keys is not publicly known will also lend itself to $\mathsf{Add\text{-}Del}$ unlinkability. We formally describe this construction in the next section.

5.1 Construction

Let KVC be a key-value commitment as described in Section 2.2.3. Let $H_1, H_2 : \{0,1\}^\lambda \times \mathcal{D} \to \mathcal{K}$ be two hash functions (that will be modeled as random oracles). Note that we will be designing an oblivious accumulator for elements from $\mathcal{D}$ and $\mathcal{K}$ denotes the space of keys for the key-value commitment, and $|\mathcal{K}| = 2^\lambda$. The only requirement from the space of values $\mathcal{V}$ for the key-value commitment is that $1 \in \mathcal{V}$. We construct our oblivious accumulator $\mathsf{OblvAcc}$ as follows:

- $(\mathsf{pp}, C) \leftarrow_\$ \mathsf{Setup}(1^\lambda)$: On input the security parameter λ, the setup algorithm runs $(\mathsf{pp}_{\mathsf{KVC}}, C_{\mathsf{KVC}}) \leftarrow_\$ \mathsf{KVC.Setup}(1^\lambda)$ and outputs the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{KVC}}, H_1, H_2)$ and the initial accumulator string $C = C_{\mathsf{KVC}}$. All other algorithms have access to the public parameters.
- $(C, w_x, u, \mathsf{aux}) \leftarrow_\$ \mathsf{Add}(C, x, U)$: On input an accumulator string C, an element $x \in \mathcal{D}$, and the digest of all update information U produced until the current point in time, the addition algorithm:
  - samples $r \leftarrow_\$ \{0,1\}^\lambda$
  - computes $k_1 = H_1(r, x)$, $k_2 = H_2(r, x)$
  - runs $(C_{\mathsf{KVC}}, \Lambda_{k_1}, u_{\mathsf{KVC}}) \leftarrow \mathsf{KVC.Insert}(C, (k_1, 1))$
  - runs $\overline{\Lambda_{k_2}} \leftarrow \mathsf{KVC.NonMemProofCreate}(k_2, U \cup \{u_{\mathsf{KVC}}\})$

  and finally outputs a new accumulator string $C = C_{\mathsf{KVC}}$, a membership proof $w_x = (\Lambda_{k_1}, \overline{\Lambda_{k_2}})$, update information $u = u_{\mathsf{KVC}}$, and auxiliary information $\mathsf{aux} = r$.
- $(C, u) \leftarrow \mathsf{Del}(C, x, U, \mathsf{aux})$: On input an accumulator string C, an element $x \in \mathcal{D}$, the digest of all update information U produced until the current point in time, and auxiliary information aux, the deletion algorithm:
  - parses $\mathsf{aux} = r$
  - computes $k_2 = H_2(r, x)$
  - runs $(C_{\mathsf{KVC}}, \Lambda_{k_2}, u_{\mathsf{KVC}}) \leftarrow \mathsf{KVC.Insert}(C, (k_2, 1))$

  and finally outputs a new accumulator string $C = C_{\mathsf{KVC}}$ and update information $u = u_{\mathsf{KVC}}$.
- $w_x \leftarrow \mathsf{MemProofUpdate}(w_x, u)$: On input a membership proof w_x and update information u, the membership proof update algorithm:
  - parses $w_x = (\Lambda_{k_1}, \overline{\Lambda_{k_2}})$
  - runs $\Lambda_k \leftarrow \mathsf{KVC.ProofUpdate}(\Lambda_{k_1}, u)$
  - $\overline{\Lambda_{k_2}} \leftarrow \mathsf{KVC.NonMemProofUpdate}(\overline{\Lambda_{k_2}}, u)$

  and finally outputs an updated membership proof $w_x = (\Lambda_{k_1}, \overline{\Lambda_{k_2}})$.
- $0/1 \leftarrow \mathsf{MemVer}(C, x, w_x, \mathsf{aux})$: On input an accumulator string C, an element $x \in \mathcal{D}$, and a membership proof w_x, the membership verification algorithm:

- parses $\mathsf{aux} = r$
- computes $k_1 = H_1(r, x)$, $k_2 = H_2(r, x)$
- parses $w_x = (\Lambda_{k_1}, \overline{\Lambda_{k_2}})$
- runs $b_1 \leftarrow \mathsf{KVC.Ver}(C, (k_1, 1), \Lambda_{k_1})$
- runs $b_2 \leftarrow \mathsf{KVC.NonMemVer}(C, k_2, \overline{\Lambda_{k_2}})$

and finally outputs $b_1 \wedge b_2$.

The correctness of the above scheme follows directly from the construction and the correctness of KVC. In the remainder of this section, we will prove the soundness and obliviousness properties of our oblivious accumulator. We thus have the following theorem.

Theorem 2. *Assuming the existence of a key-value commitment that satisfies correctness and weak (strong) key binding, then there exists an oblivious accumulator that satisfies correctness, weak (strong) soundness, element hiding, and* Add-Del *indistinguishability, in the random oracle model.*

5.2 Soundness

Lemma 2. *Assume that H_1, H_2 are random oracles. If* KVC *has weak (strong) key binding, then* OblvAcc *has weak (strong) soundness.*

Proof. Suppose we have a PPT adversary $\mathcal{A}$ that can break the weak (strong) soundness of OblvAcc. By definition, this means that $\mathcal{A}$, with knowledge of pp, can come up with an *honestly generated* (*potentially adversarially generated*) accumulator and either a membership proof that certifies membership of an element that has not been added, or a non-membership proof that certifies non-membership of an element that has been added. Suppose it is the former. Recall that in our scheme, a membership proof w_x consists of a proof Λ_{k_1} and non-membership proof $\overline{\Lambda_{k_2}}$ of KVC, where $k_1 = H_1(r, x)$, $k_2 = H_2(r, x)$. If it is the case that x has not been added, then there cannot exist an r such that both Λ_{k_1} and $\overline{\Lambda_{k_2}}$ verify (as if they do, by definition, x has been added, and not yet deleted). Therefore, if w_x certifies x that has not been added, at least one of Λ_{k_1} and $\overline{\Lambda_{k_2}}$ can be used to break the weak (strong) key binding of KVC. A similar argument can be made for the latter case. A final detail is we assume that H_1, H_2 exhibit no collisions over the inputs queried on by $\mathcal{A}$. Indeed, since $\mathcal{A}$ is PPT and $|\mathcal{K}| = 2^\lambda$, this is true with probability all but $\frac{\mathrm{poly}(\lambda)}{2^\lambda} = \mathrm{negl}(\lambda)$. $\square$

5.3 Element hiding

Lemma 3. *Assume that H_1, H_2 are random oracles.* OblvAcc *is element hiding.*

Proof. We will show that for any PPT adversary $\mathcal{A}$,

$$\Pr\left[b' = b \;\middle|\; \begin{array}{c} (\mathsf{pp}, C_0) \leftarrow_\$ \mathsf{Setup}(1^\lambda) \\ x_0, x_1 \leftarrow_\$ \mathcal{A}(\mathsf{pp}, C_0) \\ b \leftarrow_\$ \{0,1\} \\ (C_1, w_{x_b}, u_1, \mathsf{aux}) \leftarrow_\$ \mathsf{Add}(C_0, x_b, \emptyset) \\ (C_2, u_2) \leftarrow \mathsf{Del}(C_1, x_b, \{u_1\}, \mathsf{aux}) \\ b' \leftarrow_\$ \mathcal{A}(C_1, C_2, u_1, u_2) \end{array} \right] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

We assume that H_1, H_2 exhibit no collisions over the inputs queried on by $\mathcal{A}$. Indeed, since $\mathcal{A}$ is PPT and $|\mathcal{K}| = 2^\lambda$, this is true with probability all but $\frac{\text{poly}(\lambda)}{2^\lambda} = \text{negl}(\lambda)$.

Note that $(C_1, \cdot, u_1) \leftarrow \mathsf{KVC.Insert}(C_0, (k_{1,b}, 1))$, where $k_{1,b} = H_1(r_b, x_b)$, and $(C_2, \cdot, u_2) \leftarrow \mathsf{KVC.Insert}(C_1, (k_{2,b}, 1))$, where $k_{2,b} = H_2(r_b, x_b)$. Since r_b is sampled at random from $\{0,1\}^\lambda$ and H_1 and H_2 are random oracles, we have $(k_{1,b}, k_{2,b}) \equiv (k_{1,1-b}, k_{2,1-b}) \equiv (\alpha_1, \alpha_2)$, where $\alpha_1, \alpha_2 \leftarrow_\$ \mathcal{K}$. Since these are the only values needed by the challenger to play the above game, this means that (C_1, C_2, u_1, u_2) is distributed the same, regardless of b. Therefore, the claim of $\mathsf{OblvAcc}$ being element hiding follows. □

5.4 Add-Del indistinguishability

Lemma 4. *Assume that H_1, H_2 are random oracles.* $\mathsf{OblvAcc}$ *is* $\mathsf{Add\text{-}Del}$ *indistinguishable.*

Proof. We will show that for any PPT adversary $\mathcal{A}$,

$$\Pr\left[b' = b \;\middle|\; \begin{array}{c} (\mathsf{pp}, C_0) \leftarrow_\$ \mathsf{Setup}(1^\lambda) \\ x_0 \leftarrow_\$ \mathcal{A}(\mathsf{pp}, C_0) \\ (C_1, w_{x_0}, u_1, \mathsf{aux}_0) \leftarrow_\$ \mathsf{Add}(C_0, x_0, \emptyset) \\ b \leftarrow_\$ \{0,1\}, x_1 \leftarrow_\$ \mathcal{D} \\ \textbf{if } b = 0\text{: } (C_2, w_{x_1}, u_2, \mathsf{aux}_1) \leftarrow_\$ \mathsf{Add}(C_1, x_1, \{u_1\}) \\ \textbf{if } b = 1\text{: } (C_2, u_2) \leftarrow \mathsf{Del}(C_1, x_0, \{u_1\}, \mathsf{aux}_0) \\ b' \leftarrow \mathcal{A}(C_1, C_2, u_1, u_2) \end{array} \right] \leq \frac{1}{2} + \text{negl}(\lambda)$$

We assume that H_1, H_2 exhibit no collisions over the inputs queried on by $\mathcal{A}$. Indeed, since $\mathcal{A}$ is PPT and $|\mathcal{K}| = 2^\lambda$, this is true with probability all but $\frac{\text{poly}(\lambda)}{2^\lambda} = \text{negl}(\lambda)$.

Note that $(C_1, \cdot, u_1) \leftarrow \mathsf{KVC.Insert}(C_0, (k_{1,0}, 1))$, where $k_{1,0} = H_1(r_0, x_0)$, and $(C_2, \cdot, u_2) \leftarrow \mathsf{KVC.Insert}(C_1, (k_{1,1}, 1))$, where $k_{1,1} = H_1(r_1, x_1)$ if $b = 0$, and $(C_2, \cdot, u_2) \leftarrow \mathsf{KVC.Insert}(C_1, (k_{2,0}, 1))$, where $k_{2,1} = H_2(r_0, x_0)$ if $b = 1$. Since r_0, r_1 are sampled at random from $\{0,1\}^\lambda$ and H_1 and H_2 are random oracles, we have $(k_{1,0}, k_{1,1}) \equiv (k_{1,0}, k_{2,0}) \equiv (\alpha_1, \alpha_2)$, where $\alpha_1, \alpha_2 \leftarrow_\$ \mathcal{K}$. Since these are the only values needed by the challenger to play the above game, this means that (C_1, C_2, u_1, u_2) is distributed the same, regardless of b. Therefore, the claim of $\mathsf{OblvAcc}$ being $\mathsf{Add\text{-}Del}$ indistinguishable follows. □

5.5 Extension for unique accumulation of elements

Both our main construction of Section 5.1 and the modular construction of the *almost-oblivious* accumulator described in the introduction, do not guarantee that the accumulated elements are unique. This implies that the same element x can be accumulated more than once and this would go unoticed because of the element hiding property [13].

[13] In the almost-oblivious accumulator which reveals the size of the accumulated set, this might be more problematic if in the underlying application the size of the set is important and should only contain unique elements

To overcome this problem instead of accumulating commitments to x one could use a deterministic commitment (assuming also that the accumulated elements bare random and from a large enough domain to avoid guessing attacks). One approach to do so, would be to use a hash function as a commitment scheme. If guessing is still a concern, we could use a verifiable oblivious PRF (VOPRF)[23] for the generation of the committed value in the almost-oblivious construction (or for the selection of randomness r in the construction of Section 5.1). In a high level, in a VOPRF protocol, when given a PRF F, a third party can communicate with a server holding a secret key k to evaluate an argument x and get back $y = F_k(x)$, while the server learns nothing about x. If F is also verifiable, there is a way to convince a third party that y is the true output of $F_k(x)$ without revealing k. Using this approach, has the trade off of requiring a server that holds the PRF key (thus, some point of centralization), however it makes such guessing attacks harder since an attacker would have to interact with a server in order to test for elements (which might imply some actual financial cost, i.e. the server could charge a fee for its given evaluation).

6 Lower Bounds

In this section, we will first show that for an oblivious accumulator, the digest of all update information cannot be compressed with time and must grow linearly with the number of operations that have been performed. This builds off of an information-theoretic argument in the style of [9,18] to argue the claim for deletions, and then uses the obliviousness properties to argue that the claim must hold for any sequence of operations. This result appears in Section 6.1.

Next, we show that a similar claim holds even for non-oblivious accumulators like ZCash that don't satisfy Add-Del indistinguishability but have have Add-Del unlinkability. For such accumulators, we show that the digest of all update information must grow in a sense with the number of deletions that have been performed. This result appears in Section 6.2.

6.1 Oblivious Accumulators

Lemma 5. *Let* $\mathsf{OblvAcc}$ *be an oblivious accumulator over the domain* $\mathcal{D}$*. Let* $S \subseteq \mathcal{D}$ *be a set of size* n *and let* $T \subset S$ *be a set of size* $\frac{n}{2}$*. Consider performing the following sequence of operations in order:*

1. $(\mathsf{pp}, C) \leftarrow_\$ \mathsf{Setup}(1^\lambda)$
2. $(C, w_x, u, \mathsf{aux}) \leftarrow_\$ \mathsf{Add}(C, x, U)$ *for each* $x \in S$
3. $(C, u) \leftarrow \mathsf{Del}(C, x, U, \mathsf{aux})$ *for each* $x \in T$

Let C *be the accumulator string and* U *be the digest of all update information produced at the end of all the operations. Then,*

$$|C| + |U| = \Omega(n)$$

Proof. We will show that if the theorem is false, then we can encode arbitrary subsets of S of size $\frac{n}{2}$ with $o(n)$ bits, which is impossible information-theoretically from Shannon's coding theorem, as there are $\binom{n}{\frac{n}{2}} = 2^{\Omega(n)}$ possible subsets of S of size $\frac{n}{2}$.

Let $T \subset S$ be a set of size $\frac{n}{2}$. Consider two parties A and B who know the set S, and suppose A knows T and wishes to encode T for B. A and B agree upon a mutual source of randomness and thus, we can assume that both parties toss the same random coins. A proceeds as follows. A runs $(\mathsf{pp}, C) \leftarrow_\$ \mathsf{OblvAcc.Setup}(1^\lambda)$. Then, A runs $(C, w_x, u, \mathsf{aux}) \leftarrow_\$ \mathsf{OblvAcc.Add}(C, x, U)$ for each $x \in S$, followed by $(C, u) \leftarrow \mathsf{OblvAcc.Del}(C, x, U, \mathsf{aux})$ for each $x \in T$. Let C be the final accumulator string and U be the final digest of all update information. A then sends along (C, U) to B.

We now claim that B can recover T. B can run $(\mathsf{pp}, C) \leftarrow_\$ \mathsf{OblvAcc.Setup}(1^\lambda)$ and $(C, w_x, u, \mathsf{aux}) \leftarrow_\$ \mathsf{OblvAcc.Add}(C, x, U)$ for each $x \in S$ (using the same random coins as A). Now, using $\mathsf{OblvAcc.MemProofUpdate}$, B can compute membership proofs w_x for each $x \in S$ after the sequence of $\mathsf{OblvAcc.Adds}$. Notice that this point, all those proofs would verify. Then, using $\mathsf{OblvAcc.MemProofUpdate}$ and U, B can compute updated membership proofs for each $x \in S$ after the sequence of $\mathsf{OblvAcc.Dels}$. From the correctness and soundness of $\mathsf{OblvAcc}$, only the membership proof of $x \in S \backslash T$ will now verify. Thus, by attempting to invoke $\mathsf{OblvAcc.MemVer}$ on each w_x, B can learn if $x \in T$ or not. Thus (C, U) encodes T and hence the claim in the lemma follows. □

For an oblivious accumulator, call a sequence of operations $\{O_i\}_i$ *valid*, where each O_i is an Add or Del, if and only if no operation attempts to Del an element that does not exist, i.e., has not been added or has already been deleted.

Lemma 6. *Let* $\mathsf{OblvAcc}$ *be an oblivious accumulator and let* $\ell \in \mathbb{N}$. *Let* $\{O_i\}_{i \in [\ell]}$ *and* $\{O'_i\}_{i \in [\ell]}$ *be two valid sequences of operations for* $\mathsf{OblvAcc}$. *Consider performing the following operations:*

1. $(\mathsf{pp}, C_0) \leftarrow_\$ \mathsf{Setup}(1^\lambda)$
2. $(C_i, (\cdot), u_i, (\cdot)) \leftarrow_\$ O_i(C_{i-1}, \cdot, \cdot, (\cdot))$ *for* $i \in [\ell]$
3. $(C'_i, (\cdot), u'_i, (\cdot)) \leftarrow_\$ O'_i(C'_{i-1}, \cdot, \cdot, (\cdot))$ *for* $i \in [\ell]$, *where* $C'_0 = C_0$

Then, for any PPT adversary,

$$(C_1, \ldots, C_\ell, u_1, \ldots, u_\ell) \approx_c (C'_1, \ldots, C'_\ell, u'_1, \ldots, u'_\ell)$$

that is, the sequence of accumulator strings and update information released are computationally indistinguishable.

Proof. We can prove this by induction on ℓ. For $\ell = 1$, both O_1 and O'_1 must be Adds. In this case, by the element hiding of $\mathsf{OblvAcc}$, the claim of the lemma holds. Assume the claim holds for $\ell = k$, and let us consider the case of $\ell = k+1$.

For any sequence of operations $\mathsf{O} = \{\mathsf{O}_i\}_{i \in [k+1]}$, let $\mathsf{transcript}(\mathsf{O})$ denote the sequence of accumulator strings and update information released. In particular,

$$\mathsf{transcript}(\{O_i\}_{i \in [k+1]}) = (C_1, \ldots, C_{k+1}, u_1, \ldots, u_{k+1})$$

and

$$\mathsf{transcript}(\{O'_i\}_{i\in[k+1]}) = (C'_1,\ldots,C'_{k+1},u'_1,\ldots,u'_{k+1})$$

Let O be an Add. First, note that

$$\mathsf{transcript}(\{O_i\}_{i\in[k+1]}) \approx_c \mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O\})$$

This follows from just element hiding if O_{k+1} were an Add, and from Add-Del unlinkability and indistinguishability if O_{k+1} were a Del. Next, note that there is a function $f_{O,\mathsf{pp}}$ such that

$$\mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O\}) \leftarrow_\$ f_{O,\mathsf{pp}}(\mathsf{transcript}(\{O_i\}_{i\in[k]}))$$

By our inductive hypothesis,

$$\mathsf{transcript}(\{O_i\}_{i\in[k]}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k]})$$

Therefore,

$$\mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O\}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k]} \cup \{O\})$$

as

$$\mathsf{transcript}(\{O'_i\}_{i\in[k]} \cup \{O\}) \leftarrow_\$ f_{O,\mathsf{pp}}(\mathsf{transcript}(\{O'_i\}_{i\in[k]}))$$

Finally, note that

$$\mathsf{transcript}(\{O'_i\}_{i\in[k+1]}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k]} \cup \{O\})$$

which follows as before from just element hiding if O'_{k+1} were an Add, and from Add-Del unlinkability and indistinguishability if O'_{k+1} were a Del. This completes the proof of the lemma. □

Theorem 3. *Let* $\mathsf{OblvAcc}$ *be an oblivious accumulator. Let* $\{O_i\}_{i\in[n]}$ *be a valid sequence of operations for* $\mathsf{OblvAcc}$*. Consider performing the following sequence of operations in order:*

1. $(\mathsf{pp}, C_0) \leftarrow_\$ \mathsf{Setup}(1^\lambda)$
2. $(C_i, (\cdot), u_i, (\cdot)) \leftarrow_\$ O_i(C_{i-1}, \cdot, \cdot, (\cdot))$ *for* $i \in [n]$

Let C *be the accumulator string and* U *be the digest of all update information produced at the end of all the operations. Then,*

$$|C| + |U| = \Omega(n)$$

Proof. We combine Lemmas 5 and 6. We know from Lemma 5 that there is a sequence of valid operations for which the claim in this lemma is true. We claim that from Lemma 6, this claim is true for all sequences of valid operations. This follows because both (C, U) is some function of the $\mathsf{transcript}$ of a sequence of operations (as defined in Lemma 6), and since the $\mathsf{transcript}$s are indistinguishable from Lemma 6, $|C| + |U|$ must be as well. □

6.2 Oblivious Accumulators without Add-Del indistinguishability

For an oblivious accumulator, define the $\mathsf{optrace}$ of a sequence of operations $\{O_i\}_i$ to be the sequence of operation types of each operation O_i as either an Add or a Del.

Lemma 7. *Let* $\mathsf{OblvAcc}$ *be an oblivious accumulator without* Add-Del *indistinguishability and let* $\ell \in \mathbb{N}$. *Let* $\{O_i\}_{i\in[\ell]}$ *and* $\{O'_i\}_{i\in[\ell]}$ *be two valid sequences of operations for* $\mathsf{OblvAcc}$ *with the same* $\mathsf{optrace}$. *Consider performing the following operations:*

1. $(\mathsf{pp}, C_0) \leftarrow_\$ \mathsf{Setup}(1^\lambda)$
2. $(C_i, (\cdot), u_i, (\cdot)) \leftarrow_\$ O_i(C_{i-1}, \cdot, \cdot, (\cdot))$ *for* $i \in [\ell]$
3. $(C'_i, (\cdot), u'_i, (\cdot)) \leftarrow_\$ O'_i(C'_{i-1}, \cdot, \cdot, (\cdot))$ *for* $i \in [\ell]$, *where* $C'_0 = C_0$

Then, for any PPT adversary,

$$(C_1, \ldots, C_\ell, u_1, \ldots, u_\ell) \approx_c (C'_1, \ldots, C'_\ell, u'_1, \ldots, u'_\ell)$$

that is, the sequence of accumulator strings and update information released are computationally indistinguishable.

Proof. We can prove this by induction on ℓ. For $\ell = 1$, both O_1 and O'_1 must be Adds. In this case, by the element hiding of $\mathsf{OblvAcc}$, the claim of the lemma holds. Assume the claim holds for $\ell = k$, and let us consider the case of $\ell = k+1$.

Let O_{k+1} and O'_{k+1} be Adds. Then,

$$\mathsf{transcript}(\{O_i\}_{i\in[k+1]}) \approx_c \mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O'_{k+1}\})$$

This follows from just element hiding. Next, note that there is a function $f_{O'_{k+1},\mathsf{pp}}$ such that

$$\mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O'_{k+1}\}) \leftarrow_\$ f_{O'_{k+1},\mathsf{pp}}(\mathsf{transcript}(\{O_i\}_{i\in[k]}))$$

By our inductive hypothesis,

$$\mathsf{transcript}(\{O_i\}_{i\in[k]}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k]})$$

Therefore,

$$\mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O'_{k+1}\}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k+1]})$$

as

$$\mathsf{transcript}(\{O'_i\}_{i\in[k+1]}) \leftarrow_\$ f_{O'_{k+1},\mathsf{pp}}(\mathsf{transcript}(\{O'_i\}_{i\in[k]}))$$

This completes the proof of the lemma for the case that O_{k+1} and O'_{k+1} are Adds.

Now suppose that O_{k+1} and O'_{k+1} be Dels. The issue is that $\{O_i\}_{i\in[k]} \cup \{O'_{k+1}\}$ may not be a valid sequence of operations. However, we can leverage both element hiding and Add-Del unlinkability in this case.

Let $D \subseteq [k]$ and $D' \subseteq [k]$ be the indices of Adds in $\{O_i\}_{i\in[k+1]}$ and $\{O'_i\}_{i\in[k+1]}$ respectively that have corresponding Dels. We have two cases depending on whether $D \cap D' \neq \emptyset$ or $D \cap D' = \emptyset$.

If $D \cap D' \neq \emptyset$, let $j \in D \cap D'$. Let $O^{(j)}$ and $O'^{(j)}$ be the Dels that correspond to the Adds at index j in $\{O_i\}_{i\in[k+1]}$ and $\{O'_i\}_{i\in[k+1]}$ respectively. Now,

$$\mathsf{transcript}(\{O_i\}_{i\in[k+1]}) \approx_c \mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O^{(j)}\})$$

This follows from Add-Del unlinkability. By our inductive hypothesis,

$$\mathsf{transcript}(\{O_i\}_{i\in[k]}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k]})$$

Thus, from element hiding, we have that

$$\mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O^{(j)}\}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k]} \cup \{O'^{(j)}\})$$

Finally, from Add-Del unlinkability,

$$\mathsf{transcript}(\{O'_i\}_{i\in[k+1]}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k]} \cup \{O'^{(j)}\})$$

completing the proof of the lemma for the case that $D \cap D' \neq \emptyset$.

Now, suppose $D \cap D' = \emptyset$. Let $j' \in [k]$ denote the index of the Add in $\{O'_i\}_{i\in[k+1]}$ that corresponds to the Del O'_{k+1}. Let $O^{(j')}$ be the Del that corresponds to the Add at index j' in $\{O_i\}_{i\in[k+1]}$. Now,

$$\mathsf{transcript}(\{O_i\}_{i\in[k+1]}) \approx_c \mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O^{(j')}\})$$

This follows from Add-Del unlinkability. By our inductive hypothesis,

$$\mathsf{transcript}(\{O_i\}_{i\in[k]}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k]})$$

Thus, from element hiding, we have that

$$\mathsf{transcript}(\{O_i\}_{i\in[k]} \cup \{O^{(j')}\}) \approx_c \mathsf{transcript}(\{O'_i\}_{i\in[k+1]})$$

completing the proof of the lemma for the case that $D \cap D' = \emptyset$. $\square$

For an oblivious accumulator, define the $\mathsf{delspace}$ of a sequence of operations $\{O_i\}_i$ as follows:

– For each i such that O_i is a Del, define

$$\mathsf{delspace}(O_i) = i - 1 - 2 \cdot |\{j < i : O_j \text{ is a } \mathsf{Del}\}|$$

– Define

$$\mathsf{delspace}(\{O_i\}_i) = \prod_{i:O_i \text{ is a } \mathsf{Del}} \mathsf{delspace}(O_i)$$

Based on the above definition and Lemma 7, we can prove the following theorem, just as we did Theorem 3.

Theorem 4. *Let* $\mathsf{OblvAcc}$ *be an oblivious accumulator without* Add-Del *indistinguishability. Let* $\{O_i\}_{i\in[n]}$ *be a valid sequence of operations for* $\mathsf{OblvAcc}$*. Consider performing the following sequence of operations in order:*

1. $(\mathsf{pp}, C_0) \leftarrow_{\$} \mathsf{Setup}(1^\lambda)$
2. $(C_i, (\cdot), u_i, (\cdot)) \leftarrow_{\$} O_i(C_{i-1}, \cdot, \cdot, (\cdot))$ *for* $i \in [n]$

Let C *be the accumulator string and* U *be the digest of all update information produced at the end of all the operations. Then,*

$$|C| + |U| = \Omega(\log \mathsf{delspace}(\{O_i\}_{i\in[n]}))$$

Acknowledgements

Foteini Baldimtsi and Ioanna Karantaidou are supported by NSF Awards #2143287 and #2247304 as well as a Google Faculty Award. Ioanna Karantaidou is additionally supported by a Protocol Labs PhD Student Fellowship.

We thank Ian Miers and Swan Klein for initial discussions on the topic of oblivious accumulators.

Disclaimer

References

1. Acar, T., Nguyen, L.: Revocation for delegatable anonymous credentials. In: International Workshop on Public Key Cryptography. pp. 423–440. Springer (2011)
2. Agrawal, S., Raghuraman, S.: Kvac: Key-value commitments for blockchains and beyond. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 839–869. Springer (2020)
3. Au, M.H., Tsang, P.P., Susilo, W., Mu, Y.: Dynamic universal accumulators for ddh groups and their application to attribute-based anonymous credential systems. In: Topics in Cryptology–CT-RSA 2009: The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings. pp. 295–308. Springer (2009)
4. Au, M.H., Tsang, P.P., Susilo, W., Mu, Y.: Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In: Fischlin, M. (ed.) Topics in Cryptology – CT-RSA 2009. pp. 295–308. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
5. Baldimtsi, F., Camenisch, J., Dubovitskaya, M., Lysyanskaya, A., Reyzin, L., Samelin, K., Yakoubov, S.: Accumulators with applications to anonymity-preserving revocation. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 301–315. IEEE (2017)
6. Benaloh, J., de Mare, M.: One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In: Helleseth, T. (ed.) Advances in Cryptology — EUROCRYPT '93. pp. 274–285. Springer Berlin Heidelberg, Berlin, Heidelberg (May 1993), `https://www.microsoft.com/en-us/research/publication/one-way-accumulators-a-decentralized-alternative-to-digital-signatures/`
7. Benarroch, D., Campanelli, M., Fiore, D., Gurkan, K., Kolonelos, D.: Zero-knowledge proofs for set membership: Efficient, succinct, modular. Cryptology ePrint Archive, Report 2019/1255 (2019), `https://eprint.iacr.org/2019/1255`
8. Boneh, D., Bünz, B., Fisch, B.: Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. pp. 561–586. Springer, Springer International Publishing, Cham (2019), `https://eprint.iacr.org/2018/1188`
9. Camacho, P., Hevia, A.: On the impossibility of batch update for cryptographic accumulators. In: Progress in Cryptology–LATINCRYPT 2010: First International Conference on Cryptology and Information Security in Latin America, Puebla, Mexico, August 8-11, 2010, proceedings 1. pp. 178–188. Springer (2010)
10. Camenisch, J., Kohlweiss, M., Soriente, C.: An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In: Jarecki, S., Tsudik, G. (eds.) Public Key Cryptography – PKC 2009. pp. 481–500. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), `https://eprint.iacr.org/2008/539`
11. Camenisch, J., Lysyanskaya, A.: Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In: Yung, M. (ed.) Advances in Cryptology — CRYPTO 2002. pp. 61–76. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
12. Camenisch, J., Stadler, M.: Efficient group signature schemes for large groups. In: CRYPTO (1997)
13. Campanelli, M., Fiore, D., Greco, N., Kolonelos, D., Nizzardo, L.: Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In: Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon,

South Korea, December 7–11, 2020, Proceedings, Part II 26. pp. 3–35. Springer (2020)

14. Campanelli, M., Fiore, D., Han, S., Kim, J., Kolonelos, D., Oh, H.: Succinct zero-knowledge batch proofs for set accumulators. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 455–469 (2022)
15. Catalano, D., Fiore, D.: Vector commitments and their applications. In: International Workshop on Public Key Cryptography. pp. 55–72. Springer (2013)
16. Chen, B., Dodis, Y., Ghosh, E., Goldin, E., Kesavan, B., Marcedone, A., Mou, M.E.: Rotatable zero knowledge sets - post compromise secure auditable dictionaries with application to key transparency. In: Agrawal, S., Lin, D. (eds.) Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13793, pp. 547–580. Springer (2022)
17. Chepurnoy, A., Papamanthou, C., Srinivasan, S., Zhang, Y.: Edrax: A Cryptocurrency with Stateless Transaction Validation. Cryptology ePrint Archive, Report 2018/968 (2018), https://ia.cr/2018/968
18. Christ, M., Bonneau, J.: Limits on revocable proof systems, with applications to stateless blockchains. Cryptology ePrint Archive (2022)
19. Damgård, I., Triandopoulos, N.: Supporting Non-membership Proofs with Bilinear-map Accumulators. Cryptology ePrint Archive, Report 2008/538 (2008), https://eprint.iacr.org/2008/538
20. Dodis, Y., Kiayias, A., Nicolosi, A., Shoup, V.: Anonymous identification in ad hoc groups. In: Eurocrypt (2004)
21. Ghosh, E., Ohrimenko, O., Papadopoulos, D., Tamassia, R., Triandopoulos, N.: Zero-knowledge accumulators and set algebra. In: Asiacrypt (2016)
22. Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: Aggregating proofs for multiple vector commitments. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 2007–2023 (2020)
23. Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology - ASIACRYPT 2014. Lecture Notes in Computer Science, vol. 8874, pp. 233–253. Springer (2014). https://doi.org/10.1007/978-3-662-45608-8_13, https://doi.org/10.1007/978-3-662-45608-8_13
24. Karantaidou, I., Baldimtsi, F.: Efficient constructions of pairing based accumulators. In: 2021 IEEE 34th Computer Security Foundations Symposium (CSF). pp. 1–16. IEEE (2021)
25. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16. pp. 177–194. Springer (2010)
26. Lai, R.W., Malavolta, G.: Subvector commitments with application to succinct arguments. In: Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I 39. pp. 530–560. Springer (2019)
27. Leung, D., Gilad, Y., Gorbunov, S., Reyzin, L., Zeldovich, N.: Aardvark: An asynchronous authenticated dictionary with applications to account-based cryptocurrencies. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 4237–4254 (2022)

28. Li, J., Li, N., Xue, R.: Universal Accumulators with Efficient Nonmembership Proofs. In: Katz, J., Yung, M. (eds.) Applied Cryptography and Network Security. pp. 253–269. Springer Berlin Heidelberg, Berlin, Heidelberg (2007), https://www.cs.purdue.edu/homes/ninghui/papers/accumulator_acns07.pdf
29. Libert, B., Ling, S., Nguyen, K., Wang, H.: Zero-knowledge arguments for lattice-based accumulators: logarithmic-size ring signatures and group signatures without trapdoors. In: Eurocrypt (2016)
30. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In: 2013 IEEE Symposium on Security and Privacy. pp. 397–411 (May 2013). https://doi.org/10.1109/SP.2013.34
31. Nguyen, L.: Accumulators from Bilinear Pairings and Applications. In: Menezes, A. (ed.) Topics in Cryptology – CT-RSA 2005. pp. 275–292. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), https://eprint.iacr.org/2005/123
32. Nguyen, L., Safavi-Naini, R.: Efficient and provably secure trapdoor-free group signature schemes from bilinear pairings. In: Asiacrypt (2004)
33. Papamanthou, C., Shi, E., Tamassia, R., Yi, K.: Streaming authenticated data structures. In: Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32. pp. 353–370. Springer (2013)
34. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables based on cryptographic accumulators. Algorithmica **74**, 664–712 (2016)
35. Peikert, C., Pepin, Z., Sharp, C.: Vector and functional commitments from lattices. In: Theory of Cryptography: 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8–11, 2021, Proceedings, Part III 19. pp. 480–511. Springer (2021)
36. Srinivasan, S., Chepurnoy, A., Papamanthou, C., Tomescu, A., Zhang, Y.: Hyperproofs: Aggregating and maintaining proofs in vector commitments. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3001–3018 (2022)
37. Srinivasan, S., Karantaidou, I., Baldimtsi, F., Papamanthou, C.: Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2719–2733 (2022)
38. Sun, S.F., Au, M.H., Liu, J.K., Yuen, T.H.: Ringct 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. In: ESORICS (2017)
39. Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable subvector commitments for stateless cryptocurrencies. In: Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings 12. pp. 45–64. Springer (2020)
40. Tomescu, A., Bhupatiraju, V., Papadopoulos, D., Papamanthou, C., Triandopoulos, N., Devadas, S.: Transparency logs via append-only authenticated dictionaries. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1299–1316 (2019)
41. Tomescu, A., Bhupatiraju, V., Papadopoulos, D., Papamanthou, C., Triandopoulos, N., Devadas, S.: Transparency logs via append-only authenticated dictionaries. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1299–1316 (2019)
42. Tomescu, A., Xia, Y., Newman, Z.: Authenticated dictionaries with cross-incremental proof (dis) aggregation. Cryptology ePrint Archive (2020)

43. Tyagi, N., Fisch, B., Zitek, A., Bonneau, J., Tessaro, S.: Versa: Verifiable registries with efficient client audits from rsa authenticated dictionaries. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2793–2807 (2022)
44. Yu, Z., Au, M.H., Yang, R., Lai, J., Xu, Q.: Lattice-based universal accumulator with nonmembership arguments. In: Information Security and Privacy: 23rd Australasian Conference, ACISP 2018, Wollongong, NSW, Australia, July 11-13, 2018, Proceedings 23. pp. 502–519. Springer (2018)
45. Zhang, Y., Katz, J., Papamanthou, C.: An Expressive (Zero-Knowledge) Set Accumulator. In: 2017 IEEE European Symposium on Security and Privacy (EuroS P). pp. 158–173 (April 2017). https://doi.org/10.1109/EuroSP.2017.35

A Related Work on Set Compressing Primitives

We discuss related works for the primitives of accumulators, vector commitments and key-value commitments.

A.1 Accumulators

In the universal, dynamic, trapdoorless setting, the RSA [11,28] satisfies strong soundness under the strong RSA assumption. It can be instantiated with class groups in order to avoid the trusted setup and it supports batching and aggregation of proofs [8]. The bilinear accumulator [31,19,3] satisfies strong soundness under the q-strong Diffie Hellman assumption and has public parameters pp linear in the maximum accumulator capacity q. It supports batching and aggregation of proofs [37]. Finally, the lattice-based accumulator by Yu et al. [44] satisfies weak soundness based on the Short Integer Solution problem. It has expensive proof updates, since it uses a Merkle Tree construction.

A.2 Vector Commitments

Catalano and Fiore [15] give two VC instantiations that satisfy strong key binding. The first VC instantiation is secure under the CDH assumption in bilinear groups and has public parameters pp of length quadratic in the vector length q. The parameters are of the form $g^{z_i z_j}$ and are generated using q random elements $z_1, \ldots, z_q$. Extend generates an extra secret exponent z_{q_1} and appends a set of q elements $g^{z_1 z_{q+1}}, \ldots, g^{z_q z_{q+1}}$ in pp. The second VC instantiation is secure under the RSA assumption and has public parameters pp of length linear in the vector length or pp of constant length with the use of collision resistant functions. pp includes the description of a collision resistant function H that maps to prime numbers and Extend includes one hash operation $H(q+1) = e_{q+1}$ that can be run by everyone without the trapdoor. Lai and Malavolta [26] extend the constructions to support subvector openings, i.e. one concise proof for opening in multiple positions, similar to a batch proof for accumulators. Campanelli et al. extend the RSA instantiation to support incremental proof aggregation and give a second construction with the same properties, based on the RSA accumulator, using the batching and aggregation techniques by Boneh et al. [8]. Based on KZG polynomial commitments [25], aSVCs [39] is a VC scheme with aggregatable, updateable subvector opening proofs and pp linear in the size of the vector q. aSVC satisfies strong binding under the q-strong bilinear Diffie-Hellman assumption. Extend appends an extra parameter $g^{\tau^{q+1}}$ to pp and is generated using the secret trapdoor τ. Hyperproofs [36] is tree-based VC scheme with aggregatable and efficiently updateable logarithmic size subvector opening proofs and linear parameter size that satisfies strong binding under q-strong Diffie Hellman. Extend works the same as in aSVCs and uses the trapdoor. Pointproofs [22] is a linear parameter VC scheme with subvector openings that supports aggregation and cross-aggregation, meaning it has constant proofs of opening across different vector commitments. It satisfies strong binding based on the weak bilinear

Diffie-Hellman exponent problem. There is no trivial way to extend the vector length for Pointproofs. Finally, there are lattice-based VC schemes [33,35] with logarithmic sized single opening proofs satisfying strong binding based on the Short Integer Solution problem. In order to extend the vector length in [35], a random matrix A_{q+1} needs to be generated and appended to pp together with another matrix R_{q+1} that was computed using a trapdoor.

A.3 Key-Value Commitments

Boneh et al. [8] show how to build a KVC scheme using an accumulator and committing to each bit of the value v separately. Their construction does not support non-membership, since the non-membership proofs offered by the accumulator are used in order to express opening of bit equals zero. The tree-based accumulator schemes of [34] and [41] support key-value openings but do not support efficient proof updates. Aardvark [27] describes a way to get a KVC from VC schemes. It does not account for vector commitment Extend functions, instead it considers multiple VC schemes of fixed length B. Once a commitment gets full, a new one is instantiated. The final commitment is the set of all VC digests. The VC positions are filled sequentially, similar to a linked-list, with values of the form $(k, v, succ(k))$, where $succ(k)$ is the smallest key in the list larger than k. This method is used in order to realize key non-membership but at the same time, it does not allow for efficient updates. KVaC [2,43] is an RSA-based KVC scheme that satisfies strong key binding under the generalized RSA assumption. Tomescu et al. [42] propose a KVC scheme with cross-incremental proof (dis)aggregation that has strong soundness for one-hop aggregation. Their scheme also supports append-only updates. Both the above schemes have constant commitment and proof size and constant public parameters.