KVaC: Key-Value Commitments for Blockchains and Beyond

Shashank Agrawal[1] and Srinivasan Raghuraman[2]

[1] Western Digital Research
shashank.agrawal@wdc.com
[2] Visa Research
srraghur@visa.com

Abstract. As blockchains grow in size, validating new transactions becomes more and more resource intensive. To deal with this, there is a need to discover compact encodings of the (effective) state of a blockchain — an encoding that allows for efficient proofs of membership and updates. In the case of account-based cryptocurrencies, the state can be represented by a key-value map, where keys are the account addresses and values consist of account balance, nonce, etc.

We propose a new commitment scheme for key-value maps whose size does not grow with the number of keys, yet proofs of membership are of constant-size. In fact, both the encoding and the proofs consist of just two and three group elements respectively (in groups of unknown order like class groups). Verifying and updating proofs involves just a few group exponentiations. Additive updates to key values enjoy the same level of efficiency too.

Key-value commitments can be used to build dynamic accumulators and vector commitments, which find applications in group signatures, anonymous credentials, verifiable databases, interactive oracle proofs, etc. Using our new key-value commitment, we provide the most efficient constructions of (sub)vector commitments to date.

1 Introduction

Cryptocurrency space has grown quite rapidly since the introduction of Bitcoin [3] about a decade ago. The state of several leading cryptocurrencies like Ethereum [15], Ripple [26], EOS [13] and Stellar [24] can be represented by a key-value map $\mathcal{M}$ where *keys* are the public addresses of users and *values* are the attributes associated with them (balance amount, nonce, etc.). When Alice generates a transaction tx to transfer an amount x from her public key pk to Bob, the map $\mathcal{M}$ is used to check if pk has a balance of at least x. On confirmation of tx, balance of Alice and Bob (along with other bookkeeping information) is updated in $\mathcal{M}$.

In a cryptocurrency network, every node is expected to verify and store the state of the system. As the number of users increase and new accounts are created, the size of the key-value map grows, and the resource cost of running a

node increases. This drives a large proportion of users to rely on third parties to inform them of the state of the system, severely limiting the amount of decentralization in the network [27,12,11]. To provide some context, Ethereum has over 70 million accounts now [14] even though its not very widely used (5-10 transactions per second [16]). Upcoming currencies like Libra [23] can have a much higher throughput and are expected to have billions of accounts.

Key-value commitments. In this paper, we initiate a formal study of key-value commitments (KVC). These commitments allow one to produce succinct encodings of key-value maps that are amenable to both efficient membership proofs and additive updates.[3] We propose a new KVC KVaC (pronounced 'quack') in groups of unknown order, such as an RSA group or a class group [5], with the following properties:

- *Succinct encoding.* Commitment value consists of just two group elements *irrespective* of the number of items (key-value pairs) committed. If class groups of quadratic imaginary order targeting 100-bit security are used, then the commitment size would be 4096 bits or about half a kilobyte [18].
- *Succinct proofs & fast verification.* A proof to show that a certain item is in the commitment consists of just three group elements and only five exponentiations are required to verify a proof.
- *Fast updates.* Inserting new items to the map, updating values of existing items (additive updates), and updating membership proofs when items are added/changed can all be done quite efficiently — at most four exponentiations are required in any case. Furthermore, to update the value of an item, its existing value is not needed.
- *Trustless set-up.* If class groups are used as the underlying group here, then the encoding scheme could be bootstrapped in a trustless manner.
- *Aggregation & batching.* Multiple proofs can be aggregated to the size of a single proof and verified in the same amount of time as a single proof.

We prove the security of our construction under the RSA assumption in the random oracle model or the strong RSA assumption in the standard model.

Vector commitments & accumulators. KVCs can be used to construct and improve upon well-studied cryptographic primitives like accumulators [2,8,25,7,1] and more recent ones like (sub)vector commitments (VC) [10,19,22,11,4,20]. Accumulators are used in group signatures, anonymous credentials, computation on authenticated data, decentralized bulletin boards, etc., and VCs have applications to verifiable databases with updates, interactive oracle proofs, etc.

VCs have been used to build accumulators [10] and vice versa [4]. We show how to build both of them from KVCs. In particular, we provide the first VC scheme where both public parameters and proofs consist of a constant number of group elements, the public parameters do not restrict the number of elements

[3] We do not intend key-value commitments to provide any sort of hiding for the keys or values committed, similar to how vector commitments do not intend to hide the vector elements [10].

that could be committed to, and all the VC algorithms require a constant number of group operations. Furthermore, the proofs can be batched to produce constant-size subvector openings. We also show how to build a dynamic accumulator from an (insert-only) KVC whose efficiency matches with the state-of-the-art. Thus, KVCs can be seen as a more general and flexible primitive.

We discuss our work in connection to VCs and accumulators in more depth in the following section.

1.1 Applications

Stateless validation of cryptocurrencies. Todd [27] first proposed the idea of a "stateless" blockchain where nodes participate in transaction validation without storing the entire state of the ledger, but rather only a short commitment to the state. Using our new key-value commitment scheme KVaC, one can build an account-based cryptocurrency where a node needs to store only two group elements (the commitment) to validate transactions. In the new currency, an account is represented by a key-value pair where key is a public key for a signature scheme (or a hash of it) and value consists of the account balance and a counter. (The counter helps to prevent replay attacks.) Suppose Alice has an account represented by $(pk, (v||ct))$ and she wants to transfer an amount b to a public key $pk^\star$. She will generate a transaction $tx = (pk, \sigma, v, ct, b, pk^\star, \pi)$ where σ is a signature on $(v, ct, b, pk^\star, \pi)$ under pk and π is a proof of the membership of $(pk, (v||ct))$ in the commitment.

A validator node, holding a commitment C, checks if σ is a valid signature, π is a valid proof w.r.t. C, and b is at most v. If all the checks pass, it updates the commitment using the efficient update algorithm Update of KVaC. The validator first executes Update with $(pk, -b||1)$ and C to produce a new commitment C' that reflects the change to Alice's account, and then executes Update with $(pk^\star, b||0)$ and C' to produce the final commitment C'' that reflects the changes to both the accounts. Here, KVaC's ability to update the value associated with $pk^\star$ without the knowledge of the original is very useful, otherwise senders would have to keep track of receivers' balances too.

Every time the commitment value changes, users need to update their proofs so that they remain valid. This puts some additional burden on users as they have to sync up with the blockchain frequently, which brings up a natural question: can the information required to update proofs be sublinear in the number of updates made? Unfortunately, this does not seem to be the case because the new and old values of commitment can be used to find out what updates have been made. Specifically, the lower bound of Camacho and Hevia can be extended to our setting as well [6].

Use of KVCs in blockchain also adds some extra overhead of generating and transmitting proofs, but they can be batched together and only one proof needs to be stored with every block.

Vector commitments. Vector commitments (VC) were introduced by Catalano and Fiore [10]. VCs allow one to commit to an ordered sequence of values in

such a way that one can later open the commitment at specific positions. They also allow updates to individual values and corresponding updates to proofs. The constructions of Catalano and Fiore, and those of Chepurnoy et al. [11] proposed later, have public parameters that scale *linearly* with the size of the vector. Though they suggest a way to cut down the parameters to a constant, it makes the cost of VC algorithms linear in the size of the vector.

Boneh et al. [4] take a different approach to build VCs. They provide a transformation from their accumulator construction which supports both membership and non-membership proofs. Though their construction does not put any restriction on the length of the vector, the vector elements are accumulated bit by bit. So the number of group operations required to insert new elements, generate membership proofs, etc. is linear in the bit-size of vector elements. Updates to values are not explicitly discussed but one could infer that it would have a similar cost. Lastly, their proofs contain seven group elements.

KVCs can be used to build a VC in a straightforward way, with keys playing the role of indices. Our KVC leads to a VC where the public parameters are of constant size, there is no restriction on the number of elements that could be committed, and proofs consist of just three group elements. Further, all the VC algorithms require a constant number of group operations. This improves upon all the known constructions.

Subvector commitments. Lai and Malavolta [20] recently proposed the notion of subvector commitments (SVC). An SVC allows one to open a committed vector at a set of positions, where the opening size is independent of length of the committed vector and the number of positions to be opened. They show that the proofs (or openings) of Catalano and Fiore [10] can be extended for multiple elements without losing the succinctness. However, this also means that they inherit the limitations of Catalano and Fiore's constructions.

Membership proofs of our KVC can be batched, resulting in an SVC with succinct subvector openings — but without linear growth in public parameters.

Accumulators. Accumulators are a very well-studied cryptographic primitive with numerous applications. They can be based on RSA groups [2,8], Merkle hash trees [7], or bilinear maps [25]. See Baldimtsi et al. [1] for a thorough discussion of different types of accumulators and modular conversions between them. We show how to build a *dynamic positive* accumulator from an (insert-only) KVC whose efficiency matches with the state-of-the-art. (All accumulator algorithms need a constant number of group operations.) A dynamic accumulator supports both additions and deletions, and a positive one supports membership proofs.

1.2 More on Related Work

Comparison with Boneh et al. While Boneh et al.'s VC construction [4] is novel and efficient, their technique inherently works bit-by-bit. (They also discuss how to build KVCs from VCs.) The exponent is always a subset product of elements corresponding to the bits set to 1. It is not clear how to get rid of this

restriction. This limitation also translates to updates as one would have to know which bits would have to be flipped from 0 to 1 and 1 to 0. This cannot be done simply using the knowledge of update value. Our construction takes a completely different approach by encoding the values directly as a linear function, which allows us to get past the bit-by-bit issues as well as handle updates while being oblivious to prior value (very important for blockchain applications). Our KVC algorithms involve constant number of group operations as opposed to linear in bit-length; the constants are small (between three and five); and, our proof size is three as opposed to seven.

Recent work. Lately, vector commitments has been a very active area of research. Several new constructions have been proposed. We discuss a few of them here.

Campanelli et al. [9] study aggregation for subvector commitments. They introduce new incremental aggregation and disaggregation properties for SVCs. Incremental aggregation allows one to merge different subvector openings into a single concise opening, and to further aggregate merged openings (without knowing the entire committed vector). Disaggregation allows one to 'break down' openings into openings of subvectors. Campanelli et al. point out that though a form of aggregation is already present in the VC of Boneh et al. [4], it can be performed only once. They construct new VC schemes where openings can be aggregated an unbounded number of times.

Campanelli et al. provide two constructions with constant-size public parameters, a CRS in their case. However, the CRS needs to be 'specialized' for a length n before it can be used for commitment. The first construction, based on Boneh et al. [4], has a dependence on the bit-length of elements. Verification in the second construction, based on Catalano and Fiore [10], requires generation of n primes (unless they are already stored). Furthermore, Campanelli et al. do not consider updates to vector elements. Our KVC construction, on the other hand, can be used to build a VC that does not restrict the number of elements in a commitment and allows them to be updated. However, like Boneh et al., we can only support one-hop aggregation (we do not know how to further aggregate aggregated proofs).

Pointproofs [17] study aggregatable VCs in the cross-commitment setting. Proofs for subvectors of multiple commitments can be aggregated by any third party into a single proof for the multiple subvectors. When applied to smart contracts storage, Pointproofs reduces validators' storage requirements considerably. While Pointproofs have very short proofs, the public parameters are linear in the length of the vector. The scheme relies on a q-type assumption in a bilinear group. Further, in order to prove security of aggregation, they need to work in the algebraic group model.

Tomescu et al. [28] also study aggregatable subvector commitment and give an efficient construction in prime-order groups from constant-sized polynomial commitments. (They also provide a detailed and extensive overview of the literature.) Their public parameters also depend on the length of the vector though.

They discuss how to build a stateless cryptocurrency from their VC but the number of users n that need to be supported must be known in advance.

1.3 Organization

We first provide a technical overview of our construction (Section 2). We discuss two intermediate steps to building a KVC: an insert-only KVC that allows insertion of new items but no updates to existing ones, and an increment-only KVC that allows incrementing the values corresponding to keys but no inserts (it is assumed that all keys are initialized with value 0).

In Section 3, we describe the notation used in the paper, define key-value commitments formally, and state the assumptions we need to prove security. In Sections 4 and 5, we build an insert-only KVC and a full KVC, respectively. We show how the former can be used to build accumulators and the latter naturally lends itself to vector commitments. Finally, in Section 6, we show how to batch membership proofs together.

We provide a formal description of the increment-only KVC in the full version of the paper.

2 Overview

In this section, we provide a technical overview of our construction of key-value commitments. Informally, a key-value commitment 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. To motivate our construction, we build up to it in three steps.

2.1 Insert-only

We begin by describing an insert-only key-value commitment construction. By insert-only, we mean that we only insert key-value pairs into the commitment and do not the update the value corresponding to a key. Our key-value commitment C to the key-value map $\mathcal{M} = \{(k_i, v_i)\}_{i\in[q]} \subseteq \mathcal{K} \times \mathcal{V}$ takes the form

$$C_{\mathcal{M}} = \left(g^{\sum_{i\in[q]} v_i \prod_{j\in[q]\setminus\{i\}} z_j}, g^{\prod_{i\in[q]} z_i}\right),$$

where z_i are random strings[4] corresponding to the keys k_i. The crucial observation here is that the exponent in the first element of the commitment, namely $g^{\sum_{i\in[q]} v_i \prod_{j\in[q]\setminus\{i\}} z_j}$, is a linear function of the values v_i. The proof corresponding to a key $k_m \in \mathcal{K}_{\mathcal{M}}$ would be

$$\Lambda_{k_m} = \left(g^{\sum_{i\in[q]\setminus\{m\}} v_i \prod_{j\in[q]\setminus\{i,m\}} z_j}, g^{\prod_{i\in[q]\setminus\{m\}} z_i}\right)$$

[4] The precise requirement on $\{z_i\}$ will be described later.

which is essentially the key-value commitment of all the other key-value pairs in $\mathcal{M}$. Let us assume that the strings z_i can be publicly generated from the keys k_i, say by some hash function[5]. Then, the verification of the proof Λ_{k_m} for a key-value pair (k_m, v_m) would simply be the following checks:

$$\begin{aligned}
\Lambda_{k_m,1}^{z_m} \cdot \Lambda_{k_m,2}^{v_m} &= \left(g^{\sum_{i\in[q]\setminus\{m\}} v_i \prod_{j\in[q]\setminus\{i,m\}} z_j}\right)^{z_m} \cdot \left(g^{\prod_{i\in[q]\setminus\{m\}} z_i}\right)^{v_m} \\
&= g^{\sum_{i\in[q]\setminus\{m\}} v_i \prod_{j\in[q]\setminus\{i\}} z_j} \cdot g^{v_m \prod_{i\in[q]\setminus\{m\}} z_i} \\
&= g^{\sum_{i\in[q]} v_i \prod_{j\in[q]\setminus\{i\}} z_j} \\
&= C_{\mathcal{M},1}
\end{aligned}$$

and

$$\begin{aligned}
\Lambda_{k_m,2}^{z_m} &= \left(g^{\prod_{i\in[q]\setminus\{m\}} z_i}\right)^{z_m} \\
&= g^{\prod_{i\in[q]} z_i} \\
&= C_{\mathcal{M},2}
\end{aligned}$$

Note that the above design also describes a procedure to insert a key-value pair into the commitment. To insert a new pair (k_{q+1}, v_{q+1}) into $\mathcal{M}$, we set

$$C_{\mathcal{M}'} = \left(C_{\mathcal{M},1}^{z_{q+1}} \cdot C_{\mathcal{M},2}^{v_{q+1}}, C_{\mathcal{M},2}^{z_{q+1}}\right)$$

where $\mathcal{M}' = \mathcal{M} \cup \{(k_{m+1}, v_{m+1})\}$. Furthermore, the proof $\Lambda_{k_{q+1}}$ corresponding to the key k_{q+1} would be $C_{\mathcal{M}}$.

We also have a straightforward way to update proofs for the existing keys when a new key-value pair is inserted. For instance, on inserting (k_{q+1}, v_{q+1}), we can update the proof Λ_{k_m} corresponding to the key k_m with the help of the update information (k_{q+1}, v_{q+1}) as follows:

$$\Lambda'_{k_m} = \left(\Lambda_{k_m,1}^{z_{q+1}} \cdot \Lambda_{k_m,2}^{v_{q+1}}, \Lambda_{k_m,2}^{z_{q+1}}\right).$$

One can easily check that the modified proof would be successfully verified.

The final piece of the puzzle is key binding, i.e., it must be computationally infeasible for an adversary to produce a proof corresponding to a key k that verifies for a key-value pair (k, v) which is not in the key-value commitment. We analyze this as follows. Suppose an adversary comes up with a key-value pair (k, v) and a proof $\Lambda_k = (\Lambda_{k,1}, \Lambda_{k,2})$ corresponding to the key k. Let us assume that $k \in \mathcal{K}_{\mathcal{M}}$ (the other case is easier to handle). Let m denote the index of the key k in the map $\mathcal{M}$, i.e., let $k = k_m$. Let z_m be the string corresponding to k_m. We will, for reasons that will be clear shortly, require that z_m be an *odd* string.

[5] We will actually not require any private randomness in $\{z_i\}$, just that there is some concise representation of them. This will be clear from the fact that we publish the key for the hash function to enable its public evaluation.

Let v_m be the value corresponding to the key k_m in the commitment and let $\Lambda_{k_m} = (\Lambda_{k_m,1}, \Lambda_{k_m,2})$ be the corresponding proof. We first check that

$$\Lambda_{k,2}^{z_m} = C_{\mathcal{M},2}.$$

Suppose that the check passes. Then, with overwhelming probability,

$$\Lambda_{k,2} = g^{\prod_{i \in [q] \setminus \{m\}} z_i} = \Lambda_{k_m,2}.$$

This is because, otherwise,

$$\alpha = \frac{\Lambda_{k,2}}{\Lambda_{k_m,2}} \neq \pm 1$$

is a non-trivial z_mth root of unity. We know that such elements are hard to find without knowing the order of the group. In particular, this means that we have computed the order of the non-trivial element α.

We now proceed to the second and final check. We check that

$$\Lambda_{k,1}^{z_m} \cdot \Lambda_{k,2}^{v} = C_{\mathcal{M},1}.$$

Note that

$$\Lambda_{k_m,1}^{z_m} \cdot \Lambda_{k_m,2}^{v_m} = C_{\mathcal{M},1}.$$

and, with overwhelming probability, $\Lambda_{k,2} = \Lambda_{k_m,2}$. Hence

$$\left(\frac{\Lambda_{k,1}}{\Lambda_{k_m,1}}\right)^{z_m} = \Lambda_{k_m,2}^{v_m - v}.$$

We only care about the case $v_m \neq v$. Note that setting

$$\beta = \frac{\Lambda_{k,1}}{\Lambda_{k_m,1}}$$

and

$$\gamma = \Lambda_{k_m,2}^{v_m - v}$$

we have found β, a z_mth root of a non-trivial element γ, which should be hard to find in groups of unknown order.

Formally, we can show that if the RSA assumption holds, then computing β is indeed hard. This can be seen as follows. The exponent of g in $\Lambda_{k_m,2}$ contains numbers that are coprime to z_m. Furthermore, we will choose $\{z_i\}$ to be larger than the permitted value space, which ensures that $v_m - v$ is also coprime to z_m. Thus, if one can compute β, which is a z_mth root of γ, we can actually compute a z_mth root of g through an application of Shamir's trick, which would break the RSA assumption.

2.2 Increment-only

We next describe an increment-only key-value commitment construction. By increment-only, we mean that values corresponding to *all* keys are initialized to 0 and we can just update them by some amount δ every time. We propose that our commitment C to a key-value map $\mathcal{M} = \{(k_i, v_i)\}_{i \in [q]} \subseteq \mathcal{K} \times \mathcal{V}$ takes the form

$$C_{\mathcal{M}} = g^{\prod_{i \in [q]} z_i^{v_i}},$$

where z_i are random primes corresponding to the keys k_i. Let us assume that the strings z_i can be publicly generated from the keys k_i, say by some hash function.

This construction is reminiscent of the RSA-based accumulator construction of Li et al. [21] and Boneh et al. [4]. The proof corresponding to a key $k_m \in \mathcal{K}_{\mathcal{M}}$ would need to consist of two parts, one to show that the exponent of z_m is at least v_m and one to show that it is at most v_m. The first part of the proof would be

$$\Lambda_{k_m,1} = g^{\prod_{i \in [q] \setminus \{m\}} z_i^{v_i}},$$

which is essentially the key-value commitment of all the other key-value pairs in $\mathcal{M}$. It can be used to show that the exponent of z_m is at least v_m by verifying that

$$\Lambda_{k_m,1}^{z_m^{v_m}} = C_{\mathcal{M}}.$$

The second part of the proof uses the idea from Li et al. [21] and Boneh et al. [4] that was used to build non-membership proofs. Basically, we would like to show that z_m is coprime to $\prod_{i \in [q] \setminus \{m\}} z_i^{v_i}$. This can be done by leveraging Bezout coefficients $a, b \in \mathbb{Z}$ such that

$$a \cdot z_m + b \cdot \prod_{i \in [q] \setminus \{m\}} z_i^{v_i} = 1.$$

2.3 Putting it all together

We re-examine the insert-only key-value commitment, namely,

$$C_{\mathcal{M}} = \left(g^{\sum_{i \in [q]} v_i \prod_{j \in [q] \setminus \{i\}} z_j}, g^{\prod_{i \in [q]} z_i} \right)$$

and

$$\Lambda_{k_m} = \left(g^{\sum_{i \in [q] \setminus \{m\}} v_i \prod_{j \in [q] \setminus \{i,m\}} z_j}, g^{\prod_{i \in [q] \setminus \{m\}} z_i} \right).$$

The exponent of the first component of $C_{\mathcal{M}}$ is linear in the values that are being committed. Thus, in order to change the value corresponding to key k_m by δ, it is enough to perform the following operation:

$$C_{\mathcal{M}',1} = C_{\mathcal{M},1} \cdot g^{\delta \cdot \prod_{i \in [q] \setminus \{m\}} z_i}.$$

This means that one only needs to know the index m[6] and δ in order to perform an update on the value corresponding to the key k_m in the commitment. Let

$$\beta = g^{\prod_{i \in [q] \setminus \{m\}} z_i}.$$

Firstly, note that β can be generated publicly. However, one would have to generate all the z_i for $i \in [q] \setminus \{m\}$ (or retrieve it from some persistent storage) and then perform $q - 1$ exponentiations, making it a computationally intensive task. However, notice that

$$\Lambda_{k_m,2} = \beta.$$

Thus, if we assume that the party involved in the update of the mth element has access to Λ_{k_m}, we can assume that we have access to β and we do not have to recompute it every time an update is issued. However, we would like to overcome this limitation. For instance, in the case of a blockchain, a sender who wants to send another user some funds will not be able to update the key-value commitment to reflect the changes in the recipient's account efficiently unless they have access to the proof of the recipient.

Let us turn to the proofs. Note that if we perform an update at index m, the proof corresponding to the key k_m does not change. For $n \neq m$, we can update the proof corresponding to the key k_n by performing the following operation:

$$\Lambda_{k_n,1} = \Lambda_{k_n,1} \cdot g^{\delta \cdot \prod_{j \in [q] \setminus \{m,n\}} z_j}.$$

Let

$$\gamma = g^{\prod_{j \in [q] \setminus \{m,n\}} z_j}.$$

Again, note that γ can be generated publicly but it would be a computationally intensive task. The situation is even worse here because γ cannot be computed efficiently from any other information that is available as that would amount to taking z_mth roots of elements.

The construction we propose now circumvents all of these issues. It gets rid of indices and allows for efficient updates of the key-value commitment and proofs. Moreover, we can do so without involving the party (their proof) whose value is being modified. Our commitment C to the key-value map $\mathcal{M} = \{(k_i, v_i)\}_{i \in [q]} \subseteq \mathcal{K} \times \mathcal{V}$ takes the form

$$C_{\mathcal{M}} = \left(g^{\left(\sum_{i \in [q]} v_i \prod_{j \in [q] \setminus \{i\}} z_j\right) \cdot \prod_{i \in [q]} z_i^{u_i}}, g^{\prod_{i \in [q]} z_i^{u_i + 1}} \right),$$

where u_i denotes the number of updates made to the value corresponding to the key k_i. With this form, insertion can be carried out in exactly the same way as in the case of the insert-only construction.

[6] While this is already pretty good, we ideally would like to get rid of the notion of an index and carry out an update just knowing the key k_m. We do achieve this in our construction.

More interestingly, if $C_{\mathcal{M}'}$ denotes the new commitment value after changing the value corresponding to key k_m by δ, then

$$\begin{aligned} C_{\mathcal{M}',1} &= C_{\mathcal{M},1}^{z_m} \cdot g^{\delta \cdot z_m^{u'_m} \prod_{i \in [q] \setminus \{m\}} z_i^{u_i+1}} \\ &= C_{\mathcal{M},1}^{z_m} \cdot g^{\delta \cdot z_m^{u_m+1} \prod_{i \in [q] \setminus \{m\}} z_i^{u_i+1}} \\ &= C_{\mathcal{M},1}^{z_m} \cdot g^{\delta \cdot \prod_{i \in [q]} z_i^{u_i+1}} \\ &= C_{\mathcal{M},1}^{z_m} \cdot C_{\mathcal{M},2}^{\delta}, \end{aligned}$$

where $u'_m = u_m + 1$ denotes the (new) total number of updates for key k_m. Note how using the number of updates $\{u_i\}$ in the exponents of the $\{z_i\}$ lets us perform an update without having to compute values akin to β (and γ while updating proofs) which have to miss some $\{z_i\}$ in the exponent, rendering their computation inefficient.

In fact, it is now easy to see that

$$C_{\mathcal{M}'} = \left(C_{\mathcal{M},1}^{z_m} \cdot C_{\mathcal{M},2}^{\delta}, C_{\mathcal{M},2}^{z_m} \right).$$

Thus, an update works exactly how an insert would and this is an important property our constructions enjoys that we will come back to later in seeing how this primitive fits into the blockchain setting. It can also be easily deduced now that this design has a very similar consequence on enabling update to proofs efficiently. In particular, recall that updating the values corresponding to keys and inserting a new key have similar effect. Since we know how to update proofs following an insert, we can update them in exactly the same way on an update to some value.

To prove key-binding, it will be crucial for a verifier to know precisely the number of updates that have been performed on the value corresponding to the key for which a proof is provided. If verifier is not aware of this, it is possible for an attacker to generate a "fake" proof and break key-binding. Indeed, the number of updates made to the values corresponding to each key is public in the blockchain setting, but we would not want parties to store this information as it grows linearly with the number of key-value pairs that are inserted into the commitment.

For this purpose, we can include in the commitment $C_{\mathcal{M}}$ an increment-only key-value commitment that stores the number of updates performed on each key. During verification, proofs would now have to contain a proof for the right number of updates that have been performed in addition to the proof for the value itself. In this way, the verifying party need not store the number of updates that have been made to the values corresponding to each of the keys. The final step is to observe that the second component of the key-value commitment, $C_{\mathcal{M},2}$, essentially acts as an increment-only key-value commitment for the number of updates. Thus, we do not need to add anything else to $C_{\mathcal{M}}$.

3 Preliminaries

3.1 Notation

For $n \in \mathbb{N}$, let $[n] = \{1, 2, \ldots, n\}$. Let $\lambda \in \mathbb{N}$ denote the security parameter. Let $|b|$ denote the bit-length of $b \in \mathbb{N}$. Let Primes denote the set of integer primes and $\mathsf{Primes}(\lambda)$ denote the set of integer primes less than 2^λ. 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 $\mathrm{poly}(\cdot)$, we denote any function which is bounded by a polynomial in its argument. An algorithm $\mathcal{T}$ 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 $\mathcal{A}$.

3.2 Key-Value Commitments

Informally, a key-value commitment 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. 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 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:

- $(\mathsf{pp}, C) \leftarrow_\$ \mathsf{KeyGen}(1^\lambda)$: On input the security parameter λ, the key 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.
- $(C, \Lambda_k, \mathsf{upd}) \leftarrow \mathsf{Insert}(C, (k, v))$: On input a commitment string C and a key-value pair $(k, v) \in \mathcal{K} \times \mathcal{V}$, the insertion algorithm outputs a new commitment string C, a proof Λ_k (that v is the value corresponding to k), and update information upd.
- $(C, \mathsf{upd}) \leftarrow \mathsf{Update}(C, (k, \delta))$: On input a commitment string C, a key $k \in \mathcal{K}$ and an update value δ[7], the update algorithm outputs an updated string C and update information upd. Note that this algorithm does not need the value corresponding to the key k.

[7] We assume updates are additive, i.e., updating a value v by δ amounts to changing v to $v + \delta$.

– $\Lambda_k \leftarrow \mathsf{ProofUpdate}(k, \Lambda_k, \mathsf{upd})$: On input a key $k \in \mathcal{K}$, a proof Λ_k for some value corresponding to the key k and update information upd, the proof update algorithm outputs an updated proof Λ_k.
– $1/0 \leftarrow \mathsf{Ver}(C, (k, v), \Lambda_k)$: On input a 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).

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 a 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 key 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 $(k, v) \in \mathcal{M}$.

To present the requirement formally, we define a correctness game. We have an adversary in this game to capture the arbitrary order in which inserts and updates can be applied to a commitment. We do not provide it any capability to do something beyond that.

Definition 1. *For a key-value commitment* KVC *and an adversary* $\mathcal{A}$*, we define a random variable* $\mathcal{G}^{\text{correct}}_{\mathsf{KVC},\lambda,\mathcal{A}}$ *through a game between a challenger* CH *and* $\mathcal{A}$ *as follows:*

$\underline{\mathcal{G}^{\text{correct}}_{\mathsf{KVC},\lambda,\mathcal{A}}}$:

1. CH *samples* $(\mathsf{pp}, C) \leftarrow_{\$} \mathsf{KeyGen}(1^\lambda)$ *and sends them to* $\mathcal{A}$*.* CH *also maintains its own state comprising a key-value map* $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{V}$ *initialized to the empty map, a key-proof map* $\mathcal{P}$ *initialized to the empty map, and the initial commitment value* C*.*
2. $\mathcal{A}$ *issues queries of one of the following forms:*
   – $(\mathsf{Insert}, (k, v))$*:* CH *checks if* $\mathcal{M}$ *contains a tuple of the form* $(k, \cdot)$*. If so,* CH *responds with* $\perp$*. If not,* CH *updates* $\mathcal{M}$ *to* $\mathcal{M} \cup \{(k, v)\}$ *and executes* $\mathsf{Insert}(C, (k, v))$ *to obtain a new commitment* C*, along with* Λ_k *and* upd*.* CH *then updates* $\mathcal{P}$ *to* $\mathcal{P} \cup \{(k, \Lambda_k)\}$*.*
   – $(\mathsf{Update}, (k, \delta))$*:* CH *checks if* $\mathcal{M}$ *contains a tuple of the form* (k, v)*. If not,* CH *responds with* $\perp$*. If so,* CH *updates* $\mathcal{M}$ *to* $(\mathcal{M} \cup \{(k, v + \delta)\}) \setminus \{(k, v)\}$ *and executes* $\mathsf{Update}(C, (k, \delta))$ *to obtain a new commitment* C*, along with update value* upd*.*

   Any time $\mathcal{A}$ *issues a query,* CH *deals with the query as above and then performs the following updates and checks:*
   – *Let* upd *be the update information obtained by* CH *while processing* $\mathcal{A}$*'s most recent query. For each tuple* $(k, \Lambda_k) \in \mathcal{P}$*,* CH *updates* Λ_k *by executing* $\mathsf{ProofUpdate}(k, \Lambda_k, \mathsf{upd})$*.*
   – *For each tuple* $(k, v) \in \mathcal{M}$ *with the corresponding tuple* $(k, \Lambda_k) \in \mathcal{P}$*,* CH *obtains* $b_k \leftarrow \mathsf{Ver}(C, (k, v), \Lambda_k)$*. If for any* k*,* $b_k = 0$*, then* CH *outputs* $\mathsf{failure}$ *and terminates.*
3. CH *outputs* $\mathsf{success}$*.*

The value of the random variable $\mathcal{G}^{\text{correct}}_{\mathsf{KVC},\lambda,\mathcal{A}}$ *is defined to be the output of* CH*, namely,* failure *or* success*.*

Definition 2 (Correctness). *A key-value commitment* KVC *is* correct *if for every adversary* $\mathcal{A}$*, the following probability is identically zero:*

$$\mathsf{Adv}^{\text{correct}}_{\mathsf{KVC},\mathcal{A}}(\lambda) = \Pr\left[\mathsf{failure} \leftarrow_{\$} \mathcal{G}^{\text{correct}}_{\mathsf{KVC},\lambda,\mathcal{A}}\right]$$

The security requirement for key-value commitments is that of key binding. Informally, this says that it should be infeasible for any polynomially bounded adversary (with knowledge of pp) to come up with an *honestly generated* commitment and two proofs that certify to different values for the same key, or a single proof that certifies to a value for a key that has not been inserted. The adversary is, however, disallowed from executing Insert more than once with respect to any k. We present the requirement formally below. We first define a key-binding game.

Definition 3. *For a key-value commitment* KVC *and an adversary* $\mathcal{A}$*, we define a random variable* $\mathcal{G}^{\text{bind}}_{\mathsf{KVC},\lambda,\mathcal{A}}$ *through a game between a challenger* CH *and* $\mathcal{A}$ *as follows:*

$\underline{\mathcal{G}^{\text{bind}}_{\mathsf{KVC},\lambda,\mathcal{A}}:}$

1. CH *samples* $(\mathsf{pp}, C) \leftarrow_{\$} \mathsf{KeyGen}(1^{\lambda})$ *and sends them to* $\mathcal{A}$*.* CH *also maintains its own state comprising a key-value map* $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{V}$ *initialized to the empty map and the initial commitment value* C*.*
2. $\mathcal{A}$ *issues queries of one of the following forms:*
   - $(\mathsf{Insert}, (k, v))$*:* CH *checks if* $\mathcal{M}$ *contains a tuple of the form* $(k, \cdot)$*. If so,* CH *responds with* $\perp$*. If not,* CH *updates* $\mathcal{M}$ *to* $\mathcal{M} \cup \{(k, v)\}$ *and executes* $\mathsf{Insert}(C, (k, v))$ *to obtain a new commitment* C*.*
   - $(\mathsf{Update}, (k, \delta))$*:* CH *checks if* $\mathcal{M}$ *contains a tuple of the form* (k, v)*. If not,* CH *responds with* $\perp$*. If so,* CH *updates* $\mathcal{M}$ *to* $(\mathcal{M} \cup \{(k, v + \delta)\}) \setminus \{(k, v)\}$ *and executes* $\mathsf{Update}(C, (k, \delta))$ *to obtain a new commitment* C*.*
3. $\mathcal{A}$ *sends a final output to* CH *of one of the following forms:*
   - *Type 1: a key* k *such that* $\mathcal{M}$ *does not contain a tuple of the form* $(k, \cdot)$*, a value* v*, and a proof* Λ_k*.*
   - *Type 2: a key* k *such that* $\mathcal{M}$ *contains a tuple of the form* $(k, \cdot)$*, a pair of values* (v, v') *where* $v \neq v'$*, and a pair of proofs* (Λ_k, Λ'_k)*.*
4. CH *performs the following checks corresponding to* $\mathcal{A}$*'s output:*
   - *Type 1: if* $\mathsf{Ver}(C, (k, v), \Lambda_k) = 1$*, then* CH *outputs* failure*. Otherwise,* CH *outputs* success*.*
   - *Type 2: if* $\mathsf{Ver}(C, (k, v), \Lambda_k) = \mathsf{Ver}(C, (k, v'), \Lambda'_k) = 1$*, then* CH *outputs* failure*. Otherwise,* CH *outputs* success*.*

The value of the random variable $\mathcal{G}^{\text{bind}}_{\mathsf{KVC},\lambda,\mathcal{A}}$ *is defined to be the output of* CH*, namely,* failure *or* success*.*

Definition 4 (Key-binding). *A key-value commitment* KVC *is* key-binding *if for every PPT adversary* $\mathcal{A}$*, the following probability is negligible in* λ*:*

$$\mathsf{Adv}^{\mathrm{bind}}_{\mathsf{KVC},\mathcal{A}}(\lambda) = \Pr\left[\mathsf{failure} \leftarrow_{\$} \mathcal{G}^{\mathrm{bind}}_{\mathsf{KVC},\lambda,\mathcal{A}}\right].$$

Notice that in the definition of $\mathcal{G}^{\mathrm{bind}}_{\mathsf{KVC},\lambda,\mathcal{A}}$, the commitment C is honestly generated by the challenger CH based on the queries issued by the adversary $\mathcal{A}$. Also note that the definition only uses the Insert and Update routines of the key-value commitment, as these are the only two that impact the value of the commitment. Indeed, the adversary can perform all operations by itself given the public parameters. However, the purpose of the game is to define the honestly generated commitment with respect to which the adversary will attempt to produce "fake" proofs. Definition 4 states that no PPT adversary will be able to do so.

We note that key-value commitments are not required to satisfy any sort of hiding property, although one can also define key-value commitments that are hiding. Informally, a key-value commitment is hiding if an adversary cannot distinguish whether a commitment was created to a key-value map $\mathcal{M}$ or to another key-value map $\mathcal{M}'$ even after learning the values corresponding to keys that have the same value in both maps.

KVC for account-based cryptocurrency. In Section 1.1, we briefly discussed how a key-value commitment can be used to build an account-based cryptocurrency with stateless validation. When a validator receives a transaction $tx = (pk, \sigma, v, ct, b, pk^\star, \pi)$ to transfer an amount b from Alice's public key pk to Bob's public key $pk^\star$, it checks if σ is a valid signature on $(v, ct, b, pk^\star, \pi)$, π is a valid proof of the membership of $(pk, (v||ct))$ w.r.t. to the commitment state C, and b is at most v.

The validator can then execute Update with C and $(pk, -b||1)$ to produce a new commitment C' that reflects the change to Alice's account, but what should it do for Bob? Perhaps it should insert $pk^\star$ into C' if this is the first time that $pk^\star$ is being used in a transaction, or update it otherwise. However, there is no way for the validator to know this unless it keeps a copy of the blockchain. Fortunately, in our construction KVaC, insert and update work in the exact same way (see Figure 2), so the validator could always just do an update. (The only difference between these operations is that the former outputs a proof whereas the latter doesn't.)

Let us look at the situation from Alice and Bob's perspective. While Alice has a proof π for her key pk that she can update, Bob may not have any proof for $pk^\star$ if this is the first time someone is sending money to him. We could help Bob generate a proof by including the latest state of the commitment value in every block. If the transaction tx is accepted into a block ℓ, Bob could use the commitment C' from block $\ell - 1$ to quickly generate a proof for $pk^\star$ by applying all the updates in block ℓ to C' except his own.

3.3 Assumptions

We describe in this section the various hardness assumptions that we will use in this work.

Groups of Unknown Order. We assume the existence of a randomized polynomial time algorithm $\mathsf{GGen}(\lambda)$ that takes as input the security parameter λ and outputs two integers a, b along with the description of a group $\mathbb{G}$ of unknown order in the range $[a, b]$ such that a, b and $a - b$ are all integers exponential in λ (similar to Boneh et al. [4]). We will suppress a, b when they are understood or not required.

RSA Assumption. Informally, the RSA assumption states that an adversary cannot compute a random root of a random group element. In the game, the challenger runs the generation algorithm $\mathsf{GGen}(\lambda)$ to obtain integers a, b such that a, b and $a - b$ are exponential in λ and the description of a group $\mathbb{G}$ of unknown order in the interval $[a, b]$. It also samples a random group element w and a random λ-bit prime ℓ, and outputs w and ℓ to the adversary. The adversary is then supposed to return an ℓth root u of w.

Definition 5 (RSA). *The RSA assumption holds for the algorithm* GGen *if for any PPT adversary* $\mathcal{A}$*, the following probability is negligible in* λ*:*

$$\mathsf{Adv}^{\mathrm{RSA}}_{\mathcal{A}}(\lambda) = \Pr\left[u^{\ell} = w : \begin{array}{c}(a, b, \mathbb{G}) \leftarrow_{\$} \mathsf{GGen}(\lambda) \\ w \leftarrow_{\$} \mathbb{G} \\ \ell \leftarrow_{\$} \mathsf{Primes}(\lambda) \\ u \leftarrow_{\$} \mathcal{A}(a, b, \mathbb{G}, w, \ell)\end{array}\right].$$

We would like to generalize the RSA assumption. For the assumption to make sense, we must maintain that ℓ is invertible modulo Q with overwhelming probability, where Q is the order of the group. If ℓ is a prime larger than Q, ℓ is certainly coprime to Q and hence invertible modulo Q. This also means that every element has an ℓth root. In particular, for any w, $u = w^{\ell^{-1}}$ is well-defined. Intuitively, the problem of finding an ℓth root should still be hard. This leads us to a generalized form of RSA.

Definition 6 (Generalized RSA). *The Generalized RSA assumption holds for the algorithm* GGen *if for any PPT adversary* $\mathcal{A}$*, the following probability is negligible in* λ*:*

$$\mathsf{Adv}^{\mathrm{GRSA}}_{\mathcal{A}}(\lambda) = \Pr\left[u^{\ell} = w : \begin{array}{c}(a, b, \mathbb{G}) \leftarrow_{\$} \mathsf{GGen}(\lambda),\ |b| = \zeta \\ w \leftarrow_{\$} \mathbb{G} \\ \ell \leftarrow_{\$} \mathsf{Primes}(\zeta + 1) \setminus [b] \\ u \leftarrow_{\$} \mathcal{A}(a, b, \mathbb{G}, w, \ell)\end{array}\right].$$

Strong RSA Assumption. Informally, the strong RSA assumption states that an adversary cannot compute any non-trivial root of a random group element. In the game, the challenger runs the generation algorithm $\mathsf{GGen}(\lambda)$ to obtain the description of a group $\mathbb{G}$ of unknown order. It also samples a random group

element w and gives it to the adversary. The adversary is then supposed to return an ℓth root u of w for any odd prime ℓ of its choice. In particular, in the strong RSA assumption, the adversary gets to pick ℓ, while in the (regular) RSA assumption, the adversary is given a randomly chosen ℓ.

Definition 7 (Strong RSA). *The Strong RSA assumption holds for the algorithm* GGen *if for any PPT adversary* $\mathcal{A}$*, the following probability is negligible in* λ*:*

$$\mathsf{Adv}^{\mathrm{SRSA}}_{\mathcal{A}}(\lambda) = \Pr\left[\begin{array}{c} u^\ell = w \\ \ell \in \mathsf{Primes} \setminus \{2\} \end{array} : \begin{array}{c} (a, b, \mathbb{G}) \leftarrow_\$ \mathsf{GGen}(\lambda) \\ w \leftarrow_\$ \mathbb{G} \\ (u, \ell) \leftarrow_\$ \mathcal{A}(a, b, \mathbb{G}, w) \end{array}\right].$$

4 An Insert-only Key-Value Commitment

- $\mathsf{KeyGen}(1^\lambda)$: Sample the description of a group $(a, b, \mathbb{G}) \leftarrow_\$ \mathsf{GGen}(\lambda)$ of unknown order in the range $[a, b]$ where a, b and $a - b$ are exponential in λ, and a random element $g \leftarrow_\$ \mathbb{G}$[a]. Set $\mathcal{V} = [0, a)$ and $\mathcal{K} = \{0,1\}^*$. Let ζ denote the bit-length of b, i.e., $\zeta = |b|$. Sample the description of a hash function H that maps arbitrary strings to unique primes from the set $\mathsf{Primes}(\zeta + 1) \setminus [b]$[b]. Output $(\mathsf{pp}, C) = ((a, b, \mathbb{G}, g, H), (1, g))$.
- $\mathsf{Insert}(C, (k, v))$: Parse the input C as (C_1, C_2). Let $z = H(k)$. Set

  $$\Lambda_k = C, \quad C = (C_1^z \cdot C_2^v, C_2^z), \quad \mathsf{upd} = (k, v).$$

  Output $(C, \Lambda_k, \mathsf{upd})$.
- $\mathsf{ProofUpdate}(k, \Lambda_k, \mathsf{upd})$: Parse Λ_k as $(\Lambda_{k,1}, \Lambda_{k,2})$ and upd as $(\mathsf{upd}_1, \mathsf{upd}_2)$. Let $z = H(\mathsf{upd}_1)$. Set

  $$\Lambda_k = \left((\Lambda_{k,1})^z \cdot (\Lambda_{k,2})^{\mathsf{upd}_2}, (\Lambda_{k,2})^z\right).$$

  Output Λ_k.
- $\mathsf{Ver}(C, (k, v), \Lambda_k)$: Parse the input C as (C_1, C_2) and Λ_k as $(\Lambda_{k,1}, \Lambda_{k,2})$. Let $z = H(k)$. Check whether:
  - $v \in \mathcal{V}$ and $k \in \mathcal{K}$,
  - $(\Lambda_{k,2})^z = C_2$,
  - $(\Lambda_{k,1})^z \cdot (\Lambda_{k,2})^v = C_1$.

  If all the checks pass, output 1; else, output 0.

[a] A random element of $\mathbb{G}$ would be a generator of $\mathbb{G}$ with overwhelming probability and, at the very least, would not be of low order.

[b] Using existing techniques, we can sample hash functions that satisfy this property with high probability. For details, please refer to the full version.

Fig. 1. KVC-Ins: Insert-only KVC construction

We begin by describing an insert-only key-value commitment KVC-Ins. By insert-only, we mean that we only insert key-value pairs into the commitment but do not the update the value corresponding to a key. We also note that we are only concerned with the case of inserting values corresponding to distinct keys, i.e., we assume that the insert algorithm is not invoked with the same key more than once. The construction KVC-Ins is formally described in Figure 1.

Completeness and Efficiency. The correctness of the scheme follows directly from inspection. Also note that all operations involve (at most) one hash computation, three exponentiations and one multiplication. The size of the key-value commitment is constant, namely, two group elements. This is also true of the proofs corresponding to keys.

4.1 Key Binding

If we model the hash function H in the construction as a random oracle, we can prove the key binding of KVC-Ins based on the generalized RSA assumption. We note that while applying the Definitions 3 and 4 for key binding to KVC-Ins, the adversary will not be allowed to issue any update queries (KVC-Ins is an insert-only commitment scheme).

Lemma 1. *Suppose there exists a PPT adversary $\mathcal{A}$ in the random oracle model that satisfies*

$$\mathsf{Adv}^{\mathrm{bind}}_{\mathsf{KVC\text{-}Ins},\mathcal{A}}(\lambda) = \epsilon,$$

where KVC-Ins *is the key-value commitment scheme defined in Figure 1. Then, there exists a PPT adversary $\mathcal{B}$ such that*

$$\mathsf{Adv}^{\mathrm{GRSA}}_{\mathcal{B}}(\lambda) \geq \frac{\epsilon}{T_\lambda^2} - \mathsf{negl}(\lambda),$$

where T_λ denotes the running time of $\mathcal{A}$ parameterized by λ.

Proof. Assume the existence of an adversary $\mathcal{A}$ as stated in the lemma above. We now design the adversary $\mathcal{B}$. On obtaining $(a, b, \mathbb{G}) \leftarrow_{\$} \mathsf{GGen}(\lambda)$ with $|b| = \zeta$, $w \leftarrow_{\$} \mathbb{G}$ and $\ell \leftarrow_{\$} \mathsf{Primes}(\zeta + 1) \setminus [b]$ from the challenger CH, $\mathcal{B}$ first guesses the number, q, of keys that $\mathcal{A}$ would issue hash queries for or insert into the commitment, and the index, m, of the key k_m which $\mathcal{A}$ would provide the "fake" proof for at the end of its execution. Note that each of these choices are limited in number by T_λ and hence $\mathcal{B}$ makes the correct guesses with probability greater than or equal to T_λ^{-2}.

$\mathcal{B}$ chooses $q-1$ unique primes $\{z_i\}_{i\in[q]\setminus\{m\}}$ at random from the set $\mathsf{Primes}(\zeta+1) \setminus [b]$ that it will assign, under the map H, to the set of keys other than the one that $\mathcal{A}$ would provide the "fake" proof for. It computes

$$\pi = w^{\prod_{i\in[q]\setminus\{m\}} z_i}.$$

With all but negligible probability in λ, $\ell \neq z_i$ for any $i \in [q]\setminus\{m\}$. $\mathcal{B}$ sets $g = w$ and $z_m = \ell$. $\mathcal{B}$ sends $((a, b, \mathbb{G}, g), (1, g))$ to $\mathcal{A}$.

$\mathcal{B}$ maintains a key-value map $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{V}$ initialized to the empty map and the initial commitment value $C = (1, g)$. Any time $\mathcal{A}$ issues the ith query, k_i, for the computation of H, $\mathcal{B}$ returns z_i and assigns $H(k_i) = z_i$. Any time $\mathcal{A}$ issues queries as in Definition 3, note that $\mathcal{B}$ has all the values it needs to make updates to C and $\mathcal{M}$ as defined in Definition 3 and Figure 1. If $\mathcal{A}$ aborts at any point in time, $\mathcal{B}$ aborts as well. Assuming no aborts, finally, $\mathcal{A}$ responds with a tuple of the form

1. (k_m, v, Λ) where k_m wasn't inserted by $\mathcal{A}$, or
2. $(k_m, v, v', \Lambda, \Lambda')$ where k_m was inserted by $\mathcal{A}$ and $v \neq v'$.

Case 1: Let $C = (C_1, C_2)$ and $\Lambda = (\Lambda_1, \Lambda_2)$. Since $\mathsf{Ver}(C, (k_m, v), \Lambda) = 1$,

$$\Lambda_2^{\ell} = C_2.$$

Without loss of generality, we can assume that $\mathcal{A}$ inserted all keys that it queried the hash function for into the commitment (other than k_m).[8] Let $\mathcal{M} = \{(k_i, v_i)\}_{i \in [q] \setminus \{m\}}$ be the key-value map committed to in C. We have

$$C_2 = g^{\prod_{i \in [q] \setminus \{m\}} H(k_i)}.$$

Therefore,

$$\Lambda_2^{\ell} = w^{\prod_{i \in [q] \setminus \{m\}} z_i}.$$

Since $\ell \neq z_i$ for any $i \in [q] \setminus \{m\}$, ℓ is coprime to $\prod_{i \in [q] \setminus \{m\}} z_i$. $\mathcal{B}$ then computes integers θ_1, θ_2 such that

$$\theta_1 \cdot \ell + \theta_2 \cdot \prod_{i \in [q] \setminus \{m\}} z_i = 1.$$

Finally, $\mathcal{B}$ computes

$$u = w^{\theta_1} \Lambda_2^{\theta_2}.$$

Note that

$$\begin{aligned} u^{\ell} &= w^{\theta_1 \ell} \Lambda_2^{\theta_2 \ell} \\ &= w^{\theta_1 \ell} (\Lambda_2^{\ell})^{\theta_2} \\ &= w^{\theta_1 \ell} (w^{\prod_{i \in [q] \setminus \{m\}} z_i})^{\theta_2} \\ &= w^{\theta_1 \cdot \ell + \theta_2 \cdot \prod_{i \in [q] \setminus \{m\}} z_i} \\ &= w. \end{aligned}$$

$\mathcal{B}$ forwards u to CH.

[8] If this is not the case, $\mathcal{B}$ can insert (arbitrary) values for the remaining keys after the fact to complete the reduction.

Case 2: Let $\Lambda' = (\Lambda'_1, \Lambda'_2)$. We claim that

$$\Lambda_2 = \Lambda'_2.$$

Since $\mathsf{Ver}(C, (k_m, v), \Lambda) = \mathsf{Ver}(C, (k_m, v'), \Lambda') = 1$,

$$\Lambda_2^\ell = C_2 = \Lambda_2'^\ell.$$

Let

$$\alpha = \frac{\Lambda_2}{\Lambda'_2}.$$

We have that $\alpha^\ell = 1$. Since ℓ is prime, if $\alpha \neq 1$, ℓ must be the order of α in $\mathbb{G}$. But ℓ is larger than the order of $\mathbb{G}$, which is not possible. Hence $\alpha = 1$.

Without loss of generality, we can assume that $\mathcal{A}$ inserted all keys that it queried the hash function for into the commitment. Let $\mathcal{M} = \{(k_i, v_i)\}_{i \in [q]}$ be the key-value map committed to in C. Consider the proof Λ_{k_m} corresponding to the key k_m defined by

$$\Lambda_{k_m} = \left(\Lambda_{k_m,1}, \Lambda_{k_m,2}\right),$$

where

$$\Lambda_{k_m,1} = g^{\sum_{i \in [q] \setminus \{m\}} v_i \prod_{j \in [q] \setminus \{i,m\}} H(k_j)}$$

and

$$\Lambda_{k_m,2} = g^{\prod_{i \in [q] \setminus \{m\}} H(k_i)}.$$

Clearly, $\mathsf{Ver}(C, (k_m, v_m), \Lambda_{k_m}) = 1$. In particular,

$$\Lambda_{k_m,2}^\ell = C_2.$$

Extending our previous argument, we have that

$$\Lambda_2 = \Lambda'_2 = \Lambda_{k_m,2} = g^{\prod_{i \in [q] \setminus \{m\}} H(k_i)}.$$

Since $\mathsf{Ver}(C, (k_m, v), \Lambda) = \mathsf{Ver}(C, (k_m, v'), \Lambda') = 1$, we have

$$\left(\Lambda_1\right)^\ell \cdot \left(\Lambda_2\right)^v = C_1 = \left(\Lambda'_1\right)^\ell \cdot \left(\Lambda'_2\right)^{v'}.$$

This implies that

$$\left(\frac{\Lambda_1}{\Lambda'_1}\right)^\ell = \left(g^{\prod_{i \in [q] \setminus \{m\}} H(k_i)}\right)^{v'-v}.$$

Let

$$\beta = \frac{\Lambda_1}{\Lambda'_1}$$

and

$$v' - v = \delta.$$

Note that $\beta^\ell = \pi^\delta$. This implies that

$$\beta^\ell = w^{\delta \prod_{i \in [q] \setminus \{m\}} z_i}.$$

Since $\mathcal{V} = [0, a)$, $|\delta| < a < b < \ell$, and since ℓ is prime, ℓ is coprime to δ. Also, since $\ell \neq z_i$ for any $i \in [q] \setminus \{m\}$, ℓ is coprime to $\prod_{i \in [q] \setminus \{m\}} z_i$. Hence, ℓ is coprime to $\delta \prod_{i \in [q] \setminus \{m\}} z_i$. $\mathcal{B}$ then computes integers θ_1, θ_2 such that

$$\theta_1 \cdot \ell + \theta_2 \cdot \delta \prod_{i \in [q] \setminus \{m\}} z_i = 1.$$

Finally, $\mathcal{B}$ computes

$$u = w^{\theta_1} \beta^{\theta_2}.$$

Note that

$$\begin{aligned} u^\ell &= w^{\theta_1 \ell} \beta^{\theta_2 \ell} \\ &= w^{\theta_1 \ell} (\beta^\ell)^{\theta_2} \\ &= w^{\theta_1 \ell} \left(w^{\delta \prod_{i \in [q] \setminus \{m\}} z_i}\right)^{\theta_2} \\ &= w^{\theta_1 \cdot \ell + \theta_2 \cdot \delta \prod_{i \in [q] \setminus \{m\}} z_i} \\ &= w. \end{aligned}$$

$\mathcal{B}$ forwards u to CH. This completes the proof.

Removing the random oracle assumption. Intuitively, we need H to be a random oracle only because we are programming the challenge prime ℓ from the RSA assumption as one of the z's output by H. We can however get over this difficulty by letting H output arbitrary primes and letting the adversary choose ℓ as in the game for the strong RSA assumption.

Lemma 2. *Suppose there exists a PPT adversary $\mathcal{A}$ in the standard model that satisfies*

$$\mathsf{Adv}^{\text{bind}}_{\mathsf{KVC\text{-}Ins},\mathcal{A}}(\lambda) = \epsilon,$$

where KVC-Ins *is the key-value commitment scheme defined in Figure 1. Then, there exists a PPT adversary $\mathcal{B}$ such that*

$$\mathsf{Adv}^{\text{SRSA}}_{\mathcal{B}}(\lambda) \geq \epsilon - \mathsf{negl}(\lambda).$$

A proof of this lemma can be found in the full version of the paper.

4.2 Accumulators

Observe that an insert-only key-value commitment directly gives us an accumulator that supports insertions and membership proofs (just use arbitrary keys). Our construction of an insert-only key-value commitment also provides for deletions because the proof corresponding to a key is the commitment of the remainder of the key-value map, which would be the new commitment. Existing proofs can be updated using the techniques of aggregation in Section 6. The idea is that given key-value commitments to the maps $\mathcal{M} \setminus \{(k, v)\}$ and $\mathcal{M} \setminus \{(k', v')\}$, it is possible to create a commitment to the map $\mathcal{M} \setminus \{(k, v), (k', v')\}$. Thus, KVC-Ins can be used to build optimal dynamic positive accumulators [1].

5 A Complete Key-Value Commitment

In this section, we provide our main construction of a key-value commitment. The construction KVaC is formally described in Figure 2.

Completeness and Efficiency. The correctness of the scheme follows directly from inspection. Also note that all operations involve (at most) two hash computations, five exponentiations and two multiplications. The size of the key-value commitment is constant, namely, two group elements. This is also true of the proofs corresponding to keys.

5.1 Key Binding

If we model the hash function H in the construction as a random oracle, we can prove the key binding of KVaC based on the generalized RSA assumption.

Lemma 3. *Suppose there exists a PPT adversary $\mathcal{A}$ in the random oracle model that satisfies*

$$\mathsf{Adv}^{\mathrm{bind}}_{\mathsf{KVaC},\mathcal{A}}(\lambda) = \epsilon,$$

where KVaC is the key-value commitment scheme defined in Figure 2. Then, there exists a PPT adversary $\mathcal{B}$ such that

$$\mathsf{Adv}^{\mathrm{GRSA}}_{\mathcal{B}}(\lambda) \geq \frac{\epsilon}{T_\lambda^3} - \mathsf{negl}(\lambda),$$

where T_λ denotes the running time of $\mathcal{A}$ parameterized by λ.

Proof. Assume the existence of an adversary $\mathcal{A}$ as stated in the lemma above. We now design the adversary $\mathcal{B}$. On obtaining $(a, b, \mathbb{G}) \leftarrow_\$ \mathsf{GGen}(\lambda)$ with $|b| = \zeta$, $w \leftarrow_\$ \mathbb{G}$ and $\ell \leftarrow_\$ \mathsf{Primes}(\zeta + 1) \setminus [b]$ from the challenger CH, $\mathcal{B}$ first guesses the number, q, of keys that $\mathcal{A}$ would issue hash queries for or insert into the commitment, the index, m, of the key k_m which $\mathcal{A}$ would provide the "fake" proof for at the end of its execution, and u, the maximum number of updates performed on the value corresponding to any of the keys inserted by $\mathcal{A}$. Note that each of these choices are limited in number by T_λ and hence $\mathcal{B}$ makes the correct guesses with probability greater than or equal to T_λ^{-3}. We assume without loss of generality that $\mathcal{A}$ makes the same number of updates, u, to the values corresponding to each of the keys inserted into the commitment.

$\mathcal{B}$ chooses $q-1$ unique primes $\{z_i\}_{i\in[q]\setminus\{m\}}$ at random from the set $\mathsf{Primes}(\zeta+1) \setminus [b]$ that it will assign, under the map H, to the set of keys other than the one that $\mathcal{A}$ would provide the "fake" proof for. It computes

$$\pi = w^{\prod_{i\in[q]\setminus\{m\}} z_i^{u+1}}.$$

With all but negligible probability in λ, $\ell \neq z_i$ for any $i \in [q]\setminus\{m\}$. $\mathcal{B}$ sets $g = w$ and $z_m = \ell$. $\mathcal{B}$ sends $((a, b, \mathbb{G}, g), (1, g))$ to $\mathcal{A}$.

- $\mathsf{KeyGen}(1^\lambda)$: Sample the description of a group $(a, b, \mathbb{G}) \leftarrow_\$ \mathsf{GGen}(\lambda)$ of unknown order in the range $[a, b]$ where a, b and $a - b$ are exponential in λ and a random element $g \leftarrow_\$ \mathbb{G}$. Set $\mathcal{V} = [0, a)$. Let the bit-length of b, $|b| = \zeta$. Set $\mathcal{K} = \{0,1\}^*$. Sample the description of a hash function H that maps arbitrary strings to unique primes from the set $\mathsf{Primes}(\zeta + 1) \setminus [b]$. Output $(\mathsf{pp}, C) = ((a, b, \mathbb{G}, g, H), (1, g))$.

- $\mathsf{Insert}(C, (k, v))$: Parse the input C as (C_1, C_2). Let $z = H(k)$. Set

$$\Lambda_k = ((C_1, C_2), (g, 1, 1), 0)\ ^a, \quad C = (C_1^z \cdot C_2^v, C_2^z), \quad \mathsf{upd} = (\mathsf{insert}, (k, v)).$$

  Output $(C, \Lambda_k, \mathsf{upd})$.

- $\mathsf{Update}(C, (k, \delta))$: Parse the input C as (C_1, C_2). Let $z = H(k)$. Set

$$C = \left(C_1^z \cdot C_2^\delta, C_2^z\right), \quad \mathsf{upd} = (\mathsf{update}, (k, \delta)).$$

  Output (C, upd).

- $\mathsf{ProofUpdate}(k, \Lambda_k, \mathsf{upd})$: Parse the inputs Λ_k as $((\Lambda_{k,1}, \Lambda_{k,2}), (\Lambda_{k,3}, \Lambda_{k,4}, \Lambda_{k,5}), u_k)$ and upd as $(\mathsf{upd}_1, (\mathsf{upd}_2, \mathsf{upd}_3))$. Let $z = H(k)$. If $\mathsf{upd}_2 = k$, set

$$\Lambda_k = ((\Lambda_{k,1}, (\Lambda_{k,2})^z), (\Lambda_{k,3}, \Lambda_{k,4}, \Lambda_{k,5}), u_k + 1).$$

  Otherwise, let $\hat{z} = H(\mathsf{upd}_2)$. We assume that $z \neq \hat{z}$. Compute $\alpha, \beta \in \mathbb{Z}$ such that

$$\alpha \cdot z + \beta \cdot \hat{z} = 1.$$

  Compute

$$\gamma = \beta \cdot \Lambda_{k,5} \bmod z.$$

  Compute $\eta \in \mathbb{Z}$ such that

$$\gamma \cdot \hat{z} + \eta \cdot z = \Lambda_{k,5}.$$

  Set

$$\Lambda_k = \left(\left((\Lambda_{k,1})^{\hat{z}} \cdot (\Lambda_{k,2})^{\mathsf{upd}_3}, (\Lambda_{k,2})^{\hat{z}}\right), \left((\Lambda_{k,3})^{\hat{z}}, \Lambda_{k,4} \cdot \Lambda_{k,3}^{\eta}, \gamma\right), u_k\right).$$

  Output Λ_k.

- $\mathsf{Ver}(C, (k, v), \Lambda_k)$: Parse the inputs C as (C_1, C_2) and Λ_k as $((\Lambda_{k,1}, \Lambda_{k,2}), (\Lambda_{k,3}, \Lambda_{k,4}, \Lambda_{k,5}), u_k)$. Let $z = H(k)$. Check whether:
  - $v \in \mathcal{V}$ and $k \in \mathcal{K}$,
  - $u_k \in \mathbb{Z}_{\geq 0}$,
  - $(\Lambda_{k,2})^z = C_2$,
  - $(\Lambda_{k,1})^{z^{u_k+1}} \cdot (\Lambda_{k,2})^v = C_1$,
  - $(\Lambda_{k,3})^{z^{u_k+1}} = C_2$,
  - $(\Lambda_{k,4})^z \cdot (\Lambda_{k,3})^{\Lambda_{k,5}} = g$.

  If all the checks pass, output 1; else, output 0.

[a] It is possible to do away with one of these elements, but we keep them all here for ease of presentation.

Fig. 2. KVaC: Full KVC construction

$\mathcal{B}$ maintains a key-value map $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{V}$ initialized to the empty map and the initial commitment value $C = (1, g)$. Any time $\mathcal{A}$ issues the ith query, k_i, for the computation of H, $\mathcal{B}$ returns z_i and assigns $H(k_i) = z_i$. Any time $\mathcal{A}$ issues queries as in Definition 3, note that $\mathcal{B}$ has all the values it needs to make updates to C and $\mathcal{M}$ as defined in Definition 3 and Figure 2. If $\mathcal{A}$ made more than u updates to any key, $\mathcal{B}$ aborts. If $\mathcal{A}$ aborts at any point in time, $\mathcal{B}$ aborts as well. Assuming no aborts, finally, $\mathcal{A}$ responds with a tuple of the form

1. (k_m, v, Λ) where k_m wasn't inserted by $\mathcal{A}$, or
2. $(k_m, v, v', \Lambda, \Lambda')$ where k_m was inserted by $\mathcal{A}$ and $v \neq v'$.

Case 1: Let $C = (C_1, C_2)$ and $\Lambda = ((\Lambda_1, \Lambda_2), (\cdot, \cdot, \cdot), u_k)$. Since it is the case that $\mathsf{Ver}(C, (k_m, v), \Lambda) = 1$,

$$\Lambda_2^{\ell} = C_2.$$

Without loss of generality, we can assume that $\mathcal{A}$ inserted all keys that it queried the hash function for into the commitment (other than k_m). Let $\mathcal{M} = \{(k_i, v_i)\}_{i \in [q] \setminus \{m\}}$ be the map committed to in C. We have

$$C_2 = g^{\prod_{i \in [q] \setminus \{m\}} H(k_i)^{u+1}}.$$

Therefore

$$\Lambda_2^{\ell} = w^{\prod_{i \in [q] \setminus \{m\}} z_i^{u+1}}.$$

Since $\ell \neq z_i$ for any $i \in [q] \setminus \{m\}$, ℓ is coprime to $\prod_{i \in [q] \setminus \{m\}} z_i^{u+1}$. $\mathcal{B}$ then computes integers θ_1, θ_2 such that

$$\theta_1 \cdot \ell + \theta_2 \cdot \prod_{i \in [q] \setminus \{m\}} z_i^{u+1} = 1.$$

Finally, $\mathcal{B}$ computes

$$u = w^{\theta_1} \Lambda_2^{\theta_2}.$$

Note that

$$\begin{aligned} u^{\ell} &= w^{\theta_1 \ell} \Lambda_2^{\theta_2 \ell} \\ &= w^{\theta_1 \ell} (\Lambda_2^{\ell})^{\theta_2} \\ &= w^{\theta_1 \ell} (w^{\prod_{i \in [q] \setminus \{m\}} z_i^{u+1}})^{\theta_2} \\ &= w^{\theta_1 \cdot \ell + \theta_2 \cdot \prod_{i \in [q] \setminus \{m\}} z_i^{u+1}} \\ &= w. \end{aligned}$$

$\mathcal{B}$ forwards u to CH.

Case 2: Let $\Lambda' = (\Lambda'_1, \Lambda'_2, (\cdot,\cdot,\cdot), u'_k)$. Following the key-binding proof of the increment-only construction (see the full version), we have that with overwhelming probability,

$$u = u_k = u'_k.$$

As in the proof of Lemma 1,

$$\Lambda_2 = \Lambda'_2.$$

Without loss of generality, we can assume that $\mathcal{A}$ inserted all keys that it queried the hash function for into the commitment. Let $\mathcal{M} = \{(k_i, v_i)\}_{i\in[q]}$ be the key-value map committed to in C. Consider the proof Λ_{k_m} corresponding to the key k_m defined by

$$\Lambda_{k_m} = (\Lambda_{k_m,1}, \Lambda_{k_m,2}),$$

where

$$\Lambda_{k_m,1} = g^{\left(\sum_{i\in[q]\setminus\{m\}} v_i \prod_{j\in[q]\setminus\{i,m\}} H(k_j)\right)\cdot \prod_{i\in[q]\setminus\{m\}} H(k_j)^u}$$

and

$$\Lambda_{k_m,2} = g^{\ell^u \cdot \prod_{i\in[q]\setminus\{m\}} H(k_i)^{u+1}}.$$

Clearly, $\mathsf{Ver}(C, (k_m, v_m), \Lambda_{k_m}) = 1$. In particular,

$$\Lambda^{\ell}_{k_m,2} = C_2.$$

Extending our previous argument, we have that

$$\Lambda_2 = \Lambda'_2 = \Lambda_{k_m,2} = g^{\ell^u \cdot \prod_{i\in[q]\setminus\{m\}} H(k_i)^{u+1}}.$$

Since $\mathsf{Ver}(C, (k_m, v), \Lambda) = \mathsf{Ver}(C, (k_m, v'), \Lambda') = 1$, we have

$$(\Lambda_1)^{\ell^{u+1}} \cdot (\Lambda_2)^v = C_1 = (\Lambda'_1)^{\ell^{u+1}} \cdot (\Lambda'_2)^{v'}.$$

This implies that

$$\left(\frac{\Lambda_1}{\Lambda'_1}\right)^{\ell^{u+1}} = \left(g^{\ell^u \cdot \prod_{i\in[q]\setminus\{m\}} H(k_i)^{u+1}}\right)^{v'-v}.$$

Let

$$\xi = \frac{\left(\frac{\Lambda_1}{\Lambda'_1}\right)^{\ell}}{g^{(v'-v)\cdot\prod_{i\in[q]\setminus\{m\}} H(k_i)^{u+1}}} = \frac{\left(\frac{\Lambda_1}{\Lambda'_1}\right)^{\ell}}{\pi^{v'-v}}.$$

We have that $\xi^{\ell^u} = 1$. Since ℓ is prime, the order of ξ in $\mathbb{G}$ must be a power of ℓ. But ℓ is larger than the order of $\mathbb{G}$, which is not possible. Hence $\xi = 1$, that is,

$$\left(\frac{\Lambda_1}{\Lambda'_1}\right)^{\ell} = \pi^{v'-v}.$$

Let

$$\beta = \frac{\Lambda_1}{\Lambda'_1}$$

and

$$v' - v = \delta.$$

Note that $\beta^\ell = \pi^\delta$. This implies that

$$\beta^\ell = w^{\delta \prod_{i \in [q] \setminus \{m\}} z_i^{u+1}}.$$

Since $\mathcal{V} = [0, a)$, $|\delta| < a < b < \ell$, and since ℓ is prime, ℓ is coprime to δ. Also, since $\ell \neq z_i$ for any $i \in [q] \setminus \{m\}$, ℓ is coprime to $\prod_{i \in [q] \setminus \{m\}} z_i^{u+1}$. Hence, ℓ is coprime to $\delta \prod_{i \in [q] \setminus \{m\}} z_i^{u+1}$. $\mathcal{B}$ then computes integers θ_1, θ_2 such that

$$\theta_1 \cdot \ell + \theta_2 \cdot \delta \prod_{i \in [q] \setminus \{m\}} z_i^{u+1} = 1.$$

Finally, $\mathcal{B}$ computes

$$u = w^{\theta_1} \beta^{\theta_2}.$$

Note that

$$\begin{aligned} u^\ell &= w^{\theta_1 \ell} \beta^{\theta_2 \ell} \\ &= w^{\theta_1 \ell} (\beta^\ell)^{\theta_2} \\ &= w^{\theta_1 \ell} (w^{\delta \prod_{i \in [q] \setminus \{m\}} z_i^{u+1}})^{\theta_2} \\ &= w^{\theta_1 \cdot \ell + \theta_2 \cdot \delta \prod_{i \in [q] \setminus \{m\}} z_i^{u+1}} \\ &= w. \end{aligned}$$

$\mathcal{B}$ forwards u to CH. This completes the proof.

Removing the random oracle assumption. As before, we can get rid of the need for a random oracle by leveraging the stronger form of RSA.

Lemma 4. *Suppose there exists a PPT adversary $\mathcal{A}$ in the standard model that satisfies*

$$\mathsf{Adv}^{\mathrm{bind}}_{\mathsf{KVaC},\mathcal{A}}(\lambda) = \epsilon,$$

where KVaC *is the key-value commitment scheme defined in Figure 2. Then, there exists a PPT adversary $\mathcal{B}$ such that*

$$\mathsf{Adv}^{\mathrm{SRSA}}_{\mathcal{B}}(\lambda) \geq \epsilon - \mathsf{negl}(\lambda).$$

A proof of this lemma can be found in the full version of the paper.

Theorem 1. *The construction* KVaC *in Figure 2 is a key-value commitment scheme for arbitrary keys and an exponentially large value space where the commitment is of constant size, the proof corresponding to any key is of constant size, and each operation requires only a constant number of hash computations, exponentiations or multiplications. The construction is key-binding (Def. 4) based on (i) the generalized RSA assumption in the random oracle model (Lemma 3), or (ii) the strong RSA assumption in the standard model (Lemma 4).*

5.2 Performing "Double" Exponentiations

In the verification procedure, one must compute z^{u_k+1}. While this is only a polynomial time computation (as u_k can only be polynomially large), we might want to avoid "double" exponentiations such as computing $(\Lambda_{k,1})^{z^{u_k+1}}$. This can be done by accumulating these values as they are computed per each update using standard accumulators such as those in the work of Boneh et al. [4] or the insert-only key-value commitment scheme construction in Figure 1. This would only add one more group element to the commitment and a constant number (at most three) of hash computations, group exponentiations or multiplications to the operations.

5.3 Vector Commitments

A key-value commitment directly gives us a vector commitment. We can use the keys as indices. Using our construction for key-value commitments, the newly constructed vector commitment enjoys several benefits in comparison to prior constructions. For instance, additive updates on values can be performed by any party and corresponding updates to proofs can be made extremely efficiently. The commitment and the proofs are constant-sized and verification only involves a constant number of operations. An added benefit of constructing a vector commitment in this way is that the length of the vector being committed need not be known ahead of time, or, in fact, at any point in time. If one, however, did wish for a vector commitment with restrictions on the length of the vector that can be committed, or one which only allowed for appending elements, it can be trivially achieved through minor (black-box) modifications to our construction.

6 Aggregating Proofs

In this section, we describe how proofs corresponding to multiple keys can be combined, or "aggregated", into a single proof and "batch" verified in one shot. The first observation is that the proofs corresponding to keys in the increment-only construction described in the full version can be combined in a straightforward manner using the Shamir Trick described in the work of Boneh et al. [4], although this would only yield one-hop aggregation. We will discuss here how to combine proofs corresponding to multiple keys in the insert-only construction described in Figure 1. In fact, the insert-only construction supports unbounded aggregation and disaggregation in the sense of Campanelli et al. [9]. Putting these techniques together, one can combine proofs corresponding to multiple keys for the full construction described in Figure 2, but combined proofs cannot be combined further.

Suppose we had two proofs $\Lambda = (\Lambda_1, \Lambda_2)$ and $\Lambda' = (\Lambda'_1, \Lambda'_2)$ corresponding to two keys k and k' with values v and v' (with respect to the insert-only construction described in Figure 1), and let the current state of the key-value

commitment be $C = (C_1, C_2)$. Suppose $z = H(k)$ and $z' = H(k')$. Recall that if Λ and Λ' are valid proofs, it must be the case that

$$\Lambda_2^z = \Lambda_2'^{z'} = C_2$$

and

$$\Lambda_1^z \cdot \Lambda_2^v = \Lambda_1'^{z'} \cdot \Lambda_2'^{v'} = C_1.$$

In order to combine these two proofs, we would have to come up an "aggregated" proof $\Lambda'' = (\Lambda_1'', \Lambda_2'')$ whose "batch" verification would look like

$$\Lambda_2''^{zz'} = C_2$$

and

$$\Lambda_1''^{zz'} \cdot \Lambda_2''^{vz'+v'z} = C_1.$$

In other words, Λ'' represents a key-value commitment to the key-value pairs in C other than (k, v) and (k', v') which is realized by the fact that inserting (k, v) and (k', v') into Λ'' generates C.

We combine Λ and Λ' as follows. Since z and z' are distinct primes, compute $\alpha, \beta \in \mathbb{Z}$ such that

$$\alpha \cdot z + \beta \cdot z' = 1.$$

Set

$$\Lambda_2'' = \Lambda_2^\beta \cdot \Lambda_2'^\alpha$$

and

$$\Lambda_1'' = \frac{\Lambda_1^\beta \cdot \Lambda_1'^\alpha}{\Lambda_2''^{v\alpha+v'\beta}}.$$

Observe that

$$\Lambda_2''^{zz'} = \Lambda_2^{\beta zz'} \cdot \Lambda_2'^{\alpha zz'} = C_2^{\alpha z+\beta z'} = C_2$$

and

$$\begin{aligned}
\Lambda_1''^{zz'} \cdot \Lambda_2''^{vz'+v'z} &= \frac{\Lambda_1^{\beta zz'} \cdot \Lambda_1'^{\alpha zz'}}{\Lambda_2''^{v\alpha zz'+v'\beta zz'}} \cdot \Lambda_2^{\beta(vz'+v'z)} \cdot \Lambda_2'^{\alpha(vz'+v'z)} \\
&= \frac{(\Lambda_1^z \cdot \Lambda_2^v)^{\beta z'} \cdot \left(\Lambda_1'^{z'} \cdot \Lambda_2'^{v'}\right)^{\alpha z} \cdot (\Lambda_2^z)^{\beta v'} \cdot \left(\Lambda_2'^{z'}\right)^{\alpha v}}{C_2^{v\alpha+v'\beta}} \\
&= C_1^{\alpha z+\beta z'} \\
&= C_1.
\end{aligned}$$

Notice that the aggregation procedure involves only two hash computations, five exponentiations and three multiplications. The size of the combined proof is the same as the sizes of each of the separate proofs and, by construction, the combined proof can be verified in one shot. Key-binding for the combined proof can be shown in exactly the same as was done for each of the separate proofs.

We can easily extend this procedure to combine more than two proofs. In particular, an aggregated proof can be combined with a regular proof or two aggregated proofs can be combined with each other.

Acknowledgement

We thank Asiacrypt 2020 reviewers for providing valuable feedback on the paper. We thank Benedikt Bünz for suggesting several improvements to the paper.

References

1. 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 2017, Paris, France, April 26-28, 2017. pp. 301–315. IEEE (2017)
2. Benaloh, J.C., de Mare, M.: One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In: Helleseth, T. (ed.) EUROCRYPT'93. LNCS, vol. 765, pp. 274–285. Springer, Heidelberg (May 1994). https://doi.org/10.1007/3-540-48285-7_24
3. Bitcoin. https://bitcoin.org/
4. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part I. LNCS, vol. 11692, pp. 561–586. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26948-7_20
5. Buchmann, J., Hamdy, S.: A survey on iq cryptography. In: In Proceedings of Public Key Cryptography and Computational Number Theory. pp. 1–15 (2001)
6. Camacho, P., Hevia, A.: On the impossibility of batch update for cryptographic accumulators. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 178–188. Springer, Heidelberg (Aug 2010)
7. Camacho, P., Hevia, A., Kiwi, M.A., Opazo, R.: Strong accumulators from collision-resistant hashing. In: Wu, T.C., Lei, C.L., Rijmen, V., Lee, D.T. (eds.) ISC 2008. LNCS, vol. 5222, pp. 471–486. Springer, Heidelberg (Sep 2008)
8. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 61–76. Springer, Heidelberg (Aug 2002). https://doi.org/10.1007/3-540-45708-9_5
9. Campanelli, M., Fiore, D., Greco, N., Kolonelos, D., Nizzardo, L.: Vector commitment techniques and applications to verifiable decentralized storage. Cryptology ePrint Archive, Report 2020/149 (2020), https://eprint.iacr.org/2020/149
10. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 55–72. Springer, Heidelberg (Feb / Mar 2013). https://doi.org/10.1007/978-3-642-36362-7_5
11. Chepurnoy, A., Papamanthou, C., Zhang, Y.: Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968 (2018), https://eprint.iacr.org/2018/968
12. Dryja, T.: Utreexo: A dynamic hash-based accumulator optimized for the bitcoin UTXO set. Cryptology ePrint Archive, Report 2019/611 (2019), https://eprint.iacr.org/2019/611
13. EOS.io — Blockchain software architecture. https://www.eos.io
14. Etherchain – Evolution of the total number of Ethereum accounts. https://www.etherchain.org/charts/totalAccounts
15. Ethereum. https://www.ethereum.org/
16. Etherscan. https://etherscan.io/

17. Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: Aggregating proofs for multiple vector commitments. Cryptology ePrint Archive, Report 2020/419 (2020), https://eprint.iacr.org/2020/419
18. Hamdy, S., Möller, B.: Security of cryptosystems based on class groups of imaginary quadratic orders. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 234–247. Springer, Heidelberg (Dec 2000). https://doi.org/10.1007/3-540-44448-3_18
19. Krupp, J., Schröder, D., Simkin, M., Fiore, D., Ateniese, G., Nürnberger, S.: Nearly optimal verifiable data streaming. In: Cheng, C.M., Chung, K.M., Persiano, G., Yang, B.Y. (eds.) PKC 2016, Part I. LNCS, vol. 9614, pp. 417–445. Springer, Heidelberg (Mar 2016). https://doi.org/10.1007/978-3-662-49384-7_16
20. Lai, R.W.F., Malavolta, G.: Subvector commitments with application to succinct arguments. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part I. LNCS, vol. 11692, pp. 530–560. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26948-7_19
21. Li, J., Li, N., Xue, R.: Universal accumulators with efficient nonmembership proofs. In: Katz, J., Yung, M. (eds.) ACNS 07. LNCS, vol. 4521, pp. 253–269. Springer, Heidelberg (Jun 2007). https://doi.org/10.1007/978-3-540-72738-5_17
22. Libert, B., Ramanna, S.C., Yung, M.: Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In: Chatzigiannakis, I., Mitzenmacher, M., Rabani, Y., Sangiorgi, D. (eds.) ICALP 2016. LIPIcs, vol. 55, pp. 30:1–30:14. Schloss Dagstuhl (Jul 2016). https://doi.org/10.4230/LIPIcs.ICALP.2016.30
23. Libra. https://libra.org/
24. Mazieres, D.: The stellar consensus protocol: A federated model for internet-level consensus. Stellar Development Foundation (2015)
25. Nguyen, L.: Accumulators from bilinear pairings and applications. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 275–292. Springer, Heidelberg (Feb 2005). https://doi.org/10.1007/978-3-540-30574-3_19
26. Ripple - One Frictionless Experience To Send Money Globally. https://www.ripple.com
27. Peter Todd: Making UTXO Set Growth Irrelevant With Low-Latency Delayed TXO Commitments. https://petertodd.org/2016/delayed-txo-commitments
28. Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable subvector commitments for stateless cryptocurrencies. Cryptology ePrint Archive, Report 2020/527 (2020), https://eprint.iacr.org/2020/527