Universal Vector Commitments

Ojaswi Acharya[1], Foteini Baldimtsi[2], Samuel Dov Gordon[2], Daniel McVicker[2], and Aayush Yadav[2]

[1] University of Massachusetts Amherst
oacharya@umass.edu
[2] George Mason University
{foteini, gordon, dmcvicke, ayadav5}@gmu.edu

Abstract. We propose a new notion of vector commitment schemes with proofs of (non-)membership that we call universal vector commitments. We show how to build them directly from (i) Merkle commitments, and (ii) a universal accumulator and a plain vector commitment scheme. We also present a generic construction for universal accumulators over large domains from any vector commitment scheme, using cuckoo hashing. Leveraging the aforementioned generic constructions, we show that universal vector commitment schemes are implied by plain vector commitments and cuckoo hashing.

1 Introduction

The problem of set membership (and non-membership) is ubiquitous in modern cryptography. Applications such as the revocation of authentication credentials, verifiable databases and cloud storage rely on efficient proofs for set (non-)membership [CL02, BCD+17, SMP23, ZKP17, TBP+19, BdM94, BP97]. More recently, set (non-)membership has gained increased attention because of its role in blockchain systems [BBF19]. One could imagine, for instance, a system where users are able to demonstrate ownership of digital assets by proving that they appear in some large public set.

Two of the most prominent mechanisms that support set (non-)membership proofs are cryptographic accumulators and vector commitments. Briefly, a cryptographic accumulator [BdM94] is a compact representation, acc, of a set of elements S over some domain. It allows a *prover* to generate a short proof of membership (resp. non-membership) for any element that is accumulated (resp. not accumulated) in acc. A *verifier* can efficiently verify such proofs without accessing the entire set, using only the current digest acc. An accumulator is *dynamic* if it supports efficient updates by allowing changes to acc through insertion (or deletion) of elements. Furthermore, an accumulator that supports both proofs of membership and non-membership is a *universal* accumulator (UA).

A vector commitment scheme (VC) [CF13] allows one to succinctly commit to an *ordered* sequence of n values $\mathbf{x} = [x_1 \ \cdots \ x_n]$ in a way that one can later prove that the value x_i is the ith committed element. For VCs, a generalized notion of key-value commitments (KVaC) has also been suggested [AR20]. A KVaC commits a map of key-value pairs with unique keys, where each key is associated with only one value.

The case for universal vector commitments. Vector commitments have so far been defined to only support proofs of membership (i.e., a value x_i is in the ith position of the commitment). However, in certain applications, and usually for accountability reasons, one might also be interested in proving that a value is *not* in the commitment. Supporting non-membership proofs is thus an essential feature for accountable VCs, that is missing from existing formalizations.

Consider for example a supply chain management system where a VC is used to store various states of a product (a case where order matters), but non-membership proofs may be needed to demonstrate that a certain product or component is not part of the supply chain at a given point in time, which can be essential for detecting counterfeit goods or unauthorized substitutions.

Other applications of VCs with a non-membership functionality include storage of blockchain transactions where a user wants to prove *exclusion* of a token from a transaction block [Kon19], or distributed file-storage where the usual *proofs of retrievability* of a file [JK07, Fis19] could be extended to also include proofs that a certain illicit file is *not* being stored by the server.

Overview of our contributions

Defining universal vector commitments. In Section 3 we present the first formal definition of a *universal vector commitments* (UVC) by extending the notion of vector commitment schemes to include proofs of non-membership. In general, such proofs of non-membership would allow one to prove that a value is not at any position of the committed vector. This is analogous to a universal accumulator where one can generate proofs for elements that are not accumulated. The importance of formally defining UVCs and their security properties is further emphasized by the fact that although there exist constructions that realize the definition of UVC [Kon19, GV20], a formal description of its security requirements is lacking.

Concrete constructions for universal vector commitments. A straightforward way of building a UVC is to consider a Merkle tree with number of leaves equal to the domain size of the underlying set, with non-members represented as a leaf node with some generic null value. Indeed, this approach was made somewhat practically efficient using a Sparse Merkle tree (SMT) [DPP16] for the specific case of *proofs of exclusion* (of tokens from a transaction block) [Kon19]. In particular, the tokens excluded from the transaction are represented as (the hash of) null nodes in a Merkle tree, and the corresponding proof of exclusion as the Merkle path to the leaf. However, despite the use of SMT, the overall storage requirement of this approach grows in proportion to the domain size.

Towards building secure UVCs, we present two new constructions in Sections 4 and 5. Our first, more practical, construction based on Merkle commitments is a translation of the 'key accumulator' given by Goyal and Vusirikala [GV20]. Importantly, it offers compact proofs of inclusion/exclusion, and is asymptotically space-optimal (among Merkle tree-based approaches). Our second construction

describes a generic way to build UVCs from a (plain) VC and a UA, demonstrating an interesting theoretical connection between these primitives (see Figure 1).

A generic construction for universal accumulators. Given the conceptual similarities of VCs, UAs and KVaCs, one wonders about the relationship between these primitives. Interestingly, it was shown by Boneh et al. [BBF19] that VCs can be built from UAs. In a recent concurrent work, Fiore et al. [FKP23] gave a generic way to obtain UAs (and KVaCs) using VCs and Cuckoo Hashing (CH), thus closing the circle on a longstanding question. In this work, we also present a generic construction in Section 6 for (updateable) UAs from any VC, using Cuckoo Hashing. Our generic construction thus gives one of the first UAs supporting large domains (as opposed to the one from [CF13]). Looking ahead, the two techniques are quite similar — although, [FKP23] take the opposite approach of arriving at UAs from the KVaC construction, whereas we build both primitives directly. More generally we observe that our VC + CH abstraction gives many important primitives that we summarize in Figure 1. A particularly interesting observation here is that UVCs are implied by the VC + CH abstraction; this follows as a consequence of our two aforementioned generic constructions.

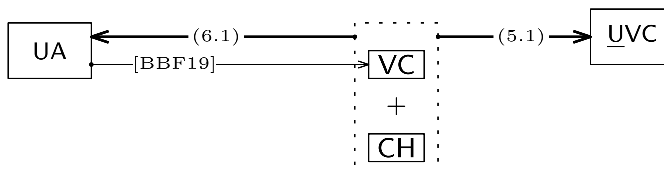

Fig. 1. The relationship between VCs, UAs and KVaCs.

Further, by appropriately instantiating our generic construction for UAs, we obtain (i) a UA from the SIS problem that is comparably efficient with existing lattice-based constructions and (ii) the first UA based on the standard RSA assumption.

Lastly, we outline even more applications of the VC + CH abstraction towards building KVaCs, and verifiable Registration-based Encryption [GV20][3] in Appendix A.

2 Preliminaries

We denote the set of all positive integers up to k as $[k] := \{1, \ldots, k\}$ and the set of all non-negative integers up to k as $[0, k] := \{0\} \cup [k]$. We use lowercase bold-face letters to denote vectors and uppercase bold-face letters for matrices. For a vector $\mathbf{x}$, x_i denotes its ith element. We summarize all important notation in Table 1.

[3] Due to space constraints, and the narrow scope of this particular observation, we limit the associated discussions only to the appendix.

Symbol	Meaning
λ	security parameter
n	input dimension of vector commitment
S	number of accumulated elements in an accumulator
S'	maximum supported number of accumulated elements
D	size of domain of individual input elements
m	number of hashes in cuckoo table
ℓ	image size of cuckoo table hashes

Table 1. Our notation.

2.1 Vector Commitments

We now review the definitions of vector commitments as introduced in [CF13].

Definition 1 (Vector Commitment Scheme). *A vector commitment scheme over a message space* $\mathcal{M}$ *is a tuple of algorithms* $\Pi_{\mathsf{VC}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *defined as follows:*

- $\mathsf{Setup}(1^\lambda, 1^n) \rightarrow \mathsf{pp}$. *Takes as input the security parameter* 1^λ *and the size of the input vector* 1^n *where* $n = poly(\lambda)$*, and outputs the public parameters* pp*. All algorithms below take* pp *as input, but we omit it for notational clarity.*
- $\mathsf{Commit}(\mathbf{x}) \rightarrow (\mathsf{com}, \mathsf{aux})$. *Takes as input a vector* $\mathbf{x} \in \mathcal{M}^n$*, and outputs a commitment* com *and possibly some auxiliary information* aux*.*
- $\mathsf{Open}(\mathsf{com}, x, i, \mathsf{aux}) \rightarrow \pi$. *Takes as input a commitment* com*, an element* $x \in \mathcal{M}$*, an index* $i \in [n]$ *and auxiliary information* aux*, and outputs a proof* π *for* x*.*
- $\mathsf{Verify}(\mathsf{com}, x, i, \pi) \rightarrow \{0, 1\}$. *Takes as input a commitment value* com*, an element* $x \in \mathcal{M}$*, an index* $i \in [n]$ *and a proof* π*, and outputs* 0 *or* 1*.*

A vector commitment scheme must satisfy the following properties:

- **Correctness:** For security parameter λ and any input vector $\mathbf{x} = \{x_1, x_2, \ldots, x_n\} \in \mathcal{M}^n,\ \forall n \geq 1$, a vector commitment scheme satisfies correctness if:

$$\Pr\left[\mathsf{Verify}(\mathsf{com}, x, i, \pi) \ :\ \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, 1^n) \\ \mathsf{com} \leftarrow \mathsf{Commit}(\mathbf{x}) \\ \forall i \in [n],\ \pi \leftarrow \mathsf{Open}(\mathsf{com}, i, x) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

- **Position binding:** For security parameter λ, for all PPT adversaries $\mathcal{A}$, a vector commitment scheme is position binding if:

$$\Pr\left[\begin{array}{l} \mathsf{Verify}(\mathsf{com}, x, i, \pi) = 1 \\ \wedge\ \mathsf{Verify}(\mathsf{com}, y, i, \pi') = 1 \end{array} \ :\ \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, 1^n) \\ \mathsf{com}, x, y, i, \pi, \pi' \leftarrow \mathcal{A}(\mathsf{pp}) \\ x \neq y \end{array}\right] \leq \mathsf{negl}(\lambda).$$

2.2 Universal Accumulators

An accumulator may be trapdoor-based or trapdoorless. A *trapdoor accumulator* is characterised by the presence of a trusted authority known as an *accumulator*

manager who holds some secret trapdoor information (such as a secret key) that allows them to perform add and delete operations on the accumulated set. Reyzin et al. [RY16] additionally define *witness-holders* and third-party *users* of the protocol. *Trapdoorless accumulators*, on the other hand, do not require a secret trapdoor, thus dismissing the need for a manager. Instead, witness-holders acting as provers in the protocol must keep track of the current accumulated set and some other auxiliary information to be able to perform add and delete operations. In our definition below, we primarily focus on the trapdoorless setting.

We now review the definitions for cryptographic accumulators, as introduced by [BdM94] along with the universal extension of [LLX07]. Our definition considers an *updateable* accumulator which allows for insertion and deletion without the efficiency requirement of dynamic accumulators.

Definition 2 (Updateable Universal Accumulator). *An updateable universal accumulator over some domain $\mathcal{X}$ is a tuple of algorithms $\Pi_{\mathsf{UA}} = (\mathsf{Gen}, \mathsf{Add}, \mathsf{Delete}, \mathsf{MemWitCreate}, \mathsf{NonMemWitCreate}, \mathsf{VerifyMem}, \mathsf{VerifyNonMem})$ defined as follows:*

- $\mathsf{Gen}(1^\lambda) \to (\mathsf{state}_0, \mathsf{acc}_0)$. *Takes input the security parameter, and outputs an initial state $\mathsf{state}_0 = (S_0, \mathsf{aux})$ where $S_0 = \varnothing$, and acc_0 is the accumulator's initial value.*
- $\mathsf{Add}(\mathsf{acc}_t, \mathsf{state}_t, x) \to \mathsf{state}_{t+1}, \mathsf{acc}_{t+1}$. *Takes as input the current accumulator value acc_t, state $\mathsf{state}_t = (S_t, \mathsf{aux})$ where S_t is the set of currently accumulated elements, and an element $x \in \mathcal{X}$ to be added to the accumulator. Outputs the new state $\mathsf{state}_{t+1} = (S_{t+1}, \mathsf{aux})$ and the updated accumulator value acc_{t+1}.*
- $\mathsf{Delete}(\mathsf{acc}_t, \mathsf{state}_t, x) \to \mathsf{state}_{t+1}, \mathsf{acc}_{t+1}$. *Takes as input the current accumulator value acc_t, state $\mathsf{state}_t = (S_t, \mathsf{aux})$ where S_t is the set of currently accumulated elements, and an element $x \in \mathcal{X}$ to be deleted from the accumulator. Outputs the new state $\mathsf{state}_{t+1} = (S_{t+1}, \mathsf{aux})$ and the updated accumulator value acc_{t+1}.*
- $\mathsf{MemWitCreate}(\mathsf{acc}_t, \mathsf{state}_t, x) \to w_t^x$. *Takes as input the current accumulator value acc_t, state $\mathsf{state}_t = (S_t, \mathsf{aux})$ where S_t is the set of currently accumulated elements, and an element $x \in \mathcal{X}$. Outputs a membership witness w_t^x for x.*
- $\mathsf{VerifyMem}(\mathsf{acc}_t, x, w_t^x) \to \{0, 1\}$. *Takes as input the current accumulator value acc_t, an element $x \in \mathcal{X}$ and its membership witness w_t^x, and outputs 0 or 1.*
- $\mathsf{NonMemWitCreate}(\mathsf{acc}_t, \mathsf{state}_t, x) \to \overline{w}_t^x$. *Takes as input the current accumulator value acc_t, state $\mathsf{state}_t = (S_t, \mathsf{aux})$ where S_t is the set of currently accumulated elements, and an element $x \in \mathcal{X}$. Outputs a non-membership witness w_t^x for x.*
- $\mathsf{VerifyNonMem}(\mathsf{acc}_t, x, \overline{w}_t^x) \to \{0, 1\}$. *Takes as input the current accumulator value acc_t, an element $x \in \mathcal{X}$ and its non-membership witness w_t^x, and outputs 0 or 1.*

The properties satisfied by an updateable universal accumulator are as follows:

- **Correctness (for positive accumulator):** For security parameter λ, and values $S_t = \{y_1, y_2, \dots, y_{x-1}, x, y_{x+1}, \dots, y_t\} \subseteq \mathcal{X}$ for any $t \geq 1$, correctness of the membership witness requires that

$$\Pr\left[\begin{array}{l}\mathsf{VerifyMem}\left(\mathsf{acc}_t, x, w_t^x\right) = 1 : \\ \hfill (\mathsf{state}_0, \mathsf{acc}_0) \leftarrow \mathsf{Gen}(1^\lambda) \\ \hfill \forall y_i \in S,\ (\mathsf{state}_i, \mathsf{acc}_i) \leftarrow \mathsf{Add}(\mathsf{acc}_{i-1}, \mathsf{state}_{i-1}, y_i) \\ \hfill w_t^x \leftarrow \mathsf{MemWitCreate}(\mathsf{acc}_t, \mathsf{state}_t, x)\end{array}\right] = 1$$

- **Correctness (for negative accumulator):** For security parameter λ, and values $\{y_1, y_2, \dots, y_t\} \subseteq \mathcal{X}$ for any $t \geq 1$, correctness of the non-membership witness requires that for any $x \in \mathcal{X}$

$$\Pr\left[\begin{array}{l}\mathsf{VerifyNonMem}\left(\mathsf{acc}_t, x, \overline{w}_t^x\right) = 1 : \\ \hfill (\mathsf{pp}, \mathsf{state}_0, \mathsf{acc}_0) \leftarrow \mathsf{Gen}(1^\lambda) \\ \hfill \forall i \in [t-1],\ (\mathsf{state}_i, \mathsf{acc}_i) \leftarrow \mathsf{Add}(\mathsf{acc}_{i-1}, \mathsf{state}_{i-1}, y_i) \\ \hfill x \in S_{t-1} \Rightarrow (\mathsf{state}_t, \mathsf{acc}_t) \leftarrow \mathsf{Delete}(\mathsf{acc}_{t-1}\mathsf{state}_{t-1}, x) \\ \hfill x \notin S_{t-1} \wedge y_t \neq x \Rightarrow (\mathsf{state}_t, \mathsf{acc}_t) \leftarrow \mathsf{Add}(\mathsf{acc}_{t-1}, \mathsf{state}_{t-1}, y_t) \\ \hfill \overline{w}_t^x \leftarrow \mathsf{NonMemWitCreate}(\mathsf{acc}_t, \mathsf{state}_t, x)\end{array}\right] = 1$$

A universal accumulator is correct if it satisfies correctness for both positive and negative accumulators. In terms of security, an accumulator should satisfy the soundness property of set binding.

- **(Strong) Set binding:** For security parameter λ, for all PPT adversaries $\mathcal{A}$ the set-binding property with respect to membership requires that for every $x \in \mathcal{X}$

$$\Pr\left[\begin{array}{l}\mathsf{VerifyMem}(\mathsf{acc}_t, x, w_t^x) = 1 \wedge \\ \mathsf{VerifyNonMem}(\mathsf{acc}_t, x, \overline{w}_t^x) = 1 : \\ \hfill (\mathsf{pp}, \mathsf{state}_0, \mathsf{acc}_0) \leftarrow \mathsf{Gen}(1^\lambda) \\ \hfill (x, \mathsf{acc}_t, w_t^x, \overline{w}_t^x) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{state}_0, \mathsf{acc}_0)\end{array}\right] \leq \mathsf{negl}(\lambda),$$

Both notions have a natural intuitive appeal. Correctness requires that for every element in (resp. not in) the accumulator, an honest prover can provide a correct witness of membership (resp. non-membership). Soundness says that, with all but negligible probability, a dishonest prover must not be able to simultaneously provide a membership and a non-membership witness for the same value.

For comparison, we also show the more standard soundness assumption used in prior constructions of universal accumulators, which we refer to as *weak set binding*. The main difference is that weak binding restricts the adversary to interacting with the accumulator via oracles, hence we may assume that the accumulator is constructed honestly according to the protocol.

(Weak) Set binding: For security parameter λ and accumulated set A_t, for all PPT adversaries $\mathcal{A}$ the set-binding property with respect to membership requires that for every $x \in \mathcal{X}$

$$\Pr\left[\begin{array}{l}(\mathsf{VerifyMem}(\mathsf{acc}_t, x, w_t^x) = 1 \ \wedge x \notin S_t) \ \vee \\ (\mathsf{VerifyNonMem}(\mathsf{acc}_t, x, w_t^x) = 1 \wedge x \in S_t) : \\ \hfill (\mathsf{pp}, \mathsf{state}_0, \mathsf{acc}_0) \leftarrow \mathsf{Gen}(1^\lambda) \\ \hfill (x, S_t, w_t^x) \leftarrow \mathcal{A}^{\mathsf{Add},\mathsf{Delete}}(\mathsf{pp}, \mathsf{state}_0, \mathsf{acc}_0)\end{array}\right] \leq \mathsf{negl}(\lambda),$$

2.3 Cuckoo hashing

Let $h_i : \mathcal{U} \rightarrow \{1+(i-1)\cdot\ell, \ldots, \ell+(i-1)\cdot\ell\}$ for $i \in [m]$ be hash functions associated with table $\mathbf{T}$, such that $|\mathbf{T}| = m\ell$.

- $\mathsf{insert}(x, \mathbf{T})$
  1. If $\mathsf{lookup}(x) \neq \perp$, do nothing and return $\mathbf{T}$.
  2. For $i = 1$ to m do:
     - If $\mathbf{T}[h_i(x)] = \varnothing$, set $\mathbf{T}[h_i(x)] \leftarrow x$ and return $\mathbf{T}$.
     - Otherwise, swap $x \leftrightarrow \mathbf{T}[h_i(x)]$.
  3. If $x \neq \varnothing$, call $\mathsf{insert}(x, \mathbf{T})$.
  4. Return $\mathbf{T}$.
- $\mathsf{delete}(x, \mathbf{T})$
  1. For $i \in [m]$, set $\mathbf{T}[h_i(x)] \leftarrow \varnothing$ if $\mathbf{T}[h_i(x)] = x$.
  2. Return $\mathbf{T}$.
- $\mathsf{lookup}(x, \mathbf{T})$
  1. Set $\mathsf{pos} \leftarrow \perp$.
  2. For $i \in [m]$, set $\mathsf{pos} \leftarrow h_i(x)$ if $\mathbf{T}[h_i(x)] = x$.
  3. Return pos.

Fig. 2. Cuckoo hashing

Cuckoo hashing [RP04] is a type of open-addressing technique that gives a dictionary with worst-case constant lookup and deletion, and expected amortized constant-time insertion operations. The main idea is to maintain multiple hash-tables, each keyed by a different hash function. A value is inserted by keying into the first table and bumping any colliding value into its keyed position in the next table. This process must be repeated for every collision, until the values stabilize. As a value can only be one of m positions, it follows that both deletion and lookup operations can be performed in constant time. Some additional care is necessary with insertion as the process can fail to stabilize. This runaway condition occurs when a particular value revisits its original position at the beginning of the current process. The resolution to this issue is to simply stash the orphaned value. This gives an expected amortized constant run-time for insertion. We now formally describe this construction, and direct the reader to the original paper [RP04] for a thorough analysis.

A cuckoo hash comprises of a hash table, $\mathbf{T} \in \mathcal{U}^{m\ell}$. With each chunk of length ℓ, there is an associated hash function $h_i : \mathcal{U} \to \{1 + (i-1)\cdot\ell, \ldots, \ell + (i-1)\cdot\ell\}$. For any key $x \in \mathcal{X} \subseteq \mathcal{U}$ we have that x is stored in at most one position $h_i(x)$.

Remark 1. The recent work of Yeo [Yeo23] proposes a more direct cryptographic treatment of cuckoo hashing with the goal of achieving negligible construction failure probability — ie., the probability that a set of n elements can not be added into the hash table according to a randomly sampled hash function. While we do not formally state Yeo's result here, we find that it is readily applicable to our work.

3 Universal Vector Commitments

The definition of an universal vector commitment scheme, in addition to the usual algorithms, consists of algorithms to prove and verify non-membership. Specifically, the $\mathsf{ProveNonMembership}$ algorithm takes in a commitment value com and an element x and proves that x does not belong to any component of the vector $\mathbf{x}$ that corresponds to the commitment com. Formally, we have the following definition:

Definition 3. *We say a vector commitment scheme is* universal *if it includes additional algorithms* $\mathsf{ProveNonMembership}, \mathsf{VerifyNonMembership}$ *described below:*

- $\mathsf{ProveNonMembership}(\mathsf{com}, x, \mathsf{aux}) \to \overline{\pi}_x$. *Takes as input a commitment value* com*, an element* $x \in \mathcal{M}$*, and auxiliary information* aux *and outputs a proof* $\overline{\pi}_x$ *that* x *is not an element of vector* $\mathbf{x}$ *whose commitment is* com.
- $\mathsf{VerifyNonMembership}(\mathsf{com}, x, \overline{\pi}_x) \to \{0,1\}$. *Takes as input a commitment value* com*, an element* $x \in \mathcal{M}$*, and a non-membership proof* $\overline{\pi}_x$ *and outputs 1 if* $\overline{\pi}_x$ *is a valid proof for the fact that* x *is not an element of vector* $\mathbf{x}$ *whose commitment is* com*, and 0 otherwise.*

The algorithms above have the following properties:

- **Correctness (for non-membership):** For security parameter λ, any input vector $\mathbf{x} = \{x_1, x_2, \ldots, x_n\} \in \mathcal{M}^n$, and an $x \in \mathcal{M}$ but $x \notin \{x_1, \ldots, x_n\}$, a vector commitment scheme with non-membership satisfies non-membership correctness if:

$$\Pr\left[\begin{array}{r} \mathsf{VerifyNonMembership}(\mathsf{com}, x, \overline{\pi}_x) = 1 : \qquad\qquad \\ \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, 1^n) \\ \mathsf{com} \leftarrow \mathsf{Commit}(\mathbf{x}) \\ \overline{\pi}_x \leftarrow \mathsf{ProveNonMembership}(\mathsf{com}, x, \mathsf{aux}) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

- **Soundness (element binding):** For security parameter λ, for all $x \in \mathcal{M}$, and for all PPT adversaries $\mathcal{A}$, a vector commitment scheme has element binding if:

$$\Pr\left[\begin{array}{l}\mathsf{Verify}(\mathsf{com}, i, x, \pi) \wedge \\ \mathsf{VerifyNonMembership}(\mathsf{com}, x, \overline{\pi}) = 1 : \\ \hfill \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \hfill \mathsf{com}, x, i, \pi, \overline{\pi} \leftarrow \mathcal{A}(\mathsf{pp})\end{array}\right] \leq \mathsf{negl}(\lambda).$$

4 Universal Vector Commitments from Merkle Trees

We now describe our Merkle tree based construction with non-membership proofs over a totally ordered message space $\mathcal{M}$. This construction closely follows the "key accumulator" of [GV20], re-written using the UVC syntax. Our main observation here is that this key accumulator quite readily gives an efficient vector commitment with non-membership. As previously stated, this construction is space-optimal among Merkle tree-based approaches since it does not require leaves with null values for proving exclusion. Instead, non-membership of a value x is demonstrated by proving that two committed values x_{lo}, x_{hi} such that $x_{\text{lo}} < x < x_{\text{hi}}$ are adjacent in the tree[4]. Thus, this construction gives a viable alternative for size *and space* efficient proofs of exclusion using Merkle commitments.

4.1 Construction

Let $\mathsf{H} : \mathcal{D} \rightarrow \{0,1\}^\lambda$ be a CRHF over some domain D. We define a vector commitment scheme with non-membership $\Pi^{\mathsf{MT}}_{\mathsf{VCNM}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify}, \mathsf{ProveNonMembership}, \mathsf{VerifyNonMembership})$ over message space $\mathcal{M}$ as follows:

- $\mathsf{Commit}\left(\mathbf{x} = \left[x_1 \cdots x_n\right]\right) \rightarrow \mathsf{com}$. It creates a Merkle commitment to $\mathbf{x}$ such that each leaf node, $\mathsf{node}_i \coloneqq (0||0^\lambda||x_i||0^\lambda)$ and each interior node, $\mathsf{node}_{l,r} \coloneqq (1||\mathsf{H}(\mathsf{node}_l)||\Gamma||\mathsf{H}(\mathsf{node}_r))$ where node_l (resp. node_r) is the node to the left (resp. right) of $\mathsf{node}_{l,r}$ and $\Gamma \in \mathcal{M}$ is the greatest value contained in (the leaf of) its left subtree[5]. It returns the root node $\mathsf{root} =: \mathsf{com}$ as the commitment.
- $\mathsf{Open}(\mathsf{com}, i, x_i) \rightarrow \pi$. For $j \in \{i-1, i, i+1\}$, it parses the jth leaf node, node_j as $(0||0^\lambda||x'_j||0^\lambda)$. If $x'_i \neq x_i$, it outputs $\perp$ and continues otherwise. It then outputs $\pi \coloneqq (\mathsf{path}_{i-1}, \mathsf{path}_i, \mathsf{path}_{i+1})$ where each path_j consists of the collection of nodes from $\mathsf{node}_j = \mathsf{node}_j$ to root.
- $\mathsf{Verify}(\mathsf{com}, x_i, i, \pi) \rightarrow \{0,1\}$. It first parses π as a set of three path collections. Then, for each $j \in [3]$, it parses path_j as $\{\mathsf{node}_{j,1}, \mathsf{node}_{j,2}, \ldots, \mathsf{node}_{j,d_j}\}$ for tree depth d_j (thus $\mathsf{node}_{j,d_j} \equiv \mathsf{node}_j$), and performs two types of checks:

[4] For soundness, we must also show that the leaves are stored in sorted order.
[5] This induces a binary search tree over the interior nodes.

(i) *Path correctness.* It checks that $\mathsf{node}_{j,1} = \mathsf{root}$. Next, each node $\mathsf{node}_{j,k}$ is in turn interpreted as $(b_{j,k}||l_{j,k}||\Gamma_{j,k}||r_{j,k})$. For each $1 < k < d_j$, it checks whether $\mathsf{node}_{j,k+1}$ is a left or right child of its parent, i.e., whether $\mathsf{H}(\mathsf{node}_{j,k+1}) = l_{j,k}$ or $r_{j,k}$. If it is a left (resp. right) child it checks that $\Gamma_{j,k'} \leq \Gamma_{j,k}$ (resp. $\Gamma_{j,k'} \geq \Gamma_{j,k}$) for all $k' > k$. For each $k < d_j$, it checks that $b_{j,k} = 1$ and conversely that that $b_{j,d_j} = 0$.

(ii) *Adjacency check.* Firstly, it checks that $\Gamma_{1,d_1} < \Gamma_{2,d_2} = x_i < \Gamma_{3,d_3}$. Let κ be the largest common prefix of the three paths, i.e., $\mathsf{node}_{1,k} = \mathsf{node}_{2,k} = \mathsf{node}_{3,k}$ for all $k \leq \kappa$. It checks that $\Gamma_{1,\kappa} = \Gamma_{1,d_1}$, and that $l_{1,\kappa} = \mathsf{H}(\mathsf{node}_{1,\kappa+1})$, and $r_{1,\kappa} = \mathsf{H}(\mathsf{node}_{3,\kappa+1})$. Finally, letting $h_2 := \mathsf{H}(\mathsf{node}_{2,\kappa+1})$, for all $k > \kappa$, if $h_2 = r_{1,\kappa}$ it checks that $r_{1,k} = \mathsf{H}(\mathsf{node}_{1,k+1})$, $l_{2,k} = \mathsf{H}(\mathsf{node}_{2,k+1})$ and $r_{2,d_2-1} = \mathsf{H}(\mathsf{node}_{3,d_3})$; otherwise if $h_2 = l_{1,\kappa}$ it checks that $r_{2,k} = \mathsf{H}(\mathsf{node}_{2,k+1})$, $l_{3,k} = \mathsf{H}(\mathsf{node}_{3,k+1})$ and $l_{2,d_2-1} = \mathsf{H}(\mathsf{node}_{1,d_1})$[6].

If any of the checks fails, it outputs 0. Otherwise it outputs 1.

- $\mathsf{ProveNonMembership}(\mathsf{com}, x) \to \overline{\pi}_x$. It parses every leaf node_i as $(0||0^\lambda||x_i|0^\lambda)$. It then performs a binary search on the x_i's, if the value x is found, it outputs $\perp$ and continues otherwise. Via the aforementioned binary search, it finds leaf nodes $\mathsf{node}_{\mathrm{lo}}$ and $\mathsf{node}_{\mathrm{hi}}$ such that $\mathsf{node}_{\mathrm{lo}}$ (resp. $\mathsf{node}_{\mathrm{hi}}$) contains the greatest (resp. smallest) value smaller (resp. greater) than x. It then outputs $\pi := (\mathsf{path}_{\mathrm{lo}}, \mathsf{path}_{\mathrm{hi}})$ where each path_j consists of the collection of nodes from node_j to root.
- $\mathsf{VerifyNonMembership}(\mathsf{com}, x, \overline{\pi}) \to \{0, 1\}$. It first parses $\overline{\pi}$ as a set of two path collections. Then, for each $j \in [2]$, it parses path_j as $\{\mathsf{node}_{j,1}, \mathsf{node}_{j,2}, \ldots, \mathsf{node}_{j,d_j}\}$ for tree depth d_j (thus $\mathsf{node}_{j,d_j} \equiv \mathsf{node}_j$), and performs two types of checks:

(i) *Path correctness.* It checks that $\mathsf{node}_{j,1} = \mathsf{root}$. Next, each node $\mathsf{node}_{j,k}$ is in turn interpreted as $(b_{j,k}||l_{j,k}||\Gamma_{j,k}||r_{j,k})$. For each $1 < k < d_j$, it checks whether $\mathsf{node}_{j,k+1}$ is a left or right child of its parent, i.e., whether $\mathsf{H}(\mathsf{node}_{j,k}) = l_{j,k+1}$ or $r_{j,k+1}$. If it is a left (resp. right) child it checks that $\Gamma_{j,k} \leq \Gamma_{j,k+1}$ (resp. $\Gamma_{j,k} \geq \Gamma_{j,k+1}$). For each $k < d_j$, it checks that $b_{j,k} = 1$ and conversely that that $b_{j,d_j} = 0$.

(ii) *Adjacency check.* Firstly, it checks that $\Gamma_{1,d_1} < x < \Gamma_{3,d_3}$. Let κ be the largest common prefix of the two paths, i.e., $\mathsf{node}_{1,k} = \mathsf{node}_{2,k}$ for all $k \leq \kappa$. Check that $\Gamma_{1,\kappa} = \Gamma_{1,d_1}$, and that $l_{1,\kappa} = \mathsf{H}(\mathsf{node}_{1,\kappa+1})$, and $r_{1,\kappa} = \mathsf{H}(\mathsf{node}_{2,\kappa+1})$. Finally, for all $k > \kappa$, it checks that $r_{1,k} = \mathsf{H}(\mathsf{node}_{1,k+1})$ and $l_{2,k} = \mathsf{H}(\mathsf{node}_{2,k+1})$[7].

If any of the checks fails, it outputs 0. Otherwise it outputs 1.

Remark 2. Verification of (non-)membership can be easily extended for instances where one of the paths is empty (this occurs at the end nodes). Without loss of

[6] This checks that if the middle path is to the right of $\mathsf{node}_{\cdot,\kappa}$, then the first path is comprised of nodes going rightward, the second path is comprised of nodes going leftward and the third path ends in a node to the immediate right of the middle node. This is similarly extended to the case when the middle path is to the left of $\mathsf{node}_{\cdot,\kappa}$

[7] This checks whether the first path is comprised of nodes going rightward, and the second path is comprised of nodes going leftward.

generality, assume that the left path, path_{i-1}) (resp. $\mathsf{path}_{\mathrm{lo}}$) is empty. Then, the membership (resp. non-membership) verifier first performs the *path correctness* check for the non-empty path(s). It then checks that $\Gamma_{2,d_2} = x_i < \Gamma_{3,d_3}$ (resp. $x < \Gamma_{3,d_3}$) and also performs the *adjacency check* for node_2 and node_3. Lastly, the verifier must ensure that node_2 (resp. $\mathsf{node}_{\mathrm{hi}}$) holds the smallest value in the tree by checking that every node along the path to the root is a left child of its parent. It outputs 0 if any of the checks fail, and 1 otherwise.

Security. Below, we present our main theorems concerning the security of Construction 4.1.

Theorem 1. *Let* H *be a collision-resistant hash function, then* $\Pi^{MT}_{VCNM} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify}, \mathsf{ProveNonMembership}, \mathsf{VerifyNonMembership})$*, constructed as in Construction 4.1 is position binding.*

Proof. If an adversary $\mathcal{A}$, given the public parameters of the protocol, outputs $(\mathsf{com}, x, y, i, \pi, \overline{\pi})$ such that $x \neq y$ and

$$\mathsf{Verify}(\mathsf{com}, x, i, \pi) = \mathsf{Verify}(\mathsf{com}, y, i, \overline{\pi}) = 1$$

for some $i \in [n]$, then somewhere along the paths to x and y, consider two distinct nodes $\mathsf{node}^{(x)}_{i,k}$ and $\mathsf{node}^{(y)}_{i,k}$ such that $\mathsf{node}^{(x)}_{i,k-1} = \mathsf{node}^{(y)}_{i,k-1}$. Clearly, such a pair of nodes must exist since otherwise $\mathsf{root}^{(x)} \neq \mathsf{root}^{(y)}$. We can thus define adversary $\mathcal{A}_{\mathsf{CRHF}}$ that uses $\mathcal{A}$ to break collision-resistance of H. Specifically, it finds two such nodes, $\nu_0 := \mathsf{node}^{(x)}_{i,k}$ and $\nu_1 := \mathsf{node}^{(y)}_{i,k}$ and returns them. Since $\mathsf{node}^{(x)}_{i,k-1} = \mathsf{node}^{(y)}_{i,k-1}$, it follows that $\mathsf{H}(\nu_0) = \mathsf{H}(\nu_1)$. □

Theorem 2. $\Pi^{MT}_{VCNM} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify}, \mathsf{ProveNonMembership}, \mathsf{VerifyNonMembership})$*, constructed as in Construction 4.1 is element binding.*

Proof. This essentially follows from the path correctness and adjacency check of the membership and non-membership proofs. Concretely, suppose an adversary $\mathcal{A}$, given the public parameters of the protocol, outputs $(\mathsf{com}, x, i, \pi, \overline{\pi})$ such that

$$\mathsf{Verify}(\mathsf{com}, i, x, \pi) = \mathsf{VerifyNonMembership}(\mathsf{com}, x, \overline{\pi}) = 1$$

for some $i \in [n]$. Then, one of two cases must hold:

- Case I. $(x \in \mathbf{x})$: In this case $\mathcal{A}$ must find two adjacent leaf nodes $\mathsf{node}_{-1}, \mathsf{node}_{+1}$ in the tree such that $x_{-1} < x < x_{+1}$ for a valid proof of non-membership. However, if such a pair of values existed then either the adjacency check fails during membership verification, or the leafs are not in sorted order. In the latter case, the path correctness condition is violated, as the Γ values will violate the total order over the induced binary search tree on the interior nodes. Thus if $\mathcal{A}$ is able to find such a pair of nodes, membership verification will fail.

- Case II. ($x \notin \mathbf{x}$) : In this case, $\mathcal{A}$ simply can not produce a valid middle path such that $\Gamma_{2,d_2} = x$. Thus it can not produce a successful proof of membership.

Thus we conclude that such an adversary $\mathcal{A}$ can not exist. □

It follows that Construction 4.1 is a secure vector commitment scheme with non-membership.

Efficiency analysis. Assuming the size of the message space, $|\mathcal{M}| = 2^\lambda$, the total amount of storage needed to hold the full tree is $\mathcal{O}(n\lambda)$ which is asymptotically optimal. Further, both membership and non-membership proofs are of size $\mathcal{O}(\log n)$. Verification for a proof of (non-)membership requires checking the correctness conditions for all $\mathcal{O}(\log n)$ nodes, as well as adjacency conditions between pairs of paths in time $\mathcal{O}(\log n)$. The time to verify is thus also $\mathcal{O}(\log n)$.

5 Universal Vector Commitments from Universal Accumulators

We now describe our construction of an universal vector commitment, assuming a universal accumulator and a vector commitment scheme without non-membership. We present the construction below.

The main idea is straightforward — the commitment to a vector of values according to $\Pi^{\mathsf{Gen}}_{\mathsf{UVC}}$ consists of, both, a commitment according to Π_{VC} as well as the accumulated value of $\mathbf{x}$ according to Π_{UA}. The opening of a commitment for some value $x_i \in \mathbf{x}$ requires opening it to the position i and demonstrating that x_i is in the accumulated set. For non-membership of a value, the committer simply provides a proof that the value is not in the accumulated set. Intuitively, this scheme has non-membership soundness as the set-binding of the underlying accumulator ensures that a dishonest committer cannot simultaneously open a commitment to a value and also disprove that it was accumulated.

5.1 Construction

Let Π_{UA} be a universal accumulator scheme with strong set binding and Π_{VC} a vector commitment scheme with strong position binding. We define a vector commitment scheme with non-membership $\Pi^{\mathsf{Gen}}_{\mathsf{UVC}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify}, \mathsf{ProveNonMembership}, \mathsf{VerifyNonMembership})$ as follows:

- $\mathsf{Setup}(1^\lambda, 1^n) \rightarrow \mathsf{pp}$. It obtains the parameters for the vector commitment as $\mathsf{pp}_{\mathsf{VC}} \leftarrow \mathsf{VC.Setup}(1^\lambda, 1^n)$, and initialises the universal accumulator, $(\mathsf{state}_0, \mathsf{acc}_0) \leftarrow \mathsf{UA.Gen}(1^\lambda)$. It outputs the protocol's public parameters $\mathsf{pp} := (\mathsf{pp}_{\mathsf{VC}}, \mathsf{state}_0, \mathsf{acc}_0)$.

- $\mathsf{Commit}\left(\mathbf{x} = \left[x_1 \cdots x_n\right], \mathsf{aux} = (\mathsf{state}_0, \mathsf{acc}_0)\right) \rightarrow \mathsf{com}$. It creates a commitment to the vector $\mathbf{x}$ as $\mathsf{com}' \leftarrow \mathsf{VC.Commit}(\mathsf{pp}_{\mathsf{VC}}, \mathbf{x})$. Then, for each $i \in [n]$, it accumulates x_i as $(\mathsf{state}_i, \mathsf{acc}_i) \leftarrow \mathsf{UA.Add}(\mathsf{acc}_{i-1}, \mathsf{state}_{i-1}, x_i)$. Finally it returns $\mathsf{com} \coloneqq (\mathsf{com}', \mathsf{acc}_n)$ as the full commitment and auxiliary value $\mathsf{aux} \coloneqq \mathsf{state}_n$.
- $\mathsf{Open}(\mathsf{com}, i, x_i, \mathsf{aux} = \mathsf{state}_n) \rightarrow \pi$. It parses com as $(\mathsf{com}', \mathsf{acc}_n)$. It then runs the vector commitment opening algorithm $\pi_{\mathsf{VC}} \leftarrow \mathsf{VC.Open}(\mathsf{com}', i, x_i)$ as well as the accumulator membership algorithm $\pi_{\mathsf{UA}} \leftarrow \mathsf{UA.MemWitCreate}(\mathsf{acc}_n, \mathsf{state}_n, x_i)$. It returns the opening full proof $\pi \coloneqq (\pi_{\mathsf{VC}}, \pi_{\mathsf{UA}})$.
- $\mathsf{Verify}(\mathsf{com}, x_i, i, \pi) \rightarrow \{0,1\}$. It parses π as $(\pi_{\mathsf{VC}}, \pi_{\mathsf{UA}})$, and com as $(\mathsf{com}', \mathsf{acc}_n)$. It returns the outcome of $\mathsf{VC.Verify}(\mathsf{com}', i, x_i, \pi_{\mathsf{VC}}) \wedge \mathsf{UA.VerifyMem}(\mathsf{acc}_n, x_i, \pi_{\mathsf{UA}})$.
- $\mathsf{ProveNonMembership}(\mathsf{com}, x, \mathsf{aux}) \rightarrow \overline{\pi}_x$. It parses com as $(\mathsf{com}', \mathsf{acc}_n)$. It then creates an accumulator non-membership proof $\overline{\pi}_x \leftarrow \mathsf{UA.NonMemWitCreate}(\mathsf{acc}_n, \mathsf{state}_n, x)$ and outputs it.
- $\mathsf{VerifyNonMembership}(\mathsf{com}, x, \overline{\pi}_x) \rightarrow \{0,1\}$. It parses π as $(\pi_{\mathsf{VC}}, \pi_{\mathsf{UA}})$, and com as $(\mathsf{com}', \mathsf{acc}_n)$. It returns the outcome of $\mathsf{UA.VerifyNonMem}(\mathsf{acc}_n, x, \overline{\pi}_x)$.

Security. Below, we present our main theorems concerning the security of Construction 5.1.

Theorem 3. *Let Π_{UA} be a universal accumulator and Π_{VC} be a vector commitment, then*
$\Pi_{\mathsf{UVC}}^{\mathsf{Gen}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify}, \mathsf{ProveNonMembership}, \mathsf{VerifyNonMembership})$, *constructed as in Construction 5.1 is position binding.*

Proof. If an adversary $\mathcal{A}$, given the public parameters of the protocol, outputs $(\mathsf{com}, x, y, i, \pi, \overline{\pi})$ such that $x \neq y$ and

$$\mathsf{Verify}(\mathsf{com}, x, i, \pi) = \mathsf{Verify}(\mathsf{com}, y, i, \overline{\pi}) = 1$$

for some $i \in [n]$, then an adversary $\mathcal{A}_{\mathsf{VC}}$ can use $\mathcal{A}$ to break the position binding property of Π_{VC}, since for the above equality to hold, we must have that

$$\mathsf{VC.Verify}(\mathsf{com}_0, x, i, \pi) = \mathsf{VC.Verify}(\mathsf{com}_0, y, i, \overline{\pi}) = 1 \quad .$$ □

Theorem 4. *Let Π_{UA} be a universal accumulator and Π_{VC} be a vector commitment, then*
$\Pi_{\mathsf{UVC}}^{\mathsf{Gen}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify}, \mathsf{ProveNonMembership}, \mathsf{VerifyNonMembership})$, *constructed as in Construction 5.1 is element binding.*

Proof. Suppose an adversary $\mathcal{A}$, given the public parameters of the protocol, outputs $(\mathsf{com}, x, i, \pi, \overline{\pi})$ such that

$$\mathsf{Verify}(\mathsf{com}, i, x, \pi) = \mathsf{VerifyNonMembership}(\mathsf{com}, x, \overline{\pi}) = 1$$

for some $i \in [n]$, then an adversary $\mathcal{A}_{\mathsf{UA}}$ can use $\mathcal{A}$ to break set binding property of Π_{UA}, since we must have that

$$\mathsf{UA.VerifyMem}(\mathsf{com}_1, x, \pi) = \mathsf{UA.VerifyNonMem}(\mathsf{com}_1, x, \overline{\pi}) \quad .$$ □

Thus, given a universal accumulator Π_{UA}, and a vector commitment Π_{VC}, Construction 5.1 is a secure vector commitment scheme with non-membership.

6 Universal Accumulators from Vector Commitments

We give a generic technique for obtaining a universal accumulator from a vector commitment scheme using cuckoo hashing. At a high level, the idea is to use a vector commitment on a cuckoo hash table containing elements from the set S being accumulated. A proof of membership for some element $x \in S$ is then a commitment opening to the position of x in the cuckoo hash table, and a proof of non-membership for some element $\overline{x} \notin S$ is a commitment opening to *all positions that* $\overline{x}$ *could have been* in the cuckoo hash table.

The resulting universal accumulator has the same membership proof size as the underlying vector commitment scheme while non-membership proofs are a factor of m larger (typically a small constant). Similarly, both prover and verifier computation is asymptotically the same as that of the vector commitment scheme. Notably, if the underlying vector commitment scheme supports sub-vector openings or batch proofs, the membership proofs for the accumulator scheme can also trivially be aggregated or batched respectively. Unfortunately, this does not translate to non-membership proofs as the proofs grow linearly with the number of aggregated (resp. batched) proofs.

6.1 Construction

Let Π_{VC} be a vector commitment scheme over the message space $\mathcal{M}$, let $\mathsf{H} : D \to \mathcal{M} \setminus \{0\}$ be a CRHF over some domain $\mathcal{D}$, and let $\mathcal{H} = \{h_i : \mathcal{M} \setminus \{0\} \to \{1 + (i-1) \cdot \ell, \ldots, \ell + (i-1) \cdot \ell\} \mid \forall i \in [m]\}$ be a family of public hash functions associated with a cuckoo hash $\Pi_{\mathsf{CH}} = (\mathsf{lookup}, \mathsf{insert}, \mathsf{delete})$. Then we define an updateable universal accumulator scheme as follows:

- $\mathsf{Setup}(1^\lambda, 1^\ell, 1^m) \to \mathsf{pp}$. It sets up the public parameters for the $\mathsf{pp} \leftarrow \mathsf{VC.Setup}(1^\lambda)$ and outputs pp. All algorithms below take pp as input but we omit it for notational clarity.
- $\mathsf{Gen}(1^\lambda) \to (\mathsf{state}_0, \mathsf{acc}_0)$. It creates an empty cuckoo table $\mathbf{T}_0 \coloneqq \mathbf{0}^{\ell \cdot m}$ and defines the set, $S_0 \coloneqq \varnothing$, to be accumulated. It creates the initial commitment $\mathsf{acc}_0 \leftarrow \mathsf{VC.Commit}(\mathbf{T}_0)$, and outputs the initial state $\mathsf{state}_0 \coloneqq (\mathbf{T}_0, S_0)$ and acc_0.
- $\mathsf{Add}(\mathsf{acc}_t, \mathsf{state}_t, x) \to \mathsf{state}_{t+1}, \mathsf{acc}_{t+1}$. It updates $S_{t+1} \coloneqq S_t \cup \{x\}$, $\mathbf{T}_{t+1} \coloneqq \mathsf{CH.insert}(\mathsf{H}(x), \mathbf{T}_t)$ and sets the new state $\mathsf{state}_{t+1} \coloneqq (S_{t+1}, \mathbf{T}_{t+1})$. Finally it computes the new accumulation value $\mathsf{acc}_{t+1} \leftarrow \mathsf{VC.Commit}(\mathbf{T}_{t+1})$ and outputs $\mathsf{state}_{t+1}, \mathsf{acc}_{t+1}$.
- $\mathsf{Delete}(\mathsf{acc}_t, \mathsf{state}_t, x) \to \mathsf{state}_{t+1}, \mathsf{acc}_{t+1}$. It updates $S_{t+1} \coloneqq S_t \setminus \{x\}$, $\mathbf{T}_{t+1} \coloneqq \mathsf{CH.delete}(\mathsf{H}(x), \mathbf{T}_t)$ and sets the new state $\mathsf{state}_{t+1} \coloneqq (S_{t+1}, \mathbf{T}_{t+1})$. Finally it computes the new accumulation value $\mathsf{acc}_{t+1} \leftarrow \mathsf{VC.Commit}(\mathbf{T}_{t+1})$ and outputs $\mathsf{state}_{t+1}, \mathsf{acc}_{t+1}$.

- $\mathsf{MemWitCreate}(\mathsf{acc}_t, \mathsf{state}_t, x) \to w_x$. If $x \notin S_t$, it sets w_x to $\perp$ and returns. Otherwise, it looks up the index $i \leftarrow \mathsf{CH.lookup}(\mathsf{H}(x), \mathbf{T}_t)$ for x in the Cuckoo Hash, and sets $w_x \leftarrow \mathsf{VC.Open}(\mathsf{pp}, \mathsf{acc}_t, i)$ and outputs it.
- $\mathsf{VerifyMem}(\mathsf{acc}_t, x, w_t^x) \to \{0, 1\}$. It returns the outcome of

$$\bigvee_{i \in [m]} \mathsf{VC.Verify}(\mathsf{pp}, \mathsf{acc}_t, h_i(\mathsf{H}(x)), \mathsf{H}(x), w_x) \quad .$$

- $\mathsf{NonMemWitCreate}(\mathsf{acc}_t, \mathsf{state}_t, x) \to \overline{w}_x$. If $x \in S_t$, it sets w_x to $\perp$ and returns. Otherwise, it sets

$$\overline{w}_x \coloneqq \bigcup_{i \in [m]} (\mathbf{T}_t[h_i(\mathsf{H}(x))], \mathsf{VC.Open}(\mathsf{pp}, \mathsf{acc}_t, h_i(\mathsf{H}(x)))) \quad .$$

- $\mathsf{VerifyNonMem}(\mathsf{acc}_t, x, \overline{w}_x) \to \{0, 1\}$. It parses $\overline{w}_x$ as $\{(y_i, \pi_i)_{i=1}^m\}$. and returns the outcome of

$$\bigwedge_{i \in [m]} \mathsf{VC.Verify}(\mathsf{pp}, \mathsf{acc}_t, h_i(\mathsf{H}(x)), y_i, \pi_i) = 1 \wedge \mathsf{H}(x) \neq y_i \quad .$$

It should be noted that in our universal accumulator construction, we do not formally describe a mechanism for updating an existing proof for some element. Instead, proof updates are done by re-running the witness creation algorithms on the new accumulator value. While we do not rule it out entirely, an efficient proof update is nonetheless challenging in our generic construction for the simple reason that when an element is added into the set, a new commitment to the updated table must be generated and every opening proof is now with respect to this new commitment.

Security. Below, we present our main theorems concerning the security of Construction 6.1. Due to space constraints, we provide the proofs in Appendix **??**.

Theorem 5. *Let* Π_{VC}= (Setup, Commit, Open, Verify) *is a vector commitment and* $\mathcal{H}$ *be a family of hash functions, then* Π = (Gen, Add, Delete, MemWitCreate, VerifyMem, NonMemWitCreate, VerifyNonMem) *constructed as given in Construction 6.1 is a universal accumulator.*

Proof. As a lemma, we first argue that the values inserted and deleted from the cuckoo table $\mathbf{T}_i$ at each time $i \in [t]$ form a one-to-one correspondence with the elements added and removed from the accumulated set S_i at the same time. If not, i.e. we have some $i, j \in [t]$ such that $y_i \neq y_j$ yet $\mathsf{H}(y_i) = \mathsf{H}(y_j)$, then we have a break in the CRHF.

To show the correctness of $\mathsf{VerifyMem}$, suppose the accumulator currently represents state S_t, and $x \in S_t$. This implies there exists an $j \in [t]$ such that the operation performed was $\mathsf{Add}(\mathsf{pp}, \mathsf{acc}_{j-1}, \mathsf{state}_{j-1}, x)$ and for any $k \in [t]$ such that the operation was $\mathsf{Delete}(\mathsf{pp}, \mathsf{acc}_{k-1}, \mathsf{state}_{k-1}, x)$, $k < j$. Thus $\mathsf{H}(x)$ has been inserted into the CH table and not been removed. Recall that once an element is

inserted into a cuckoo table, it always exists at a location defined by its image under some hash h_i, with future operations only swapping which of the m hashes it uses. This implies there exists some $i \in [m]$ such that position i in $\mathbf{T}_t$ is $\mathsf{H}(x)$, which is the value returned by $\mathsf{CH.lookup}(\mathsf{H}(x), \mathbf{T}_t)$. By the correctness of Π_{VC}, $\mathsf{VC.Verify}(\mathsf{pp}, \mathsf{acc}_t, h_i(\mathsf{H}(x)), \mathsf{H}(x), w_x) = 1$ with probability $1 - \mathsf{negl}(\lambda)$. Hence by union bound $\mathsf{VerMem}(\mathsf{pp}, \mathsf{acc}_t, x, \mathsf{MemWitCreate}(\mathsf{pp}, \mathsf{acc}_t, \mathsf{state}_t, x))$ accepts with probability
bounded below by $1 - \mathsf{negl}(\lambda)$.

Our proof of correctness for $\mathsf{VerifyNonMem}$ proceeds similarly: $x \notin S$ implies there is no i, j such that position j of $\mathbf{T}_t = h_i(\mathsf{H}(x))$. By the correctness of $\mathsf{VC.Verify}$, $\mathsf{VC.Verify}(\mathsf{pp}, \mathsf{acc}_t, h_i(\mathsf{H}(x)), y_i, \pi_i) = 1$ with all but negligible probability for each $i \in [m]$, thus $\mathsf{VerifyNonMem}$ accepts with all but negligible probability.

We now argue our accumulator is strong set binding. Suppose an adversary is able to provide x, acc_t, w_t^x, and $\overline{w}_t^x$ which breaks the set binding property. From the correctness of $\mathsf{VerifyMem}$, there exists an $i \in [m]$ such that $\mathsf{VC.Verify}(\mathsf{pp}, \mathsf{acc}_t, h_i(\mathsf{H}(x)), \mathsf{H}(x), w_t^x) = 1$. From the correctness of $\mathsf{VerifyNonMem}$, $\overline{w}_t^x$ includes (y_i, π_i) such that $\mathsf{VC.Verify}(\mathsf{pp}, \mathsf{acc}_t, h_i(\mathsf{H}(x)), y_i, \pi_i) = 1$ and $y_i \neq \mathsf{H}(x)$. This violates the position binding property of Π_{VC}. □

Efficiency analysis. One can easily verify that our public parameter and membership proof sizes are exactly the same as the parameter and proof size of the vector commitment scheme over a vector of $m \cdot \ell$ elements. Note that since each element is from the image of a CRHF $\mathsf{H}(\cdot)$, the size $|\mathcal{M}|$ of each element is $\mathcal{O}(\lambda)$ rather than $\mathcal{O}(\log |D|)$. As for our nonmembership proof, we output m copies of a vector commitment opening along with elements from the VC message space, hence the size is $\mathcal{O}(m \cdot V)$, where V is the size of a single VC opening proof over $\mathcal{M}^{m\ell}$. Thus, we achieve our claimed domain-independence.

By applying the result of [Yeo23], we can build a cuckoo table with negligible failure probability (and hence an accumulator with all but negligible correctness) using $m = \mathcal{O}\left(\sqrt{\frac{\lambda}{log\ S'}}\right)$ hashes, each containing $\ell = O(S')$ cells. In practice, it is sufficient to set m to some small constant, e.g. $m = 2$. However, this leads to failure probability $\frac{1}{poly(\lambda)}$; by including a definition of the cuckoo hash functions along with the accumulator rather than as a one-time setup, we may use the standard strategy of resampling the hash functions and creating a new table whenever a failure occurs. With this we achieve the desired correctness probability at the cost of larger (expected) runtime and $\mathcal{O}(\lambda)$ additional accumulator size. Note that allowing the accumulator to use (potentially maliciously-chosen) hash functions does not break the security of the accumulator, as strong soundness only relies on the hashes being public deterministic functions. At worst, adversarially-chosen cuckoo hashes reduces the efficiency of insert and delete operations.

Optimizing for small domains. Our above construction and analysis assumes that $\log D \gg \lambda$. In smaller domains where this is not the case, it is concretely

more efficient to omit the CRHF and instead operate directly on $\mathcal{M} = \mathcal{D}$. More precisely, we define our cuckoo table $\mathbf{T}$ as a vector over $\mathcal{D}$ and during insert, delete, and lookup we use x as input rather than $\mathsf{H}(x)$. This adds a factor of $\log D$ to our accumulator and proof sizes, leading to the results claimed in Table 2.

6.2 Instantiations

Scheme	Setup	Assumption	Binding	acc	w_x	$\overline{w}_x$
[YAY+18]	Public	SIS	Weak	$\log D$	$\log S \log D$	$\log S \log D$
[dCP23]	Public	SIS	Strong	$\log D$	$\log^2 S' \log D$	$\log^2 S' \log D$
[GV20]	Public	SIS	Strong	$\log D$	$\log S \log D$	$\log S \log D$
Our Claim (1)	Public	SIS	Strong	$\log D$	$\log S' \log D$	$\log S' \log D$

Table 2. Comparison of Lattice-based universal accumulator schemes. For brevity we omit terms derived from the security parameter λ.

By appropriately instantiating our generic construction, we show that one can obtain an efficient universal accumulators from SIS and standard RSA.

To briefly summarize the VC of [LLNW16]: it is a Merkle Tree using a hash function derived from a public random matrix, with an additional step at each layer to ensure the output is "small" in terms of the SIS problem. We may make our construction dynamic by adding the improvements of [LNWX17], which note that an update to one element of the input only requires updates to proofs holding nodes on that element's path to the root. The output of the hash (and thus the size of the accumulator) is a single element from the domain, and an opening proof is $\log S$ domain elements. Verification consists of recomputing the root hash from the opened element and the siblings along the path to the root. For our instantiation, our domain is fixed-size due to the CRHF. The input size is the total number of cells in the cuckoo table rather than the total number of accumulated elements, i.e. $\mathcal{O}(m\ell) = \mathcal{O}(S')$ instead of $\mathcal{O}(S)$. So,

Claim 1. *Together with the vector commitment of [LLNW16] (or [LNWX17]) based on the SIS assumption, we have a universal accumulator based on SIS.*

In Table 2 we compare our instantiations with the state-of-the-art lattice-based universal accumulators [YAY+18, dCP23, GV20]. Our instantiations have competitive accumulated values and proof sizes for both membership and non-membership.

Now, to briefly summarize the VC of [CF13]: given a large semiprime modulus N we have n bases s_1 to s_n and exponents e_1 to e_n corresponding to the indices in our VC. The commitment stores values $m_i \in \mathcal{M}$ as $s_i^{\mu_i}$, and the commitment c is the product of these values. An opening proof for μ_i is the e_i^{th} root mod N of $\pi_i = \Pi_{j \neq i} s_j^{\mu_j}$, yielding the simple verification algorithm $c = s_i^{\mu_i} \cdot \pi_i^{e_i}$. The construction supports batch updates for both commitments and proofs for the same cost as one update, thus our instantiation can easily support updating

all changes to the cuckoo table as a result from a single add or delete with no additional overhead. So,

Claim 2. *Together with the vector commitment of [CF13] based on the standard RSA assumption, we have a universal accumulator based on standard RSA.*

6.3 Generically Upgrading to VCs to UVCs

Finally, we observe that there is a compiler from any plain vector commitment scheme to an universal vector commitment scheme:

Claim 3. *Let Π_{VC} be a vector commitment scheme, and let Π_{UA} be a universal accumulator built using Π_{VC} using Construction 6.1. Then the Construction 5.1 using Π_{VC} and Π_{UA} is a secure universal vector commitment scheme.*

This construction's opening proof consists of a Π_{VC} opening proof plus a Π_{UA} membership proof, which as discussed in section 6 is itself a single Π_{VC} opening proof. The non-membership proof is a single Π_{UA} non-membership proof, which as discussed in section 6 is m Π_{VC} opening proofs, and $m = 2$ for typical parameters. Thus our compiler adds non-membership proofs with just a factor of 2 overhead on the original commitment scheme for both proof types.

7 Conclusion

We proposed the notion of universal vector commitment schemes that extend plain vector commitments with proofs of non-membership that allow one to prove that a value is not at any position of the committed vector. We also presented two constructions for universal vector commitments, the former of which offers practical efficiency while the latter offers some theoretical insights into our new primitive. We believe that the ability for vector commitments to provide non-membership proofs presents exciting practical opportunities for building more accountable systems. In particular, we gave the example of proofs of exclusion in blockchain-based transactions [Kon19], as well as in increasing accountability in distributed file-storage system by extending the usual proofs of retrievability [JK07, Fis19] to also include proofs of exclusion of illicit files.

We also presented a generic construction for universal accumulators using a vector commitment and a cuckoo hash. Importantly, we pointed out that this technique is broadly applicable to not just universal accumulators, but to many other related primitives including universal vector commitments.

Bibliography

[AR20] Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. 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 III*, page 839–869, Berlin, Heidelberg, 2020. Springer-Verlag.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

[BCD+17] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *2017 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 301–315, 2017.

[BdM94] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 274–285. Springer Berlin Heidelberg, 1994.

[BP97] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 480–494. Springer Berlin Heidelberg, 1997.

[CF13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 55–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[dCP23] Leo de Castro and Chris Peikert. Functional commitments for all functions, with transparent setup and from sis. In *Advances in Cryptology – EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*, page 287–320, Berlin, Heidelberg, 2023. Springer-Verlag.

[DPP16] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees. In Billy Bob Brumley and Juha Röning, editors, *Se-*

cure IT Systems, pages 199–215, Cham, 2016. Springer International Publishing.

[Fis19] Ben Fisch. Tight proofs of space and replication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 324–348, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[FKP23] Dario Fiore, Dimitris Kolonelos, and Paola de Perthuis. Cuckoo commitments: Registration-based encryption and key-value map commitments for large spaces. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023*, pages 166–200, Singapore, 2023. Springer Nature Singapore.

[GHMR18] Sanjam Garg, Mohammad Hajiabadi, Mohammad Mahmoody, and Ahmadreza Rahimi. Registration-based encryption: Removing private-key generator from IBE. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018, Part I*, volume 11239 of *LNCS*, pages 689–718, Panaji, India, November 11–14, 2018. Springer, Heidelberg, Germany.

[GV20] Rishab Goyal and Satyanarayana Vusirikala. Verifiable registration-based encryption. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 621–651, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[JK07] Ari Juels and Burton S. Kaliski Jr. Pors: proofs of retrievability for large files. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 584–597, Alexandria, Virginia, USA, October 28–31, 2007. ACM Press.

[Kon19] Georgios Konstantopoulos. Plasma cash: Towards more efficient plasma constructions, 2019.

[LLNW16] Benoît Libert, San Ling, Khoa Nguyen, and Huaxiong Wang. Zero-knowledge arguments for lattice-based accumulators: Logarithmic-size ring signatures and group signatures without trapdoors. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 1–31, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[LLX07] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[LNWX17] San Ling, Khoa Nguyen, Huaxiong Wang, and Yanhong Xu. Lattice-based group signatures: Achieving full dynamicity with ease. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Applied Cryptography and Network Security*, pages 293–312, Cham, 2017. Springer International Publishing.

[RP04] Flemming Friche Rodler Rasmus Pagh. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[RY16] Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed pki. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 292–309, Cham, 2016. Springer International Publishing.

[SMP23] Daria Schumm, Rahma Mukta, and Hye-young Paik. Efficient credential revocation using cryptographic accumulators. In *2023 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 127–134, 2023.

[TBP+19] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1299–1316. ACM, 2019.

[YAY+18] Zuoxia Yu, Man Ho Au, Rupeng Yang, Junzuo Lai, and Qiuliang Xu. Lattice-based universal accumulator with nonmembership arguments. In Willy Susilo and Guomin Yang, editors, *Information Security and Privacy*, pages 502–519, Cham, 2018. Springer International Publishing.

[Yeo23] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 197–230, Cham, 2023. Springer Nature Switzerland.

[ZKP17] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. An expressive (zero-knowledge) set accumulator. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 158–173. IEEE, 2017.

A More Generic Constructions from Cuckoo Hashing

As we previously asserted — the vector commitment and cuckoo hashing abstraction turns out to be quite useful for building many related primitives. This is also summarized in the figure below.

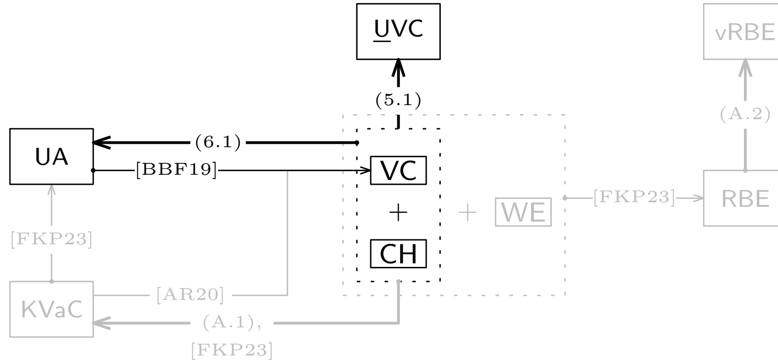

Fig. 3. Extended relationship between VCs, UAs, KVaCs and (v)RBEs.

In this section, we outline high-level constructions for Key-Value Commitments [AR20] and Verifiable Registration-based Encryption Scheme [GV20]. Since these are essentially observations on exiting work, we do not give formal definitions of these objects and instead direct the reader to the cited literature.

A.1 Dynamic Key-Value Commitments

The universal accumulator construction in Construction 6.1 can be generalized further to obtain commitments for key-value maps. In particular, we can once again leverage the cuckoo hashing technique in a manner identical to Construction 6.1 while modifying $\mathsf{CH.insert}$ to insert a key-value pair $x = (k, v) \in \mathcal{K} \times \mathcal{V}$ according to key k as shown in Figure 4. Then, if all keys are unique, updates can be performed by changing the value in the appropriate data field and generating a fresh commitment. The rest of the protocol behaves identically[8].

The definition in [AR20] requires that a secure KVaC must satisfy *key binding*. Informally it says that it should be infeasible for any polynomially bounded adversary with oracle access to Add, Delete and Update, to come up with an *honestly* generated commitment and either two certificates to different values under the same key, or a certificate for a value under a key not in the map. We claim the the above construction satisfies this property.

Claim 4. *Let Π_{VC} be a vector commitment scheme, then the KVaC construction outlined above is key binding.*

[8] This is basically also the construction in [FKP23] where they use two VCs to store k and v separately whereas we store a single (k, v) tuple.

Let $h_i : \mathcal{U} \to \{1+(i-1)\cdot\ell, \ldots, \ell+(i-1)\cdot\ell\}$ for $i \in [m]$ be hash functions associated with table $\mathbf{T}$, such that $|\mathbf{T}| = m\ell$.

- $\mathsf{insert}(k, v, \mathbf{T})$
  1. If $\mathsf{lookup}(k) \neq \perp$, do nothing and return $\mathbf{T}$.
  2. For $i = 1$ to m do:
     - If $\mathbf{T}[h_i(k)] = \varnothing$, set $\mathbf{T}[h_i(k)] \leftarrow (k, v)$ and return $\mathbf{T}$.
     - Otherwise, swap $(k, v) \leftrightarrow \mathbf{T}[h_i(k)]$.
  3. If $(k, v) \neq \varnothing$, call $\mathsf{insert}(k, v, \mathbf{T})$.
  4. Return $h_1(k), \mathbf{T}$.
- $\mathsf{delete}(k, \mathbf{T})$
  1. For $i \in [m]$, set $\mathbf{T}[h_i(k)] \leftarrow \varnothing$ if $\mathbf{T}[h_i(k)] = (k, \cdot)$.
  2. Return $\mathbf{T}$.
- $\mathsf{lookup}(k, \mathbf{T})$
  1. Set $\mathsf{pos} \leftarrow \perp$.
  2. For $i \in [m]$, set $\mathsf{pos} \leftarrow h_i(k)$ if $\mathbf{T}[h_i(k)] = (k, \cdot)$.
  3. Return pos.

Fig. 4. Modified cuckoo hashing

Sketch. Note that it is impossible, by construction, for an adversary to come up with two proofs for different values under the same key k (with respect to some honestly generated commitment) given it can only perform a single insertion under any k. Additionally, proving membership of a value under some key that has not been inserted would amount to breaking position binding of the VC. So the KVaC is key binding under insertion.

A.2 Verifiable Registration-based Encryption

Registration-based Encryption was introduced by Garg et al. [GHMR18] as an alternative to Identity-based Encryption wherein a trusted central authority is responsible for generating the user's secret. In an RBE, users instead generate their own keys and register themselves (with their public keys) with a central authority known as the key *curator*, who in turn maintains the system state and its public parameters for all registered users. Thus, an RBE facilitates a public key infrastructure where parties can send encrypted messages knowing only each others' public keys (and identities), and the public parameters of the system. To decrypt a message, the user must posses the corresponding secret key and a piece of opening information that is retrievable via the curator. Besides requiring traditional semantic security for encryption, an RBE scheme must have compact public parameters, both encryption and decryption should be sublinear in n, the number of registered users.

Verifiable RBEs were proposed by Goyal and Vusirikala [GV20] to introduce accountability to the RBE curator. The requirement in a vRBE is that a malicious curator must be able to prove unique registration (resp. non-registration) for every registered (resp. unregistered) user. This is accomplished with the help of two algorithms $\mathsf{PreProve}$ and $\mathsf{PostProve}$ that respectively capture the case of

proving that a user is not yet registered and that a user has been (uniquely) registered. While Goyal and Vusirikala give a Merkle Tree based construction (their curator is essentially what we use in Section 4), it is clear with hindsight that more generally, verifiability in RBE schemes can be achieved somewhat generically if the curator uses a UA to store the public keys.

Fiore et al. [FKP23] gave an RBE construction from VC, CH and an RBE scheme. For security, they additionally require a new primitive called *witness encryption for vector commitments* (VCWE). A VCWE encrypts a message using a VC and a value x at index i in the committed vector. Decryption is performed via an opening proof for (x, i) with respect to the commitment. The registration mechanism in their RBE proceeds by inserting the users' identities into a CH and committing to it with the VC (notice here the resemblance with our UA construction). To encrypt a message, a user runs the VCWE with respect to the receiver's identity and position in the table. Using the opening proof for their identity in the CH as the opening information, the receiver is able to decrypt the ciphertext successfully[9].

A straightforward observation is that their registration process is also our UA construction from Section 6. Importantly for our purposes, this means that their curator readily gives a vRBE when instantiated as per our Construction 6.1, where the pre- and post-registration proofs are the non-membership and membership proofs respectively and the *soundness of pre- and post-registration verifiability* follow from the soundness of the UA.

[9] We have omitted many technical details here as they would be redundant. Please see [FKP23] for the construction.