Authenticated Dictionaries with Cross-Incremental Proof (Dis)aggregation

Alin Tomescu[1] Yu Xia[2] Zachary Newman[2]

[1] *VMware Research*
[2] *MIT CSAIL*

Wednesday, October 7th, 2020

Abstract

Authenticated dictionaries (ADs) are a key building block of many cryptographic systems, such as transparency logs, distributed file systems and cryptocurrencies. In this paper, we propose a new notion of *cross-incremental proof (dis)aggregation* for authenticated dictionaries, which enables aggregating multiple proofs with respect to *different dictionaries* into a single, succinct proof. Importantly, this aggregation can be done *incrementally* and can be later reversed via *disaggregation*. We give an efficient authenticated dictionary construction from hidden-order groups that achieves cross-incremental (dis)aggregation. Our construction also supports updating digests, updating (cross-)aggregated proofs and precomputing all proofs efficiently. This makes it ideal for stateless validation in cryptocurrencies with smart contracts. As an additional contribution, we give a second authenticated dictionary construction, which can be used in more malicious settings where dictionary digests are adversarially-generated, but features only "one-hop" proof aggregation (with respect to the same digest). We add support for *append-only proofs* to this construction, which gives us an *append-only authenticated dictionary (AAD)* that can be used for transparency logs and, unlike previous AAD constructions, supports updating and aggregating proofs.

Contents

1 Introduction **3**
1.1 Related Work . . . 3

2 Preliminaries **5**
2.1 Algorithms . . . 5
2.2 Proofs of Knowledge of Co-prime Roots (PoKCR) . . . 6
2.3 Proofs of Knowledge of Exponent (PoKE) . . . 6
2.4 RSA Accumulators . . . 6
2.4.1 Membership and subset witnesses . . . 7
2.4.2 Non-membership and disjointness witnesses . . . 7
2.5 Incrementally Aggregatable Vector Commitments from RSA . . . 7

3 Enhancements to RSA Accumulators **9**
3.1 Faster Witness Extraction in PoKCR . . . 9
3.2 Computing All Non-membership Witnesses Across Different, Related Accumulators . . . 10
3.3 Aggregating Non-membership Witnesses Across Different, Related Accumulators . . . 11

4 Authenticated Dictionaries from Hidden-Order Groups **11**
4.1 Definitions . . . 11
4.2 Updatable Authenticated Dictionary for Stateless Validation . . . 13
4.3 Append-only Authenticated Dictionary for Transparency Logs . . . 15

5 Future Work **17**

A Updating RSA Non-membership Witnesses **20**

B Security Proofs **20**
B.1 Definitions and Cryptographic Assumptions . . . 20
B.2 Warm-up: Key Binding for Individual Proofs . . . 21
B.2.1 Case 1: $v \neq \perp$ and $v' \neq \perp$ . . . 21
B.2.2 Case 2: $v = \perp$ or $v' = \perp$ . . . 22
B.2.3 Weak Key Binding for Individual UAD Proofs . . . 22
B.3 Key Binding for Aggregated Proofs . . . 22
B.3.1 Case 1: $v_z \neq \perp$ and $v'_z \neq \perp$ . . . 22
B.3.2 Case 2: $v_z = \perp$ or $v'_z = \perp$ . . . 23
B.3.3 Weak Key Binding for Aggregated UAD Proofs . . . 23
B.4 Warm-up: Append-only Security w.r.t. Individual Proofs . . . 23
B.5 Append-only Security for Aggregated Proofs . . . 24

1 Introduction

An *authenticated dictionary (AD)* scheme securely outsources storage of a set of *keys-value pairs* to an *untrusted prover*. In this setting, the prover can convince any *verifier*, who has a succinct *digest* of the dictionary, that a key has a particular value in the outsourced dictionary by sending him a *lookup proof* for that key. Authenticated dictionaries are a fundamental building block of numerous applications, including authenticated file systems [15], privacy-preserving web applications [9, 11], cryptocurrencies [30], stateless validation in cryptocurrencies [6, 14] and transparency logs [7, 17, 23, 29]. In this paper, we enhance authenticated dictionaries in two ways.

ADs for Stateless Validation. In Section 4.2, we propose a new *updatable authenticated dictionary (UAD)* which supports a new notion of *cross-incremental proof (dis)aggregation*. Specifically, our UAD supports aggregating many lookup proofs, even if those proofs are with respect to different dictionaries with different digests. Importantly, such a *cross-aggregated proof* can also be disaggregated to recover the original lookup proofs. Cross-incremental aggregation generalizes previous notions of *incremental aggregation* of proofs with respect to the same digest [4] and *one-hop, cross-commitment aggregation* of proofs with respect to different digests [10]. Furthermore, our UAD offers efficiently updatable proofs and digests as well as efficient proof pre-computation. We prove our UAD satisfies *weak key binding* (see Definition B.2), which assumes digests are honestly generated.

Our UAD can be used for stateless, smart contract-based cryptocurrencies. In such systems, the memory of each smart contract is just a dictionary that maps memory locations from $\{0,1\}^{256}$ to their value, and can thus be authenticated using our UAD. Because previous work cannot authenticate dictionaries with large key space, it restricts the smart contract memory to be much smaller ($\{0,1\}^{10}$) and authenticates it using a vector [10]. Our work naturally overcomes this limitation. Furthermore, using cross-incremental aggregation, miners can now *incrementally* build a block's cross-aggregated proof as proofs for different smart contract executions arrive. This makes block proposal faster, as miners do not have to wait for all proofs before starting to aggregate. It also opens up opportunities for aggregating proofs inside the P2P network of the cryptocurrency, reducing communication in the network.

Beyond Stateless Validation. In Section 4.3, we modify our UAD to have stronger security so it can be used in more malicious settings. Specifically, our new construction satisfies *strong key binding* (see Definition 4.3), which means security holds even when digests are maliciously constructed by the adversary. This is the case in many applications, such as transparency logs, authenticated file systems and privacy-preserving web applications. For the transparency log setting, we show our construction supports efficient *append-only proofs* [29] that one dictionary is a subset of another. Furthermore, we add support for *non-membership proofs* of keys that are *not* in the dictionary. As a result, we obtain an *append-only authenticated dictionary (AAD)* which, unlike previous schemes [27, 29], supports updating proofs and *one-hop proof aggregation*.

New Techniques for RSA Accumulators. As a building block for our AAD, we develop techniques for computing and aggregating *RSA non-membership witnesses* across different-but-related *RSA accumulators* [2]. We also develop a faster algorithm for witness extraction in Boneh et al.'s *proof-of-knowledge of co-prime roots (PoKCR)* protocol (see Section 2.2). We believe these techniques could be of independent interest.

1.1 Related Work

We survey recent work that builds authenticated dictionaries from group-theoretic assumptions, rather than traditional Merkle-based techniques, which are inherently expensive to aggregate.

Streaming Authenticated Data Structure (SADS). Papamanthou et al. [20] present an elegant lattice-based construction that generalized Merkle trees using an algebraic hash function. However, their construction does not support aggregating proofs nor append-only proofs, making it ill-suited both for stateless validation and transparency logs.

Authenticated Hash Tables (AHTs). Papamanthou et al. [21] build *authenticated hash tables (AHTs)* from both bilinear accumulators [18] and RSA accumulators [2]. Their tree-based approach uses an accumulator rather than a normal collision-resistant hash function for authentication. However, their construction assumes digests are generated honestly and is thus only secure under weak key binding. Also, they do not support proof updates, proof aggregation, nor append-only proofs.

AD scheme	Aggregatable π's?	Binding	Updatability?	Update hint-free?	Non-memb. π's?	Append-only π's?	Prove all fast?
Merkle tree	×	Strong	I	×	✓	×	✓
SADS [20]	×	Strong	DI	✓	✓	×	✓
AHTs [21]	×	Weak	×	n/a	✓	×	✓
KVC_1 [3]	One-hop	Strong	DI	×	✓	×	✓
KVC_2 [3]	One-hop	Weak	DI	×	✓	×	✓
AAD [27]	×	Strong	×	n/a	✓	✓	✓
Aardvark [14]	One-hop	Weak	DI	×	✓	×	×
KVaC [1]	One-hop	Weak	DI	✓	×	×	×
Our **UAD**	Cross-incr.	Weak	ADIX	×	✓*	✓	✓
Our **AAD**	One-hop	Strong	aDI	×	✓	✓	✓

*Our UAD supports non-membership proofs, but they can only be "one-hop" aggregated.

Table 1: Comparison of our AD with other ADs based on lattices, pairing-friendly or hidden-order groups. In "Updatability", we indicate updatability of: individual lookup proofs (I), aggregated lookup proofs (A), aggregated lookup proofs, but only after changes to existing keys (a), cross-aggregated lookup proofs (X) and digests (D).

Key-Value Commitments (KVC). Boneh et al. [3] briefly describe two AD constructions from their RSA-based vector commitment scheme. Their first construction, which we dub KVC_1, satisfies strong key binding (see Definition 4.3) while their second construction, KVC_2, relaxes security to weak key binding for efficiency gains. Unlike our UAD construction, they do not support incremental (dis)aggregation, nor cross-aggregation of proofs. Also, they cannot update aggregated proofs and do not support append-only proofs.

AADs. Tomescu et al. [29] give an append-only authenticated dictionary (AAD) construction from bilinear accumulators [18]. Later on, Tomescu [27] generalizes this construction to RSA accumulators. Both constructions support append-only proofs and non-membership proofs and can be used, in theory, for transparency logs. They also use an amortization technique [19] to precompute all lookup proof and all append-only proofs in quasilinear time and support appending to the dictionary in *amortized* polylogorithmic time. Importantly, their AAD supports pre-computing *all* non-membership proofs. Unfortunately, neither construction supports proof aggregation, nor proof updates (individual or aggregated). Also, their RSA-based AAD requires $O(\lambda)$ more exponentiations per key-value pair added to the dictionary than ours.

Aardvark. Leung et al. [14] propose Aardvark, an authenticated dictionary built on top of the cross-aggregatable VC by Gorbunov et al [10]. Aardvark supports one-hop aggregation of proofs and relies on pairing-friendly groups. As a result, Aardvark is fast in practice and has aggregated proof sizes of only 48 bytes. In contrast, our UAD's aggregated proof, consisting of two hidden-order group elements, is at least 512 bytes. However, Aardvark has a few drawbacks. First, the digest's size is non-constant: n/B group elements, where B is a construction-specific bucket size and n is the dictionary size. Second, Aardvark is only secure under *weak key binding*, by assuming dictionary digests are generated honestly. Third, although an Aardvark dictionary proof is built by cross-aggregating VC proofs, Aardvark does not explore further cross-aggregating their dictionary proofs themselves. Even if cross-aggregation were possible in Aardvark, it would not be incremental and it would not support proof disaggregation. Last, Aardvark does not support fast pre-computation of proofs.

KVaC. Agrawal and Raghumaran also introduced an authenticated dictionary from the RSA assumption, which they dub *KVaC*. Their elegant construction also builds on top of [4, 5, 13] and has the advantage of not requiring any auxiliary information (i.e., *update hints*) for updating proofs and digests. Furthermore, KVaC also supports one-hop proof aggregation. This makes KVaC very useful for stateless validation. However, their construction has a few drawbacks. First, they do not support incremental (dis)aggregation nor cross-commitment aggregation, which helps with smart contract based validation [10]. Second, they do not discuss updating aggregated proofs, which helps with stateless validation in the smart contract setting. Third, they do not explore fast proof pre-computation, which is a necessary ingredient for proof-serving nodes in stateless cryptocurrencies [6, 28]. Fourth, their construction only satisfies weak key binding (see Definition B.2), which is sufficient in the stateless cryptocurrency setting, but not in other settings, such as transparency logs. Fifth, their construction does not support non-membership proofs for keys that were never inserted in the dictionary, which is also necessary for transparency logs. We believe it would

$\mathsf{ShamirTrick}\left(\left(g^{1/x_i}\right)_{i\in[n]}, \boldsymbol{x}\right) \rightarrow g^{1/x^*}$

if $n = 1$, **then**

  return $x_1, g^{1/x_1}$

else

  Split $[n]$ into halves L and R

  $\boldsymbol{g}_L \leftarrow [g^{1/x_1}, \ldots, g^{1/x_{n/2}}]$,

  $\boldsymbol{g}_R \leftarrow [g^{1/x_{n/2+1}}, \ldots, g^{1/x_n}]$

  $(x_L, g^{\frac{1}{x_L}}) \leftarrow \mathsf{ShamirTrick}\left(\boldsymbol{g}_L, \boldsymbol{x}_L\right)$

  $(x_R, g^{\frac{1}{x_R}}) \leftarrow \mathsf{ShamirTrick}\left(\boldsymbol{g}_R, \boldsymbol{x}_R\right)$

  $(a, b) \rightarrow \mathsf{EEA}(x_L, x_R)$

  ▷ i.e., $ax_L + bx_R = 1$.

  $g^{\frac{1}{x_L \cdot x_R}} \rightarrow \left(g^{\frac{1}{x_L}}\right)^b \left(g^{\frac{1}{x_R}}\right)^a$

  return $\left(x_L \cdot x_R, g^{\frac{1}{x_L \cdot x_R}}\right)$

endif

$\mathsf{RootFactor}(g, \boldsymbol{x}) \rightarrow \left(g^{x^*/x_i}\right)_{i\in[n]}$

if $n = 1$, **then return** g, **else**

  Split $[n]$ into halves L and R

  $g_L \leftarrow g^{\prod_{i\in L} x_i}$

  $g_R \leftarrow g^{\prod_{i\in R} x_i}$

  $L \leftarrow \mathsf{RootFactor}(g_L, \boldsymbol{x}_R)$

  $R \leftarrow \mathsf{RootFactor}(g_R, \boldsymbol{x}_L)$

  return $L||R$

endif

$\mathsf{MultiRootExp}(\boldsymbol{\alpha}, \boldsymbol{x}) \rightarrow y$

if $n = 1$, **then return** α_1, **else**

  Split $[n]$ into halves L and R

  $\alpha_L \leftarrow g^{\prod_{i\in L} x_i}$

  $\alpha_R \leftarrow g^{\prod_{i\in R} x_i}$

  $x^*_L \leftarrow \prod_{i\in L} x_i$

  $x^*_R \leftarrow \prod_{i\in R} x_i$

  $L \leftarrow \mathsf{MultiRootExp}(\alpha_L, \boldsymbol{x}_R)$

  $R \leftarrow \mathsf{MultiRootExp}(\alpha_R, \boldsymbol{x}_L)$

  return $L^{x^*_R} \cdot R^{x^*_L}$

endif

Figure 1: We frequently rely on these algorithms, where $\boldsymbol{x} = [x_1, \ldots, x_n], n = 2^k, x^* = \prod_{i\in[n]} x_i$ and $|x_i| = O(\ell)$ bits.

be interesting to see to what extent our techniques can enhance KVaCs and vice-versa.

2 Preliminaries

Notation. Let λ denote our security parameter. Let ℓ denote the length (in bits) of vector elements and of dictionary values. Let $\mathbb{G}_?$ denote a hidden-order group and g be a random group element in $\mathbb{G}_?$. Let $H : \{0,1\}^* \rightarrow \mathsf{Primes}_{\ell+1}$ be a collision-resistant hash function that outputs $(\ell+1)$-bit primes. We typically use bolded variables $\boldsymbol{x}$ to denote vectors $[x_1, x_2, \ldots, x_n]$ of elements. We also use $\boldsymbol{x}_I = (x_i)_{i\in I}$ to denote an I-subvector of $\boldsymbol{x}$ with only the values at indices in I.

2.1 Algorithms

Our work makes frequent use of the following algorithms.

Extended Euclidean Algorithm (EEA). Given two integers x, y such that $\gcd(x, y) = 1$, $(a, b) \leftarrow \mathsf{EEA}(x, y)$ returns *Bézout coefficients* (a, b) such that $ax + by = 1$ in $O(m \log^2 m \log\log m)$ bit operations [25], where $m = \max(|x|, |y|)$. Importantly, the returned coefficients satisfy $a \le y$ and $b \le x$.

Shamir's trick. For any $g \in \mathbb{G}_?$, given integers x, y and $g^{\frac{1}{x}}, g^{\frac{1}{y}}$, this classic algorithm by Shamir [26] efficiently computes $g^{\frac{1}{xy}}$ as $\left(g^{1/x}\right)^b \left(g^{1/y}\right)^a$ where a, b are Bézout coefficients such that $ax + by = 1$. If $\ell = \max\{|x|, |y|\}$, the cost is dominated by $O(\ell)$ group operations. Shamir's trick can be extended to $n > 2$ inputs $\{x_i, g^{1/x_i}\}_{i\in[n]}$ by building a binary computation tree whose leaves are the individual $(x_i, g^{1/x_i})$'s (see Fig. 1). Next, if a node's left and right children store $(x_L, g^{1/x_L})$ and $(x_R, g^{1/x_R})$, respectively, then that node computes and stores $(x_L \cdot x_R, g^{\frac{1}{x_L x_R}})$. This way, the root will compute the desired $(\prod_{i\in[n]} x_i, g^{\frac{1}{\prod_{i\in[n]} x_i}})$. Shamir's (recursive) trick on n inputs takes $T(n) = 2T(n/2) + O(\ell n) = O(\ell n \log n)$ group operations which, in practice, dominate the cost of EEAs and integer multiplications.

$\mathsf{RootFactor}$. Given $g \in \mathbb{G}_?$ and integers $\boldsymbol{x} = [x_1, \ldots, x_n]$, outputs g^{x^*/x_i} for all x_i, where $x^* = \prod_{i\in[n]} x_i$. This algorithm takes $O(\ell n \log n)$ group operations, where $\ell = \max_i |x_i|$, and was introduced by Sander et al [24].

MultiRootExp. Given group elements $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_n]$ and integers $\boldsymbol{x} = [x_1, \ldots, x_n]$, outputs $y = \prod_{i\in[n]} \alpha_i^{x^*/x_i}$ where $x^* = \prod_{i\in[n]} x_i$. This algorithm takes in $O(\ell n \log n)$ group operations, where $\ell = \max_i |x_i|$, and was introduced by Boneh et al. [3] (but under the name **MultiExp**).

2.2 Proofs of Knowledge of Co-prime Roots (PoKCR)

The PoKCR protocol by Boneh et al. [3] proves knowledge of w_i's such that $w_i^{x_i} = \alpha_i$ to a verifier that has the α_i's and x_i's. In other words, the protocol proves the following relation holds:

$$\mathcal{R}_{\mathsf{PoKCR}} = \{[\alpha_1, \ldots, \alpha_n] \in \mathbb{G}_?^n, [x_1, \ldots, x_n] \in \mathbb{G}_?^n : w_i^{x_i} = \alpha_i, \forall i \in [n]\} \tag{1}$$

To prove the relation holds, $\mathsf{PoKCR.Prove}(\boldsymbol{\alpha}, \boldsymbol{x}, \boldsymbol{w})$ simply returns $W = \prod_{i\in[n]} w_i$. To verify, $\mathsf{PoKCR.Ver}(\boldsymbol{\alpha}, \boldsymbol{x}, W)$ first computes $x^* = \prod_{i\in[n]} x_i$ and then checks if $W^{x^*} \stackrel{?}{=} \mathsf{MultiRootExp}(\boldsymbol{\alpha}, \boldsymbol{x}) = \prod_{i\in[n]} \alpha_i^{x^*/x_i}$.

Knowledge soundness. Boneh et al. [3] argue the PoKCR protocol is a proof of knowledge by showing the verifier can extract any w_i's given a W, $\boldsymbol{x}$ and $\boldsymbol{\alpha}$ as follows. Let $z_i = x^*/x_i$ and $z_{i,j} = x^*/(x_i x_j)$. Since W is valid, $W^{x*} = \prod_{i\in[n]} \alpha_i^{z_i}$. This means $W^{x^*} = (\prod_{j\in[n]\setminus\{i\}} \alpha_j^{z_j})\alpha_i^{z_i} = (\prod_{j\in[n]\setminus\{i\}} \alpha_j^{z_{i,j}})^{x_i}\alpha_i^{z_i} \stackrel{\mathsf{def}}{=} A_j^{x_i}\alpha_i^{z_i}$. Also note that $W^{z_i} = (W^{x^*})^{1/x_i} = (A_j^{x_i}\alpha_i^{z_i})^{1/x_i} = A_j(\alpha_i^{z_i})^{1/x_i}$. Thus, we can let $u = W^{z_i}/A_j = (\alpha_i^{z_i})^{1/x_i} = (\alpha_i^{1/x_i})^{z_i}$. Next, note that a $\mathsf{ShamirTrick}(u, \alpha_i, x_i, z_i)$ on $u = (\alpha_i^{1/x_i})^{z_i} = (\alpha_i^{x^*/x_i})^{1/x_i}$ and $\alpha_i = (\alpha_i^{x^*/x_i})^{1/z_i}$, gives exactly α_i^{1/x_i}.

Time to extract. Assume each x_i is at most ℓ bits. Note that $A_j = \prod_{j\in[n]\setminus\{i\}} \alpha_j^{z_{i,j}} = \mathsf{MultiRootExp}(\boldsymbol{\alpha}_{[n]\setminus\{i\}}, \boldsymbol{x}_{[n]\setminus\{i\}})$ can be computed in $O(\ell n \log n)$ group operations. This dominates the time to compute W^{z_i} and to do the Shamir trick.

2.3 Proofs of Knowledge of Exponent (PoKE)

The PoKE protocol by Boneh et al. [3] proves knowledge of an $x \in \mathbb{Z}$ such that $w = u^x$ to a verifier who has w and u. In other words, it proves that the relation $\mathcal{R}_{\mathsf{PoKE}} = \{w \in \mathbb{G}_?, u \in \mathbb{G}_? : w = u^x\}$ holds. The protocol makes use

$\mathsf{PoKE.Prove}(w = u^x, u \in \mathbb{G}_?, x \in \mathbb{Z}) \to \pi^{\mathsf{PoKE}}$	$\mathsf{PoKE.Ver}(w \in \mathbb{G}_?, u \in \mathbb{G}_?, \pi^{\mathsf{PoKE}}) \to \{0,1\}$
$g \leftarrow H_{\mathbb{G}_?}(u, w)$ **and** $z = g^x$	Parse $(z, Q, r) \leftarrow \pi^{\mathsf{PoKE}}$
$\ell \leftarrow H_{\mathsf{Primes}_{2\lambda}}(u, w, z)$ **and** $\alpha \leftarrow H_{2\lambda}(u, w, z, \ell)$	$g \leftarrow H_{\mathbb{G}_?}(u, w)$
Let $q \in \mathbb{Z}, r \in [0, \ell)$ s.t. $x = q\ell + r$.	$\ell \leftarrow H_{\mathsf{Primes}_{2\lambda}}(u, w, z)$ **and** $\alpha \leftarrow H_{2\lambda}(u, w, z, \ell)$
Let $Q = (ug^\alpha)^q$ **and** $\pi^{\mathsf{PoKE}} = (z, Q, r)$.	**return** $Q^\ell (ug^\alpha)^r \stackrel{?}{=} wz^\alpha$

Figure 2: Non-interactive proof-of-knowledge of exponent (PoKE) protocol [3].

of three different hash functions $H_{\mathbb{G}_?} : \mathbb{G}_?^2 \to \mathbb{G}_?$, $H_{\mathsf{Primes}_{2\lambda}} : \mathbb{G}_?^3 \to \mathsf{Primes}_{2\lambda}$ and $H_{2\lambda} : \mathbb{G}_?^3 \times \{0,1\}^{2\lambda} \to \{0,1\}^{2\lambda}$ modeled as random oracles. Correctness holds because:

$$Q^\ell u^r g^{\alpha r} = (u^q g^{\alpha q})^\ell u^r g^{\alpha r} = u^{q\ell} g^{\alpha q \ell} u^r g^{\alpha r} = u^{q\ell + r} g^{\alpha q \ell + \alpha r} = u^x g^{\alpha x} = wz^\alpha \tag{2}$$

Knowledge soundness holds in the generic group model (see [3, Proof of Thm. 3 in Appendix C.2]). A PoKE proof contains two elements in $\mathbb{G}_?$ and a 2λ-bit number r and can be computed in $O(|x|)$ group operations.

2.4 RSA Accumulators

RSA accumulators were introduced by Benaloh and de Mare [2]. Given a set $T = \{e_1, e_2, \ldots, e_n\}$ of *unique*, ℓ-bit *prime* elements, an RSA accumulator is a commitment $c = g^{\prod_{e_i \in T} e_i}$ to the set T. Here, g is a random element of $\mathbb{G}_?$. RSA accumulators support proving *membership* of any e_i, as well as *non-membership* of any $e \notin T$. Furthermore, RSA accumulators support proving *subset* relations $S \subseteq T$ as well as *disjointness* relations $X \cap T = \varnothing$.

2.4.1 Membership and subset witnesses

To prove that $e_i \in T$ w.r.t. the accumulator c, the prover computes a *membership witness* $w_i = g^{\prod_{e_j \in T, e_j \neq e_i}} = c^{1/e_i}$. The verifier checks the witness as $w_i^{e_i} \stackrel{?}{=} c$. Sander et al. [24] showed that all of T's membership witnesses can be computed in $O(\ell n \log n)$ time as $(w_i)_{i\in[n]} \leftarrow \mathsf{RootFactor}(g, [e_1, e_2, \ldots e_n])$ (see Fig. 1). Furthermore, a *subset witness* for several elements $S \subseteq T$ can be computed as $w_S = g^{\prod_{e_j \in T, e_j \notin S}} = c^{1/\prod e_j \in S}$. The verifier checks the witness as $w_i^{\prod_{e_j \in S}} \stackrel{?}{=} c$. Note that such a witness can be aggregated from the individual witnesses c^{1/e_j}'s as $w_S = \mathsf{ShamirTrick}((c^{1/e_j})_{j\in S}, (e_j)_{j\in S})$.

Cross-accumulator aggregation. Boneh et al. [3] show it is possible to *cross-aggregate* membership witnesses (and thus subset witnesses too) with respect to *different* accumulators, under the restriction that the elements being witnessed are pairwise co-prime. Recall that, given n witnesses w_i, each for an element e_i w.r.t. a different accumulator a_i, a cross-aggregated witness should prove that $w_i^{e_i} = a_i, \forall i \in [n]$. But, as Boneh et al. observe, this is equivalent to proving the PoKCR relation from Eq. (1) holds with $x_i = e_i$ and $\alpha_i = a_i$. Thus, assuming all pairs of e_i's are co-prime, a PoKCR proof can be used to aggregate all w_i's into a single $w = \prod_{i\in[n]} w_i$ and verified using $\mathsf{PoKCR.Ver}(\boldsymbol{a}, \boldsymbol{x}, w)$. In Section 3.1, we give a faster algorithm for extracting all witnesses w_i from such a cross-aggregated witness w.

2.4.2 Non-membership and disjointness witnesses

Li et al. [16] introduced *non-membership witnesses* for RSA accumulators. A non-membership witness for e w.r.t. to accumulator $c = g^u$ is $(a, B = g^b)$ where $(a, b) = \mathsf{EEA}(u, e)$, such that $au + be = 1$. The witness is verified by checking if $c^a B^e = g$ in $O(\ell)$ group operations. Li et al. also show how to update non-membership witnesses after additions or deletions to the accumulator (see Fig. 3).

$\mathsf{Acc.NonMemWitUpdAdd}\,(c, a, B, x, x')$	$\mathsf{Acc.NonMemWitUpdDel}\,(c, c', a, B, x, x')$
Let $(s, t) = \mathsf{EEA}(x, x')$	▷ *Note that $c' = c^{1/x'}$ is the updated accumulator without x'.*
Let $q \in \mathbb{Z}, r \in [0, x)$ s.t. $at = qx + r$	Let $q \in \mathbb{Z}, r \in [0, x)$ s.t. $x'a = qx + r$
return $(a', B') = (r, c^{qx'+as}B)$	**return** $(a', B') = (r, (c')^q B)$

Figure 3: Algorithms by Li et al. [16] for updating an RSA non-membership witness in $O(\ell)$ group operations. For an intuitive explanation, see Appendix A.

Boneh et al. [3] give algorithms for aggregating several non-membership witnesses $(a_i, B_i)_{i\in[n]}$ for e_i w.r.t. the same accumulator c into a single *disjointness witness* for all e_i's, which we describe in Fig. 4. Specifically, $\mathsf{Acc.NonMemWitAgg}^*$ returns an $O(\ell n)$-sized disjointness witness and $\mathsf{Acc.NonMemWitAgg}$ returns a *constant-sized* disjointness witness by using a PoKE proof.

2.5 Incrementally Aggregatable Vector Commitments from RSA

Our work builds upon Catalano and Fiore's RSA-based vector commitment (VC) scheme [5], which was later enhanced by Lai and Malavolta [12] with *subvector proofs* and by Campanelli et al. [4] with *proof (dis)aggregation*, efficient proof pre-computation and constant-sized auxiliary information for updates.

Public parameters. Let $e_i = H(i)$ be distinct primes corresponding to each position $i \in [n]$ in the vector. We often use $e_I = \prod_{i\in I} e_i$ for any set $I \subseteq [n]$. Let $S = g^{\prod_{j\in[n]} e_j}$ and $S_i = S^{1/e_i} = g^{\prod_{j\in[n]\setminus\{i\}} e_j}$. Note that S can be regarded as an RSA accumulator [2] and S_i as an RSA membership witness for e_i. Also note that the vector can be extended with new positions by adding more primes $(e_{n+1}, e_{n+2}, \ldots)$ to S (and thus to the S_i's too). The public parameters consist of a *proving key* $\mathsf{prk} = (g, H)$ and a *verification key* $\mathsf{vrk} = (g, H)$. The S_i's are called *update keys* since they serve as auxiliary information when updating digests and proofs.

Commitment. A commitment to $\boldsymbol{v} = (v_i)_{i\in[n]}$ consists of $S = g^{e_{[n]}}$ and $c = \prod_{i\in[n]} S_i^{v_i}$. The committing time is dominated by computing all S_i's in $O(\ell n \log n)$ group operations via $\mathsf{RootFactor}(g, [e_1, \ldots, e_n])$. The commitment

$\underline{\mathsf{Acc.NonMemWitAgg}^*\left(c,(a_i,B_i,e_i)_{i\in[n]}\right)} \to (a,B)$	$\underline{\mathsf{Acc.NonMemWitAgg}\left(c,(a_i,B_i,e_i)_{i\in[n]}\right)} \to \pi$
if $n=1$, **then return** (a_1,B_1), **else**	$(a,B) \leftarrow \mathsf{Acc.NonMemWitAgg}^*(c,(a_i,B_i,e_i)_{i\in[n]})$
Split $[n]$ into two halves L and R	**return** $\pi = (c^a, \pi_a^{\mathsf{PoKE}} = \mathsf{PoKE.Prove}(c^a,c,a), B)$
$(a_L,B_L) \leftarrow \mathsf{Acc.NonMemWitAgg}^*\left(c,(a_i,b_i,e_i)_{i\in L}\right)$	
$(a_R,B_R) \leftarrow \mathsf{Acc.NonMemWitAgg}^*\left(c,(a_i,b_i,e_i)_{i\in R}\right)$	$\underline{\mathsf{Acc.NonMemWitAggVer}\left(c,\pi,(e_i)_{i\in[n]}\right)} \to \{0,1\}$
Let $e_L = \prod_{i\in L} e_i$ and $e_R = \prod_{i\in R} e_i$	Let $e_I = \prod_{i\in[n]} e_i$
$(s,t) \leftarrow \mathsf{EEA}(e_L,e_R)$	Parse $(w,\pi_a^{\mathsf{PoKE}},B) \leftarrow \pi$
$a' \leftarrow t\cdot a_L \cdot e_R + s\cdot a_R\cdot e_L$	**return**
Let $q\in\mathbb{Z}, r\in[0,e_Le_R)$ s.t. $a' = q\cdot(e_Le_R)+r$	$1 \stackrel{?}{=} \mathsf{PoKE.Ver}(w,c,\pi_a^{\mathsf{PoKE}}) \land$
return $(r, c^q(B_L)^t(B_R)^s)$	$g \stackrel{?}{=} wB^{e_I}$

Figure 4: Algorithms by Boneh et al. [3] for aggregating RSA non-membership witnesses. (For intuition, see [3, pg. 18]).

can be updated after extending the vector with a new position v_{n+1} as $S' = S^{e_{n+1}}$ and $c' = c^{e_{n+1}}S^{v_{n+1}}$. Note that the vector can be extended with multiple positions by applying this update sequentially.

The commitment can also be updated after changing any set of values $(v_j)_{j\in J}$ by δ_j, if the individual S_j's are given. Then, the commitment is updated as $S' = S$ and $c' = c\cdot\prod_{j\in J} S_j^{\delta_j}$. If $b=|J|$, this takes $O(\ell b)$ group operations. Alternatively, if an *aggregated update key* $S_J = S^{1/e_J}$ is given, each S_j is first computed via $\mathsf{RootFactor}(S_J,(e_j)_{j\in J})$ in $O(\ell b\log b)$ group operations.

Subvector proofs. An *I-subvector proof* π_I for $\boldsymbol{v}_I = (v_i)_{i\in I}, I\subseteq[n]$ consists of (1) $S_I = S^{\frac{1}{e_I}}$ and (2) $\Lambda_I = \left(\prod_{j\in[n]\setminus I} S_j^{v_j}\right)^{1/e_I} = \prod_{j\in[n]\setminus I} S_{I,j}^{v_j}$, where $S_{I,j} = S_j^{1/e_I}$. Let $b=|I|$. The prover computes (1) $S_I = g^{\prod_{i\in[n]\setminus I} e_i}$ and (2) all $S_{I,j}$'s via $\mathsf{RootFactor}(g,(e_i)_{i\in[n]\setminus I})$. The prover time is dominated by the $O(\ell(n-b)\log(n-b))$ group operations from $\mathsf{RootFactor}$. To verify π_I, one checks if $(S_I)^{e_I} = S$, computes $(S_i)_{i\in I}$ via $\mathsf{RootFactor}(S_I,(e_i)_{i\in I})$ and checks if $c = \Lambda_I^{e_I}\prod_{i\in I} S_i^{v_i}$. The verification time is dominated by the $O(\ell b\log b)$ group operations from $\mathsf{RootFactor}$.

If $|I|=1$, then an I-subvector proof is referred to as an *individual proof.* A useful fact to notice is that π_I is simply the commitment to a vector "without positions I in it," since $S_I = S^{1/e_I}$ and $\Lambda_I = \left(c/\prod_{i\in I} S_i^{v_i}\right)^{1/e_I}$. We often use this observation in Section 4.

(Dis)aggregating proofs. Campanelli et al. introduce *incremental (dis)aggregation* of subvector proofs (see Fig. 5). Specifically, they show how to aggregate π_I,π_J for $\boldsymbol{v}_I$ and $\boldsymbol{v}_J$ into a subvector proof $\pi_{I\cup J}$ for $\boldsymbol{v}_{I\cup J}$ via $\mathsf{CFG.Agg}$. They also show how to disaggregate any proof π_I into a proof $\pi_{I\setminus K}$ for a smaller subvector via $\mathsf{CFG.Disagg}$. Lastly, they give a $\mathsf{CFG.AggManyToOne}$ algorithm that aggregates individual proofs for v_i's, $i\in I$, into a subvector proof for $\boldsymbol{v}_I$ Fig. 5 summarizes these algorithms as well as $\mathsf{CFG.Agg}^*$, a relaxed version of $\mathsf{CFG.Agg}$ that assumes $I\cap J=\varnothing$.

Let $b=\max\{|I|,|J|\}$. Then, $\mathsf{CFG.Agg}^*$ takes $O(\ell b\log b)$ group operations. $\mathsf{CFG.AggManyToOne}$ uses $\mathsf{CFG.Agg}^*$ recursively and takes $T(b) = 2T(b/2)+O(\ell b\log b) = O(\ell b\log^2 b)$ group operations. $\mathsf{CFG.Disagg}$ takes $O(\ell(|I|-|K|)\log(|I|-|K|))$ group operations. Since the worst-case time to disaggregate π_I and π_J is $O(\ell b\log b)$ group operations, $\mathsf{CFG.Agg}$ is also $O(\ell b\log b)$.

Updating $\boldsymbol{I}$-subvector proofs A proof $\pi_I = (S_I,\Lambda_I)$ for $(v_i)_{i\in I}$ can be updated to $\pi_I' = (S_I',\Lambda_I')$ after the vector changes. Previous work [4,5] shows how to update proofs if (1) several v_j's change or if (2) the vector is extended with extra positions $n+1,n+2,\ldots,n+\Delta$. In Section 4.2, we show how to handle the case where positions in the vector are "removed," which is necessary for dictionaries.

Case 1: Extending vector with new values $v_{n+1},\ldots,v_{n+\Delta}$: In this case, we have:

$$S_I' = S_I^{\prod_{j\in[\Delta]} e_{n+j}} \qquad \Lambda_I' = \Lambda_I^{\prod_{j\in[\Delta]} e_{n+j}} \prod_{j\in[\Delta]}\left((S_I')^{1/e_{n+j}}\right)^{v_{n+j}} \tag{3}$$

Note that this takes $O(\ell\Delta)$ group operations if done sequentially (as described in the commitment update paragraph above) rather than using $\mathsf{RootFactor}$ to compute all $S_K'^{1/e_{n+j}}$'s.

$\underline{\mathsf{CFG.Agg}^* (I, J, \boldsymbol{v}_I, \boldsymbol{v}_J, \pi_I, \pi_J)} \to \pi_{I \cup J}$

$\triangleright$ *Assume proofs verify against digest* $d = (c, S = g^{\prod_{i \in [n]} e_i})$

Parse $(S_I, \Lambda_I) \leftarrow \pi_I$ and $(S_J, \Lambda_J) \leftarrow \pi_J$

$\triangleright$ *where* $S_I^{e_I} = S, \quad \Lambda_I = \prod_{j \in [n] \setminus I} S_{I,j}^{v_j}, \quad \Lambda_I^{e_I} \prod_{i \in I} S_i^{v_i} = c$

$$S_{I \cup J} \leftarrow \mathsf{ShamirTrick}(S_I, S_J, e_I, e_J), \text{ s.t. } S_{I \cup J}^{e_I e_J} = S$$

$$(S_{I,j})_{j \in J} \leftarrow \mathsf{RootFactor}(S_{I \cup J}, (e_j)_{j \in J}), \text{ s.t. } S_{I,j} = S_I^{1/e_j}$$

$$(S_{J,i})_{i \in I} \leftarrow \mathsf{RootFactor}(S_{I \cup J}, (e_i)_{i \in I}), \text{ s.t. } S_{J,i} = S_J^{1/e_i}$$

$$\Lambda_I^* \leftarrow \frac{\Lambda_I}{\prod_{j \in J} S_{I,j}^{v_j}} = \frac{\prod_{j \in [n] \setminus I} S_{I,j}^{v_j}}{\prod_{j \in J} S_{I,j}^{v_j}} = \prod_{\substack{j \in [n] \\ j \notin I \cup J}} S_{I,j}^{v_j}$$

$$\Lambda_J^* \leftarrow \frac{\Lambda_J}{\prod_{i \in I} S_{J,i}^{v_i}} = \frac{\prod_{i \in [n] \setminus J} S_{J,i}^{v_i}}{\prod_{i \in I} S_{J,i}^{v_i}} = \prod_{\substack{i \in [n] \\ i \notin I \cup J}} S_{J,i}^{v_i}$$

$$\Lambda_{I \cup J} \leftarrow \mathsf{ShamirTrick}(\Lambda_I^*, \Lambda_J^*, e_I, e_J) = \prod_{\substack{k \in [n] \\ k \notin I \cup J}} S_{I \cup J, k}^{v_k}$$

return $\pi_{I \cup J} = (S_{I \cup J}, \Lambda_{I \cup J})$

$\underline{\mathsf{CFG.AggManyToOne}(I, \boldsymbol{v}_I, (\pi_i)_{i \in I})} \to \pi_I$

if $|I| = 1$, **then return** π_1, **else**

Split I into two halves L and R

$\pi_L \leftarrow \mathsf{CFG.AggManyToOne}(L, v_L, (\pi_i)_{i \in L})$

$\pi_R \leftarrow \mathsf{CFG.AggManyToOne}(R, v_R, (\pi_i)_{i \in R})$

$\pi_I \leftarrow \mathsf{CFG.Agg}^*(L, R, v_L, v_R, \pi_L, \pi_R)$

return π_I

endif

$\underline{\mathsf{CFG.Disagg}(I, K, \boldsymbol{v}_I, \pi_I = (S_I, \Lambda_I))} \to \pi_K$

$S_K \leftarrow S_I^{e_{I \setminus K}} = S_I^{e_I / e_K} = S^{1/e_K}$

$(S_{K,i}) \leftarrow \mathsf{RootFactor}(S_I, (e_i)_{i \in I \setminus K})$

$\Lambda_K \leftarrow \Lambda_I^{e_{I \setminus K}} \prod_{i \in I \setminus K} S_{K,i}^{v_i}$

return $\pi_K = (S_K, \Lambda_K)$

$\underline{\mathsf{CFG.Agg}(I, J, \boldsymbol{v}_I, \boldsymbol{v}_J, \pi_I, \pi_J)} \to \pi_{I \cup J}$

$\pi_{J \setminus I} \to \mathsf{CFG.Disagg}(J, J \setminus I, \boldsymbol{v}_J, \pi_J)$

$\pi_{I \cup J} \leftarrow \mathsf{CFG.Agg}^*(I, J \setminus I, \boldsymbol{v}_I, \boldsymbol{v}_{J \setminus I}, \pi_I, \pi_{J \setminus I})$

return $\pi_{I \cup J}$

Figure 5: Algorithms by Campanelli et al. [4] for incrementally (dis)aggregating proofs. $\mathsf{CFG.Agg}^*$ assumes $I \cap J = \varnothing$, $\mathsf{CFG.Disagg}$ assumes $K \subset I, K \neq \varnothing$, $\mathsf{CFG.Agg}$ assumes $I \not\subseteq J$ and $J \not\subseteq I$ and $\mathsf{CFG.AggManyToOne}$ assumes $|I| = 2^k$.

Case 2: Changing each $(v_j)_{j \in J}$ *by* δ_j: If $J \subseteq I$, then $\pi'_I = \pi_I$. Otherwise, $S' = S_I$ and $\Lambda'_I = \Lambda_I \prod_{j \in J \setminus I} (S_{I,j})^{\delta_j}$. In order to compute all $S_{I,j}$'s, we must be given either the aggregate update key S_J or the individual S_j's (from which S_J can be computed), First, computes $S_{J \setminus I} = S_J^{e_J / e_{J \setminus I}} = S^{1/e_{J \setminus I}}$ in $O(\ell |J|)$ group operations. Second, computes $S_{I \cup J} = \mathsf{ShamirTrick}(S_I, S_{J \setminus I}, e_I, e_{J \setminus I})$. Third, computes all $S_{I,j}, j \in J \setminus I$'s via $\mathsf{RootFactor}(S_{I \cup J}, (e_j)_{j \in J \setminus I})$.

Precomputing all proofs. All individual proofs can be computed fast in $O(\ell n \log^2 n)$ time via *proof disaggregation* in a recursive manner [4] by disaggregating the proof for the full vector $\boldsymbol{v}$ (i.e., $\pi_{[n]} = (g, 1_{\mathbb{G}_?})$) into a subvector proof for its left half $\boldsymbol{v}_L$ and a subvector proof for its right half $\boldsymbol{v}_R$. Then, this can be repeated recursively to obtain all individual proofs (see $\mathsf{VC.DisaggOneToMany}$ in [4, Fig. 1]).

3 Enhancements to RSA Accumulators

In this section, we give a new technique for performing faster witness extraction in the PoKCR protocol by Boneh et al. [3] from Section 2.2. This helps us verify cross-aggregated proofs faster in our UAD from Section 4.2. Then, we show how to compute *all* RSA non-membership witnesses $(a_i, B_i)_{i \in I}$ for e_i w.r.t. its own accumulator c^{1/e_i}, which we use to precompute proofs fast in our AAD from Section 4.3. Lastly, we also show how to aggregate such witnesses across different accumulators, which we leverage to aggregate lookup proofs in our AAD. The e_i's are assumed to be ℓ-bit primes.

3.1 Faster Witness Extraction in PoKCR

Suppose we have a PoKCR proof W of x_ith roots of $\alpha_i, \forall i \in [m]$ as per Section 2.2. We know that each root $w_i = \alpha_i^{1/x_i}$ can be extracted in $O(\ell m \log m)$ group operations, where ℓ is the max length in bits of the x_i's. This means extracting all w_i's takes $O(\ell m^2 \log m)$ group operations, which can be slow. Here, we give an $O(\ell m \log^2 m)$ time algorithm called $\mathsf{PoKCR.Extract}$ for computing **all** w_i's. The key idea is to split up $[m]$ into halves L and R and show how to extract $W_L = \prod_{i \in L} w_i = \prod_{i \in L} \alpha_i^{1/x_i}$ and $W_R = \prod_{i \in L} \alpha_i^{1/x_i}$. Then, the algorithm can recurse, eventually extracting all individual w_i's.

Let $x_L = \prod_{i \in L} x_i$, $x_R = \prod_{i \in R} x_i$ and $x^* = x_L x_R$. Since W verifies, we have:

$$W^{x^*} = \prod_{i \in [m]} \alpha_i^{x^*/x_i} = \left(\prod_{i \in L} \alpha_i^{x^*/x_i}\right)\left(\prod_{i \in R} \alpha_i^{x^*/x_i}\right) \Leftrightarrow \tag{4}$$

$$W^{x_L x_R} = \left(\prod_{i \in L} \alpha_i^{1/x_i}\right)^{x_L x_R} \left(\prod_{i \in R} \alpha_i^{1/x_i}\right)^{x_R x_L} \Rightarrow \tag{5}$$

$$W^{x_R} = (W_L)^{x_R} (W_R)^{x_R} \textbf{ and } W^{x_L} = (W_L)^{x_L} (W_R)^{x_L} \tag{6}$$

Note that the following terms are computable in $O(\ell m \log m)$ group operations:

$$(W_L)^{x_L} = \left(\prod_{i \in L} \alpha_i^{1/x_i}\right)^{x_L} = \mathsf{MultiRootExp}(\boldsymbol{\alpha}_L, \boldsymbol{x}_L) \tag{7}$$

$$(W_R)^{x_R} = \left(\prod_{i \in R} \alpha_i^{1/x_i}\right)^{x_R} = \mathsf{MultiRootExp}(\boldsymbol{\alpha}_R, \boldsymbol{x}_R) \tag{8}$$

Next, W^{x_L} and W^{x_R} can be computed in $O(\ell m)$ group operations. This means $W_R^{x_L} = W^{x_L}/(W_L)^{x_L}$ and $W_L^{x_R} = W^{x_R}/(W_R)^{x_R}$ can be computed too! Thus, we can use Shamir's trick to obtain W_L and W_R in $O(\ell m \log m)$ group operations.

$$\mathsf{ShamirTrick}(w_L^{x_L}, w_L^{x_R}, x_R, x_L) = \mathsf{ShamirTrick}(w_L^{x^*/x_R}, w_L^{x^*/x_L}, x_R, x_L) = w_L^{x^*/x_L x_R} = w_L \tag{9}$$

$$\mathsf{ShamirTrick}(w_R^{x_L}, w_R^{x_R}, x_R, x_L) = \mathsf{ShamirTrick}(w_R^{x^*/x_R}, w_R^{x^*/x_L}, x_R, x_L) = w_R^{x^*/x_L x_R} = w_R \tag{10}$$

Thus, all roots $w_i = \alpha_i^{1/x_i}$ can be obtained via recursion. Note that our final $\mathsf{PoKCR.Extract}(W, \boldsymbol{\alpha}, \boldsymbol{x})$ algorithm runs in time $T(m) = 2T(m/2) + O(\ell m \log m) = O(\ell m \log^2 m)$ group operations, where $m = |\boldsymbol{\alpha}| = |\boldsymbol{x}|$.

3.2 Computing All Non-membership Witnesses Across Different, Related Accumulators

In this section, we give an algorithm called $\mathsf{Acc.NonMemWitCrossProve}$ that, on input g and $(e_i)_{i \in [n]}$, computes all RSA non-membership witnesses (a_i, B_i)'s for each e_i w.r.t. $c_i = g^{e^*/e_i}$ where $e^* = \prod_{i \in [n]} e_i$. Later on, we use this algorithm in Section 4.3 to precompute all lookup proofs in our AD construction.

First, the algorithm computes a non-membership witness $(a, B = g^b)$ for e^* w.r.t. to the empty accumulator g^1, where $a \cdot 1 + b \cdot e^* = 1$. Next, let $e_L = \prod_{i \in [1, n/2]} e_i$ and $e_R = \prod_{i \in (n/2, n]} e_i$ such that $e = e_L e_R$. The algorithm recursively updates (a, B) into two witnesses: (a_L, B_L) for e_L w.r.t g^{e_R}, and (a_R, B_R) for e_R w.r.t. g^{e_L}. As a result, at the bottom of this recursion tree, the algorithm outputs all non-membership witnesses (a_i, B_i) for e_i w.r.t. g^{e^*/e_i}.

We show how to compute (a_L, B_L) and note (a_R, B_R) follows by symmetry. The algorithm computes Bézout coefficients (s, t) such that $s \cdot e_L + t \cdot e_R = 1 \Rightarrow a = as \cdot e_L + at \cdot e_R$. Then, note that:

$$g = g^a B^e = g^{as \cdot e_L + at \cdot e_R} (B^{e_R})^{e_L} \tag{11}$$

$$= (g^{as})^{e_L} (g^{e_R})^{at} (B^{e_R})^{e_L} = (g^{e_R})^{at} (g^{as} B^{e_R})^{e_L} \tag{12}$$

We could now set $a_L = at$ and $B_L = (g^{as} B^{e_R})$ as the witness, but $|a_L| = |at| = |e_L e_R| + |e_L|$ is too large for efficient recursion. To fix this, we reduce at modulo e_L by writing it as $at = q \cdot e_L + r$ for some integers (q, r), with $r < e_L$.

$$g = (g^{e_R})^{at} (g^{as} B^{e_R})^{e_L} = (g^{e_R})^{r + q \cdot e_L} (g^{as} B^{e_R})^{e_L} \tag{13}$$

$$= (g^{e_R})^r (g^{e_R})^{q \cdot e_L} (g^{as} B^{e_R})^{e_L} = (g^{e_R})^r (g^{q e_R} g^{as} B^{e_R})^{e_L} \tag{14}$$

Now, we can set $a_L = at \bmod e_L = r$ and $B_L = (g^{q e_R} g^{as} B^{e_R})$. Note that $|a_L| = |e_L| = O(\ell n/2)$ and B_L is a group element. Both (a_L, B_L) can be computed in $O(\ell n)$ group operations. Our final $\mathsf{Acc.NonMemWitCrossProve}$ recursive algorithm will take $T(n) = 2T(n/2) + O(\ell n) = O(\ell n \log n)$ group operations.

3.3 Aggregating Non-membership Witnesses Across Different, Related Accumulators

In this subsection, we give an algorithm called $\mathsf{Acc.NonMemWitCrossAgg}$ that aggregates n witnesses $(a_i, B_i)_{i\in[n]}$ for e_i w.r.t. c^{1/e_i} into a single non-membership witness (a, B) for $e^* = \prod_{i\in[n]} e_i$ w.r.t. $c' = c^{1/e^*}$. Later on, we use this algorithm in Section 4.3 to aggregate lookup proofs in our AD construction. We first describe how $\mathsf{Acc.NonMemWitCrossAgg}$ algorithm works when $n = 2$ and then define it recursively for $n > 2$.

Case $n = 2$: Suppose we are given just two witnesses $\pi_0 = (a_0, B_0)$ for e_0 w.r.t. c^{1/e_0}, and $\pi_1 = (a_1, B_1)$ for e_1 w.r.t. c^{1/e_1}, where:

$$(c^{1/e_0})^{a_0}(B_0)^{e_0} = g \qquad\qquad (c^{1/e_1})^{a_1}(B_1)^{e_1} = g \tag{15}$$

We want to aggregate them into a disjointness witness $\pi = (a, B)$ for $e_0 \cdot e_1$ w.r.t. $c' = c^{\frac{1}{e_0\cdot e_1}}$, such that $(c')^a B^{e_0\cdot e_1} = g$. First, we update the witness for e_0 w.r.t. c^{1/e_0} into a witness w.r.t c', which removed e_1, as $(a'_0, B'_0) = \mathsf{Acc.NonMemWitUpdDel}(c^{1/e_0}, c', a_0, B_0, e_0, e_1)$. Second, we update the witness for e_1 w.r.t. c^{1/e_1} into a witness w.r.t c', which removed e_0, as $(a'_1, B'_1) = \mathsf{Acc.NonMemWitUpdDel}(c^{1/e_1}, c', a_1, B_1, e_1, e_0)$. Now, we have two non-membership witnesses for e_0 and e_1 w.r.t. c':

$$(c')^{a'_0}(B'_0)^{e_0} = g \qquad\qquad (c')^{a'_1}(B'_1)^{e_1} = g \tag{16}$$

Next, we can aggregate these witnesses as $(a, B) = \mathsf{Acc.NonMemWitAgg}^*(c', a'_0, B'_0, e_0, a'_1, B'_1, e_1)$ such that $(c')^a B^{e_0 e_1} = g$. The time complexity of this is $O(\ell)$ group operations. Note that $|a| = |e_0 e_1|$, but we reduce it next.

Case $n > 2$: Suppose we are given an arbitrary number of witnesses n. Then, we lay out a computation tree with the (a_i, B_i)'s in the leaves such that every node runs the aggregation explained above on its two children. This results in the root computing a non-membership witness (a, B) for e^* w.r.t $c' = c^{1/e^*}$. Finally, we compute a PoKE proof $\pi_a^{\mathsf{PoKE}} = \mathsf{PoKE.Prove}((c')^a, c', a)$ to "compress" a, which is of size $|a| = |e^*|$. The final aggregated disjointness witness will be $\pi = ((c')^a, \pi_a^{\mathsf{PoKE}}, B)$ and verifies via $\mathsf{Acc.NonMemWitAggVer}(c', \pi, (e_i)_{i\in[n]})$ (see Fig. 4). Thus, our final $\mathsf{Acc.NonMemWitCrossAgg}$ algorithm will take $T(n) = 2T(n/2) + O(\ell n) = O(\ell n \log n)$ group operations.

4 Authenticated Dictionaries from Hidden-Order Groups

In this section, we first formalize authenticated dictionaries. Then, we give two constructions from hidden-order groups, both built on top of the VC scheme from Section 2.5. Our first construction is an *updatable authenticated dictionary (UAD)* for stateless validation in the smart contract setting (see Section 4.2). Our UAD supports a new notion of *cross-incremental proof (dis)aggregation*, which generalizes the notion of *cross-commitment aggregation* by Gorbunov et al. [10]. Our second construction is an *append-only authenticated dictionary (AAD)* for applications with stronger security requirements, such as transparency logs (see Section 4.3). Our AAD additionally supports *non-membership proofs* of keys that are not in the dictionary and *append-only proofs* to prove a dictionary has only been extended with new key-value pairs. However, our AAD's stronger security comes at the cost of downgrading from cross-incremental (dis)aggregation to "one-hop" proof aggregation [3].

Notation. We use $k \in D$ to indicate key k is in the dictionary with some value $v \neq \perp$. When k is not in the dictionary, we say it has value $v = \perp$. We also use $(k, v) \in D$ to indicate key k has value $v \neq \perp$ in the dictionary. We often denote a subset of a dictionary as a pair (K, V) where $V(k)$ stores the value of each key $k \in K$.

4.1 Definitions

The following APIs capture the essential operations in an AD scheme and are useful for formalizing security.

$\mathsf{AD.Setup}(1^\lambda) \to (\mathsf{prk}, \mathsf{vrk})$. Returns the AD's *proving key* and *verification key*.

$\mathsf{AD.Commit}(\mathsf{prk}, D) \to d$. Returns a digest d of the dictionary d.

$\mathsf{AD.ProveLookup}(\mathsf{prk}, D, K) \to \pi$. Returns a *lookup proof* π that each $k \in K$ has value $D(k)$.

$\mathsf{AD.VerLookup}(\mathsf{vrk}, d, K, V, \pi) \to \{0, 1\}$. Verifies the proof π that each $k \in K$ has value $V(k)$ in the dictionary with digest d.

$\mathsf{AD.CrossAgg}((\mathsf{vrk}_i, d_i, K_i, V_i, \pi_i)_{i\in[m]}) \to \pi$. Given lookup proofs π_i for each $k \in K_i$ having value $V_i(k)$ w.r.t. digest d_i built using the public parameters from vrk_i (with all $\mathsf{vrk}_i \neq \mathsf{vrk}_j$), returns a succinct, *cross-aggregated proof* π.

$\mathsf{AD.CrossVerLookup}((\mathsf{vrk}_i, d_i, K_i, V_i)_{i\in[m]}, \pi) \to \{0, 1\}$. Verifies the *cross-aggregated proof* π that each $k \in K_i$ has value $V_i(k)$ in the dictionary with digest d_i, for all $i \in [m]$. When $m = 1$, simply returns $\mathsf{AD.VerLookup}(\mathsf{vrk}_1, d_1, K_1, V_1, \pi)$. In other words, proofs cross-aggregated amongst $m = 1$ digests are just normal aggregated proofs.

$\mathsf{AD.ProveAppendOnly}(\mathsf{prk}, D, D') \to \pi$. Returns an append-only proof π that the dictionary D is a subset of D'.

$\mathsf{AD.VerAppendOnly}(\mathsf{vrk}, d, d', \pi) \to \{0, 1\}$. Verifies the proof π that the dictionary with digest d is a subset of the dictionary with digest d'.

Definition 4.1 (Correctness)**.** An authenticated dictionary scheme is correct if, $\forall$ public parameters $(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{AD.Setup}(1^\lambda)$, $\forall$ dictionaries D with digest $d \leftarrow \mathsf{AD.Commit}(\mathsf{prk}, D)$, the following hold:

Lookup correctness: $\forall$ sets of keys K, if $\pi = \mathsf{AD.ProveLookup}(\mathsf{prk}, D, K)$ and $V(k) = D(k), \forall k \in K$, then $\mathsf{AD.VerLookup}(\mathsf{vrk}, d, K, V, \pi) = 1$.

Append-only correctness: $\forall$ dictionaries D' such that $D \subseteq D'$ and $d' \leftarrow \mathsf{AD.Commit}(\mathsf{prk}, D)$, if $\pi = \mathsf{AD.ProveAppendOnly}(\mathsf{prk}, D, D')$, then $\mathsf{AD.VerAppendOnly}(\mathsf{vrk}, d, d', \pi) = 1$.

Definition 4.2 (Cross-lookup Correctness)**.** An authenticated dictionary scheme has cross-lookup correctness if, $\forall m$, $\forall \left((\mathsf{prk}_i, \mathsf{vrk}_i) \leftarrow \mathsf{AD.Setup}(1^\lambda)\right)_{i\in[m]}$, $\forall$ dictionaries $(D_i)_{i\in[m]}$, each with digest $d_i = \mathsf{AD.Commit}(\mathsf{prk}_i, D_i)$, $\forall$ sets of keys $(K_i)_{i\in[m]}$ with values $V_i(k) = D_i(k), k \in K_i$, if $\pi_i \leftarrow \mathsf{AD.ProveLookup}(\mathsf{prk}_i, D_i, K_i), \forall i \in [m]$ and $\pi \leftarrow \mathsf{AD.CrossAgg}((\mathsf{vrk}_i, d_i, K_i, V_i, \pi_i)_{i\in[m]})$, then $\mathsf{AD.CrossVerLookup}((\mathsf{vrk}_i, d_i, K_i, V_i)_{i\in[m]}, \pi) = 1$.

Definition 4.3 (Strong Key Binding)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$, there exists negligible function $\mathsf{negl}(\cdot)$, such that:

$$\Pr\left[\begin{array}{c} (\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{AD.Setup}(1^\lambda), \\ (d, K, K', V, V', \pi, \pi') \leftarrow \mathcal{A}(1^\lambda, \mathsf{prk}, \mathsf{vrk}) : \\ \mathsf{AD.VerLookup}(\mathsf{vrk}, d, K, V, \pi) = 1 \wedge \\ \mathsf{AD.VerLookup}(\mathsf{vrk}, d, K', V', \pi') = 1 \wedge \\ \exists k \in K \cap K' \text{ s.t. } V(k) \neq V'(k) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: In some applications, such as stateless validation for cryptocurrencies, a weaker definition where the adversary outputs a dictionary and the digest is correctly computed from it suffices (see Definition B.2).

Definition 4.4 (Strong Cross Binding)**.** $\forall M = \mathsf{poly}(\lambda)$, $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$, there exists negligible function $\mathsf{negl}(\cdot)$, such that:

$$\Pr\left[\begin{array}{c} \left((\mathsf{prk}_i, \mathsf{vrk}_i) \leftarrow \mathsf{AD.Setup}(1^\lambda)\right)_{i\in[M]}, \\ ((d_i, K_i, V_i)_{i\in I}, (d'_j, K'_j, V'_j)_{j\in J}, \pi, \pi') \leftarrow \mathcal{A}(1^\lambda, (\mathsf{prk}_i, \mathsf{vrk}_i)_{i\in[M]}) : \\ \mathsf{AD.CrossVerLookup}((\mathsf{vrk}_i, d_i, K_i, V_i)_{i\in I}, \pi) = 1 \wedge \\ \mathsf{AD.CrossVerLookup}((\mathsf{vrk}_j, d'_j, K'_j, V'_j)_{j\in J}, \pi') = 1 \wedge \\ \exists i \in I, j \in J, k \in K_i \cap K'_j \text{ such that} \\ \mathsf{vrk}_i = \mathsf{vrk}_j \ \wedge \ d_i = d'_j \ \wedge \ V_i(k) \neq V'_j(k) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: Note that I, J are subsets of $[M]$. Also note that this definition requires the public parameters of the dictionaries whose proofs are being cross-aggregated to be different. An ideal cross-aggregation scheme should also work for dictionaries built using the same public parameters.

Definition 4.5 (Append-only Security). $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$, there exists negligible function $\mathsf{negl}(\cdot)$, such that:

$$\Pr\left[\begin{array}{c}(\mathsf{prk},\mathsf{vrk}) \leftarrow \mathsf{AD.Setup}(1^\lambda),\\ (d,d',K,K',V,V',\pi,\pi',\pi_\subseteq) \leftarrow \mathcal{A}(1^\lambda,\mathsf{prk},\mathsf{vrk}):\\ \mathsf{AD.VerAppendOnly}(\mathsf{vrk},d,d',\pi_\subseteq)=1\wedge\\ \mathsf{AD.VerLookup}(\mathsf{vrk},d,K,V,\pi)=1\wedge\\ \mathsf{AD.VerLookup}(\mathsf{vrk},d',K',V',\pi')=1\wedge\\ \exists k\in K\cap K' \text{ s.t. } V(k)\neq\perp\wedge V(k)\neq V'(k)\end{array}\right]\leq \mathsf{negl}(\lambda)$$

Observation: This definition can be generalized to work with cross-aggregated proofs too.

4.2 Updatable Authenticated Dictionary for Stateless Validation

At a high level, we obtain an *updatable authenticated dictionary (UAD)* by mapping keys k to primes $e_k = H(k)$ in the VC from Section 2.5 (instead of mapping vector indices i to primes e_i). This idea was developed concurrently in [1], however, our work extends it in a different direction. First, we add *cross-incremental (dis)aggregation of proofs*, which allows *incrementally* cross-aggregating multiple lookup proofs across different digests as well as disaggregating such cross-aggregated proofs. This is useful for stateless validation in the smart contract setting [10]. Second, our construction supports updating aggregated proofs and proof pre-computation. One important caveat is that our cross-aggregation requires that the digests were built using different public parameters. Although this is different than previous work [10], which allows the public parameters to be the same, it can still be used in the smart contract setting, since each contract can easily use its own (constant-sized) public parameters.

We prove our UAD has *weak key binding* (see Definition B.2) in Appendix B.3 under the Strong RSA assumption (see Definition B.1). Similar to previous work [3, 4], our UAD can update proofs and digests, but requires *auxiliary information* related to the keys $k \in K$ that changed. Although eliminating the need for such auxiliary information is important [1], we believe it is worth the cost given that it enables our UAD to support cross-incremental proof (dis)aggregation with the ability to update even cross-aggregated proofs! Furthermore, in the stateless validation setting, this auxiliary information is not too problematic, since it can be easily included in the transaction or served by *proof-serving nodes* [28].

Public parameters. The public parameters remain $\mathsf{prk} = \mathsf{vrk} = (g, H)$, where g is a generator for the hidden-order group $\mathbb{G}_?$ and H is a CRHF that maps keys (not vector indices) to $\ell+1$ bit primes. Values in the dictionary must be ℓ-bit numbers.

Digest. Let D be a dictionary and K be the set of keys with a value $D(k) \neq \perp$ in the dictionary. We often use $e_K = \prod_{k\in K} e_k$ to denote the product of the prime representatives of all keys in K. Let $S = g^{e_K}$ and $S_k = S^{\frac{1}{e_k}}$. The digest $d = (c, S)$ of our UAD resembles the VC from Section 2.5, where $c = \prod_{(k,v)\in D}(S_k)^v$.

Lookup proofs. Similar to Section 2.5, a proof π_k for k having value $v \neq \perp$ in the dictionary D with digest $d = (c, S)$ consists of two parts. The first part is a commitment Λ_k to D without (k, v) in it, where $\Lambda_k = \prod_{(k',v)\in D, k'\neq k}\left((S_{k'})^v\right)^{\frac{1}{e_k}}$. The second part, is an RSA membership witness S_k for k w.r.t. the RSA accumulator S in the digest, where $S_k = S^{\frac{1}{e_k}}$. As before, to verify the proof, one checks if $S = (S_k)^{e_k}$ and if $c = (S_k)^v(\Lambda_k)^{e_k}$.

(Dis)aggregating lookup proofs. The incremental proof (dis)aggregation from Fig. 5 carries over to our construction. This because the $\mathsf{CFG.Agg}$, $\mathsf{CFG.Disagg}$ and $\mathsf{CFG.AggManyToOne}$ algorithms by Campanelli et al. [4] are agnostic to whether $e_i = H(i)$ is obtained from the hash of a vector index i or a dictionary key i. Thus, our aggregated lookup proofs resemble the I-subvector proofs from Section 2.5. Specifically, given lookup proofs $(\pi_k)_{k\in K}$, for each key k having value $v_k \neq \perp$, we can aggregate them as $(S_K, \Lambda_K) \leftarrow \mathsf{CFG.AggManyToOne}(K, (v_k)_{k\in K}, (\pi_k)_{k\in K})$, such that they verify as $S = (S_K)^{e_K}$ and $c = (\Lambda_K)^{e_K}\prod_{k\in K}(S_k)^{v_k}$.

Updating the digest. The digest $d = (c, S)$ can be easily updated given additions of new key-value pairs or changes to existing keys in the dictionary, similar to how the VC is updated in Section 2.5. Additionally, we also show how to update the digest after *removing* keys from the dictionary. This might be useful in the stateless cryptocurrency setting, for example, to delete users whose balance is zero. Specifically, we observe that, given the proof $\pi_k = (S_k, \Lambda_k)$ for the removed key k w.r.t. the old digest $d = (c, S)$, we simply set the new digest to be $d' = (c', S')$ with $c' = \Lambda_k$ and

$S' = S_k$. This is because the proof for the removed key k is exactly the digest of the dictionary without key k in it (see Section 2.5)! Lastly, to update the digest after multiple keys $k \in K$ were removed, we first aggregate the proofs into a (Λ_K, S_K) via CFG.Agg or CFG.AggManyToOne and let the new digest $d' = (\Lambda_K, S_K)$.

Updating proofs. Let $\pi_K = (S_K, \Lambda_K)$ be either an individual or an aggregated lookup proof for all $k \in K$ having value v_k w.r.t. digest $d = (c, S)$. Updating π_K after adding new key-value pairs or after changing existing keys' values works the same as in the VC from Section 2.5. As in the VC, updating after changing existing keys $\hat{k} \in \hat{K}$ requires at least the *aggregate update key* $S_{\hat{K}}$ (or the individual $S_{\hat{k}}$'s).

We show how to update π_K to $\pi'_{K \setminus \hat{K}}$ after removing several keys $\hat{k} \in \hat{K}$. We stress that the updated proof $\pi'_{K\setminus\hat{K}}$ has to be for keys $K \setminus \hat{K}$ since that is the subset of K left in the dictionary after removing all keys in $\hat{K}$. One consequence of this is that, when $K = \hat{K}$, the updated proof $\pi'_{K\setminus\hat{K}}$ would need to be a non-membership proof for all $k \in K$. While our UAD does not support this, our AAD in Section 4.3 does. Thus, we only care about the case when $K \neq \hat{K}$.

In this case, assume we are given an aggregated proof $\pi_{\hat{K}'}$ for each $\hat{k}$ having value $v_{\hat{k}}$ (or individual proofs $\pi_{\hat{k}}$ which can be aggregated into $\pi_{\hat{K}}$). Note that we can aggregate $\pi_{K\cup\hat{K}} = (S_{K\cup\hat{K}}, \Lambda_{K\cup\hat{K}})$ via CFG.Agg($K, \hat{K}$, $(v_k)_{k\in K}, (v_{\hat{k}})_{\hat{k}\in\hat{K}}, \pi_K, \pi_{\hat{K}}$), obtaining a valid proof for all keys $K \cup \hat{K}$ w.r.t. the old digest d. Interestingly, we observe that $\pi_{K\cup\hat{K}}$ is also a valid proof for all keys $K \setminus \hat{K}$ w.r.t. the new digest $d' = (c', S') = (\Lambda_{\hat{K}}, S_{\hat{K}})$. (Recall from above that the new digest d' is just the aggregated proof $\pi_{\hat{K}}$ for the removed keys.) We explain how this works next.

First, partition $K \cup \hat{K}$ into $K \setminus \hat{K}$ and $\hat{K}$ so that $e_{K\cup\hat{K}} = e_{K\setminus\hat{K}} e_{\hat{K}}$. Second, to see how $S_{K\cup\hat{K}}$ verifies against S', note that, since $S_{K\cup\hat{K}}$ verifies against S and $K \cup \hat{K}$, we have $S = (S_{K\cup\hat{K}})^{e_{K\setminus\hat{K}} e_{\hat{K}}} \Leftrightarrow S^{1/e_{\hat{K}}} = (S_{K\cup\hat{K}})^{e_{K\setminus\hat{K}}} \Leftrightarrow S' = (S_{K\cup\hat{K}})^{e_{K\setminus\hat{K}}}$ Third, to see how $\Lambda_{K\cup\hat{K}}$ verifies against c' and $K \setminus \hat{K}$, note that, since $\Lambda_{K\cup\hat{K}}$ verifies against c and and $K \cup \hat{K}$, we have:

$$c = (\Lambda_{K\cup\hat{K}})^{e_{K\setminus K} e_{\hat{K}}} \prod_{k\in K\setminus\hat{K}} (S_k)^{v_k} \prod_{k\in\hat{K}} (S_k)^{v_k} \Leftrightarrow \tag{17}$$

$$c / \prod_{k\in\hat{K}} (S_k)^{v_k} = (\Lambda_{K\cup\hat{K}})^{e_{K\setminus K} e_{\hat{K}}} \prod_{k\in K\setminus\hat{K}} (S_k)^{v_k} \Leftrightarrow \tag{18}$$

$$\left(c / \prod_{k\in\hat{K}} (S_k)^{v_k} \right)^{1/e_{\hat{K}}} = (\Lambda_{K\cup\hat{K}})^{e_{K\setminus K}} \left(\prod_{k\in K\setminus\hat{K}} (S^{1/e_k})^{v_k} \right)^{1/e_{\hat{K}}} \Leftrightarrow \tag{19}$$

$$\Lambda_{\hat{K}} = (\Lambda_{K\cup\hat{K}})^{e_{K\setminus K}} \prod_{k\in K\setminus\hat{K}} \left((S^{1/e_{\hat{K}}})^{v_k} \right)^{1/e_k} \Leftrightarrow \tag{20}$$

$$c' = (\Lambda_{K\cup\hat{K}})^{e_{K\setminus\hat{K}}} \prod_{k\in K\setminus\hat{K}} ((S')^{1/e_k})^{v_k} \tag{21}$$

Thus, we can set $\pi'_{K\setminus\hat{K}} = \pi_{K\cup\hat{K}}$ as the new updated proof for all keys in $K \setminus \hat{K}$.

Cross-incremental (dis)aggregation. Suppose we are given m aggregated proofs π_i for keys $k \in K_i$ with values $v = V_i(k)$, where each π_i is w.r.t. its own digest d_i. We would like to *cross-aggregate* [10] these proofs into a single, constant-sized proof π that verifies against these m digests and the (K_i, V_i)'s. Let $d_i = (c_i, A_i)$ and $\pi_i = (W_i, \Lambda_i)$ where $W_i = A_i^{1/e_{K_i}}$.

Our first difficulty is that we must cross-aggregate all of the RSA subset witnesses W_i into a single witness that verifies w.r.t. to the m different A_i accumulators. This seems possible using a PoKCR proof (see Section 2.2), similar to how RSA accumulator witnesses were cross-aggregated in Section 2.4.1. Unfortunately, these techniques require that $\gcd(e_{K_i}, e_{K_j}) = 1, \forall i, j$, which is not necessarily the case when the K_i's share common keys. Our key observation is that we can ensure this GCD property holds by requiring that all keys in K_i's be hashed with a K_i-specific hash function H_i. This ensures that the same $k \in K_i \cap K_j$ gets mapped to two different prime representatives, which in turn ensures $\gcd(e_{K_i}, e_{K_j}) = 1, \forall i, j$. Put differently, the public parameters for the ith AD with digest d_i must have its own, unique hash function H_i.

We are now left with cross-aggregating the Λ_i's. Recall that each Λ_i would verify as $c_i = \Lambda_i^{e_{K_i}} \prod_{k \in K_i} (A_{i,k})^{V_i(k)} \Leftrightarrow c_i / \prod_{k \in K_i} (A_{i,k})^{V_i(k)} = \Lambda_i^{e_{K_i}}$. Thus, checking all the Λ_i's is equivalent to checking a PoKCR relation holds (i.e., set $w_i = \Lambda_i$, $x_i = e_{K_i}$ and $\alpha_i = c_i / \prod_{k \in K_i} (A_{i,k})^{V_i(k)}$ in Eq. (1)). As a result, we can aggregate the Λ_i's using a PoKCR, just like the W_i's. Finally, the cross-aggregated proof $\pi = (W, \Lambda)$ can be computed in $O(m)$ group operations as $W = \prod_{i \in [m]} W_i$ and $\Lambda = \prod_{i \in [m]} \Lambda_i$.

Verifying cross-aggregated proofs. Let $e^* = \prod_{i \in [m]} e_{K_i}$ where $e_{K_i} = \prod_{k \in K_i} H_i(k)$. One first verifies the cross-aggregated RSA subset witness W via $W^{e^*} \stackrel{?}{=} \prod_{i \in [m]} A_i^{e^*/e_{K_i}} = \mathsf{MultiRootExp}((A_i)_{i \in [m]}, (e_{K_i})_{i \in [m]})$ (similar to Section 2.4.1). Second, the verifier extracts all W_i's from W such that $W_i^{e_{K_i}} = A_i$ via $\mathsf{PoKCR.Extract}(W, (A_i)_{i \in [m]}, (e_{K_i})_{i \in [m]})$ from Section 3.1. Importantly, this takes $O(\ell bm \log^2 (bm))$ group operations (rather than $O(\ell bm^2)$, if done naively). Third, the verifier computes, for all $i \in [m]$, $(A_{i,k})_{k \in K_i} \leftarrow \mathsf{RootFactor}(W_i, (H_i(k))_{k \in K_i})$, where $A_{i,k} = W_i^{e_{K_i}/H_i(k)} = A_i^{1/H_i(k)}$. Last, the verifier checks if $\Lambda^{e^*} = \prod_{i \in [m]} (\alpha_i)^{e^*/e_{K_i}} = \mathsf{MultiRootExp}(\boldsymbol{\alpha}, (e_{K_i})_{i \in [m]})$, where $\alpha_i = c_i / \prod_{k \in K_i} (A_{i,k})^{V_i(k)}$. The total verification time is dominated by the $O(\ell bm \log^2 \ell bm)$ group operations from $\mathsf{PoKCR.Extract}$.

Soundness of cross-aggregation. Recall that all the aggregated lookup proofs $\pi_i = (W_i, \Lambda_i)$'s can be extracted from a cross-aggregated proof (W, Λ) via $\mathsf{PoKCR.Extract}$. This means that, two cross-aggregated proofs π and π' that are inconsistent for some $k \in K_i \cap K'_j$ can be used to obtain two inconsistent aggregated lookup proofs π_i and π'_j for K_i and K'_j, respectively, w.r.t. the same digest $d_i = d'_j$. As a result, the security reduction from Appendix B.3 can be used to break the Strong RSA assumption.

Incremental (dis)aggregation. Since the W_i's and Λ_i's can be recovered from the cross-aggregated proof (W, Λ) via $\mathsf{PoKCR.Extract}$, our construction also supports *disaggregation* of cross-aggregated lookup proofs. Furthermore, cross-aggregation can be done *incrementally* as follows. Assume we are given two cross-aggregated proofs π, π' that verify against $(\mathsf{vrk}_i, d_i, K_i, V_i)_{i \in [m]}$ and $(\mathsf{vrk}'_i, d'_i, K'_i, V'_i)_{i \in [m']}$, we can combine them as follows. If $\mathsf{vrk}_i \neq \mathsf{vrk}'_j, \forall i \in [m], j \in [m']$, then we can simply multiply the two proofs together. Otherwise, for each i, j where $\mathsf{vrk}_i = \mathsf{vrk}'_j$ and $d_i = d'_j$, we extract π_i from π and π'_j from π' via $\mathsf{PoKCR.Extract}$ and remove them from π and π'. (Since we cannot cross-aggregate across different digests with the same vrk, we require that $d_i = d'_j$.) Then, we aggregate π_i and π'_j together via $\mathsf{CFG.Agg}$ into a $\pi_{i,j}$. Finally, we add $\pi_{i,j}$ back into either π or π' (but not both). After repeating this for all i, j where $\mathsf{vrk}_i = \mathsf{vrk}'_j$, π and π' will not share any verification keys and can be aggregated by multiplying them as before.

Updatability. As a consequence of disaggregation, our cross-aggregated proofs can be updated by disaggregating them, updating the aggregated proof and re-aggregating them.

4.3 Append-only Authenticated Dictionary for Transparency Logs

In this subsection, we extend our updatable authenticated dictionary (UAD) from Section 4.2 with strong key binding, *non-membership proofs*, and append-only proofs. An *append-only proof* [29] can convince any verifier that a dictionary D is a subset of D', meaning all key-value pairs in D are also in D'. This gives us an *append-only authenticated dictionary (AAD)*, which can be used for transparency logging [7, 17, 23, 29] as well as any other application with stronger security requirements, where our UAD's weak binding does not suffice.

However, adding strong key binding comes at a cost. First, our AAD no longer supports incremental (dis)aggregation of proofs nor cross-aggregation. Instead, it only supports *"one-hop" proof aggregation* [3], or aggregating b individual lookup proofs into a constant-sized, aggregated proof. Second, while such aggregated proofs can still be updated after changes to existing keys' values, they can no longer be updated after adding keys to or removing keys from the dictionary. Nonetheless, our AAD maintains full updatability of individual proofs, updatability of digests and efficient pre-computation of all proofs.

Strong binding. To prove security against adversaries that output arbitrary digests, we need to augment our lookup proofs. Specifically, in addition to (S_k, Λ_k), we include an additional RSA non-membership witness (a_k, B_k) for k w.r.t. to S_k (since S_k can also be seen as an RSA accumulator too). To verify the proof $\pi_k = (S_k, \Lambda_k, a_k, B_k)$ for (k, v_k) against the digest $d = (c, S)$, the verifier checks S_k and Λ_k as explained in Section 4.2 and additionally checks that $(S_k)^{a_k} (B_k)^{e_k} = g$.

Non-membership proofs. A non-membership proof π_k for key k is a non-membership of e_k in the RSA accumulator S (see Section 2.4.2). Specifically, $\pi_k = (\perp, \perp, a_k, B_k)$ where $S^{a_k} (B_k)^{e_k} = g$. The verifier can easily check the proof satisfies the equation above in $O(\ell)$ group operations.

Aggregating lookup proofs. Aggregation works as before, except we must now account for the additional RSA non-membership witnesses (a_k, B_k) used for key binding or for non-membership. Given lookup proofs $(\pi_k)_{k\in K}$, we partition the set of keys K into two sets: (1) K_1, the set of keys k with values $v_k \neq \perp$ in the dictionary, and (2) K_0, the set of keys that are not in the dictionary (i.e., $v_k = \perp$). Recall that $\pi_k = (S_k, \Lambda_k, a_k, B_k)$, with S_k and Λ_k set to $\perp$, when proving non-membership.

For $k \in K_0$: In this case, recall that (a_k, B_k) is just an RSA non-membership witness for e_k w.r.t. the accumulator S, which we can aggregate as $\pi_{K_0} = \mathsf{Acc.NonMemWitAgg}(S, (a_k, B_k, e_k)_{k\in K_0})$ (see Fig. 4). This does come at the cost of using a PoKE proof (see Section 2.3), which impedes both incremental (dis)aggregation and updating aggregated proofs when keys are removed or added to the dictionary (which changes S and thus (a_k, B_k)).

For $k \in K_1$: First, the (S_k, Λ_k)'s can be aggregated into (S_{K_1}, Λ_{K_1}) as explained in Section 4.2. Next, recall that (a_k, B_k) is an RSA non-membership witness for e_k w.r.t. to S_k (not w.r.t. S, as was the case for $k \in K_0$). We combine all (a_k, B_k)'s into a disjointness witness for e_{K_1} w.r.t. S_{K_1} as $\pi_{K_1} \leftarrow \mathsf{Acc.NonMemWitCrossAgg}((a_k, B_k, e_k, S_k)_{k\in K_1})$ (see Section 3.3). In Appendix B.3, we show this is enough for strong key binding. Note that this aggregation also comes at the cost of using a PoKE proof.

The final aggregated proof is $\pi_K = (S_{K_1}, \Lambda_{K_1}, \pi_{K_1}, \pi_{K_0})$, and verifies as:

$$S = (S_{K_1})^{e_{K_1}} \tag{22}$$

$$S_k = S^{1/e_k}, \forall k \in K_1, \text{ computed via } \mathsf{RootFactor}(S_{K_1}, (e_k)_{k\in K_1}) \tag{23}$$

$$c = \prod_{k\in K_1} (S_k)^{v_k} (\Lambda_{K_1})^{e_{K_1}} \tag{24}$$

$$1 = \mathsf{Acc.NonMemWitAggVer}(S_{K_1}, \pi_{K_1}, (e_k)_{k\in K_1}) \text{ (see Fig. 4)} \tag{25}$$

$$1 = \mathsf{Acc.NonMemWitAggVer}(S, \pi_{K_0}, (e_k)_{k\in K_0}) \tag{26}$$

Updating individual proofs. Let $\pi_k = (S_k, \Lambda_k, a_k, B_k)$ be an *individual* lookup proof for a single key-value pair (k, v) w.r.t. digest $d = (c, S)$. The updated proof will be $\pi'_k = (S'_k, \Lambda'_k, a'_k, B'_k)$, computed as follows, based on two cases.

Case 1: Adding a new key-value pair $(\hat{k}, \hat{v})$. If $v \neq \perp$, then (S'_k, Λ'_k) are computed as before in Section 4.2, while $(a'_k, B'_k) = \mathsf{Acc.NonMemWitUpdAdd}(S_k, a_k, B_k, e_k, e_{\hat{k}})$. If $v = \perp$, then $(a'_k, B'_k) = \mathsf{Acc.NonMemWitUpdAdd}(S, a_k, B_k, e_k, e_{\hat{k}})$.

Case 2: Changing existing key $\hat{k}$'s value by $\hat{\delta}$. If $v \neq \perp$, the proof update is done as explained in Section 4.2. Otherwise, if $v = \perp$, then the non-membership proof π_k remains the same (since only c, not S, changed in the digest). Note that, in this case, our AAD also supports updating aggregated lookup proofs $\pi_K = (S_K, \Lambda_K, a_K, B_K)$, since a change to one or more keys $\hat{k}$ does not affect the (a_K, B_K) part of the proof (which cannot be updated due to its use of PoKE proofs).

Case 3: Removing a key $\hat{k}$ with value $\hat{v}$. If $k = \hat{k}$, then, we must update the lookup proof for (k, v) to a non-membership proof for k. Recall that π_k contains a non-membership witness (a_k, B_k) w.r.t. the RSA accumulator S_k. Importantly, observe that the new digest $d' = (c', S')$, after removing k, has exactly $S' = S_k$. Thus, we simply set $\pi'_k = (\perp, \perp, a_k, B_k)$. Otherwise, if $k \neq \hat{k}$, we have two cases based on whether π_k is a non-membership proof.

Subcase $k \neq \hat{k}$ and $v = \perp$. In this subcase, we have to update the (a_k, B_k) non-membership witness for e_k w.r.t. S to be a witness w.r.t. $S' = S_{\hat{k}}$ as $(a'_k, B'_k) = \mathsf{Acc.NonMemWitUpdDel}(S, S', a_k, B_k, e_k, e_{\hat{k}})$. If multiple keys $\hat{k} \in R$ are removed, then recall from Section 4.2 that the new digest has $S' = S^{1/e_R}$. Thus, we can update the proof as $(a'_k, B'_k) = \mathsf{Acc.NonMemWitUpdDel}(S, S', a_k, B_k, e_k, e_R)$. (Note that an aggregated proof π_R for all $\hat{k} \in R$ would suffice in this case, since it contains $S_R = S^{1/e_R} = S'$.)

Subcase $k \neq \hat{k}$ and $v \neq \perp$. In this subcase, S'_k and Λ'_k are updated as explained in Section 4.2. Then, the (a_k, B_k) non-membership witness for e_k w.r.t. S_k is updated into a witness w.r.t. $S_{k,\hat{k}} = \mathsf{ShamirTrick}(S_k, S_{\hat{k}}, e_k, e_{\hat{k}})$. This can be done as $(a'_k, B'_k) = \mathsf{Acc.NonMemWitUpdDel}(S_k, S_{k,\hat{k}}, a_k, B_k, e_k, e_{\hat{k}})$. If multiple keys $\hat{k} \in R$ are being removed, this is handled as in the previous subcase by computing $S_{R,k} = (S_R)^{1/e_k} = \mathsf{ShamirTrick}(S_R, S_k, e_R, e_k)$ and then setting $(a'_k, B'_k) = \mathsf{Acc.NonMemWitUpdDel}(S_k, S_{R,k}, a_k, B_k, e_k, e_R)$. As in the previous subcase, an aggregate proof π_R for all $\hat{k} \in R$ suffices.

Append-only proofs. Suppose new key-value pairs $(k, v_k)_{k\in K}$ were added to the dictionary with digest $d = (c, S)$, obtaining a new digest $d' = (c', S')$. Then, note that $S' = S^{e_K}$ and $c' = c^{e_K} \prod_{k\in K} (S')^{\frac{v_i}{e_k}} = c^{e_K} \prod_{k\in K} (S^{e_K})^{\frac{v_i}{e_k}} = c^{e_K} S^{\sum_{k\in K} v_i \cdot \frac{e_K}{e_k}}$ Let $z = \sum_{k\in K} v_i \cdot e_{K\setminus\{k\}}$. The append-only proof π between $d = (c, S)$ and $d = (c', S')$ consists of three PoKE proofs. Specifically, $\pi_{\subseteq}^{\mathsf{PoKE}} = \mathsf{PoKE.Prove}(S', S, u)$, $\pi_u^{\mathsf{PoKE}} = \mathsf{PoKE.Prove}(c^u, c, u)$ and $\pi_z^{\mathsf{PoKE}} = \mathsf{PoKE.Prove}(S^z, S, z)$. Note that these PoKE proofs can be further compressed via the PoKCR protocol [3]. The final append-only proof is $\pi = (\pi_{\subseteq}^{\mathsf{PoKE}}, U, \pi_u^{\mathsf{PoKE}}, Z, \pi_z^{\mathsf{PoKE}})$, where $U = c^u$, $Z = S^z$.

To verify π, one checks if $c' = UZ$, $\mathsf{PoKE.Ver}(S', S, \pi_{\subseteq}^{\mathsf{PoKE}}) \stackrel{?}{=} 1$, $\mathsf{PoKE.Ver}(U, c, \pi_u^{\mathsf{PoKE}}) \stackrel{?}{=} 1$ and $\mathsf{PoKE.Ver}(Z, S, \pi_z^{\mathsf{PoKE}}) \stackrel{?}{=} 1$. We stress that our UAD from Section 4.2 also supports this type of append-only proof, as highlighted in Table 1.

Computing all lookup proofs fast. To compute all $\pi_k = (S_k, \Lambda_k, a_k, B_k)$ efficiently, first, the S_k's and Λ_k's can be computed as before in Section 4.2. Second, each (a_k, B_k) is an RSA non-membership witness for e_k w.r.t. $S_k = S^{1/e_k}$ and, as we have shown in Section 3.2, we can compute all of them in $O(\ell n \log n)$ group operations via $\mathsf{Acc.NonMemWitCrossAgg}(g, (e_k)_{k\in D})$.

5 Future Work

Our work leaves several interesting open questions. First, can we build an AD with strong key binding **and** cross-incremental (dis)aggregation of proofs? Second, can we eliminate the need for auxiliary information during updates in such an AD? Third, can we *de-amortize* [19] and efficiently pre-compute *all* non-membership proofs? Lastly, we did not formalize nor prove append-only security with cross-aggregated proofs but, since cross-aggregated proofs can be disaggregated, security follows naturally as argued in Section 4.3.

References

[1] S. Agrawal and S. Raghuraman. KVaC: Key-Value Commitments for Blockchains and Beyond. Cryptology ePrint Archive, Report 2020/1161, 2020. https://eprint.iacr.org/2020/1161.

[2] J. Benaloh and M. de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In T. Helleseth, editor, *EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[3] D. Boneh, B. Bünz, and B. Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. Cryptology ePrint Archive, Report 2018/1188, 2018. https://eprint.iacr.org/2018/1188.

[4] M. Campanelli, D. Fiore, N. Greco, D. Kolonelos, and L. Nizzardo. Vector Commitment Techniques and Applications to Verifiable Decentralized Storage. Cryptology ePrint Archive, Report 2020/149, 2020. https://eprint.iacr.org/2020/149.

[5] D. Catalano and D. Fiore. Vector Commitments and Their Applications. In K. Kurosawa and G. Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 55–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] A. Chepurnoy, C. Papamanthou, and Y. Zhang. Edrax: A Cryptocurrency with Stateless Transaction Validation, 2018. https://eprint.iacr.org/2018/968.

[7] S. A. Crosby and D. S. Wallach. Efficient Data Structures for Tamper-Evident Logging. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, page 317–334, USA, 2009. USENIX Association.

[8] I. Damgård and M. Koprowski. Generic Lower Bounds for Root Extraction and Signature Schemes in General Groups. In L. R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 256–271, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[9] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 31, USA, 2012. USENIX Association.

[10] S. Gorbunov, L. Reyzin, H. Wee, and Z. Zhang. Pointproofs: Aggregating Proofs for Multiple Vector Commitments, 2020. https://eprint.iacr.org/2020/419.

[11] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 895–913, May 2016.

[12] R. W. Lai and G. Malavolta. Subvector Commitments with Application to Succinct Arguments. Cryptology ePrint Archive, Report 2018/705, 2018. https://eprint.iacr.org/2018/705.

[13] R. W. F. Lai and G. Malavolta. Subvector Commitments with Application to Succinct Arguments. In A. Boldyreva and D. Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 530–560, Cham, 2019. Springer International Publishing.

[14] D. Leung, Y. Gilad, S. Gorbunov, L. Reyzin, and N. Zeldovich. Aardvark: A Concurrent Authenticated Dictionary with Short Proofs. Cryptology ePrint Archive, Report 2020/975, 2020. https://eprint.iacr.org/2020/975.

[15] J. Li, M. Krohn, D. Mazières, D. Shasha, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[16] J. Li, N. Li, and R. Xue. Universal Accumulators with Efficient Nonmembership Proofs. In J. Katz and M. Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[17] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: Bringing Key Transparency to End Users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, Washington, D.C., August 2015. USENIX Association.

[18] L. Nguyen. Accumulators from Bilinear Pairings and Applications. In A. Menezes, editor, *CT-RSA '05*, pages 275–292, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[19] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168 – 173, 1981.

[20] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. Streaming Authenticated Data Structures. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 353–370, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[21] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated Hash Tables Based on Cryptographic Accumulators. *Algorithmica*, 74(2):664–712, 2016.

[22] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, feb 1978.

[23] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS'14*, February 2014.

[24] T. Sander, A. Ta-Shma, and M. Yung. Blind, Auditable Membership Proofs. In Y. Frankel, editor, *Financial Cryptography*, pages 53–71, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[25] A. Schönhage. Schnelle Berechnung Von Kettenbruchentwicklungen. *Acta Inf.*, 1(2):139–144, June 1971.

[26] A. Shamir. On the generation of cryptographically strong pseudo-random sequences. In S. Even and O. Kariv, editors, *Automata, Languages and Programming*, pages 544–550, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

[27] A. Tomescu. *How to Keep a Secret and Share a Public Key (Using Polynomial Commitments)*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2020.

[28] A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovich. Aggregatable Subvector Commitments for Stateless Cryptocurrencies. In C. Galdi and V. Kolesnikov, editors, *Security and Cryptography for Networks*, pages 45–64, Cham, 2020. Springer International Publishing.

[29] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas. Transparency Logs via Append-Only Authenticated Dictionaries. In *ACM CCS'19*, CCS '19, page 1299–1316, New York, NY, USA, 2019. Association for Computing Machinery.

[30] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf, 2014.

A Updating RSA Non-membership Witnesses

In this section, we explain why the algorithms for updating RSA non-membership witnesses from Fig. 3 work. Suppose we have an RSA accumulator $c = g^{\prod_{e \in T} e} = g^u$ over a set T (of prime representatives). Suppose $x \nmid u$, so x is not in the RSA accumulator c. Then, a non-membership witness for x consists of (commitments to) Bézout coefficients (a, b) such that $au + bx = 1$. Specifically, the witness is $\pi_x = (a, B = g^b)$ and verifies as:

$$c^a B^x = g \Rightarrow g^{ua} g^{bx} = 1 \Rightarrow au + bx = 1 \tag{27}$$

Updates after adding an element x'. Next, suppose we added a new element x' to the set, obtaining a new accumulator $c' = c^{x'} = g^{ux'}$. We would like to update the Bézout coefficients (a, b) from π_x to $(a', B' = g^{b'})$ such that $(c')^{a'} (B')^x = g$. Since x and x' are both primes, we have $\gcd(x, x') = 1$. Thus, $\exists s, t$ such that $sx + tx' = 1$, which implies $a = asx + atx'$. Replacing in Eq. (27), we get:

$$c^{asx+atx'} B^x = g \Leftrightarrow (c^{as})^x (c^{x'})^{at} B^x = g \Leftrightarrow (c')^{at} (c^{as} B)^x = g \tag{28}$$

However, note that if we let $a' = at$, then a' would be of size $2|x|$ and would keep getting larger after subsequent witness updates. Therefore, we would like to reduce its size back to $|x|$. We can do this easily, by reducing it modulo x! Specifically, $\exists q, r$ with $r < x$ such that $at = qx + r$. Thus, we get:

$$(c')^{qx+r} (c^{as} B)^x = g \tag{29}$$

$$(c')^r (c^{x'})^{qx} (c^{as} B)^x = g \tag{30}$$

$$(c')^r ((c')^q c^{as} B)^x = g \tag{31}$$

$$(c')^r (c^{qx'+as} B)^x = g \tag{32}$$

As a result, to update the witness for x after adding x' co-prime with x such that $\exists s, t, sx + tx' = 1$, we let $a' = r$ and $B' = c^{qx'+as} B$ as in Fig. 3.

Updates after deleting an element x'. Next, suppose we deleted an element x' from the set, obtaining a new accumulator $c' = c^{1/x'} = g^{u/x'}$. We would like to update the Bézout coefficients (a, b) from π_x to (a', B') such that $(c')^{a'} (g^\beta)^x = g$. Note that we can rewrite Eq. (27) as $c^a B^x = g \Leftrightarrow (c')^{x'a} B^x = g$. We could let $B' = B$ and $a' = x'a$, but $|a'|$ would be too large, so we would like to reduce $x'a$ modulo x! Since $\exists q, r$ such that $x'a = qx + r$, we can rewrite:

$$(c')^{x'a} B^x = g \Leftrightarrow (c')^{qx+r} B^x = g \Leftrightarrow (c')^r \left((c')^q B\right)^x = g \tag{33}$$

As a result, the updated witness is $B' = (c')^q B$ and $a' = r$ as in Fig. 3.

B Security Proofs

B.1 Definitions and Cryptographic Assumptions

Our work often relies on the Strong RSA assumption, which we define below.

Definition B.1 (Strong RSA Assumption)**.** $\mathsf{GenHidOrdGr}$ satisfies this assumption if, $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathbb{G}_? \leftarrow \mathsf{GenHidOrdGr}(1^\lambda), g \in_R \mathbb{G}_?, \\ (u, e) \leftarrow \mathcal{A}(1^\lambda, \mathbb{G}_?, g) : \\ u^e = g \textbf{ and } e \text{ is prime} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Informally, this assumption says that no probabilistic polynomial-time (PPT) adversary can compute *any* eth prime root of a random element g. This is a generalization of the RSA assumption [22], which says that, for a *fixed* e, no PPT adversary can compute an eth root of a random g.

Weak Key Binding. Some authenticated dictionary schemes are only secure if the digest d is produced correctly, rather than adversarially. This security notion is called *weak key binding* and is modeled by having the adversary return a dictionary D, whose commitment d is correctly computed. In contrast, in the *strong key binding* definition (see Definition 4.3), the adversary is allowed to output a digest d directly, which means he could maliciously generate it.

Definition B.2 (Weak Key Binding)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$, there exists negligible function $\mathsf{negl}(\cdot)$, such that:

$$\Pr\left[\begin{array}{c}(\mathsf{prk},\mathsf{vrk})\leftarrow \mathsf{AD.Setup}(1^\lambda),\\ (D,K,K',V,V',\pi,\pi')\leftarrow\mathcal{A}(1^\lambda,\mathsf{prk},\mathsf{vrk}):\\ d\leftarrow\mathsf{AD.Commit}(\mathsf{prk},D)\wedge\\ \mathsf{AD.VerLookup}(\mathsf{vrk},d,K,V,\pi)=1\ \wedge\\ \mathsf{AD.VerLookup}(\mathsf{vrk},d,K',V',\pi')=1\ \wedge\\ \exists k\in K\cap K' \text{ s.t. } V(k)\neq V'(k)\end{array}\right]\leq\mathsf{negl}(\lambda)$$

B.2 Warm-up: Key Binding for Individual Proofs

First, we prove our AAD construction from Section 4.3 satisfies strong key binding (see Definition 4.3) under the Strong RSA assumption (see Definition B.1), when proofs are *not* aggregated: i.e., adversary outputs two inconsistent proofs for an individual key k. (This proof does **not** use the generic group model [8].)

Proof. Assume an adversary $\mathcal{A}$ breaks strong key binding and outputs a digest $d=(c,S)$ and two inconsistent proofs $\pi=(S_k,\Lambda_k,a_k,B_k),\pi'=(S'_k,\Lambda'_k,a'_k,B'_k)$ for a key k having two different values v and v'. Then, we show how to build another adversary $\mathcal{B}$ that breaks a random Strong RSA problem instance g by outputting g^{1/e_k} for some prime e_k. Our adversary $\mathcal{B}$ sets up the AD scheme with prk and vrk set to g and the hash function H. Depending on v and v', we have two cases.

B.2.1 Case 1: $v\neq\perp$ and $v'\neq\perp$

Let $e_k=H(k)$ be the prime representative of the key k. If both proofs pass verification, the following relations must hold:

$$S=S_k^{e_k}=(S'_k)^{e_k} \tag{34}$$

$$c=\Lambda_k^{e_k}S_k^{v}=(\Lambda'_k)^{e_k}(S'_k)^{v'} \tag{35}$$

Since $S_k^{e_k}=(S'_k)^{e_k}$, it follows that $S_k=S'_k$. Thus, we can rewrite Eq. (35) as:

$$\Lambda_k^{e_k}S_k^{v}=(\Lambda'_k)^{e_k}S_k^{v'}\Leftrightarrow(\Lambda_k/\Lambda'_k)^{e_k}=(S_k)^{v'-v}\Leftrightarrow \tag{36}$$

$$\Lambda^{e_k}=(S_k)^{v'-v},\ \text{where } \Lambda=\Lambda_k/\Lambda'_k \tag{37}$$

Recall π contains (a_k,B_k) such that $(S_k)^{a_k}(B_k)^{e_k}=g$. $\mathcal{B}$ calls $(x_k,Y_k)=\mathsf{Acc.NonMemWitUpdAdd}(S_k,a_k,B_k,e_k,$ $v'-v)$[1] (see Section 2.4.2). In other words, $\mathcal{B}$ updates the non-membership witness (a_k,B_k) for e_k w.r.t. the RSA accumulator S_k into a non-membership witness (x_k,Y_k) for e_k w.r.t. the RSA accumulator $(S_k)^{v'-v}$. Thus, we have $(S_k)^{(v'-v)x_k}(Y_k)^{e_k}=g\Leftrightarrow(S_k)^{v'-v}=((Y_k)^{-e_k}g)^{1/x_k}$ Next, rewrite $\Lambda^{e_k}=(S_k)^{v'-v}$ from Eq. (37) as:

$$\Lambda^{e_k}=\left((Y_k)^{-e_k}g\right)^{1/x_k}\Leftrightarrow\Lambda^{x_ke_k}=(Y_k)^{-e_k}g\Leftrightarrow\Lambda^{x_k}=Y_k^{-1}g^{1/e_k} \tag{38}$$

Thus, $\mathcal{B}$ breaks Strong RSA by outputting $(e_k,\Lambda^{x_k}Y_k=g^{1/e_k})$.

[1]Note that since e_k is an $(\ell+1)$-bit prime and the values v and v' are both ℓ-bit wide, it follows that $\gcd(e_k,v'-v)=1$, which means this witness update is possible!

B.2.2 Case 2: $v = \bot$ or $v' = \bot$

Assume without loss of generality, that $v \neq \bot$ and $v' = \bot$. Recall that S_k is an RSA membership witness for e_k w.r.t. to the RSA accumulator S. Also, recall that (a'_k, B'_k) is an RSA non-membership witness for e_k w.r.t. to that same RSA accumulator S. This clearly breaks the RSA accumulator's security [3, Def. 6, pg. 13]. Since there exists an adversary $\mathcal{C}$ that, on input the accumulator S and the two inconsistent witnesses for e_k, breaks the Strong RSA assumption, $\mathcal{B}$ can use $\mathcal{C}$ to break Strong RSA. □

B.2.3 Weak Key Binding for Individual UAD Proofs

The *weak key binding* security proof for our UAD construction from Section 4.2 follows in the same fashion, except that at Eq. (37), the security reduction knows that $S_k = g^{e_{K\setminus\{k\}}}$ where K is the set of all keys in the dictionary D, which the adversary $\mathcal{A}$ now outputs. Then, Eq. (37) becomes:

$$\Lambda^{e_k} = (S_k)^{v'-v} \Leftrightarrow \Lambda^{e_k} = (g^{e_{K\setminus\{k\}}})^{v'-v} = g^{e_{K\setminus\{k\}}(v'-v)} \tag{39}$$

Thus, $\mathcal{B}$ can compute $e_{K\setminus\{k\}}$ and find Bézout coefficients (x_k, y_k) such that $x_k e_{K\setminus\{k\}}(v'-v) + y_k e_k = 1$. As a result, we have:

$$\Lambda^{e_k} = g^{\frac{1-y_k e_k}{x_k}} \Leftrightarrow \Lambda^{x_k e_k} = g/g^{y_k e_k} \Leftrightarrow \Lambda^{x_k e_k} g^{y_k e_k} = g \Leftrightarrow \Lambda^{x_k} g^{y_k} \qquad = g^{1/e_k} \tag{40}$$

Thus, $\mathcal{B}$ can break Strong RSA by outputting $(e_k, \Lambda^{x_k} g^{y_k} = g^{1/e_k})$.

B.3 Key Binding for Aggregated Proofs

We are now ready to prove *key binding* (see Definition 4.3) holds for our AAD from Section 4.3, even with aggregated lookup proofs, under the Strong RSA assumption (see Definition B.1) in the generic group model [8]. Our proof strategy is very simple: we disaggregate the inconsistent aggregated proofs into inconsistent individual proofs and re-use our security reduction from Appendix B.2.

Proof. Assume an adversary $\mathcal{A}$ breaks key binding and outputs a digest $d = (c, S)$, and two inconsistent proofs $\pi = (S_{I_1}, \Lambda_{I_1}, \pi_{I_1}, \pi_{I_0}), \pi' = (S_{J_1}, \Lambda_{J_1}, \pi_{J_1}, \pi_{J_0})$ for I, V and J, V'. Recall from Section 4.3 that $I = I_1 \cup I_0$ and $k \in I_1$ always has value $V(k) \neq \bot$ while $k \in I_0$ has value $\bot$. (J_1 and J_0 are defined similarly.) Let $v_k = V(k), \forall k \in I$ and $v'_k = V'(k), \forall k \in J$. Since $\mathcal{A}$ broke key binding, $\exists$ a key $z \in I \cap J$ with $v_z \neq v'_z$. We show how to build another adversary $\mathcal{B}$ that breaks a random Strong RSA problem instance g by outputting $g^{1/e}$ for some prime e. Our adversary $\mathcal{B}$ sets up the AD scheme with prk and vrk set to g and the hash function H. Recall that, for any set T of keys, $e_T = \prod_{k \in T} e_k$.

B.3.1 Case 1: $v_z \neq \bot$ and $v'_z \neq \bot$

First, $\mathcal{B}$ can obtain $(S_z, \Lambda_z) \leftarrow \mathsf{CFG.Disagg}(I_1, \{z\}, (v_k)_{k \in I_1}, \pi)$ (see Fig. 5) where $S_z = S^{1/e_z}$. Second, $\mathcal{B}$ must extract (a_z, B_z) from π_{I_1} such that $(S_z)^{a_z}(B_z)^{e_z} = g$. Recall that $\pi_{I_1} = (A_{I_1}, \pi^{\mathsf{PoKE}}_{I_1}, B_{I_1})$ is a disjointness proof for all keys in I_1 (including z) such that $\mathsf{PoKE.Ver}(A_{I_1}, S_{I_1}, \pi^{\mathsf{PoKE}}_{I_1}) = 1$ and $A_{I_1} B^{e_{I_1}}_{I_1} = g$. Since the PoKE proof verifies, $\mathcal{B}$ extracts a_{I_1} from the PoKE proof such that $A_{I_1} = S^{a_{I_1}}_{I_1}$ with non-negligible probability. Thus, $\mathcal{B}$ now has $S^{a_{I_1}}_{I_1} B^{e_{I_1}}_{I_1} = g$.

Next, $\mathcal{B}$ can disaggregate the RSA disjointness witness (a_{I_1}, B_{I_1}) for e_{I_1} not being in the accumulator S_{I_1} into individual non-membership witnesses $(a_k, B_k)_{k \in I_1}$ for e_k not being in accumulator $S_k = S^{1/e_k}$ as follows. $\mathcal{B}$ gets non-membership witness $(\hat{a}_k, \hat{B}_k)_{k \in I_1}$ for each e_k w.r.t. S_{I_1} via $\mathsf{BreakUpNonMemWit}(S_{I_1}, a_{I_1}, B_{I_1}, (e_k)_{k \in I_1})$ from [3, Fig. 3, pg. 16]. Then, $\mathcal{B}$ uses $\mathsf{Acc.NonMemWitUpdAdd}(S_{I_1}, \hat{a}_z, \hat{B}_z, e_z, e_{I_1}/e_z)$ to update $(\hat{a}_z, \hat{B}_z)$ into a non-membership witness (a_z, B_z) for e_z w.r.t. to S_z.

Thus, $\pi_z = (S_z, \Lambda_z, a_z, B_z)$ is an individual lookup proof for z having value v_z that verifies against $d = (c, S)$. In exactly the same fashion, $\mathcal{B}$ disaggregates π' into an individual lookup proof $\pi'_z = (S'_z, \Lambda'_z, a'_z, B'_z)$ for z having value $v'_z \neq v_z$ that verifies against $d = (c, S)$. Finally, $\mathcal{B}$ can call the adversary from Appendix B.2 with $d = (c, S)$ and these two inconsistent proofs as input and break Strong RSA.

B.3.2 Case 2: $v_z = \perp$ or $v'_z = \perp$

Without loss of generality, assume that $v_z \neq \perp$ and $v'_z = \perp$. As in the previous case, $\mathcal{B}$ can disaggregate π and obtain S_z, which is an RSA membership witness for e_z w.r.t. the accumulator S: i.e., $S_z = (S_{I_1})^{e_{I_1 \setminus \{z\}}}$.

Next, recall that the proof π' contains an RSA disjointness witness $\pi_{J_0} = (A_{J_0}, \pi^{\mathsf{PoKE}}_{J_0}, B_{J_0})$ for all keys $k \in J_0$ that have value $\perp$ w.r.t. the accumulator S, such that $A_{J_0}(B_{J_0})^{e_{J_0}} = g$ and $\mathsf{PoKE.Ver}(A_{J_0}, S, \pi^{\mathsf{PoKE}}_{J_0}) = 1$. Thus, since the PoKE proof verifies, $\mathcal{B}$ can extract a_{J_0} such that $A_{J_0} = S^{a_{J_0}}$ with non-negligible probability. Thus, $\mathcal{B}$ now has $S^{a_{J_0}}(B_{J_0})^{e_{J_0}} = g$.

Next, $\mathcal{B}$ can disaggregate this (a_{J_0}, B_{J_0}) RSA disjointness witness for e_{J_0} w.r.t. S, into RSA non-membership witnesses $(a_k, B_k)_{k \in J_0}$ for each e_k w.r.t. S via $\mathsf{BreakUpNonMemWit}(S, a_{J_0}, B_{J_0}, (e_k)_{k \in J_0})$. Therefore, $\mathcal{B}$ can obtain a non-membership witness (a_z, B_z) for e_z w.r.t. S.

$\mathcal{B}$ now has both a membership and a non-membership witness for e_z w.r.t. S. This clearly breaks the RSA accumulator's security [3, Def. 6, pg. 13]. Since there exists an adversary $\mathcal{C}$ that, on input S and the two inconsistent witnesses for e_z, breaks the Strong RSA assumption, $\mathcal{B}$ can use $\mathcal{C}$ to break Strong RSA. $\square$

B.3.3 Weak Key Binding for Aggregated UAD Proofs

We can use the same disaggregation-based approach as above to prove weak key binding for our UAD construction from Section 4.2. Importantly, there is no need for the generic group model [8] when disaggregating UAD proofs, since our UAD does not use PoKE proofs. Once the inconsistent UAD aggregated proofs are disaggregated into inconsistent UAD individual proofs, the security reduction from Appendix B.2.3 can be invoked with the individual proofs, which will break the Strong RSA problem.

B.4 Warm-up: Append-only Security w.r.t. Individual Proofs

Suppose an adversary $\mathcal{A}$ breaks append-only security (see Definition 4.5) of our AAD from Section 4.3 and outputs a lookup proof $\pi_k = (S_k, \Lambda_k, a_k, B_k)$ for key k having value $v \neq \perp$ w.r.t. digest $d = (c, S)$, a lookup proof $\pi'_k = (S_k,' \Lambda'_k, a'_k, B'_k)$ for k having value $v' \neq v$ w.r.t. digest $d' = (c', S')$, with v' possibly equal to $\perp$, and an append-only proof $\pi_\subseteq$ between d and d'. Then, we show how to build another adversary $\mathcal{B}$ that, in the generic group model [8], breaks the Strong RSA problem (see Definition B.1).

Recall from Section 4.3 that $\pi_\subseteq = (\pi^{\mathsf{PoKE}}_\subseteq, U, \pi^{\mathsf{PoKE}}_u, Z, \pi^{\mathsf{PoKE}}_z)$ where $c' = UZ$, $\mathsf{PoKE.Ver}(S', S, \pi^{\mathsf{PoKE}}_\subseteq) = 1$, $\mathsf{PoKE.Ver}(U, c, \pi^{\mathsf{PoKE}}_u) = 1$ and $\mathsf{PoKE.Ver}(Z, S, \pi^{\mathsf{PoKE}}_z) = 1$. Since the PoKE proofs verify, $\mathcal{B}$ can extract with non-negligible probability (u, z) such that $S' = S^u$ and $c' = c^u S^z$. Next, we split into cases.

Case 1: $v' \neq \perp$. Since π'_k verifies against d', we know that $S' = (S'_k)^{e_k}$, which implies that $(S'_k)^{e_k} = S^u$. But since π_k verifies against d, we have $S = (S_k)^{e_k}$. Thus, $(S'_k)^{e_k} = (S_k^{e_k})^u \Rightarrow S'_k = S_k^u$. From the validity of π'_k, we also know that $(S'_k)^{v'}(\Lambda'_k)^{e_k} = c' = c^u S^u$. Similarly, from π_k, we know that $(S_k)^v(\Lambda_k)^{e_k} = c$. Thus:

$$(S'_k)^{v'}(\Lambda'_k)^{e_k} = ((S_k)^v(\Lambda_k)^{e_k})^u\,((S_k)^{e_k})^z = (S'_k)^v(\Lambda_k)^{ue_k}(S_k)^{ze_k} \Leftrightarrow \tag{41}$$

$$(S'_k)^{v'-v} = \frac{(\Lambda_k)^{ue_k}}{(\Lambda'_k)^{e_k}}(S_k)^{ze_k} \Rightarrow (S'_k)^{v'-v} = \Lambda^{e_k},\ \text{with}\ \Lambda = \frac{(\Lambda_k)^u}{\Lambda'_k}(S_k)^z \tag{42}$$

Next, we know from π'_k that (a'_k, B'_k) is a non-membership witness for e_k w.r.t. the RSA accumulator S'_k. Since $\gcd(e_k, v' - v) = 1$ (because e_k is a $(\ell+1)$-bit prime and values v and v' are ℓ bits), we can update this to a non-membership witness for the RSA accumulator $(S'_k)^{v'-v}$ as $(x_k, Y_k) = \mathsf{Acc.NonMemWitUpdAdd}(S'_k, a'_k, B'_k, e_k, v' - v)$ such that $\left((S'_k)^{v'-v}\right)^{x_k}(Y_k)^{e_k} = g \Leftrightarrow (S'_k)^{v'-v} = ((Y_k)^{-e_k}g)^{1/x_k}$. Finally, replace $(S'_k)^{v'-v}$ in Eq. (42) to get $((Y_k)^{-e_k}g)^{1/x_k} = \Lambda^{e_k} \Leftrightarrow (Y_k)^{-e_k}g = \Lambda^{x_k e_k} \Leftrightarrow Y_k^{-1}g^{1/e_k} = \Lambda^{x_k}$. Thus, $g^{1/e_k} = \Lambda^{x_k}Y_K$, and $\mathcal{B}$ breaks Strong RSA on g by outputting $(e_k, \Lambda^{x_k}Y_K)$.

Case 2: $v' = \perp$. In this case, π_k contains a membership witness S_k for e_k in S such that $(S_k)^{e_k} = S$. Recall that $\mathcal{B}$ extracted u from the append-only proof $\pi_\subseteq$ such that $S' = S^u$. Since $((S_k)^u)^{e_k} = ((S^k)^{e_k})^u = S^u = S'$, this implies $(S_k)^u$ is a membership witness for e_k w.r.t. S' as well. However, since π'_k says key k is not in the new dictionary, it includes a (contradicting) non-membership witness (a'_k, B'_k) for e_k w.r.t. the accumulator S'. This clearly breaks the RSA accumulator's security [3, Def. 6, pg. 13]. Since there exists an adversary $\mathcal{C}$ that, on input the accumulator S' and the two inconsistent witnesses for e_k, breaks the Strong RSA assumption, $\mathcal{B}$ can use $\mathcal{C}$ to break Strong RSA.

B.5 Append-only Security for Aggregated Proofs

Proof. Suppose an adversary $\mathcal{A}$ breaks append-only security as defined in Definition 4.5 and outputs a lookup proof π for keys $k \in K$ having value $v_k = V(k)$ w.r.t. digest (c, S), a lookup proof π' for keys $k \in K'$ having value $v'_k = V'(k)$ w.r.t. digest (c', S'), such that $\exists z \in K \cap K'$ with $v_z \neq \perp$ and $v_z \neq v'_z$, and an append-only proof $\pi_\subseteq$ between d and d'. Then, we show how to build another adversary $\mathcal{B}$ that, in the generic group model [8], breaks the Strong RSA problem (see Definition B.1) As with our proof for strong key binding of aggregated proofs (see Appendix B.3), we take advantage of the fact that our construction supports disaggregating proofs. Note that $K = K_1 \cup K_0$ and $k \in K_1$ always has value $v_k \neq \perp$ while $k \in K_0$ has value $\perp$. Similarly, K' is also partitioned into K'_1 and K'_0. As before, we need to consider two separate cases.

Case 1: $\boldsymbol{v_z \neq \perp}$ and $\boldsymbol{v'_z \neq \perp}$. In this case, $\mathcal{B}$ proceeds similar to Appendix B.3.1. Specifically, given the aggregated lookup proof π, $\mathcal{B}$ disaggregates a proof π_z for z having value v_z w.r.t. digest (c, S). Similarly, given π', $\mathcal{B}$ disaggregates π'_z for z having value $v'_z \neq v_z$ w.r.t. digest (c', S'). Now, $\mathcal{B}$ calls the security reduction from Appendix B.4 with $(d, d', \pi_z, \pi'_z, z, v_z, v'_z, \pi_\subseteq)$ as input, which breaks the Strong RSA problem.

Case 2: $\boldsymbol{v_z \neq \perp}$ and $\boldsymbol{v'_z = \perp}$. In this case $z \in K_1$ and $z \in K'_0$. Recall that $\pi = (S_{K_1}, \Lambda_{K_1}, \pi_{K_1}, \pi_{K_0})$ and $\pi' = (S_{K'_1}, \Lambda_{K'_1}, \pi_{K'_1}, \pi_{K'_0})$. $\mathcal{B}$ proceeds as follows. $\mathcal{B}$ computes an RSA membership witness S_z for e_z w.r.t. the accumulator S as $S_z = (S_{K_1})^{e_{K_1 \setminus \{z\}}}$. As explained in Appendix B.4, $\mathcal{B}$ extracts u from $\pi_\subseteq$ such that $S' = S^u$. Then, $\mathcal{B}$ updates S_z to a membership witness $(S_z)^u$ for e_z w.r.t. S'. Similar to Appendix B.3.2, $\mathcal{B}$ extracts with non-negligible probability $(a_{K'_0}, B_{K'_0})$ from $\pi_{K'_0} = (A_{K'_0}, \pi^{\mathsf{PoKE}}_{K'_0}, B_{K'_0})$ where $A_{K'_0} = (S')^{a_{K'_0}}$ and $(S')^{a_{K'_0}} B_{K'_0}^{e_{K'_0}} = g$. In other words, $\mathcal{B}$ extracts an RSA disjointness witness $(a_{K'_0}, B_{K'_0})$ for $e_{K'_0}$ w.r.t. S'. Next, $\mathcal{B}$ can disaggregate this disjointness witness into individual non-membership witnesses $(a_k, B_k)_{k \in K'_0}$ for each e_k w.r.t. S' via $\mathsf{BreakUpNonMemWit}(S', a_{K'_0}, B_{K'_0}, (e_k)_{k \in K'_0})$ from [3, Fig. 3, pg. 16]. Thus, $\mathcal{B}$ obtains a non-membership witness (a_z, B_z) for e_z w.r.t S' (since $z \in K'_0$). Since $\mathcal{B}$ now has both a membership and a non-membership witness for e_z w.r.t. S', this clearly breaks the RSA accumulator's security [3, Def. 6, pg. 13]. Since there exists an adversary $\mathcal{C}$ that, on input the accumulator S' and the two inconsistent witnesses for e_z, breaks the Strong RSA assumption, $\mathcal{B}$ can use $\mathcal{C}$ to break Strong RSA. $\square$