Authenticated Dictionaries with Cross-Incremental Proof (Dis)aggregation

Alin Tomescu\textsuperscript{1}  
Yu Xia\textsuperscript{2}  
Zachary Newman\textsuperscript{2}

\textsuperscript{1}VMware Research  
\textsuperscript{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.
Updatability, we indicate updatability of: individual lookup proofs (I), aggregated lookup proofs (A), aggregated lookup proofs, but only after changes to existing keys (i), cross-aggregated lookup proofs (X) and digests (D).

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 (i), cross-aggregated lookup proofs (X) and digests (D).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Merkle tree</td>
<td>×</td>
<td>Strong</td>
<td>I</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>SADS [20]</td>
<td>×</td>
<td>Strong</td>
<td>DI</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>AHTs [21]</td>
<td>×</td>
<td>Weak</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>KVC1 [3]</td>
<td>One-hop</td>
<td>Strong</td>
<td>DI</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>KVC2 [3]</td>
<td>One-hop</td>
<td>Weak</td>
<td>DI</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>AAD [27]</td>
<td>×</td>
<td>Strong</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Aardvark [14]</td>
<td>One-hop</td>
<td>Weak</td>
<td>DI</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>KVaC [1]</td>
<td>One-hop</td>
<td>Weak</td>
<td>DI</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

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 KVC1, satisfies strong key binding (see Definition 4.3) while their second construction, KVC2, 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 polylogarithmic 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
This way, the root will compute the desired 

\[
\text{RootFactor}(g, x) \rightarrow (g^{e^*/x})_{i \in [n]}
\]

if \( n = 1 \), then return \( g^{e^*/x} \)

else

Split \([n]\) into halves \( L \) and \( R \)

\( g_L \leftarrow [g^{1/x_1}, \ldots, g^{1/x_{n/2}}] \)

\( g_R \leftarrow [g^{1/x_{n/2+1}}, \ldots, g^{1/x_n}] \)

\((x_L, g^\frac{1}{x_L}) \leftarrow \text{ShamirTrick}(g_L, x_L)\)

\((x_R, g^\frac{1}{x_R}) \leftarrow \text{ShamirTrick}(g_R, x_R)\)

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

\( \triangleright \text{i.e. } a x_L + b x_R = 1 \)

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

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

end if

end

Figure 1: We frequently rely on these algorithms, where \( x = [x_1, \ldots, x_n] \), \( n = 2^k \), \( x^* = \prod_{i \in [n]} x_i \) 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 \( \lambda \) denote our security parameter. Let \( \ell \) denote the length (in bits) of vector elements and of dictionary values. Let \( G_\ell \) denote a hidden-order group and \( g \) be a random group element in \( G_\ell \). Let \( H : \{0, 1\}^\ast \rightarrow \text{Primes}_{\ell+1} \) be a collision-resistant hash function that outputs \((\ell + 1)\)-bit primes. We typically use bolded variables \( x \) to denote vectors \([x_1, x_2, \ldots, x_n]\) of elements. We also use \( x_I = (x_i)_{i \in I} \) to denote an \( I \)-subvector of \( x \) 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 \( \text{gcd}(x, y) = 1 \), \((a, b) \leftarrow \text{EEA}(x, y)\) returns \( \text{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 \( a \leq y \) and \( b \leq x \).

Shamir’s trick. For any \( g \in G_\ell \), given integers \( x, y \) and \( g^{\frac{1}{x}}, g^\frac{1}{y} \), this classic algorithm by Shamir [26] efficiently computes \( g^{1/x} \) as \((g^{1/y})^b (g^{1/y})^a \) where \( a, b \) are \( \text{Bézout coefficients} \) 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 EEA’s and integer multiplications.

RootFactor. Given \( g \in G_\ell \) and integers \( x = [x_1, \ldots, x_n] \), outputs \( g^{x^*/x} \) 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].
**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 \( \alpha_i \)'s and \( x_i \)'s. In other words, the protocol proves the following relation holds:

\[
R_{PoKCR} = \{ [\alpha_1, \ldots, \alpha_n] \in \mathbb{G}^n_\ell \mid x_1, \ldots, x_n \in \mathbb{G}_\ell^n : w_i^{x_i} = \alpha_i, \forall i \in [n] \}.
\]

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

**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 \( W, x \) and \( \alpha \) as follows. Let \( z_i = x^*/x_i \) and \( z_{i,j} = x^*/(x_ix_j) \). Since \( W \) is valid, \( W^{x^*} = \prod_{i \in [n]} \alpha_i^{z_i} \). This means \( W^{x_i} = (\prod_{i \in [n]} (z_i \alpha_i^{z_i})) \}

\[
\text{To verify, PoKCR.Ver}(\alpha, x, W)
\]

**Time to extract.** Assume each \( x_i \) is at most \( \ell \) bits. Note that \( A_j = \prod_{i \in [n]} (z_i \alpha_i^{z_{i,j}}) = \text{MultiRootExp}(\alpha | \{ x_i \}, x | \{ x_i \}) \) 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 \( R_{PoKE} = \{ w \in \mathbb{G}_\ell, u \in \mathbb{G}_2 : w = u^x \} \) holds. The protocol makes use of three different hash functions \( H_{G_\ell} : \mathbb{G}_\ell^2 \rightarrow \mathbb{G}_\ell, H_{\text{Primes2}_\lambda} : \mathbb{G}_\ell^3 \rightarrow \text{Primes}_{2\lambda} \) and \( H_{2\lambda} : \mathbb{G}_\ell^3 \times \{ 0, 1 \}^{2\lambda} \rightarrow \{ 0, 1 \}^{2\lambda} \) modeled as random oracles. Correctness holds because:

\[
Q^\ell u^r g^{a(r)} = (u^q g^{a(q)})^\ell u^r g^{a(r)} = u^{q\ell + r} g^{a(q)\ell + a(r)} = u^\ell + r g^{a(q)\ell + a(r)} = w^z g^{ax} = w z^\alpha
\]

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 \( \mathbb{G}_\ell \) and a 2\( \lambda \)-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, \( \ell \)-bit prime elements, an RSA accumulator is a commitment \( c = g^{\prod_{i \in T} e_i} \) to the set \( T \). Here, \( g \) is a random element of \( \mathbb{G}_\ell \). 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 \( X \cap T = \emptyset \).
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 \(w_i = g^{\prod_{j \in T \setminus I} e_i} = c^{1/e_i}\). The verifier checks the witness as \(w_i^{e_i} \equiv c\). Sander et al. [24] showed that all of \(T\)'s membership witnesses can be computed in \(O(n \log n)\) time as \((w_i)_{i \in [n]} \leftarrow \text{RootFactor}(g, [e_1, e_2, \ldots, e_n])\) (see Fig. 1). Furthermore, a subset witness for several elements \(S \subseteq T\) can be computed as \(w_S = g^{\prod_{e_j \in S} e_j} = c^{\prod_{e_j \in S} e_j}\). The verifier checks the witness as \(w_i \equiv c\). Note that such a witness can be aggregated from the individual witnesses \(c^{1/e_i}\)'s as \(w_S = \text{ShamirTrick}((c^{1/e_j})_{j \in S}, (e_j)_{j \in S})\).

Cross-aggregator accumulation. 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 \(a_i = a_i\). 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 PoKCRVer\((a, x, w)\). 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 \((a, b) = \text{EEA}(u, e)\), such that \(au + be = 1\). The witness is verified by checking if \(e^uB^e = g\) 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).

<table>
<thead>
<tr>
<th>Acc.NonMemWitUpdAdd ((c, a, B, x, x'))</th>
<th>Acc.NonMemWitUpdDel ((c, c', a, B, x, x'))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let ((s, t) = \text{EEA}(x, x'))</td>
<td>Note that (c' = c^{1/t}) is the updated accumulator without (x').</td>
</tr>
<tr>
<td>Let (q \in \mathbb{Z}, r \in [0, x)) s.t. (at = qx + r)</td>
<td>Let (q \in \mathbb{Z}, r \in [0, x)) s.t. (x'a = qx + r)</td>
</tr>
<tr>
<td>return ((a', B') = (r, e^{x' + as}B))</td>
<td>return ((a', B') = (r, (c')^sB))</td>
</tr>
</tbody>
</table>

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, Acc.NonMemWitAgg\(^*\) returns an \(O(\ell n)\)-sized disjointness witness and Acc.NonMemWitDel\(^*\) 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 \(S = g^{\prod_{i \in [n]} e_i}\) and \(S_I = S^{1/e_i}\). 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 \(prk = (g, H)\) and a verification key \(vrk = (g, H)\). The \(S_I\)'s are called update keys since they serve as auxiliary information when updating digests and proofs.

Commitment. A commitment to \(v = (v_i)_{i \in [n]}\) consists of \(S = g^{v_i}\) and \(c = \prod_{i \in [n]} S_i^{v_i}\). The committing time is dominated by computing all \(S_I\)'s in \(O(n \log n)\) group operations via RootFactor\((g, [e_1, \ldots, e_n])\). The commitment
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 \( S' = S^{v_{n+1}} \) and \( c' = c^{v_{n+1}} \). 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 \( \delta_j \), 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 \( O(b) \) group operations. Alternatively, if an aggregated update key \( S_J = S^{1/e_j} \) is given, each \( S_j \) is first computed via RootFactor(\( S_J, (e_j)_{j \in J} \)) in \( O(b \log b) \) group operations.

**Subvector proofs.** An \( I \)-subvector proof \( \pi_I \) for \( v_I = (v_i)_{i \in I}, I \subseteq [n] \) consists of (1) \( S_I = S^{v_I} \), and (2) \( \Lambda_I = \left( \prod_{j \in [n] \setminus I} S_j^{v_j} \right)^{1/e_I} = \prod_{j \in [n] \setminus I} S_j^{v_j/e_I} \), 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 RootFactor. The verification time is dominated by the \( O(b \log b) \) group operations from RootFactor. To verify \( \pi_I \), one checks if \( (S_I)^{e_I} = S \), computes \( (S_I)_{i \in I} \) via RootFactor(\( S_I, (e_i)_{i \in I} \)) and checks if \( e = \Lambda_I \prod_{i \in I} S_i^{e_i} \). The verification time is dominated by the \( O(b \log b) \) group operations from RootFactor. If \( |I| = 1 \), then an \( I \)-subvector proof is referred to as an individual proof. A useful fact to notice is that \( \pi_I \) is simply the commitment to a vector "without positions \( I \) in it," since \( S_I = S^{1/e_I} \) and \( \Lambda_I = (c/\prod_{i \in I} S_i^{v_i})^{1/e_I} \). We often use this observation in Section 4.

**Disaggregating proofs.** Campanelli et al. introduce incremental (dis)aggregation of subvector proofs (see Fig. 5). Specifically, they show how to aggregate \( \pi_I, \pi_J \) for \( v_I, v_J \) and \( v_{I \cup J} \) via CFG.Agg. They also show how to disaggregate any proof \( \pi_I \) into a proof \( \pi_I \setminus K \) for a smaller subvector via CFG.Disagg. Lastly, they give a CFG.AggsManyToOne algorithm that aggregates individual proofs for \( v_i \)'s, \( i \in I \), into a subvector proof for \( v_I \) via CFG.Aggs. Fig. 5 summarizes these algorithms as well as CFG.Aggs, a relaxed version of CFG.Agg that assumes \( I \cap J = \emptyset \).

Let \( b = \max\{|I|, |J|\} \). Then, CFG.Aggs takes \( O(b \log b) \) group operations. CFG.AggsManyToOne uses CFG.Aggs recursively and takes \( T(b) = 2T(b/2) + O(b \log b) = O((b \log b)^2) \) group operations. CFG.Disagg takes \( O((|I| - |K|) \log(|I| - |K|)) \) group operations. Since the worst-case time to disaggregate \( \pi_I \) and \( \pi_J \) is \( O(b \log b) \) group operations, CFG.Aggs is also \( O(b \log b) \).

**Updating I-subvector proofs** A proof \( \pi_I = (S_I, \Lambda_I) \) for \( v_I \) can be updated to \( \pi' = (S'_I, \Lambda'_I) \) after the vector changes. Previous work [4, 5] shows how to update proofs if (1) several \( v_i \)'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:

\[
S'_I = S_I^{\prod_{j \in [\Delta]} v_{n+j}} \quad \Lambda'_I = \Lambda_I^{\prod_{j \in [\Delta]} (S_j^{v_{n+j}})}
\]

Note that this takes \( O(\ell \Delta) \) group operations if done sequentially (as described in the commitment update paragraph above) rather than using RootFactor to compute all \( S_I^{v_{n+j}} \)'s.
Figure 5: Algorithms by Campanelli et al. [4] for incrementally (dis)aggregating proofs. CFG.Aggs assumes $I \cap J = \emptyset$, CFG.Disagg assumes $K \subset I, K \neq \emptyset$, CFG.Aggs assumes $I \nsubseteq J$ and $J \nsubseteq I$ and CFG.AggsManyToOne assumes $|I| = 2^k$.

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

Precomputing all proofs. All individual proofs can be computed fast in $O(n \log^2 n)$ time via proof disaggregation in a recursive manner [4] by disaggregating the proof for the full vector $v$ (i.e., $\pi_I = (g, 1, e_I)$) into a subvector proof for its left half $v_L$ and a subvector proof for its right half $v_R$. Then, this can be repeated recursively to obtain all individual proofs (see 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 verify cross-aggregated proofs faster in our UAD from Section 4.2. Then, we show how to compute all RSA non-membership witnesses $(u_i, B_i)_{i \in I}$ for $e_I$ w.r.t. its own accumulator $e_I^{1/e_I}$, 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 $\ell$-bit primes.

3.1 Faster Witness Extraction in PoKCR

Suppose we have a PoKCR proof $W$ of $x_i$'s 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 $\ell$ 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 PoKCRExtract 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 $W_R = \prod_{i \in R} \alpha_i^{1/x_i}$. 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:

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

\[
W^{x L x_R} = \left( \prod_{i \in L} \alpha_i^{1/x_i} \right)^{x_L x_R} \left( \prod_{i \in R} \alpha_i^{1/x_i} \right)^{x_R x_L} \Rightarrow
\]

\[
W^{x} = (W_L)^{x L} (W_R)^{x R} \quad \text{and} \quad W^{x L x_R} = (W_L)^{x L} (W_R)^{x R}
\]

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

\[
(W_L)^{x L} = \left( \prod_{i \in L} \alpha_i^{1/x_i} \right)^{x_L} = \text{MultiRootExp}(\alpha_L, x_L)
\]

\[
(W_R)^{x R} = \left( \prod_{i \in R} \alpha_i^{1/x_i} \right)^{x_R} = \text{MultiRootExp}(\alpha_R, x_R)
\]

Next, \( W^{x L} \) and \( W^{x R} \) can be computed in \( O(\ell m) \) group operations. This means \( W^{x L} = W^{x L}/(W_L)^{x L} \) and \( W^{x R} = W^{x R}/(W_R)^{x R} \) 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.

\[
\text{ShamirTrick}(w_L^{x L}, w_R^{x R}, x_L, x_L) = \text{ShamirTrick}(w_L^{x^*/x_R}, w_L^{x^*/x_L}, x_R, x_L) = w_L^{x^*/x_L x_R} = w_L
\]

\[
\text{ShamirTrick}(w_L^{x L}, w_R^{x R}, x_L, x_L) = \text{ShamirTrick}(w_L^{x^*/x_R}, w_L^{x^*/x_L}, x_R, x_L) = w_R^{x^*/x_L x_R} = w_R
\]

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

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

In this section, we give an algorithm called \text{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. \( e_i = g^{e_i^*} \), 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. \( g^{e_L} \), and \( (a_R, B_R) \) for \( e_R \) w.r.t. \( g^{e_R} \). 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. \( g^{e_i^*} \).

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 \( s \cdot e_L + t \cdot e_R = 1 \Rightarrow a = a \cdot e_L + a \cdot e_R \). Then, note that:

\[
g = g^{0 B^e} = g^{a s e_L + a t e_R (B^e R)^e_L}
\]

\[
= (g^{a s} e_L (g^{e_R})^{at} (B^e R)^e_L) = (g^{e_R})^{at} (g^{a s} B^{e R})^{e_L}
\]

We could now set \( a_L = at \) and \( B_L = (g^{a s} B^{e R})^e_L \) 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 \( r < e_L \).

\[
g = (g^{e_R})^{at} (g^{a s} B^{e R})^{e_L} = (g^{e_R})^{q e_L} (g^{a s} B^{e R})^{e_L}
\]

\[
= (g^{e_R})^{q e_L} (g^{a s} B^{e R})^{e_L} = (g^{e_R})^{r} (g^{a s} B^{e R})^{e_L}
\]

Now, we can set \( a_L = at \mod e_L = r \) and \( B_L = (g^{a s} g^{a s} B^{e R}) \). 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 \text{Acc.NonMemWitCrossProve} recursive algorithm will take \( T(n) = 2T(n/2) + O(\ell n) \) group operations.
3.3 Aggregating Non-membership Witnesses Across Different, Related Accumulators

In this subsection, we give an algorithm called Acc.NonMemWitCrossAgg that aggregates \( n \) witnesses \((a_i, B_i)\) 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. \( c^{1/e^*} \). Later on, we use this algorithm in Section 4.3 to aggregate lookup proofs in our AD construction. We first describe how 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 \( c^{1/\epsilon_0} \), and \( \pi_1 = (a_1, B_1) \) for \( e_1 \) w.r.t. \( c^{1/\epsilon_1} \), where:

\[
\begin{align*}
(c^{1/\epsilon_0})^{a_0}(B_0)^{\epsilon_0} &= g \\
(c^{1/\epsilon_1})^{a_1}(B_1)^{\epsilon_1} &= g
\end{align*}
\]

We want to aggregate them into a disjointness witness \( \pi = (a, B) \) for \( e_0 \cdot e_1 \) w.r.t. \( c^{1/\epsilon_0} \) into a witness w.r.t. \( c' \), which removed \( e_1 \), as \((a'_0, B'_0) = \text{Acc.NonMemWitUpdDel}(c^{1/\epsilon_0}, c', a_0, B_0, e_0, e_1)\). Second, we update the witness for \( e_1 \) w.r.t. \( c^{1/\epsilon_1} \) into a witness w.r.t. \( c' \), which removed \( e_0 \), as \((a'_1, B'_1) = \text{Acc.NonMemWitUpdDel}(c^{1/\epsilon_1}, c', a_1, B_1, e_1, e_0)\). Now, we have two non-membership witnesses for \( e_0 \) and \( e_1 \) w.r.t. \( c' \):

\[
\begin{align*}
(c')^{a'_0}(B'_0)^{\epsilon_0} &= g \\
(c')^{a'_1}(B'_1)^{\epsilon_1} &= g
\end{align*}
\]

Next, we can aggregate these witnesses as \((a, B) = \text{Acc.NonMemWitAgg}(c', a'_0, B'_0, e_0, a'_1, B'_1, e_1)\) such that \((c')^{a}(B)^{\epsilon} = g\). 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. \( c^{1/e^*} \). Finally, we compute a PoKE proof \( \pi_a^\text{PoKE} = \text{PoKE.Proof}(\langle c^a, c', a \rangle) \) to “compress” \( a \), which is of size \(|a| = |c^a|\). The final aggregated disjointness witness will be \( \pi = \langle \langle c^a, \pi_a^\text{PoKE}, B \rangle \rangle \) and verifies via \( \text{Acc.NonMemWitAggVer}(c', \pi, (e_i)_{i \in [n]} \rangle \) (see Fig. 4). Thus, our final \( \text{Acc.NonMemWitCrossAgg} \) 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 \textit{updatable authenticated dictionary (UAD)} for stateless validation in the smart contract setting (see Section 4.2). Our UAD supports a new notion of \textit{cross-incremental proof (dis)aggregation}, which generalizes the notion of \textit{cross-commitment aggregation} by Gorbunov et al. [10]. Our second construction is an \textit{append-only authenticated dictionary (AAD)} for applications with stronger security requirements, such as transparency logs (see Section 4.3). Our AAD additionally supports \textit{non-membership proofs of keys} that are not in the dictionary and \textit{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].

\textbf{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.

\textbf{AD.Setup}(1^k) \to (prk, vrk). Returns the AD’s proving key and verification key.

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

\textbf{AD.ProveLookup}(prk, D, K) \to \pi. Returns a lookup proof \( \pi \) that each \( k \in K \) has value \( D(k) \).
Definition 4.1 (Correctness). An authenticated dictionary scheme is correct if, \( \forall \) public parameters \((prk, vrk) \leftarrow AD\text{-}Setup(1^\lambda)\), \( \forall \) dictionaries \( D \) with digest \( d \leftarrow AD\text{-}Commit(prk, D) \), the following hold:

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

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

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

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

\[
\Pr \left[ \left( \langle prk, vrk \rangle \leftarrow AD\text{-}Setup(1^\lambda), (\langle d_i, K_i, V_i, (V'_i, \pi_i) \rangle_{i \in [m]} \leftarrow A(1^\lambda, \langle prk, vrk \rangle) : AD\text{-}VerLookup(vrk, d, K, V, \pi) = 1 \land AD\text{-}VerLookup(vrk, d, K', V', \pi') = 1 \land \exists k \in K \cap K' \text{ s.t. } V(k) \neq V'(k) \right) \right] \leq \text{negl}(\lambda)
\]

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). \( \forall M = poly(\lambda) \), \( \forall \) adversaries \( A \) running in time \( poly(\lambda) \), there exists negligible function \( \text{negl}(\cdot) \), such that:

\[
\Pr \left[ \left( \langle (prk_i, vrk_i) \leftarrow AD\text{-}Setup(1^\lambda) \rangle_{i \in [M]}, (\langle d_{ij}, K_{ij}, V_{ij}, (V'_{ij}, \pi'_j) \rangle_{j \in J, \pi} \leftarrow A(1^\lambda, \langle prk_i, vrk_i \rangle_{i \in [M]}): AD\text{-}CrossVerLookup((vrk_i, d_{ij}, K_{ij}, V_{ij})_{i \in J, \pi} = 1 \land AD\text{-}CrossVerLookup((vrk_i, d_{ij}', K_{ij}', V_{ij}')_{j \in J, \pi'}) = 1 \land \exists i \in I, j \in J, k \in K_i \cap K_j \text{ such that } vrk_i = vrk_j \land d_i = d_j' \land V_i(k) \neq V'_j(k) \right) \right] \leq \text{negl}(\lambda)
\]

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). ∀ adversaries \( \mathcal{A} \) running in time \( \text{poly}(\lambda) \), there exists negligible function \( \text{negl}(\cdot) \), such that:

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

Observation: This definition can be generalized to worked 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 \( e_k = H(k) \) 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. 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 \( prk = vrk = (g, H) \), where \( g \) is a generator for the hidden-order group \( G_T \) and \( H \) is a CRHF that maps keys (not vector indices) to \( \ell + 1 \) bit primes. Values in the dictionary must be \( \ell \)-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 \( S = g^v \) and \( S_k = S^{\frac{1}{\pi_k}} \). The digest \( d = (c, S) \) of our UAD resembles the VC from Section 2.5, where \( c = \prod_{k \in D} (S_k)^{\pi_k} \).

Lookup proofs. Similar to Section 2.5, a proof \( \pi_k \) 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 \( \Lambda_k \) to \( D \) without \( (k, v) \) in it, where \( \Lambda_k = \prod_{(k', v') \in D, k' \neq k} ((S_{k'})^{\pi_{k'}})^{\pi_k} \). The second part, is an RSA membership witness \( S_k \) for \( k \) w.r.t. the RSA accumulator \( S \) in the digest, where \( S_k = S^{\frac{1}{\pi_k}} \). 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 CFG.Aggregate, CFG.Digest and CFG.AggregateManyToOne 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 \( (S_K, \Lambda_K) \leftarrow \text{CFG.AggregateManyToOne}(K, (v_k)_{k \in K}, (\pi_k)_{k \in K}) \), 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 \( c' = \Lambda_k \).
is that we can ensure this GCD property holds by requiring that all keys in $K$ and $\pi(K)$ when all aggregate update key values works the same as in the VC from Section 2.5. As in the VC, updating after changing existing keys $k \in \hat{K}$ requires at least the aggregate update key $S_{\hat{K}}$ (or the individual $S_{\hat{K}}$).

We show how to update $\pi_K$ 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 $\pi_{K'}$ for each $k$ having value $v_k$ (or individual proofs $\pi_k$ which can be aggregated into $\pi_{K'}$). Note that we can aggregate $\pi_{K \cup \hat{K}} = (S_{K \cup \hat{K}}, \Lambda_{K \cup \hat{K}})$ via CFG.Aggs ($\hat{K}$, $K$, $(v_k)_{k \in \hat{K}}$, $(v_k)_{k \in K}$, $\pi_K$, $\pi_{\hat{K}}$), 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 $d' = (c', S') = (\Lambda_{K \cup \hat{K}}, S_{K \cup \hat{K}})$. (Recall from above that the new digest $d'$ is just the aggregated proof $\pi_{K}$ 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 $S = (S_{K \cup \hat{K}})^{e_{K \cup \hat{K}}e_{K \setminus \hat{K}}} \Leftrightarrow S^{1/e_{K \setminus \hat{K}}} = (S_{K \cup \hat{K}})^{e_{K \cup \hat{K}}} \Leftrightarrow S' = (S_{K \cup \hat{K}})^{e_{K \setminus \hat{K}}}$. 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:

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

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

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

$$\Lambda_{\hat{K}} = (\Lambda_{K \cup \hat{K}})^{e_{K \cup \hat{K}}} \prod_{k \in K \setminus \hat{K}} ((S_k')^{v_k})^{1/e_{K \setminus \hat{K}}} \Leftrightarrow$$

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

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 $\pi_i$ for keys $k \in K_i$ with values $v = V_i(k)$, where each $\pi_i$ is w.r.t. its own digest $d_i$. We would like to cross-aggregate [10] these proofs into a single, constant-sized proof $\pi$ 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 $i$th AD with digest $d_i$ must have its own, unique hash function $H_i$. 

14
We are now left with cross-aggregating the \( \Lambda_i \)'s. Recall that each \( \Lambda_i \) would verify as \( c_i = \Lambda_i^{\nu_k} \prod_{k \in K_i} (A_i, k)^{V_i(k)} \iff c_i / \prod_{k \in K_i} (A_i, k)^{V_i(k)} = \Lambda_i^{\nu_k} \). Thus, checking all the \( \Lambda_i \)'s is equivalent to checking a PoKCR relation holds (i.e., set \( w_i = A_i, x_i = e_{K_i} \) 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 \( \Lambda_i \)'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 \( W e^* \equiv \prod_{i \in [m]} A_i^{e^*/e_{K_i}} = \text{MultiRootExp}(\{A_i\}_{i \in [m]}, \{e_{K_i}\}_{i \in [m]}) \) (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 PoKCR.Extract(\( W, \{A_i\}_{i \in [m]}, \{e_{K_i}\}_{i \in [m]} \)).

Importantly, this takes \( O(\ell b m \log^2 \ell b m) \) group operations (rather than \( O(\ell b m^2) \), if done naively). Third, the verifier computes, for all \( i \in [m] \), \( A_{i,k} = W_i^{e_{K_i}/H_i(k)} = A_i^{1/H_i(k)} \). Last, the verifier checks if \( \Lambda e^* = \prod_{i \in [m]} (\alpha_i)^{e^*/e_{K_i}} = \text{MultiRootExp}(\alpha, \{e_{K_i}\}_{i \in [m]}) \), where \( \alpha_i = c_i / \prod_{k \in K_i} (A_i, k)^{V_i(k)} \). The total verification time is dominated by the \( O(\ell b m \log^2 \ell b m) \) group operations from 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 \( (W, \Lambda) \) via PoKCR.Extract. This means that, two cross-aggregated proofs \( \pi, \pi' \) that are inconsistent for some \( k \in K_i \cap K_i' \) can be used to obtain two inconsistent aggregated lookup proofs \( \pi_i, \pi_i' \) for \( K_i, K_i' \), respectively, w.r.t. the same digest \( d_i = d_i' \). 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 \( \Lambda_i \)’s can be recovered from the cross-aggregated proof \( (W, \Lambda) \) via 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 \( \pi, \pi' \) that verify against \( (\text{vrk}_i, d_i, K_i, V_i)_{i \in [m]} \) and \( (\text{vrk}_i', d_i', K_i', V_i')_{i \in [m']} \), we can combine them as follows. If \( \text{vrk}_i \neq \text{vrk}_i', \forall i \in [m], j \in [m'] \), then we can simply multiply the two proofs together. Otherwise, for each \( i, j \) where \( \text{vrk}_i = \text{vrk}_j' \) and \( d_i = d_j' \), we extract \( \pi_i, \pi_j \) from \( \pi \) and \( \pi_j \) from \( \pi' \) via PoKCR.Extract and remove them from \( \pi \) and \( \pi' \). (Since we cannot cross-aggregate across different digests with the same vrk, we require that \( d_i = d_j' \)). Then, we aggregate \( \pi_i \) and \( \pi_j \) together via CFG.Aggr into a \( \pi_{i,j} \). Finally, we add \( \pi_{i,j} \) back into either \( \pi \) or \( \pi' \) (but not both). After repeating this for all \( i, j \) where \( \text{vrk}_i = \text{vrk}_j' \), \( \pi \) and \( \pi' \) 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 (AAD) from Section 4.2 with strong key binding, non-membership proofs, and append-only proofs. An appendix-only proof [29] can convince any verifier that a dictionary \( D \) is a subset of \( D' \), and 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 \( (S_k, \Lambda_k) \), 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 \( \Lambda_k \) as explained in Section 4.2 and additionally checks that \( (S_k)^a_k (B_k)^c_k = g \).
Non-membership proofs. A non-membership proof $\pi_k$ for key $k$ is a non-membership of $e_k$ in the RSA accumulator $S$ (see Section 2.4.2). Specifically, $\pi_k = (\bot, \bot, a_k, B_k)$ where $S^{a_k}(B_k)^e_k = g$. The verifier can easily check the proof satisfies the equation above in $O(l)$ 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 $v_k = \bot$ in the dictionary, and (2) $K_0$, the set of keys that are not in the dictionary (i.e., $v_k = \bot$). Recall that $\pi_k = (S_k, \Lambda_k, a_k, B_k)$, with $S_k$ and $\Lambda_k$ set to $\bot$, 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 $\pi_{K_0} = \text{Acc.NomMemWitAgg}(S, (a_k, B_k, e_k)_{k \in K_0})$ (see Section 4.2). 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 $(S_k, \Lambda_k)$’s can be aggregated into $(S_{K_1}, \Lambda_{K_1})$ 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 $e_k$, w.r.t. $S_k$, as $\pi_{K_1} \leftarrow \text{Acc.NomMemWitCrossAgg}((a_k, B_k, e_k, S_k)_{k \in K_1})$ (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:

$$ S = (S_{K_1})^{e_{K_1}} $$

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

$$ e = \prod_{k \in K_1} (S_k)^{v_k} (\Lambda_k)^{e_{K_1}} $$

$$ 1 = \text{Acc.NomMemWitAggVer}(S_{K_1}, \pi_{K_1}, (e_k)_{k \in K_1}) \text{ (see Fig. 4) (23) } $$

$$ 1 = \text{Acc.NomMemWitAggVer}(S, \pi_{K_0}, (e_k)_{k \in K_0}) \text{ (26) } $$

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 $\pi_k' = (\pi_k, S_k', \Lambda_k', a_k', B_k')$, computed as follows, based on two cases.

Case 1: Adding a new key-value pair $(\hat{k}, \hat{v})$. If $v \neq \bot$, then $(S_{\hat{k}}', \Lambda_{\hat{k}}')$ are computed as before in Section 4.2, while $(a_{\hat{k}}', B_{\hat{k}}') = \text{Acc.NomMemWitUpdAdd}(S_k, a_k, B_k, e_k, e_{\hat{k}})$. If $v = \bot$, then $(a_{\hat{k}}', B_{\hat{k}}') = \text{Acc.NomMemWitUpdAdd}(S, a_k, B_k, e_k, e_{\hat{k}})$.

Case 2: Changing existing key $\hat{k}$’s value by $\hat{v}$. If $v \neq \bot$, the proof update is done as explained in Section 4.2. Otherwise, if $v = \bot$, then the non-membership proof $\pi_k$ 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 $\pi_k$ contains a non-membership witness $(a_k, B_k)$ w.r.t. the RSA accumulator $S_k$. Importantly, observe that the new digest $d' = (\pi', S')$, after removing $\hat{k}$, has exactly $S' = S_k$. Thus, we simply set $\pi_k' = (\bot, \bot, a_k, B_k)$. Otherwise, if $k \neq \hat{k}$, we have two cases based on whether $\pi_k$ is a non-membership proof.

Subcase $\hat{k} \neq k$ and $v = \bot$. In this subcase, we have to update the $(\hat{k}, \hat{v})$ non-membership witness for $e_{\hat{k}}$ w.r.t. $S$ to be a witness w.r.t. $S' = S_k$ as $(a_{\hat{k}}', B_{\hat{k}}') = \text{Acc.NomMemWitUpdDel}(S, a_k, B_k, e_k, e_{\hat{k}})$. 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 $(a_{\hat{k}}', B_{\hat{k}}') = \text{Acc.NomMemWitUpdDel}(S, S', a_k, B_k, e_k, e_{\hat{k}})$. (Note that an aggregated proof $\pi_R$ for all $k \in R$ would suffice in this case, since it contains $S_R = S^{1/e_R} = S'$)

Subcase $\hat{k} \neq k$ and $v \neq \bot$. In this subcase, $S_{\hat{k}}'$ and $\Lambda_{\hat{k}}'$ 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. $S'_{\hat{k}, k} = \text{ShamirTrick}(S_k, S_{\hat{k}, k}, e_k, e_{\hat{k}})$. This can be done as $(a_{\hat{k}}', B_{\hat{k}}') = \text{Acc.NomMemWitUpdDel}(S_k, S'_{\hat{k}, k}, a_k, B_k, e_k, e_{\hat{k}})$. If multiple keys $\hat{k} \in R$ are being
removed, this is handled as in the previous subcase by computing \( S_{R,k} = (S_R)^{1/e_k} \) = ShamirTrick\((S_R, S_k, e_R, e_k)\) and then setting \((a'_k, B'_k) = \text{Acc.NonMemWitUpdDel}(S_k, S_{R,k}, a_k, B_k, e_k, e_R)\). As in the previous subcase, an aggregate proof \( \pi_R \) for all \( k \in R \) 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 \( c' = c^{e_K} \prod_{k \in K} (S'_k)^{\pi_k} = c^{e_K} \prod_{k \in K} (S'^e_K)^{\pi_k} = c^{e_K} S^{e_K} \sum_{k \in K} v_k \cdot e_K \setminus \{k\} \). Let \( z = \sum_{k \in K} v_k \cdot e_K \setminus \{k\} \). The append-only proof \( \pi \) between \( d = (c, S) \) and \( d' = (c', S') \) consists of three PoKE proofs. Specifically, \( \pi^\text{PoKE}_{\subseteq} = \text{PoKE.Prove}(S', S, u), \pi^\text{PoKE}_u = \text{PoKE.Prove}(c^u, c, u) \) and \( \pi^\text{PoKE}_z = \text{PoKE.Prove}(S^z, S, z) \). Note that these PoKE proofs can be further compressed via the PoKCR protocol [3]. The final append-only proof is \( \pi = (\pi^\text{PoKE}_{\subseteq}, \pi^\text{PoKE}_u, Z, \pi^\text{PoKE}_z) \), where \( U = c^u, Z = S^z \).

To verify \( \pi \), one checks if \( c' = UZ, \text{PoKE.Ver}(S', S, \pi^\text{PoKE}_u) \triangleright 1 \), \( \text{PoKE.Ver}(U, c, \pi^\text{PoKE}_u) \triangleright 1 \) and \( \text{PoKE.Ver}(Z, S, \pi^\text{PoKE}_z) \triangleright 1 \). 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 \( \Lambda_k \)'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 \( \text{Acc.NonMemWitCrossAgg}(g, (e_k)_{k \in D}) \).

### 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


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 $c = g^{11x^e} = g^u$ 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:

$$c^a B^x = g \Rightarrow g^{ua} g^{bx} = 1 \Rightarrow au + bx = 1$$ \hspace{1cm} (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 $(a, b)$ 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:

$$c^{asx+atx'} B^x = g \Leftrightarrow (c^{as})^x (c^{at})^{tx'} B^x = g \Leftrightarrow (c')^{at} (c^{as} B)^x = g$$ \hspace{1cm} (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:

$$(c')^{qx+r} (c^{as} B)^x = g$$ \hspace{1cm} (29)

$$(c')^r (c^{as} B)^x = g$$ \hspace{1cm} (30)

$$(c')^r (c^{as} B)^x = g$$ \hspace{1cm} (31)

$$(c')^r (c^{as} B)^x = g$$ \hspace{1cm} (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 $B' = c^{rx+as} B$ as in Fig. 3.

**Updates after deleting an element $x'$.** Next, suppose we deleted an element $x'$ from the set, obtaining a new accumulator $c' = c^{1/x'} = g^{ux'}$. We would like to update the Bézout coefficients $(a, b)$ to $(a', B')$ such that $(c')^{a'} (g^b)^x = g$. Note that we can rewrite Eq. (27) as $c^a B^x = g \Leftrightarrow (c')^{ax} B^x = g$. We could let $B' = B$ and $a' = ax'$, 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:

$$(c')^{aq+r} B^x = g \Leftrightarrow (c')^{ar} B^{qx+r} = g \Leftrightarrow (c')^r ((c')^{aq} B)^x = g$$ \hspace{1cm} (33)

As a result, the updated witness is $B' = (c')^{aq} B$ 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). GenHidOrdGr satisfies this assumption if, $\forall$ adversaries $A$ running in time $\text{poly}(\lambda)$:

$$\Pr \left[ \begin{array}{l} \mathcal{G}_T \leftarrow \text{GenHidOrdGr}(1^\lambda), g \in_R \mathcal{G}_T, \\
(u, e) \leftarrow A(1^\lambda, \mathcal{G}_T, g) : \\
we = g \text{ and } e \text{ is prime} \end{array} \right] \leq \text{negl}(\lambda)$$

Informally, this assumption says that no probabilistic polynomial-time (PPT) adversary can compute any $e$th 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 $e$th 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 \( \text{poly}(\lambda) \), there exists negligible function \( \text{negl}(\cdot) \), such that:

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

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. (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 \( \text{prk} \) and \( \text{vrk} \) 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:

\[
S = S'^e_k = (S'_k)^{e_k} \tag{34}
\]

\[
c = \Lambda^{e_k} S'^e_k = (\Lambda'_k)^{e_k} (S'_k)^{v'} \tag{35}
\]

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

\[
\Lambda^{e_k} S'^e_k = (\Lambda'_k)^{e_k} S'_k \iff (\Lambda_k/\Lambda'_k)^{e_k} = (S_k)^{v'-v} \iff \Lambda^{e_k} = (S_k)^{v'-v}, \text{ where } \Lambda = \Lambda_k/\Lambda'_k \tag{36}
\]

Recall \( \pi \) contains \( (a_k, B_k) \) such that \( (S_k)^{e_k} (B_k)^{e_k} = g \). \( \mathcal{B} \) calls \( (x_k, Y_k) = \text{Acc.} \text{NonMemWitUpdAdd}(S_k, a_k, B_k, e_k, v' - v) \) (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 \( (S_k)^{v'-v} x_k (Y_k)^{e_k} = g \iff (S_k)^{v'-v} = ((Y_k)^{-e_k} g)^{1/x_k} \). Next, rewrite \( \Lambda^{e_k} = (S_k)^{v'-v} \) from Eq. (37) as:

\[
\Lambda^{e_k} = ((Y_k)^{-e_k} g)^{1/x_k} \iff \Lambda^{x_k e_k} = (Y_k)^{-e_k} g \iff \Lambda^{x_k} = Y_k^{-1} g^{1/e_k} \tag{38}
\]

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

\(^1\text{Note that since } e_k \text{ is an } (\ell + 1)\text{-bit prime and the values } v \text{ and } v' \text{ are both } \ell\text{-bit wide, it follows that } \gcd(e_k, v' - v) = 1, \text{ which means this witness update is possible!} \)
B.3.1 Case 1: $v = \bot$ 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 $C$ that, on input the accumulator $S$ and the two inconsistent witnesses for $e_k$, breaks the Strong RSA assumption, $B$ can use $C$ to break Strong RSA.

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

Thus, $B$ can compute $e_k$ and find Bézout coefficients $(x_k, y_k)$ such that $x_k e_k + y_k e_k = 1$. As a result, we have:

$$\Lambda^{e_k} = g^{(x_k e_k)} \Leftrightarrow \Lambda^{x_k e_k} = g^{y_k e_k} \Leftrightarrow \Lambda^{x_k e_k} g^{y_k e_k} = g \Leftrightarrow \Lambda^{x_k e_k} g^{y_k} = g^{1/e_k}$$

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

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 $A$ 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 $A$ broke key binding, $\exists$ a key $z \in I \cap J$ with $v_z \neq v'_z$. We show how to build another adversary $B$ that breaks a random Strong RSA problem instance $g$ by outputting $g^{1/e}$ for some prime $e$. Our adversary $B$ sets up the AD scheme with $prk$ and $v_k$ 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, $B$ can obtain $(S_z, \Lambda_z) \leftarrow$ CFG.Disagg$(I_1, \{z\}, (v_k)_{k \in I_1}, \pi)$ (see Fig. 5) where $S_z = S^{1/e_z}$. Second, $B$ must extract $(a_z, B_z)$ from $\pi_{I_1}$ such that $(S_z)^{a_z} (B_z)^{e_z} = g$. Recall that $\pi_{I_1} = (A_{I_1}, \pi_{I_1}^\text{PoKE}, B_{I_1})$ is a disjointness proof for all keys in $I_1$ (including $z$) such that $\text{PoKE}.\text{Ver}(A_{I_1}, S_{I_1}, \pi_{I_1}^\text{PoKE}) = 1$ and $A_{I_1} B_{I_1}^{e_{I_1}} = g$. Since the PoKE proof verifies, $B$ extracts $a_z$ from the PoKE proof such that $A_{I_1} = S_{I_1}^{1/e_z}$ with non-negligible probability. Thus, $B$ now has $S_z a_z B_{I_1}^{e_{I_1}} = g$.

Next, $B$ can disaggregate the RSA disjointness witness $(a_z, B_z)$ 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. $B$ gets non-membership witness $(\tilde{a}_k, \tilde{B}_k)_{k \in I_1}$ for each $e_k$ w.r.t. $S_{I_1}$ via BreakUpNonMemWit$(S_{I_1}, a_z, B_z, e_z, e_{I_1}/e_z)$ to update $(\tilde{a}_z, \tilde{B}_z)$ 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, $B$ disaggregates $\pi' = (S'_{I_1}, \Lambda'_{I_1}, a'_{I_1}, B'_{I_1})$ for $z$ having value $v'_z \neq v_z$ that verifies against $d = (c, S)$. Finally, $B$ 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, $B$ can disaggregate $\pi$ and obtain $S_z$, which is an RSA membership witness for $e_z$ w.r.t. the accumulator $S$ i.e., $S_z = (S_{t_z})^{e_{i_z} + 1}$.

Next, recall that the proof $\pi'$ contains an RSA disjointness witness $\pi_{k_0} = (A_{k_0}, \pi_{0k}, B_{k_0})$ for all keys $k \in J_0$ that have value $\perp$ w.r.t. the accumulator $S$, such that $A_{k_0}(B_{k_0})_{e_{k_0}} = g$ and PoKE.Ver$(A_{k_0}, S, \pi_{0k}) = 1$. Thus, since the PoKE proof verifies, $B$ can extract $a_{k_0}$ such that $A_{k_0} = S_a^{a_{k_0}}$ with non-negligible probability. Thus, $B$ now has $S^{a_{k_0}}(B_{j_0})_{e_{j_0}} = g$.

Next, $B$ can disaggregate this $(a_{k_0}, B_{j_0})$ RSA disjointness witness for $e_{j_0}$ w.r.t. $S$, into RSA non-membership witnesses $(a_k, B_k)_{e_k} \in J_0$ for each $e_k$ w.r.t. $S$. Then, $B$ can invoke $\text{BreakUpNonMemWit}(S, a_{k_0}, B_{j_0}, (e_k)_{k \in J_0})$. Therefore, $B$ can obtain a non-membership witness $(a_z, B_z)$ for $e_z$ w.r.t. $S$.

$B$ 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 $C$ that, on input $S$ and the two inconsistent witnesses for $e_z$, breaks the Strong RSA assumption, $B$ can use $C$ to break Strong RSA.

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 $A$ 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 $v' = \perp$ w.r.t. digest $d' = (c', S')$, with $v'$ possibly equal to $\perp$, and an append-only proof $\pi_{\perp}$ between $d$ and $d'$. Then, we show how to build another adversary $B$ that, in the generic group model [8], breaks the Strong RSA problem (see Definition B.1).

Recall from Section 4.3 that $\pi_{\perp} = (\pi_u^{\text{PoKE}}, U, \pi_u^{\text{PoKE}}, Z, \pi_z^{\text{PoKE}})$ where $c' = UZ$, PoKE.Ver$(S', S, \pi_u^{\text{PoKE}}) = 1$, PoKE.Ver$(U, c, \pi_u^{\text{PoKE}}) = 1$ and PoKE.Ver$(Z, S, \pi_z^{\text{PoKE}}) = 1$. Since the PoKE proofs verify, $B$ can extract with non-negligible probability $(u, z)$ such that $S' = S^u$ and $c' = c^uS^z$. Next, we split into cases.

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

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

Next, we know from $\pi'_k$ that $(a'_k, B'_k)$ is a non-membership witness for $e_k$ w.r.t. the RSA accumulator $S'_k$. Since $\text{gcd}(e_k, v' - v) = 1$ (because $e_k$ is a $(\ell + 1)$-bit prime and values $v$ and $v'$ are $\ell$ bits), we can update this to a non-membership witness for the RSA accumulator $(S'_k)^{v' - v} = (x_k, Y_k) = \text{Acc}.\text{NonMemWitUpdAdd}(S'_k, a'_k, B'_k, e_k, v' - v)$ such that $(S'_k)^{v' - v} = (x_k, Y_k) = (Y_k)^{-e_k}g^{1/2}$. Finally, replace $(S'_k)^{v' - v}$ in Eq. (42) to get $((Y_k)^{-e_k}g^{1/2})^{1/2} = \Lambda^{e_k} \Leftrightarrow (Y_k)^{-e_k}g = \Lambda^{e_k}$. Thus, $g^{1/2} = \Lambda^{e_k}Y_k$ and $B$ breaks Strong RSA on $g$ by outputting $(\Lambda^{e_k}, \Lambda^{e_k}Y_k)$.

Case 2: $v' = \perp$. In this case, $\pi_k$ contains a membership witness $S_k$ for $e_k$ in $S$ such that $(S_k)^{e_k} = S$. Recall that $B$ extracted $u$ from the append-only proof $\pi_{\perp}$ such that $S' = S^u$. Since $(S_k)^{e_k} = (S_k)^{e_k}u = S^u = S'$, this implies $(S_k)^{e_k}$ is a membership witness for $e_k$ w.r.t. $S'$ as well. However, since $\pi'_k$ keys $k$ 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 $C$ that, on input the accumulator $S'$ and the two inconsistent witnesses for $e_k$, breaks the Strong RSA assumption, $B$ can use $C$ to break Strong RSA.
B.5 Append-only Security for Aggregated Proofs

Proof. Suppose an adversary $A$ breaks append-only security as defined in Definition 4.5 and outputs a lookup proof $\pi$ for keys $k \in K$ having value $v_k = V(k)$ w.r.t. digest $(e, S)$, a lookup proof $\pi'$ for keys $k \in K'$ having value $v'_k = V'(k)$ w.r.t. digest $(e', S')$, such that $\exists z \in K \cap K'$ with $v_z \neq \bot$ 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 $B$ 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 $v_k \neq \bot$ while $k \in K_0$ has value $\bot$. Similarly, $K'$ is also partitioned into $K'_1$ and $K'_0$. As before, we need to consider two separate cases.

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

Case 2: $v_z \neq \bot$ and $v'_z = \bot$. 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 $\pi' = (S_{K'_1}, \Lambda_{K'_1}, \pi_{K'_1}, \pi_{K'_0})$. $B$ proceeds as follows. $B$ computes an RSA membership witness $S_z$ for $e_z$ w.r.t. the accumulator $S$ as $S_z = (S_{K_1})^{e_{K_1}}(z)$. As explained in Appendix B.4, $B$ extracts $s$ from $\pi_\subseteq$ such that $S' = S^w$. Then, $B$ updates $S_z$ to a membership witness $(S_z)^u$ for $e_z$ w.r.t. $S'$. Similar to Appendix B.3.2, $B$ extracts with non-negligible probability $(a_{K'_0}, B_{K'_0})$ from $\pi_{K'_0} = (A_{K'_0}, \pi_{K'_0}^{\text{PoKE}}, B_{K'_0})$ where $A_{K'_0} = (S')^{a_{K'_0}}$ and $(S')^{a_{K'_0}} B_{K'_0}^{a_{K'_0}} = g$. In other words, $B$ extracts an RSA disjointness witness $(a_{K'_1}, B_{K'_1})$ for $e_{K'_0}$ w.r.t. $S'$. Next, $B$ 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 BreakUpNonMemWit($S'$, $a_{K'_1}, B_{K'_1}, (e_k)_{k \in K'_0}$) from [3, Fig. 3, pg. 16]. Thus, $B$ obtains a non-membership witness $(a_z, B_z)$ for $e_z$ w.r.t. $S'$ (since $z \in K'_0$). Since $B$ 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 $C$ that, on input the accumulator $S'$ and the two inconsistent witnesses for $e_z$, breaks the Strong RSA assumption, $B$ can use $C$ to break Strong RSA.

□