HACC: A Hierarchical Accumulator with Constant Public Parameters and Logarithm Time Complexity for Large Evolving Sets

Borui Chen
Fudan University

Liang Zhang
Hong Kong University of Science and Technology

Kexin Li
Fudan University

Dongliang Cai
Fudan University

Jiamian Yan
Fudan University

Haibin Kan
Fudan University

Abstract

Dynamic universal accumulators compress an evolving set into a short digest with membership and non-membership witnesses for each element. Bilinear Pairing (BP) accumulators offer succinct witnesses and fast verification, but their public parameters size, witness generation and dynamic costs scale linearly with the set size in the trapdoorless setting. To address this bottleneck, we present HACC, a trapdoorless hierarchical accumulator that organizes capacity-bounded BP accumulators over ordered buckets into a t-ary tree. By fixing the node capacity t, HACC is a pairing-based accumulator whose public parameters can be independent of the set size n. Its ordered buckets yield native non-membership proofs and $O(t \log_t n)$ witness generation cost, and non-cascading bucket split and merge mechanisms keep it fully dynamic at worst-case $O(t \log_t n)$ dynamic cost with amortized $O(1)$ witness updates. Its witnesses are of size $O(\log_t n)$ and verify with as many pairings, which an optional path compression via polynomial multiproofs reduces to a constant for read-heavy epoch-based settings. We prove HACC correct and sound under the t-SDH assumption in the random oracle model. Experimentally, under comparable budgets, HACC accelerates element update and witness generation by 8.0×–5087× over a monolithic BP accumulator (Nguyen, CT-RSA'05, Damgård et al., eprint'08, and Srinivasan et al., CCS'22) while consuming 15.6×–15933× smaller public parameters.

1 Introduction

Cryptographic accumulators compress a set of elements to a short digest and provide short witnesses that let a verifier check membership or non-membership of elements against that digest. This primitive underlies authenticated-query applications such as stateless blockchains [8, 19], anonymous-credential revocation systems [4, 12, 13], and Proofs of Liabilities for cryptocurrency exchanges and custodians [14, 16, 25, 45]. In these deployments, accumulated sets are large and continuously updated, so practical designs must simultaneously achieve scalability and update efficiency.

An accumulator supports membership witnesses is called *positive*, and is *universal* if it additionally supports non-membership witnesses. It is classified as *static* if the accumulated set is immutable, *additive* or *subtractive* if it supports only element additions or deletions respectively, and *dynamic* if it supports both. Regarding trust assumptions, constructions are further distinguished by their reliance on secret information: *trapdoor-based* accumulators retain a trapdoor to maintain the accumulator digest, whereas *trapdoorless* accumulators maintain the accumulator using only public parameters, with any trapdoor securely destroyed after a trusted setup.

Existing constructions make different trade-offs. Hash-based constructions [19, 33, 38] provide transparent setup and practical scalability, but typically incur logarithmic witness size and rely on additional machinery for non-membership. RSA accumulators [8, 13, 30] support compact witnesses and consume constant public parameters, but they usually depends on hashing to primes and their deletion and witness generation operations are expensive. Bilinear-pairing (BP) accumulators [18, 35, 41, 43] are attractive because of succinct witnesses and efficient verification for both membership and non-membership. However, in the trapdoorless setting, BP accumulators' maximum capacity is limited by public parameter size, and both witness generation and accumulator updates require work linear in the accumulated set size.

These limitations motivate a question: *can a trapdoorless pairing-based accumulator accumulate a large growing set under constant-sized public parameters, with efficient witness generation and dynamic maintenance, while remaining universal?* We answer affirmatively with **HACC**, an efficient and scalable hierarchical accumulator for large evolving sets.

Our approach. We first organize capacity-bounded BP accumulators into a t-ary tree: each node accumulates at most t values, so one $O(t)$-size public parameter serves every node no matter how large the set is, and the set grows by growing the tree (of height only $O(\log_t n)$). Membership under such a tree is straightforward akin to a Merkle tree. The central difficulty is to (i) make the *capacity-bounded* hierarchy

support non-membership, together with (ii) efficient dynamic maintenance: a hierarchy does not by itself locate an *absent* element, and naively rebalancing the tree after element additions or deletions can move many elements and invalidate many witnesses at once.

HACC resolves both through one coherent design built on *ordered buckets*. It splits the accumulated domain into contiguous intervals $[K_L, K_R)$, one per bucket, so every queried element, present or absent, falls into exactly one bucket determined by its value; this makes non-membership as direct as membership, each reducing to a single native BP accumulator proof inside the target bucket plus an authenticated path to the root. Keeping this order under element additions and deletions, however, is quite tricky: were the intervals laid out in sorted order directly on the tree, growing one bucket would ripple through all buckets after it. HACC breaks this coupling by keeping the logical order of intervals *separate* from the physical placement of buckets in the tree, maintained by a *Free Index Pool* (FIP) and a *Bucket Mapping Table* (BMT). An overflowing or underflowing bucket is then repaired *in place* by a split or merge that reuses a free slot, so the structural change is non-cascading: it rewrites only the affected bucket's interval metadata and one or two root-to-leaf paths, while every other node stays intact.

Contributions. We make three main contributions:

1. **Ordered buckets and non-cascading split and merge mechanisms.** We introduce an ordered partition of the accumulated domain into interval-indexed buckets, and a pair of non-cascading bucket split and merge operations that keep buckets within capacity through minimal structural changes, enabled by FIP and BMT. On top of these, we design a witness-update procedure with a case analysis over membership and non-membership witnesses and over whether a split or merge is triggered, refreshing witnesses non-interactively and efficiently.

2. **An efficient and scalable hierarchical accumulator.** Based on these mechanisms, we construct a hierarchical accumulator HACC. It is a trapdoorless pairing-based dynamic universal accumulator whose public parameters are of constant size $O(t)$, so that a single setup serves a continuously growing set. HACC supports $O(t \log_t n)$ witness generation, sustains worst-case $O(t \log_t n)$ additions and deletions with amortized $O(1)$ witness updates, and keeps witnesses of size $O(\log_t n)$; an optional polynomial-multiproof compression further reduces verification to a constant number of pairings for epoch-based scenarios.

3. **Formal analysis and evaluation.** We formalize the syntax and security model, prove correctness and soundness of HACC under the t-SDH assumption in the random-oracle model, and evaluate a prototype against a monolithic BP accumulator, with a cross-family asymptotic comparison to RSA and hash-based constructions.

Table 1: Key asymptotic complexity comparisons.

Metric / Operation	**BP Acc.**	**Our HACC**
Public Parameters (SRS Size)	$O(n)$	$O(t)$
Witness Size ($\pi_y/\overline{\pi}_y$ Size)	$O(1)$	$O(\log_t n)$
Witness Generation (MWG/NMWG)	$O(n)$E	$O(t \log_t n)$E
Elementary Operations (Add/Del)	$O(n)$E	$O(t \log_t n)$E
Witness Verification (MV/NMV)	$O(1)$P	$O(\log_t n)$P
Witness Verification Optimized	–	$O(1)$P

Legend: E: Group Exponentiation; P: Bilinear Pairing.

Notes: n is the number of accumulated elements and t is a pre-defined parameter of HACC (the capacity bound of each BP accumulator node).

Table 1 summarizes the key asymptotic comparisons. t is a pre-defined parameter independent of n. The detailed analysis appears in Section 8.1.

2 Related Work

Cryptographic accumulators. Benaloh and de Mare [6] introduced one-way accumulators as a way to compress a set into a short value while still supporting succinct membership proofs. Barić and Pfitzmann [5] then formalized collision-free accumulators and clarified the security goal of preventing distinct sets from opening to the same digest. Subsequent works proposed various constructions under different assumptions, as detailed in the following three paragraphs.

RSA accumulators. RSA accumulators are realized through exponentiation in groups of unknown order. Camenisch and Lysyanskaya [13] gave the first dynamic construction under strong RSA assumption, whereas a trapdoor is needed for element deletion. Goodrich et al. [23] studied dynamic and distributed accumulator maintenance using RSA one-way accumulators. Li et al. [30] introduced universal accumulators with efficient non-membership proofs by combining RSA-style accumulation with Bézout witnesses. On the setup side, Sander [40] initiated the study of accumulators without a trapdoor, and Lipmaa [31] later obtained secure accumulators from Euclidean rings without trusted setup. Boneh et al. [8] further developed batching techniques in unknown-order groups, especially for read-heavy verification scenarios. The main advantage of RSA accumulators is constant-size parameters and witnesses with no preset capacity bound, but the exponentiation on RSA groups can be expensive.

Hash-based accumulators. Hash-based accumulators are realized by hashing a structured representation of the set. The Merkle tree [33] is the canonical example, but it is only positive and additive, and proving non-membership requires additional structure. Two lines of work add this structure. One imposes an ordering on the elements: authenticated dictionaries over skip lists or search trees [22,34], fixed-universe sparse Merkle trees [17], and indexed Merkle trees [42]. The other builds universal accumulators directly from hashing: Camacho et al. [11] from collision-resistant hashing, Jhanwar and

Tiwari [24] trading accumulation size for witness size via subset differences, and Petkus [38] shortening (non-)membership trees via multicollision resistance. Such constructions are transparent and need no trusted setup, but their witnesses and verification are logarithmic in the set size.

Bilinear-pairing accumulators. Bilinear-pairing (BP) accumulators use characteristic-polynomial commitments in pairing groups to obtain membership and non-membership proofs. Nguyen [35] proposed the first BP accumulator which is positive and dynamic. Damgård and Triandopoulos [18] extended this paradigm to support non-membership proofs, thereby making BP accumulators universal. Srinivasan et al. [41] further developed batching, aggregation, and zero-knowledge techniques for BP accumulators. In trapdoor-based settings, Karantaidou and Baldimtsi [26] gave a more efficient construction, and Vitto and Biryukov [43] proposed a dynamic universal accumulator with batch update. This family offers the most succinct witnesses and the most efficient verification, but in the trapdoorless setting its capacity is bounded by the size of public parameters, and its dynamic maintenance costs grows linearly with the accumulated set size.

Other tree-structured constructions. Several other constructions organize accumulators or commitments in a tree. Goodrich et al. [23] build an *authenticated dictionary* from a trapdoor-based RSA one-way accumulator. They also proposed a hierarchical construction for efficiency, where the tree depth is fixed to a constant ε and the node arity is $O(n^{\varepsilon})$. Papamanthou et al. [36, 37] extend this to *authenticated hash tables* instantiated with RSA or BP accumulators, and handle non-membership by storing pairs of consecutive elements. Verkle trees [28] replace the hash children of a Merkle tree with a constant-size vector commitment, yielding shallow trees with short membership proofs.

3 Preliminaries

Let $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ be multiplicative cyclic groups of prime order p, with g_1 and g_2 being generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. Let $\mathbb{Z}_p$ denote the finite field of integers modulo p and $\mathbb{Z}_p[X]$ denote the polynomial ring over $\mathbb{Z}_p$ in one variable. For a polynomial $f(X) \in \mathbb{Z}_p[X]$, we denote $f(x)$ as the evaluation of $f(X)$ at x, and $\deg(f)$ as the degree of $f(X)$.

Bilinear Pairings. A bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an *efficiently computable* map satisfying (i) *Bilinearity*: For any $u \in \mathbb{G}_1, v \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$, $e(u^a, v^b) = e(u, v)^{ab}$. and (ii) *Non-degeneracy*: $e(g_1, g_2) \neq 1_{\mathbb{G}_T}$.

We denote $\mathsf{bp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2) \leftarrow \mathsf{BilGen}(1^{\lambda})$ as a bilinear pairing instance with security parameter λ.

Polynomial Remainder Theorem (PRT) [2]. For a univariate polynomial $f(X) \in \mathbb{Z}_p[X]$ and a point $x \in \mathbb{Z}_p$, there exist a unique polynomial $q(X) \in \mathbb{Z}_p[X]$ and $r = f(-x) \in \mathbb{Z}_p$ such that $f(X) = q(X)(X + x) + r$.

Bézout's identity [1]. For two polynomials $f(X), g(X) \in \mathbb{Z}_p[X]$, there exist two polynomials $a(X), b(X) \in \mathbb{Z}_p[X]$ such that $a(X)f(X) + b(X)g(X) = d(X)$, where $d(X) = \gcd(f(X), g(X))$. Additionally, $\deg(a) < \deg(g) - \deg(d)$ and $\deg(b) < \deg(f) - \deg(d)$.

3.1 Polynomial Commitment Schemes

Given a polynomial $f(X) \in \mathbb{Z}_p[X]$ of degree at most d, a polynomial commitment scheme (PCS) allows a prover to compute a succinct commitment C to $f(X)$ and later provide a short proof π certifying the evaluation $f(x) = v$ at any given point x. The verifier can then efficiently check this proof against the commitment C.

The KZG scheme [27] is a foundational PCS that uses bilinear pairings to achieve constant-size commitments and proofs. The commitment to a polynomial is evaluated at a secret trapdoor s ($C = g_1^{f(s)}$), and the evaluation proof at point x is constructed by committing to the quotient polynomial $q(X) = (f(X) - f(x))/(X - x)$. Verification relies on a single pairing equation checking the structural relation. Subsequent works [15, 20, 21, 32] expanded KZG to support multiproof techniques. Boneh et al. [9] proposed an efficient polynomial multiproof scheme capable of aggregating the evaluations of multiple polynomials at various points into a single constant-size proof. We leverage this batching capability to compress the verification path in our epoch-based optimizations.

3.2 Accumulator based on Bilinear Pairing

Let $\mathcal{D}$ be the accumulated domain. Under multiplicative group notation, for an accumulated set $X = \{x_1, \ldots, x_n\} \subseteq \mathcal{D}$, its characteristic accumulator polynomial is defined as $X(S) = \prod_{x \in X}(S + x)$. The core algorithms of the bilinear pairing (BP) based accumulator are defined as follows, with abbreviated identifiers of some algorithm names utilized for brevity. We use this BP accumulator as a building block to design our hierarchical accumulator.

$\mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^{\lambda}, n)$: Let $\mathsf{bp} \leftarrow \mathsf{BilGen}(1^{\lambda})$ and n be the maximum capacity. Sample a trapdoor $s \xleftarrow{\$} \mathbb{Z}_p^*$ and generate the public parameters $\mathsf{pp} = \left(\mathsf{bp}, \{g_1^{s^i}\}_{i=0}^{n}, g_2, g_2^s, \mathcal{D} = \mathbb{Z}_p \setminus \{s\}\right)$.
$acc \leftarrow \mathsf{Acc.Commit}(X)$: Compute the accumulator digest acc for set $X \subseteq \mathcal{D}$. $acc = g_1^{X(s)} = g_1^{\prod_{x \in X}(s+x)}$.
$acc \leftarrow \mathsf{Acc.Add}(acc, X, x)$: Add a new element $x \notin X$ to the accumulator and update $acc \leftarrow g_1^{X'(s)} = (acc)^{(s+x)}$.
$acc \leftarrow \mathsf{Acc.Del}(acc, X, x)$: Delete an element $x \in X$ from the accumulator and update $acc \leftarrow g_1^{X'(s)} = (acc)^{1/(s+x)}$.
$\pi_y \leftarrow \mathsf{Acc.MWG}(X, y)$: (Membership Witness Generation). Prove that $y \in X$ resides in the accumulator

and output the membership witness $\pi_y = (acc)^{\frac{1}{s+y}} = g_1^{\frac{\prod_{x\in X\setminus\{y\}}(s+x)}{}}$.

$\overline{\pi}_y \leftarrow \mathsf{Acc.NMWG}(X,y)$: (Non-Membership Witness Generation). Prove that $y \notin X$ is absent from the accumulator. By Bézout's identity, there exist constants α and polynomial $\beta(S)$ satisfying $\alpha \cdot X(S) + \beta(S)\cdot(S+y) = 1$. The non-membership witness is $\overline{\pi}_y = (\alpha,\ g_1^{\beta(s)})$.

$b \in \{0,1\} \leftarrow \mathsf{Acc.MV}(acc, y, \pi_y)$: (Membership Verification). Verify the membership of y by checking the bilinear pairing equation $e(\pi_y, g_2^y \cdot g_2^s) \stackrel{?}{=} e(acc, g_2)$.

$b \in \{0,1\} \leftarrow \mathsf{Acc.NMV}(acc, y, \overline{\pi}_y)$: (Non-Membership Verification). Verify the non-membership of y by checking the equation $e(acc, g_2)^{\alpha} \cdot e(g_1^{\beta(s)}, g_2^y \cdot g_2^s) \stackrel{?}{=} e(g_1, g_2)$.

$\pi_y \leftarrow \mathsf{Acc.MWUpd_{add}}(y, \pi_y, acc_{old}, x)$: (Membership Witness Update on Add). x is the newly added element and acc_{old} is the digest before adding x. The membership of $y \in X$ is updated by $\pi_y \leftarrow acc_{old} \cdot (\pi_y)^{x-y}$.

$\pi_y \leftarrow \mathsf{Acc.MWUpd_{del}}(y, \pi_y, acc_{new}, x)$: (Membership Witness Update on Delete). x is the newly deleted element and acc_{new} is the digest after deleting x. The membership of $y \in X$ is updated by $\pi_y \leftarrow (\pi_y / acc_{new})^{1/(x-y)}$.

$\overline{\pi}_y \leftarrow \mathsf{Acc.NMWUpd_{add}}(y, \overline{\pi}_y, acc_{old}, x)$: (Non-Membership Witness Update on Add). The non-membership of $y \notin X$ is updated by $\alpha' = \alpha \cdot (x-y)^{-1}$ and $\overline{\pi}_y \leftarrow (\alpha',\ g_1^{\beta(s)}(acc_{old})^{-\alpha'})$.

$\overline{\pi}_y \leftarrow \mathsf{Acc.NMWUpd_{del}}(y, \overline{\pi}_y, acc_{new}, x)$: (Non-Membership Witness Update on Delete). The non-membership of $y \notin X$ is updated by $\alpha' = \alpha \cdot (x-y)$ and $\overline{\pi}_y \leftarrow (\alpha',\ g_1^{\beta(s)}(acc_{new})^{\alpha})$.

4 Syntax and Security Definitions

4.1 Definition of Hierarchical Accumulator

A *hierarchical accumulator* (denoted HAcc) over an accumulated domain $\mathcal{D}$ comprises the following ten polynomial-time algorithms. It realizes a standard accumulator with two minor differences: (i) an explicit accumulator state st, and (ii) explicit auxiliary messages aux that drives non-interactive witness maintenance.

$\mathsf{pp} \leftarrow \mathsf{HAcc.Setup}(1^\lambda, t)$: Take a security parameter λ and a capacity bound t for each BP accumulator as input. Output the public parameters pp, after which the underlying trapdoor is destroyed. These parameters pp are implicitly required by all subsequent algorithms.

$(acc, st) \leftarrow \mathsf{HAcc.Commit}(X)$: Take as input a set of initial elements $X \subseteq \mathcal{D}$, output the HACC digest acc and an internal state st that stores the tree structure, bucket interval boundaries, and tree node indices and evaluations.

$(acc', st', \mathtt{aux}_{add}) \leftarrow \mathsf{HAcc.Add}(st, x)$: Add an element $x \in \mathcal{D}\setminus X$ to the accumulator. Output the new HACC digest acc', the updated state st', and an auxiliary update message $\mathtt{aux}_{add}$. The update message $\mathtt{aux}_{add}$ is necessary for non-interactive witness updates.

$(acc', st', \mathtt{aux}_{del}) \leftarrow \mathsf{HAcc.Del}(st, x)$: Delete an element $x \in X$ from the accumulator. Output the new HACC digest acc', the updated state st', and a message $\mathtt{aux}_{del}$.

$\pi_y \leftarrow \mathsf{HAcc.MemWitGen}(st, y)$: Generate a membership witness π_y for element $y \in X$ using the internal state st.

$\overline{\pi}_y \leftarrow \mathsf{HAcc.NonMemWitGen}(st, y)$: Generate a non-membership witness $\overline{\pi}_y$ for an element $y \in \mathcal{D}\setminus X$.

$b \in \{0,1\} \leftarrow \mathsf{HAcc.MemVerify}(acc, y, \pi_y)$: Verify the membership witness π_y for an element y against acc. Output 1 if the witness is valid, and 0 otherwise.

$b \in \{0,1\} \leftarrow \mathsf{HAcc.NonMemVerify}(acc, y, \overline{\pi}_y)$: Verify the non-membership witness $\overline{\pi}_y$ for an element y against acc. Output 1 if the witness is valid, and 0 otherwise.

$\pi'_y \leftarrow \mathsf{HAcc.MemWitUpdate}(y, \pi_y, \mathtt{aux})$: Update the membership witness π_y for element $y \in X$ with the auxiliary update message $\mathtt{aux} \in \{\mathtt{aux}_{add}, \mathtt{aux}_{del}\}$. This operation requires no interaction with the state st.

$\overline{\pi}'_y \leftarrow \mathsf{HAcc.NonMemWitUpdate}(y, \overline{\pi}_y, \mathtt{aux})$: Update the non-membership witness $\overline{\pi}'_y$ for $y \in \mathcal{D}\setminus X$ with aux, without interacting with the state st.

4.2 Security Model

We use a dynamic semantic model. The challenger maintains an accumulator state (acc_i, st_i), initialized from $\mathsf{HAcc.Commit}(S_0)$ and updated by oracle Add and Del. Let $S_i := \mathsf{AccSet}(st_i)$ denote the accumulated set at epoch i. The adversary has adaptive access to Add, Del, MemWitGen (abbreviated as MWG), and NonMemWitGen (abbreviated as NMWG), which exposes the same public information available in stateless settings (updates and honest witnesses).

Concretely, the game starts from

$$(acc_0, st_0) \leftarrow \mathsf{HAcc.Commit}(S_0), \quad S_0 = \mathsf{AccSet}(st_0),$$

then processes adaptive updates one by one. For the i-th valid update query, the state moves from epoch i to epoch $i+1$:

if the query is $\mathsf{Add}(x)$,
$(acc_{i+1}, st_{i+1}, \mathtt{aux}_i) \leftarrow \mathsf{HAcc.Add}(st_i, x),\ S_{i+1} = S_i \cup \{x\}$;
if the query is $\mathsf{Del}(x)$,
$(acc_{i+1}, st_{i+1}, \mathtt{aux}_i) \leftarrow \mathsf{HAcc.Del}(st_i, x),\ S_{i+1} = S_i \setminus \{x\}$.

Definition 1 (Correctness)**.** *HACC is correct if, for every honestly generated state (st_i, acc_i) and every $y \in S_i$,*

$$\Pr\left[\begin{array}{l}\pi_y \leftarrow \mathit{HAcc.MemWitGen}(st_i, y);\\ \mathit{HAcc.MemVerify}(acc_i, y, \pi_y) = 1\end{array}\right] = 1,$$

and for every $y \notin S_i$,

$$\Pr\left[\begin{array}{l}\overline{\pi}_y \leftarrow \mathit{HAcc.NonMemWitGen}(st_i, y);\\ \mathit{HAcc.NonMemVerify}(hacc_i, y, \overline{\pi}_y) = 1\end{array}\right] = 1.$$

For witness updates, correctness requires preservation for unchanged predicates: if $y \in S_i \cap S_{i+1}$ (resp. $y \notin S_i \cup S_{i+1}$), then an accepting membership (resp. non-membership) witness before the update remains accepting after applying the corresponding update algorithm with $\mathtt{aux}_i$.

Definition 2 (Dynamic membership soundness)**.** *HACC satisfies dynamic membership soundness if, for every PPT adversary $\mathcal{A}$ with adaptive oracle access to Add, Del, MWG, and NMWG, there exists a negligible function* negl *such that*

$$\Pr\left[\begin{array}{l}\mathrm{pp} \leftarrow \mathit{HAcc.Setup}(1^\lambda, t);\\ S_0 \leftarrow \mathcal{A}(1^\lambda, \mathrm{pp});\\ (hacc_0, st_0) \leftarrow \mathit{HAcc.Commit}(S_0);\\ (y^*, \pi^*) \leftarrow \mathcal{A}^{\mathit{Add,Del,MWG,NMWG}}(\mathrm{pp}, hacc_0);\\ y^* \notin S_q \ \wedge\ \mathit{HAcc.MemVerify}(hacc_q, y^*, \pi^*) = 1\end{array}\right] \le \mathsf{negl}(\lambda),$$

where q is the final update epoch.

Definition 3 (Dynamic non-membership soundness)**.** *HACC satisfies dynamic non-membership soundness if, for every PPT adversary $\mathcal{A}$ with the same oracle access, there exists a negligible function* negl *such that*

$$\Pr\left[\begin{array}{l}\mathrm{pp} \leftarrow \mathit{HAcc.Setup}(1^\lambda, t);\ S_0 \leftarrow \mathcal{A}(1^\lambda, \mathrm{pp});\\ (hacc_0, st_0) \leftarrow \mathit{HAcc.Commit}(S_0);\\ (y^*, \overline{\pi}^*) \leftarrow \mathcal{A}^{\mathit{Add,Del,MWG,NMWG}}(\mathrm{pp}, hacc_0);\\ y^* \in S_q \ \wedge\\ \mathit{HAcc.NonMemVerify}(hacc_q, y^*, \overline{\pi}^*) = 1\end{array}\right] \le \mathsf{negl}(\lambda),$$

where q is the final update epoch.

5 Basic Concepts and Key Mechanisms

Before presenting the full protocol in Section 6, this section introduces the terminology, data structures, and core mechanisms that HACC relies on. Figure 1 provides a global map of the architecture, including the t-ary accumulator tree, ordered buckets over a sorted domain, hash binding, the resulting HACC digest, and the auxiliary structures used for dynamic maintenance. Each of these components is explained in detail in the subsections that follow. Table 6 in Appendix A summarizes the notations in HACC protocol.

The remainder of this section is organized by mechanism. We first describe the t-ary tree structure, ordered buckets, hash binding, and witness layout. We then present the dynamic maintenance rules, including the Free Index Pool (FIP), Bucket Mapping Table (BMT), and bucket split/merge mechanisms [1]. Finally, we define the auxiliary update message $\mathtt{aux}$ for witness updates and explain how witnesses are updated after element additions/deletions.

[1] Appendix F illustrates a specific example for dynamic maintenance.

5.1 The t-ary Ordered Tree Structure

HACC is organized as an ordered t-ary tree hierarchy where each node maintains a standard BP accumulator with a capacity threshold t. Leaf nodes (name as *buckets*) accumulate the elements in the accumulated set $X = \{x_1, x_2, \ldots, x_n\}$, while internal nodes accumulate their children nodes, serving as structural aggregators for their respective sub-trees. Each bucket is initialized to be half full to avoid frequent splits and merges during dynamic updates. We assume the accumulated domain $\mathcal{D}$ is equipped with a unified total order, enabling a global ordering of all elements. In practice, this can be achieved by injectively mapping elements into $\mathbb{Z}_p$.

Ordered Buckets. To make sure each element corresponds to a unique bucket in the tree, we partition the accumulated domain $\mathcal{D}$ into disjoint, continuous intervals. Each leaf node (bucket) v is responsible for an interval $[K_{L,v}, K_{R,v})$ and accumulates elements that fall within its assigned interval. Buckets that manage two contiguous intervals are referred to as *logically adjacent buckets*. The boundaries of these intervals are determined by the distribution of the accumulated elements; specifically, the boundary between two logically adjacent buckets is initialized as the median between the maximum element of the preceding bucket and the minimum element of the succeeding one.

Each bucket is assigned a unique index idx (starting from 1) that denotes its *physical position* in the tree. Since the tree is organized as a t-ary hierarchy, the parent of a node at index idx can be located at index $\lceil idx/t \rceil$. This indexing scheme allows us to efficiently determine the path from any bucket to the root and compute or update the corresponding accumulator digest at each level.

Hash Binding. To enable hierarchical accumulation and prevent substitution attacks (e.g., moving a node from one branch to another or swapping nodes across levels), HACC enforces a *Hash Binding* mechanism. Each node v is represented by a binding scalar $\mathsf{Bind}_v \in \mathbb{Z}_p$. Let $acc_v \in \mathbb{G}_1$ be the BP accumulator digest of v. The binding scalar is defined as:

$$\mathsf{Bind}_v = \begin{cases} \mathcal{H}(acc_v \parallel lvl_v \parallel idx_v) & v \text{ is internal} \\ \mathcal{H}(acc_v \parallel 0 \parallel idx_v \parallel K_{L,v} \parallel K_{R,v}) & v \text{ is bucket} \end{cases} \tag{1}$$

where $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_p$ is a hash function modeled as a random oracle, lvl_v denotes the node height (with leaves at level 0), and idx_v represents the physical position index of the node within its respective level. For leaf nodes, the binding scalar also incorporates its interval boundaries $K_{L,v}$ and $K_{R,v}$ to prevent an adversary from tampering with bucket boundaries to forge witnesses.

Hierarchical Accumulation. The ordered buckets and hash binding mechanisms facilitate the bottom-up construction of the HACC tree. Elements in the accumulated set are sorted and accumulated to a corresponding leaf bucket. Then the children nodes are recursively accumulated to their parent

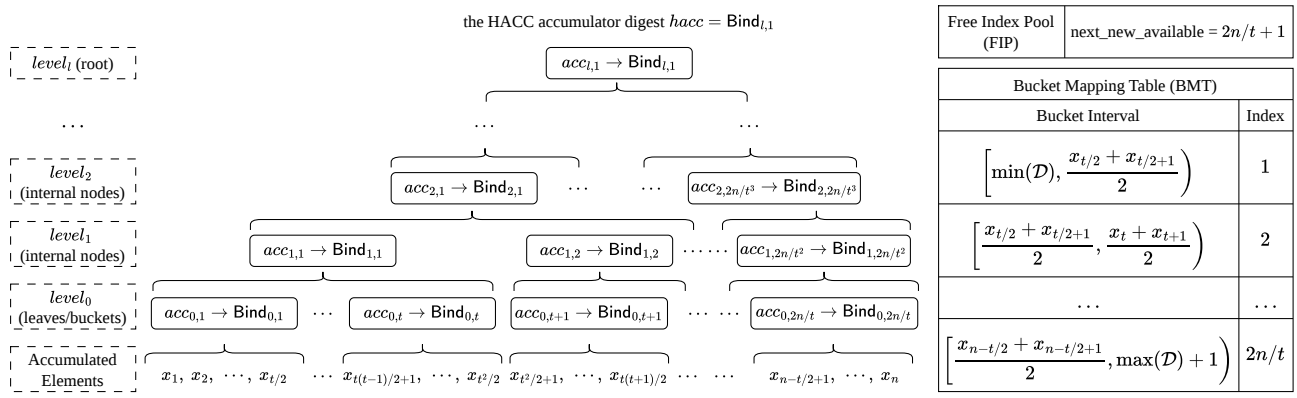


Figure 1: Overall architecture of HACC (illustrated at initialization). Left: a t-ary tree of capacity-bounded BP accumulators. Leaf buckets store contiguous slices of the sorted elements (in initial state, $x_{(i-1)t/2+1}, \ldots, x_{it/2}$ for the i-th bucket) within intervals $[K_{i-1}, K_i)$, internal nodes accumulate the binding scalars Bind of their children, and the root binding scalar $\mathsf{Bind}_{l,1}$ serves as the HACC digest *hacc*. Right: the Bucket Mapping Table (BMT) maps logical interval order to physical leaf indices, with interval boundaries set between adjacent elements (midpoints when initialized), while the Free Index Pool (FIP) tracks reusable leaf indices (no such indices in the initial state) and the next new available index for bucket splits/merges.

nodes: the set of elements accumulated by an internal parent node v consists of the binding scalars $\{\mathsf{Bind}_{u_i}\}$ of its children $\{u_i\}_{i=1}^{k}$ (where $k \le t$).

Witness Structure. A (non-)membership witness consists of a bucket (non-)membership witness π_{bucket} ($\overline{\pi}_{bucket}$) for the corresponding bucket and a path witness

$$\mathsf{PathWit} = \left\{ \left(\pi_{path}^{(level)},\ acc_{level+1} \right) \right\}_{level=0}^{l-1}$$

that validates the integrity of the path from that bucket to the root, where $\pi_{path}^{(level)}$ is the membership witness of the node at level $level$ and $acc_{level+1}$ is the accumulator digest of its parent node. l is the height of the tree.

5.2 Dynamic Maintenance

Adding or deleting an element involves inserting it into or removing it from the corresponding bucket, followed by a bottom-up update of the accumulator digests along the bucket-to-root path. However, frequent additions and deletions may cause buckets to overflow or underflow, necessitating buckets splitting and merging.

FIP and BMT. To avoid cascading adjustments to parent and sibling nodes (e.g. moving elements across buckets or adjusting parent-child relationships) triggered by bucket splits or merges, HACC maintains a global *Free Index Pool* (FIP) to decouple logical interval management from physical index allocation. The FIP tracks all currently reusable physical indices for leaf slots in the tree, as well as the next new available index for future bucket allocations. When a new bucket is created, HACC allocates a free index from the FIP to mark its physical position; if no free indices are available, it allocates the next new available index. When a bucket is merged, its physical index is recycled back into the FIP. The FIP always recycles the larger index and allocates the smallest available index to maintain tree compactness.

HACC also maintains a *Bucket Mapping Table* (BMT) to record the mapping between each bucket's logical interval and its physical index. This allows buckets to be physically arranged in the tree without following their logical order, and any element in a specific interval can be located to its bucket.

Dynamic maintenance of the tree relies on the following two mechanisms, Bucket_Split and Bucket_Merge. The pseudocodes are shown in Algorithms 1 and 2, respectively. Besides updating the affected leaf buckets, both procedures also prepare the Hints payload introduced in Section 5.3.

Algorithm 1 $\mathsf{Bucket_Split}(st,\ idx_0,\ idx_{new},\ X^*)$

Retrieve $[K_L, K_R)$ from $st[0, idx_0]$.
Identify the median $K_{mid} \leftarrow \mathsf{median}(X^*)$.
$X_{left} \leftarrow \{x \in X^* \mid x < K_{mid}\}$, $X_{right} \leftarrow X^* \setminus X_{left}$.
$acc_{left}, acc_{right} \leftarrow \mathsf{Acc.Commit}(X_{left}), \mathsf{Acc.Commit}(X_{right})$.
Update Bind_{0,idx_0} and compute $\mathsf{Bind}_{0,idx_{new}}$.
$st[0, idx_0] \leftarrow (X_{left}, acc_{left}, \mathsf{Bind}_{0,idx_0}, [K_L, K_{mid}))$.
$st[0, idx_{new}] \leftarrow (X_{right}, acc_{right}, \mathsf{Bind}_{0,idx_{new}}, [K_{mid}, K_R))$.
$\mathsf{Hints} \leftarrow \emptyset$.
for each $y \in X^*$ **do**
    Compute (A_y, Ω_y) as per Equation (4).
    $idx \leftarrow idx_0$ if $y \in X_{left}$ else idx_{new}.
    Append (y, A_y, Ω_y, idx) to Hints.
end for
return Hints.

Bucket_Split. Triggered when an insertion causes a leaf bucket v to exceed capacity t. Its logical interval $[K_L, K_R)$ is

Algorithm 2 Bucket_Merge(st, idx_0, idx_{sib}, X^*)

Retrieve $[K_L, K_M), [K_M, K_R)$ from $st[0, idx_0]$ and $st[0, idx_{sib}]$.
$acc_{unified} \leftarrow \mathsf{Acc.Commit}(X^*)$. Update Bind_{0,idx_0}.
Update $st[0, idx_0] \leftarrow (X^*, acc_{unified}, \mathsf{Bind}_{0,idx_0}, [K_L, K_R))$.
Delete $st[0, idx_{sib}]$.
$\mathsf{Hints} \leftarrow \emptyset$.
for each $y \in X^*$ **do**
    Compute (A_y, Ω_y) as per Equation (6).
    Append $(y, A_y, \Omega_y, idx_0)$ to Hints.
end for
return Hints.

partitioned at a pivot K_{mid}, dividing it into two buckets of $t/2$ elements each. The original bucket v is updated to govern $[K_L, K_{mid})$ while retaining its physical index idx_v. Simultaneously, a vacant identifier idx_{new} is retrieved from the FIP and allocated to a newly instantiated leaf v_{new}, which assumes the interval $[K_{mid}, K_R)$. v_{new} is then assigned to the physical leaf position indexed by idx_{new}, and its binding scalar $\mathsf{Bind}_{v_{new}}$ is recursively accumulated into the tree hierarchy.

Bucket_Merge. Triggered when deletions cause two logically adjacent buckets to collectively hold no more than $t/2$ elements. Their contiguous intervals are assigned to a single bucket v, which retains its existing index idx_v. The redundant bucket $v_{obsolete}$ is removed, and its physical index $idx_{obsolete}$ is returned to FIP.

5.3 Witness Update Auxiliary Message

While BP accumulator supports non-interactive witness updates using old and new digests and the added or deleted elements, HACC requires a more complex approach due to hierarchical structure and bucket split and merge mechanisms. To address this, we explicitly define an auxiliary message aux to enable witness updates.

$$\mathsf{aux} = (\mathsf{op},\ x,\ \mathsf{BucketInfo},\ \mathsf{PathInfo},\ \mathsf{Hints},\ \mathsf{HeightDelta})$$

where $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ indicates the type of update operation, x is the element being added or deleted, BucketInfo contains the updated bucket's logical interval and physical index information, PathInfo is a sequence of updates for each node along the path from the modified bucket to the root, HeightDelta is the signed tree-height variation caused by this update, and Hints provides additional information for witness updates in cases bucket splitted or merged. Information contained in aux is discussed in two cases.

Case 1: No Bucket Splits or Merges. If an addition or deletion does not trigger a bucket split or merge, it only affects a single path from the modified bucket to the root. The modification of each node along this path is a deletion of a child's old binding scalar and an addition of its new binding scalar. The auxiliary message aux contains the information about the modified bucket, defined as BucketInfo, the new and old digests of each node along this path, defined as PathInfo, as well as the element x that was added or deleted. The witness can then be non-interactively updated by applying BP accumulator update algorithms.

$$\mathsf{BucketInfo} = \{(idx,\ acc_{0,idx},\ K_{L,idx},\ K_{R,idx})\}$$

where idx is the physical index of the modified bucket, $acc_{0,idx}$ is the new accumulator digest of that bucket, and $K_{L,idx}$ and $K_{R,idx}$ are the left and right boundaries of that bucket's interval.

$$\mathsf{PathInfo} = \left\{(level,\ child_idx,\ \mathsf{op},\ \mathsf{Bind},\ acc^{before},\ acc^{after})\right\}$$

where each tuple in PathInfo records one elementary child update at some internal node: $level$ indicates the node's level in the tree, $child_idx$ indicates which child is modified, $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ indicates whether the change is an addition or deletion of a child binding scalar, Bind is the added or deleted child binding scalar, acc^{before} is the accumulator digest of the node before adding or deleting Bind, and acc^{after} is the digest after the operation.

Case 2: Bucket Splitted or Merged. If an addition or deletion triggers a bucket split or merge, the update is equivalent to a batch of element additions and deletions. Camacho et al. show a linear lower bound on witness update complexity in this setting [10], so we provide additional information in

$$\mathsf{Hints} = \{(y, A_y, \Omega_y, idx)\}$$

for efficient witness updates. Each tuple in Hints corresponds to an affected element y in the modified bucket, $A_y \in \mathbb{Z}_p$ and $\Omega_y \in \mathbb{G}_1$ are the update coefficients for y's witness, and idx indicates the physical index of the bucket that contains y. The update strategy differs between membership and non-membership witnesses.

For membership witnesses, since HACC maintains the element set X_v of the affected bucket, it can compute Hints for each affected member $y \in X_v$ (no more than t members in total) to enable a $O(1)$ update. When an overflowing bucket v partitions into v_1 and v_2 with respective element subsets X_{v_1} and X_{v_2}, assume $y \in X_{v_1}$. Let $X_{v_1}(S)$ and $X_{v_2}(S)$ denote the accumulator polynomials of the respective subsets. By PRT (Section 3), $X_{v_2}(S) = Q(S)(S+y) + R$, where $R = X_{v_2}(-y)$. The original witness can be expressed as:

$$\pi_y = g_1^{\frac{X_v(s)}{s+y}} = g_1^{\frac{X_{v_1}(s)[Q(s)(s+y)+R]}{s+y}} = (acc_{v_1})^{Q(s)} \cdot (\pi'_y)^R \tag{2}$$

The updated witness is derived as:

$$\pi'_y = (\pi_y)^{R^{-1}} \cdot (acc_{v_1})^{-Q(s)R^{-1}} \tag{3}$$

The Hints contains the tuple $(y, A_y, \Omega_y, idx_{v_1})$ where

$$A_y = R^{-1}, \quad \Omega_y = (acc_{v_1})^{-Q(s)R^{-1}} = g_1^{-(X_{v_1} \cdot Q)(s)R^{-1}} \tag{4}$$

and the witness can be updated in $O(1)$ time: $\pi'_y = (\pi_y)^{A_y} \cdot \Omega_y$.

When two buckets v_1 and v_2 merge into v, assuming $y \in X_{v_1}$, PRT yields $X_{v_2}(S) = Q(S)(S+y) + R$, where $R = X_{v_2}(-y)$. The membership witness after merging is computed as:

$$\pi'_y = g_1^{\frac{X_{v_1}(s) X_{v_2}(s)}{s+y}} = (\pi_y)^R \cdot (acc_{v_1})^{Q(s)} \tag{5}$$

The Hints contains the tuple $(y, A_y, \Omega_y, idx_{v_1})$ where

$$A_y = R, \quad \Omega_y = (acc_{v_1})^{Q(s)} = g_1^{(X_{v_1} \cdot Q)(s)} \tag{6}$$

For non-membership witnesses, any element $y \in [K_{L,v}, K_{R,v}) \setminus X_v$ may query, making precomputing hints for all potential non-members across the interval computationally infeasible. Instead, when a split or merge occurs, witness holders first reconstruct the new accumulated set X_v of the affected bucket by collecting the first components of all tuples in Hints whose bucket index equals the updated physical index, and then compute the new non-membership witness with $\mathsf{Acc.NMWG}(X_v, y)$. This requires $O(t)$ multi-scalar multiplications, but since the bucket is initialized to half capacity, this cost is amortized over $\Omega(t)$ updates without bucket splits or merges, resulting in an amortized update cost of $O(1)$. Moreover, if the logical interval $L = K_{R,v} - K_{L,v}$ of a bucket v is small, the number of potentially affected non-members is bounded. In such case, we can iterate over all $y \in [K_{L,v}, K_{R,v}) \setminus X_v$ and compute Bézout hints in $O(t \cdot L)$ time. Broadcasting these hints enables non-membership witness update in $O(1)$ time.

Bucket splits and merges may also cause changes in tree height, affecting all bucket-to-root paths. Therefore, aux also includes the signed tree-height variation

$$\mathsf{HeightDelta} = l_{new} - l_{old}.$$

6 The Full HACC Protocol

In this section, we formally define each algorithm of HACC defined in Section 4 and present the full protocol. We also provide an optional optimization for path witness verification in Section 6.3.

6.1 Static Operations

$\mathsf{pp} \leftarrow \mathsf{HAcc.Setup}(1^\lambda, t)$. Given a security parameter λ and a capacity bound t, the public parameters are instantiated by directly invoking the base BP accumulator setup defined in Section 3:

$$\mathsf{pp} \leftarrow \mathsf{Acc.Setup}(1^\lambda, t)$$

$(hacc, st) \leftarrow \mathsf{HAcc.Commit}(X)$. Elements in the accumulated set $X \subseteq \mathcal{D}$ are sorted in ascending order and partitioned into $2n/t$ groups, $X_1, X_2, \ldots, X_{2n/t}$, each containing $t/2$ elements. Each group X_i is committed to a leaf bucket as $acc_{0,i}$. Notably, buckets are initialized to half-capacity to provide a $\Omega(t)$ buffer that prevents immediate overflows or underflows during subsequent element additions or deletions. The binding scalars $\{\mathsf{Bind}_{0,i}\}$ of the leaf buckets are then grouped by a factor of t and committed to internal nodes at level 1. This process continuously proceeds until the hierarchy converges to a single root node $acc_{l,1}$. Its binding scalar $\mathsf{Bind}_{l,1}$ serves as the final HACC digest $hacc$. The state of each node in the tree, including its accumulated set, accumulator digest, binding scalar, and for leaf nodes, interval boundaries, is recorded in st. The full algorithm is specified in Algorithm 3.

Algorithm 3 $\mathsf{HAcc.Commit}(X)$

Sort X in ascending order: $x_1 < x_2 < \cdots < x_n$. $N \leftarrow 2n/t$.
Initialize tree state $st \leftarrow \emptyset$.
// Step 1: Leaf Bucket Instantiation (Level 0)
Set initial bound $K_0 \leftarrow \min(\mathcal{D})$ and $K_N \leftarrow \max(\mathcal{D}) + 1$.
for $idx = 1$ **to** N **do**
    Allocate continuous $t/2$ elements $X_{idx} \subseteq X$ to bucket idx.
    $K_{idx} \leftarrow (\max(X_{idx}) + \min(X_{idx+1}))/2$ if $idx \neq N$.
    $acc_{0,idx} \leftarrow \mathsf{Acc.Commit}(X_{idx})$.
    $\mathsf{Bind}_{0,idx} \leftarrow \mathcal{H}(acc_{0,idx} \parallel 0 \parallel idx \parallel K_{idx-1} \parallel K_{idx})$.
    $st[0, idx] \leftarrow (X_{idx}, acc_{0,idx}, \mathsf{Bind}_{0,idx}, [K_{idx-1}, K_{idx}))$.
end for
$\mathcal{L}_{curr} \leftarrow \{\mathsf{Bind}_{0,1}, \ldots, \mathsf{Bind}_{0,N}\}$. $level \leftarrow 0$.
// Step 2: Bottom-Up Recursive Tree Construction
while $|\mathcal{L}_{curr}| > 1$ **do**
    $level \leftarrow level + 1$. $\mathcal{L}_{next} \leftarrow \emptyset$.
    Segment $\mathcal{L}_{curr}$ into chunks $C_1, \ldots, C_k$, each containing t binding scalars except the last one.
    for each partitioned chunk $C_m = \{d_1, \ldots, d_{|C_m|}\}$ **do**
        $acc_{level,m} \leftarrow \mathsf{Acc.Commit}(C_m)$.
        $\mathsf{Bind}_{level,m} \leftarrow \mathcal{H}(acc_{level,m} \parallel level \parallel m)$.
        $st[level, m] \leftarrow (C_m, acc_{level,m}, \mathsf{Bind}_{level,m})$.
        Append $\mathsf{Bind}_{level,m}$ to $\mathcal{L}_{next}$.
    end for
    $\mathcal{L}_{curr} \leftarrow \mathcal{L}_{next}$.
end while
return the digest $hacc \leftarrow \mathcal{L}_{curr}[0]$ and the tree state st.

$\pi_y \leftarrow \mathsf{HAcc.MemWitGen}(st, y)$. To generate a membership witness for an element y, the algorithm first locates the target leaf bucket idx that corresponds to y from BMT. It retrieves the state of this bucket and then generates the bucket membership witness π_{bucket} for y using the BP accumulator's membership witness generation algorithm. Finally, it constructs the path witness PathWit by iteratively retrieving the parent nodes up to the root and generating membership witnesses for the corresponding binding scalars at each level. The complete algorithm is specified in Algorithm 4.

$\overline{\pi}_y \leftarrow \mathsf{HAcc.NonMemWitGen}(st, y)$. The non-membership witness generation algorithm is similar to the membership case, with the key difference being that in Step 2, it asserts

Algorithm 4 $\mathsf{HAcc.MemWitGen}(st, y)$

// Step 1: Bucket Location
Locate target leaf idx where $K_{idx-1} \le y < K_{idx}$ from BMT.
Retrieve $st[0, idx] = (X_{idx}, acc_{0,idx}, \mathsf{Bind}_{0,idx}, [K_L, K_R))$.
// Step 2: Local Witness Generation
Assert $y \in X_{idx}$. Compute $\pi_{bucket} \leftarrow \mathsf{Acc.MWG}(X_{idx}, y)$.
// Step 3: Path Witness Generation
$\mathsf{PathWit}_y \leftarrow \emptyset$, $curr_idx \leftarrow idx$. Let l denote the tree height.
for $level = 0$ **to** $l-1$ **do**
  $parent \leftarrow \lceil curr_idx/t \rceil$.
  Retrieve $C_{parent}, acc_{level+1,parent}, \mathsf{Bind}_{level+1,parent}$ from st.
  $\pi_{path}^{(level)} \leftarrow \mathsf{Acc.MWG}(C_{parent}, \mathsf{Bind}_{level,curr_idx})$.
  $\mathsf{PathWit}_y.\mathsf{append}\left(\pi_{path}^{(level)},\ acc_{level+1,parent}\right)$.
  $curr_idx \leftarrow parent$.
end for
return $\pi_y = (\pi_{bucket}, K_L, K_R, idx, acc_{0,idx}, \mathsf{PathWit}_y)$.

$y \notin X_{idx}$ and generates a bucket non-membership witness $\overline{\pi}_{bucket} \leftarrow \mathsf{Acc.NMWG}(X_{idx}, y)$ for y. The path witness generation remains the same.

$b \in \{0,1\} \leftarrow \mathsf{HAcc.MemVerify}(hacc, y, \pi_y)$. To verify a membership witness π_y for element y, the algorithm first validates the bucket witness π_{bucket} to ensure that y is indeed accumulated in the correct bucket. It then iteratively verifies the path witness $\mathsf{PathWit}_y$ from the leaf to the root, checking that each node's binding scalar is accumulated in its parent node. Finally, it asserts that the computed root binding scalar matches the HACC digest $hacc$. The complete verification algorithm is specified in Algorithm 5.

Algorithm 5 $\mathsf{HAcc.MemVerify}(hacc, y, \pi_y)$

Parse π_y as $(\pi_{bucket}, K_L, K_R, idx, acc_0, \mathsf{PathWit}_y)$.
Parse $\mathsf{PathWit}_y$ as $\{(\pi_{path}^{(0)}, acc_1), \ldots, (\pi_{path}^{(l-1)}, acc_l)\}$.
// Step 1: Local Bucket Validation
Assert $K_L \le y < K_R$ and $\mathsf{Acc.MV}(acc_0, y, \pi_{bucket})$.
// Step 2: Upward Path Verification
$\mathsf{Bind}_{curr} \leftarrow \mathcal{H}(acc_0 \parallel 0 \parallel idx \parallel K_L \parallel K_R)$.
$curr_idx \leftarrow idx$.
for $level = 0$ **to** $l-1$ **do**
  Assert $\mathsf{Acc.MV}(acc_{level+1}, \mathsf{Bind}_{curr}, \pi_{path}^{(level)})$.
  $\mathsf{Bind}_{curr} \leftarrow \mathcal{H}(acc_{level+1} \parallel level+1 \parallel \lceil curr_idx/t \rceil)$.
  $curr_idx \leftarrow \lceil curr_idx/t \rceil$.
end for
Assert $\mathsf{Bind}_{curr} == hacc$.
return TRUE

$b \in \{0,1\} \leftarrow \mathsf{HAcc.NonMemVerify}(hacc, y, \overline{\pi}_y)$. The verification of a non-membership is similar to the membership case, with the key difference being that in Step 1, it asserts $\mathsf{Acc.NMV}(acc_0, y, \overline{\pi}_{bucket})$ in the target bucket. The path verification remains the same.

6.2 Dynamic Operations

$(hacc', st', \mathtt{aux}_{add}) \leftarrow \mathsf{HAcc.Add}(st, x)$. To add an element x to the accumulated set, the algorithm first locates the target bucket index idx corresponding to x and retrieves its state. It checks if adding x would exceed the bucket's capacity t. If not, it simply add x into the bucket by BP accumulator's addition algorithm. If adding x causes an overflow, it triggers a Bucket_Split, which creates a new bucket and redistributes elements. The algorithm then computes node modification infomation $\Delta^{(0)}$, where each entry is a tuple of the form $(idx, \mathsf{Bind}^{old}, \mathsf{Bind}^{new})$ representing the modified bucket index and its old and new binding scalars. Finally, a subrutine Path_Refresh is invoked to perform an upward (i.e. bottom-up) path refresh, returning the new HACC digest $hacc'$, the updated tree state st', and the auxiliary witness update message aux defined in Section 5.3. The complete algorithm is specified in Algorithm 6, and formal details of Path_Refresh can be found in Appendix B.1.

Algorithm 6 $\mathsf{HAcc.Add}(st, x)$

// Step 1: Target Bucket Localization
Locate target bucket index idx such that $K_{idx-1} \le x < K_{idx}$.
Retrieve bucket state $st[0, idx]$.
$\mathsf{is_splitted} \leftarrow \mathsf{FALSE}$, $\mathsf{Hints} \leftarrow \emptyset$.
// Step 2: Capacity Bounds Check and Bucket Update
Assert $x \notin X_{idx}$. Let $X_{idx}^* \leftarrow X_{idx} \cup \{x\}$.
if $|X_{idx}^*| > t$ **then** **// Bucket Overflow - Trigger Split**
  $\mathsf{is_splitted} \leftarrow \mathsf{TRUE}$.
  Request new bucket index idx_{new} from FIP.
  $\mathsf{Hints} \leftarrow \mathsf{Bucket_Split}(st,\ idx,\ idx_{new},\ X_{idx}^*)$.
else **// Standard Addition**
  $acc_{0,idx} \leftarrow \mathsf{Acc.Add}(acc_{0,idx}, x)$.
  $X_{idx} \leftarrow X_{idx}^*$. Update $\mathsf{Bind}_{0,idx}$ and $st[0, idx]$.
end if
// Step 3: Upward Path Refresh
$\Delta^{(0)} \leftarrow \{\left(idx, \mathsf{Bind}_{0,idx}^{old}, \mathsf{Bind}_{0,idx}^{new}\right)\}$.
if $\mathsf{is_splitted}$ **then**
  $\Delta^{(0)} \leftarrow \Delta^{(0)} \cup \{\left(idx_{new}, \emptyset, \mathsf{Bind}_{0,idx_{new}}^{new}\right)\}$.
end if
return $\mathsf{Path_Refresh}(st, \Delta^{(0)}, \mathsf{op} = \mathsf{add}, x, \mathsf{Hints})$.

$(hacc', st', \mathtt{aux}_{del}) \leftarrow \mathsf{HAcc.Del}(st, x)$. The deletion algorithm is similar to the addition case, with the key difference being that it checks if removing x would cause the bucket to underflow (i.e., two logically adjacent buckets contain no more than $t/2$ elements). If so, it triggers a Bucket_Merge.

The algorithm then computes the node modification information $\Delta^{(0)}$ and invokes Path_Refresh to update the tree and generate the auxiliary message aux.

$\pi'_y \leftarrow \mathsf{HAcc.MemWitUpdate}(y, \pi_y, \mathsf{aux})$. The witness update algorithm takes an existing membership witness π_y for an element y and an auxiliary message aux generated from a recent addition or deletion operation. It first parses the auxiliary message to determine if the update affects the bucket containing y. If so, it updates the bucket witness π_{bucket} accordingly, either through a standard BP accumulator update or by applying hints in the case of a bucket split or merge. Finally, it updates the path witness $\mathsf{PathWit}_y$ using the Sync_Path subroutine. For any intermediate node along the path that has been modified, its modification is a deletion of a child's old binding scalar and an addition of its new binding scalar. Using the $(acc^{before}, acc^{after})$ transcript in PathInfo, Sync_Path can either refresh an existing path level or initialize a newly entered parent level when the bucket's physical index changes. The complete algorithm of HAcc.MemWitUpdate is specified in Algorithm 7, and details of Sync_Path in Appendix B.2.

Algorithm 7 $\mathsf{HAcc.MemWitUpdate}(y, \pi_y, \mathsf{aux})$

Parse π_y as $(\pi_{bucket}, K_L, K_R, idx, acc_0, \mathsf{PathWit}_y)$ and aux as
  (op, x, BucketInfo, PathInfo, Hints, HeightDelta).
// Step 1: Bucket State & Witness Update
affected $\leftarrow$ FALSE.
$idx^{old} \leftarrow idx$, $K_L^{old} \leftarrow K_L$, $K_R^{old} \leftarrow K_R$.
$\mathsf{Bind}_{old}^{(0)} \leftarrow \mathcal{H}(acc_0 \parallel 0 \parallel idx^{old} \parallel K_L^{old} \parallel K_R^{old})$.
for each $(idx', acc_0^{new}, K'_L, K'_R) \in$ BucketInfo **do**
  if $K'_L \le y < K'_R$ **then**
    $idx \leftarrow idx'$, $K_L \leftarrow K'_L, K_R \leftarrow K'_R$.
    $acc_0^{old} \leftarrow acc_0$, $acc_0 \leftarrow acc_0^{new}$.
    affected $\leftarrow$ TRUE, **break**
  end if
end for
if affected **then**
  if Hints $== \emptyset$ **then** // Standard Update
    $acc_{temp} \leftarrow acc_0^{old}$ if op == add else acc_0.
    $\pi_{bucket} \leftarrow \mathsf{Acc.MWUpd}_{\mathsf{op}}(y, \pi_{bucket}, acc_{temp}, x)$.
  else // Structural Restructuring
    Extract the tuple (y, A_y, Ω_y, idx) from Hints.
    $\pi_{bucket} \leftarrow (\pi_{bucket})^{A_y} \cdot \Omega_y$.
  end if
end if
// Step 2: Path Witness Update
$\mathsf{Bind}_{new}^{(0)} \leftarrow \mathcal{H}(acc_0 \parallel 0 \parallel idx \parallel K_L \parallel K_R)$.
$\mathsf{PathWit}_y \leftarrow \mathsf{Sync_Path}(idx^{old}, idx, \mathsf{PathWit}_y,$
  $\mathsf{PathInfo}, \mathsf{HeightDelta}, \mathsf{Bind}_{old}^{(0)}, \mathsf{Bind}_{new}^{(0)})$.
return $\pi'_y = (\pi_{bucket}, K_L, K_R, idx, acc_0, \mathsf{PathWit}_y)$.

$\overline{\pi}'_y \leftarrow \mathsf{HAcc.NonMemWitUpdate}(y, \overline{\pi}_y, \mathsf{aux})$. The update of a non-membership witness is similar to the membership case, with the key difference being that it updates the bucket non-membership witness $\overline{\pi}_{bucket}$ instead of π_{bucket} in Step 1, as introduced in Section 5.3. The path witness update in Step 2 remains the same.

6.3 Path Witness Optimization

In the standard HACC framework described above, verifying a witness π_y ($\overline{\pi}_y$) of element y requires one pairing check of π_{bucket} ($\overline{\pi}_{bucket}$) and $O(\log_t n)$ pairing operations to validate the $\mathsf{PathWit} = \{\pi_{path}^{(k-1)}, acc_k\}_{k=1}^{l}$. While $\log_t n$ can be small, the cumulative cost of pairing computations introduces non-trivial overhead in resource-constrained environments, such as EVM, where gas consumption is a primary concern. To address this efficiency bottleneck, we introduce a *Path Witness Optimization*, reducing the validation complexity of PathWit to one pairing check.

Note that BP accumulator membership witnesses are KZG-style opening witnesses at the hidden evaluation point s. For an accumulator polynomial $X(S) = \prod_{x \in X}(S + x)$ and digest $acc = g_1^{X(s)}$, a membership witness $\pi_y = g_1^{X(s)/(s+y)}$ is exactly a commitment to the quotient polynomial for opening at the evaluation point $S = -y$ (where $X(-y) = 0$). In HACC path verification, each tuple check $\mathsf{Acc.MV}(acc_k, \mathsf{Bind}_{k-1}, \pi_{path}^{(k-1)})$ is thus a KZG-style opening check at the point $(-\mathsf{Bind}_{k-1})$ for the polynomial committed by its parent acc_k. Hence, the l path-level checks can be aggregated into one statement using KZG polynomial multiproofs [9].

Compressed Path Proof Generation. This part of the protocol can be appended to the end of HAcc.MemWitGen (Algorithm 4) and HAcc.NonMemWitGen. Let Bind_k denote the binding scalar relative to acc_k and $f_k(S)$ be the corresponding accumulator polynomial such that $acc_k = g_1^{f_k(s)}$. Let $T_k = \{\mathsf{Bind}_k\}$ and $T = \cup_{k=1}^{l} T_k$. We define polynomials $Z_T(X) = \prod_{k=1}^{l}(X + \mathsf{Bind}_k)$ and $Z_{T \setminus T_k}(X) = \prod_{i \neq k}(X + \mathsf{Bind}_i)$. A Fiat-Shamir scalar challenge $\gamma = \mathcal{H}_{FS}(hacc, acc_1, \ldots, acc_l)$ is generated. Let polynomial $h(S) = \sum_{k=1}^{l} \gamma^{k-1}(f_k / Z_{T_k})(S)$ and a commitment $W \in \mathbb{G}_1$ is computed as

$$W = g_1^{h(s)} = \prod_{k=1}^{l} \left(\pi_{path}^{(k)}\right)^{\gamma^{k-1}} \tag{7}$$

A secondary scalar challenge parameter $z = \mathcal{H}_{FS}(\gamma, W)$ is sequentially derived. Let polynomial $f_z(S) = \sum_{k=1}^{l} \gamma^{k-1} Z_{T \setminus T_k}(z) f_k(S)$ and calculate the quotient polynomial $L'(S) = \frac{f_z(S) - Z_T(z) h(S)}{S - z}$. Another commitment $W' \in \mathbb{G}_1$ is computed as

$$W' = g_1^{L'(s)} \tag{8}$$

The compressed path proof σ_{path} is then defined as:

$$\sigma_{path} = \left(W,\ W',\ \{acc_k\}_{k=1}^{l}\right) \tag{9}$$

Compressed Path Proof Verification. This part of the protocol can replace the original path verification step in HAcc.MemVerify (Algorithm 5, Step 2) and HAcc.NonMemVerify. Given σ_{path}, the stateless verifier deterministically recreates the challenges γ, z, and evaluates $Z_{T\setminus T_k}(z), Z_T(z)$. The verifier then computes $F \in \mathbb{G}_1$ as:

$$F = \left(\prod_{k=1}^{l} (acc_k)^{\gamma^{k-1} \cdot Z_{T\setminus T_k}(z)} \right) \cdot W^{-Z_T(z)} \tag{10}$$

and checks the following pairing equation:

$$e(F \cdot (W')^z, g_2) \stackrel{?}{=} e(W', g_2^s) \tag{11}$$

While this batching optimization achieves a constant-time pairing complexity for verification, it compromises the non-interactive witness updatability inherent in the standard HACC. Because the batched proofs (W, W') are tied to random challenges (γ, z) that will be altered by any element update. Consequently, this optimization is specifically intended for periodically synchronized, read-intensive applications, such as Proof of Liability (PoL) or stateless blockchain protocols. In these epoch-based architectures—where periodic snapshots provide temporary stability—the system can take advantage of the $O(1)$ verification complexity during the static phase between each epoch.

7 Security Analysis

This section states the main security conclusions for HACC in the model of Section 4.2. We keep only the high-level proof strategy here; full proofs are deferred to Appendix C.

Assumptions. We assume the structural binding hash is modeled as a random oracle and the BP accumulator security is based on the t-SDH assumption [7, 18, 35].

Theorem 1 (Security guarantee)**.** *HACC satisfies correctness. Under the assumptions above, HACC satisfies dynamic membership soundness and dynamic non-membership soundness.*

Proof idea. The proof of correctness is trivial. The proof of soundness follows a two-layer reduction. First, accepted path witnesses are reduced to path authenticity: either the adversary causes a hash-binding inconsistency (random-oracle bad event), or it forges an internal BP accumulator membership witness. Second, once the path is authentic, the claim is tied to a unique honest bucket by interval checks; therefore any false accepted membership/non-membership statement induces a forgery against the corresponding local BP accumulator soundness property under t-SDH. Combining these events gives negligible adversarial advantage. □

For complete proofs, see Appendix C, especially Theorem 4 for correctness, and Theorems 5, 6, and 7 for soundness.

8 Performance Analysis

8.1 Theoretical Complexity

We establish the asymptotic complexity of HACC in comparison to RSA accumulator [13, 30], Indexed Merkle Tree [42] (IMT) and BP accumulator [18, 35, 41], as summarized in Table 2. Let n be the number of accumulated elements and t the node capacity of HACC, yielding a tree depth $l = O(\log_t n)$.

Table 2: Asymptotic comparison of universal accumulator constructions. n is the set size; t is HACC's node capacity.

Metric / Property	RSA Acc.	IMT	BP Acc.	Our HACC
Underlying primitive	RSA group	hash	pairing	pairing
Transparent setup	no	yes	no	no
Public parameters	$O(1)$	$O(1)$	$O(n)$	$O(t)$
Membership witness	$O(1)$	$O(\log n)$	$O(1)$	$O(\log_t n)$
Non-membership witness	$O(1)$	$O(\log n)$	$O(1)$	$O(\log_t n)$
Witness generation	$O(n)$R	$O(\log n)$H	$O(n)$E	$O(t \log_t n)$E
Element addition	$O(1)$R	$O(\log n)$H	$O(n)$E	$O(t \log_t n)$E
Element deletion	$O(n)$R†	–‡	$O(n)$E	$O(t \log_t n)$E
Witness update	$O(1)$R	$O(\log n)$H	$O(1)$E	am. $O(1)$E
Verification	$O(1)$R	$O(\log n)$H	$O(1)$P	$O(\log_t n)$P
Verification (compressed)	–	–	–	$O(1)$P

Legend: R: RSA group exponentiation; H: hash; E: group exponentiation; P: pairing. am.: amortized.
† Trapdoorless RSA Acc. deletion recomputes over the remaining set ($O(n)$ elements).
‡ IMT is append-only: labels are never removed, only inserted and their values updated.

For space complexity, a BPACC needs an SRS as large as the set it commits to ($O(n)$), which caps the accumulable set. HACC instead commits only to buckets of capacity t, so its SRS is $O(t)$ and independent of n — matching the constant public parameters of RSA and hash-based accumulators. The price is witness size: a BPACC has a constant-size witness, whereas a HACC witness bundles one bucket witness and l path witnesses over its tree, giving $\Theta(l) = O(\log_t n)$ (reduced to constant by the optional path compression). The update auxiliary message aux is amortized $O(\log_t n)$ by Theorem 2.

Theorem 2 (Amortized size of update message)**.** *Over any sequence of valid additions and deletions, the amortized size of HACC auxiliary message* aux *is* $O(\log_t n)$.

Proof sketch. Every update carries one modified leaf-to-root transcript contributing $O(\log_t n)$. Split/merge events add $O(t)$ hint payload, but these events are sparse because buckets are rebalanced around half capacity; the extra $O(t)$ term amortizes to $O(1)$ per update. Full proof is in Appendix D. □

For time complexity, we focus on the pairing family where HACC and BPACC share the same operations. Committing to a set of n elements costs $O(n)$E for both. Witness generation drops from $O(n)$E for BPACC to $O(t \log_t n)$E for HACC, which builds one bucket witness and one per-level path witness, each over at most t elements. For dynamic operations, BPACC recomputes the whole accumulator at $O(n)$E, while HACC updates only the (at most two) affected leaf-to-root paths at $O(t \log_t n)$E; witness update stays $O(1)$E for BPACC

Table 3: Public-parameter size (bytes) as n grows. The last column is the BPACC/HACC ratio at t=256.

n	BPACC	HACC t=32	HACC t=64	HACC t=128	HACC t=256	Ratio (t=256)
2^{12}	263,560	2512	4560	8656	16848	15.6×
2^{14}	1,049,992	2512	4560	8656	16848	62.3×
2^{16}	4,195,720	2512	4560	8656	16848	249.0×
2^{18}	16,778,632	2512	4560	8656	16848	995.9×
2^{20}	67,110,280	2512	4560	8656	16848	3983×
2^{22}	268,436,872	2512	4560	8656	16848	15,933×

and amortized $O(1)\mathsf{E}$ for HACC (Theorem 3). The one metric where HACC pays is verification: BPACC verifies in $O(1)$ pairings, whereas HACC runs one check per level, $O(\log_t n)\mathsf{P}$, which the path optimization reduces back to $O(1)\mathsf{P}$.

Theorem 3 (Amortized witness-update cost)**.** *Over any sequence of valid additions and deletions, the amortized cost of HACC witness update is* $O(1)\mathsf{E}$.

Proof sketch. The proof has two components. (i) *Bucket-local part*: for update epoch u, let n_u be current set size and let $m_u \le t$ be size of the touched bucket. A uniformly sampled witness falls in that bucket with probability $m_u/n_u \le t/n_u$. Conditioned on being in the touched bucket, split/merge has $O(t)\mathsf{E}$ cost but occurs only once per $\Omega(t)$ bucket updates, so conditional amortized cost is $O(1)\mathsf{E}$. Hence $\mathbb{E}[C_u^{\text{bucket}}] = O(t/n_u)\mathsf{E} \le O(1)\mathsf{E}$. (ii) *Path part*: let L be the lowest node shared by the added/deleted element and the path. For entries in the path, those below L remain unchanged, and those above L are retrieved from `aux`. Only the single entry at L needs a BP witness update, so $C_u^{\text{path}} = O(1)\mathsf{E}$. Combining both yields the theorem. Full proof is in Appendix D. □

8.2 Implementation and Evaluation

We implemented both our HACC and a baseline BP accumulator (BPACC) in Go, using `gnark-crypto` over the BN254 curve. Our evaluation reports three groups of results: (i) public-parameter size (Table 3); (ii) runtime of six core operations—Setup, Commit, MemWitGen, NonMemWitGen, Add, Del (Figure 2, Table 4); and (iii) witness size and verification cost, standard and compressed (Figure 3, Table 5). Witness update cost, split/merge triggered addition/deletion cost, and auxiliary message sizes are deferred to Appendix E. We use $t \in \{32, 64, 128, 256\}$ and n ranging over 2^{12} to 2^{24}; BPACC is reported up to $n = 2^{22}$, beyond which its runtime exceeds practical benchmarking budget on our test platform. All experiments run on a MacBook with an Apple M4 CPU, 16 GB RAM, macOS, and Go 1.23.

Public parameter size. Table 3 shows the scalability separation: at every fixed t, HACC's public parameters are constant in n (ranging from 2.5 KB at t=32 to 16.8 KB at t=256), whereas BPACC grows linearly to 268 MB at $n = 2^{22}$. At

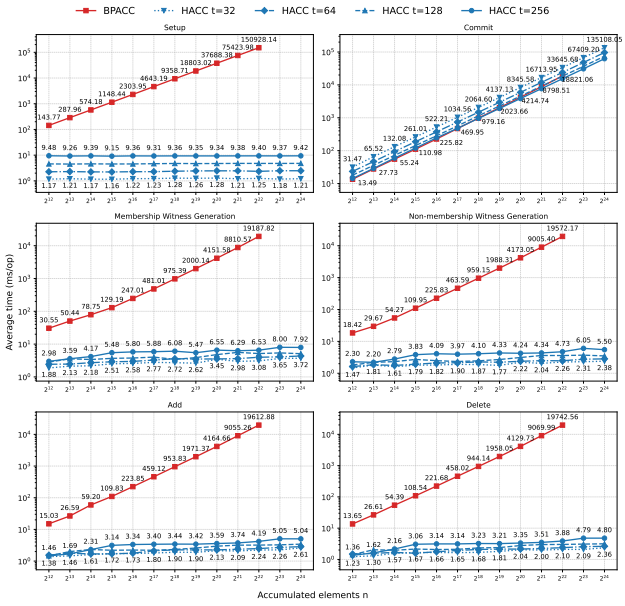


Figure 2: Runtime of six core operations (average ms/op).

Table 4: Speedup (BPACC/HACC) on the six core operations.

n	Setup	Commit	MemWitGen	NonMemWitGen	Add	Del
2^{12}	15.2×	0.9×	10.2×	8.0×	10.3×	10.1×
2^{14}	61.2×	0.9×	18.9×	19.4×	25.7×	25.2×
2^{16}	246×	0.9×	42.6×	55.2×	67.1×	70.6×
2^{18}	1000×	1.0×	160×	234×	277×	292×
2^{20}	4033×	1.1×	634×	984×	1161×	1233×
2^{22}	16064×	1.2×	2938×	4136×	4680×	5087×

t=256 the BPACC/HACC ratio grows from 15.6× at n=2^{12} to 15,933× at n=2^{22}, matching the $O(n)$ vs. $O(t)$ asymptotics. Once t is chosen, a single setup serves an arbitrarily growing set with constant public parameters.

Efficiency on six core operations. Figure 2 plots six core operations on log–log axes as n grows. BPACC rises linearly on every operation. HACC, at every fixed t, stays far below and grows only sublinearly: Setup is essentially flat because it provisions only the $O(t)$ parameters, while MemWitGen, NonMemWitGen, Add, and Del rise in discrete steps as the tree gains height and its top level fills from 2 to t children. Table 4 quantifies the speedups at t=256. The gap widens as n grows, and the largest speedups are 2938× (MemWitGen), 4136× (NonMemWitGen), 4680× (Add), and 5087× (Del). For Commit, HACC and BPACC are quite comparable because they both perform $O(n)$ global commitment work.

Witness size and verification. As a hierarchical construction, HACC trades a larger witness and higher verification cost for its parameter and update advantages. Figure 3 shows that at t=256, HACC's witness is 328 B (360 B for non-membership) at tree depth 2 and 456 B (488 B for non-membership) at depth 3, versus BPACC's constant 64 B (96 B non-membership); all stay within a few hundred bytes. Verification follows the same

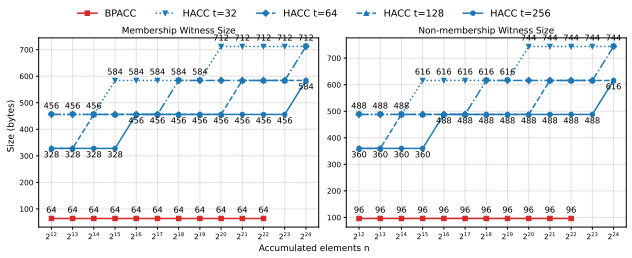


Figure 3: Witness size (bytes).

Table 5: Verification time (ms/op). Cells are shown as (BPACC/HACC); compressed columns are HACC-only.

n	MemVerify	NonMemVerify	Cmp. MemVerify	Cmp. NonMemVerify
2^{12}	(0.83/3.49)	(1.04/3.53)	(−−/3.22)	(−−/3.22)
2^{14}	(0.65/3.14)	(0.82/3.44)	(−−/2.83)	(−−/2.83)
2^{16}	(0.68/4.27)	(0.84/4.02)	(−−/2.54)	(−−/2.60)
2^{18}	(0.69/4.32)	(0.84/4.06)	(−−/2.54)	(−−/2.62)
2^{20}	(0.77/4.35)	(0.93/4.06)	(−−/2.53)	(−−/2.64)
2^{22}	(1.05/3.92)	(1.28/3.85)	(−−/2.38)	(−−/2.57)

pattern (Table 5), but all remains in a low millisecond range.

The choice of t. These results make the role of t concrete: a larger t shortens the tree, reducing witness size and verification cost, while a smaller t lowers per-operation cost. A small t already covers a vast range: with $t = 256$ (16.8 KB public parameters), a tree of depth 3 holds up to $256^3 \approx 16$ million elements and depth 4 holds over $256^4 \approx 4$ billion.

Overall, the experimental results indicate that HACC achieves significant efficiency gains, preserves practical verification costs, and offers remarkably better scalability with constant public parameter size.

9 Applications of HACC

HACC serves as a scalable state-commitment primitive for trust-minimized systems that maintain large dynamic sets and require succinct public verification.

Stateless Blockchains. Stateless blockchains commit to state through an accumulator so that validators and light clients need not store it in full [8, 19]. HACC can commit to a growing UTXO or nullifier set, with block producers publishing the new root and witness-update message each block. Over a trapdoorless BP accumulator, it keeps succinct membership and non-membership witnesses while cutting public parameters from $O(n)$ to $O(t)$ and update and witness-generation cost from $O(n)$ to $O(t \log_t n)$—the scaling that continuously growing state demands.

Anonymous Credentials. Anonymous credential systems use accumulators for privacy-preserving revocation and related dynamic state [3,4,12,13,29,39,44], where accumulated elements are valid or revoked credential handles. HACC preserves dynamic universal functionality while avoiding the linear public-parameter and proving cost of a trapdoorless BP accumulator over the whole credential population, so frequent issuance, revocation, and issuer rotation scale gracefully.

Proof of Liabilities. Proofs of Liabilities let an exchange commit to its liability ledger so users can verify their own balances [14, 16, 25]; Notus [45] shows accumulators fit dynamic liability snapshots. The exchange accumulates liability records into a HACC root per audit epoch and distributes witnesses. This setting is a natural fit: it is epoch-based and read-heavy, so after each snapshot the compressed-path optimization gives verifiers a constant, account-count-independent online cost, while HACC's $O(t)$ parameters keep the prover side scalable for large ledgers.

10 Discussion: Beyond Bilinear Pairings

Although we instantiate HACC with a bilinear-pairing accumulator, its organizing mechanism — capacity-bounded ordered buckets, Free Index Pool and Bucket Mapping Table, and non-cascading bucket splits and merges — is independent of the underlying accumulator. Replacing each tree node with an RSA accumulator yields an RSA variant that likewise bounds every node to t elements and maintains the set by local restructuring. The benefit is reducing the $O(n)$ cost for witness generation and element deletion to $O(t \log_t n)$. The path compression relies on KZG-style polynomial multiproofs and does not carry over in RSA groups, so an RSA variant takes $O(\log_t n)$ group exponentials for witness verification. In short, the mechanism is general and per-operation speedup works out in both BP and RSA family.

11 Conclusion

We presented HACC, a trapdoorless hierarchical accumulator built from capacity-bounded bilinear pairing accumulators over ordered intervals. By fixing the node capacity t, HACC is a pairing-based accumulator with constant $O(t)$ public parameters, independent of the accumulated set size. Hierarchical accumulation over ordered buckets yields $O(t \log_t n)$ witness generation cost, while non-cascading bucket splits and merges mechanisms guarantee worst-case $O(t \log_t n)$ updates and amortized $O(1)$ witness maintenance.

We formalized the syntax and security model and proved correctness and soundness under the t-SDH assumption in the random oracle model. Our complexity analysis and prototype evaluation confirm that HACC removes the linear parameter and update bottleneck of a monolithic BP accumulator while retaining succinct witnesses, and an optional path compression reduces verification to constant pairings for read-heavy deployments. These properties make HACC a practical accumulation primitive for large, continuously evolving sets such as stateless blockchains, anonymous credentials, and Proofs of Liabilities.

References

[1] Bézout's identity. *Wikipedia*, March 2026.

[2] Polynomial remainder theorem. *Wikipedia*, January 2026.

[3] Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. Dynamic Universal Accumulators for DDH Groups and Their Application to Attribute-Based Anonymous Credential Systems. In Marc Fischlin, editor, *Topics in Cryptology – CT-RSA 2009*, pages 295–308, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-00862-7_20.

[4] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with Applications to Anonymity-Preserving Revocation. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 301–315, April 2017. doi:10.1109/EuroSP.2017.13.

[5] Niko Barić and Birgit Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 480–494, Berlin, Heidelberg, 1997. Springer. doi:10.1007/3-540-69053-0_33.

[6] Josh Benaloh and Michael de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 274–285, Berlin, Heidelberg, 1994. Springer. doi:10.1007/3-540-48285-7_24.

[7] Dan Boneh and Xavier Boyen. Short Signatures Without Random Oracles. In *Advances in Cryptology - EUROCRYPT 2004*, pages 56–73, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-24676-3_4.

[8] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 561–586, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-26948-7_20.

[9] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Efficient polynomial commitment schemes for multiple points and polynomials, 2020.

[10] Philippe Camacho and Alejandro Hevia. On the Impossibility of Batch Update for Cryptographic Accumulators. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *Progress in Cryptology – LATINCRYPT 2010*, pages 178–188, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-14712-8_11.

[11] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong Accumulators from Collision-Resistant Hashing. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security*, pages 471–486, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-85886-7_32.

[12] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In Stanisław Jarecki and Gene Tsudik, editors, *Public Key Cryptography – PKC 2009*, pages 481–500, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-00468-1_27.

[13] Jan Camenisch and Anna Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 61–76, Berlin, Heidelberg, 2002. Springer. doi:10.1007/3-540-45708-9_5.

[14] Konstantinos Chalkias, Kevin Lewi, Payman Mohassel, and Valeria Nikolaenko. Distributed Auditing Proofs of Liabilities, 2020.

[15] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology – EUROCRYPT 2020*, pages 738–768, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-45721-1_26.

[16] Gaby G. Dagher, Benedikt Bünz, Joseph Bonneau, Jeremy Clark, and Dan Boneh. Provisions: Privacy-preserving Proofs of Solvency for Bitcoin Exchanges. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 720–731, New York, NY, USA, October 2015. Association for Computing Machinery. doi:10.1145/2810103.2813674.

[17] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient Sparse Merkle Trees. In Billy Bob Brumley and Juha Röning, editors, *Secure IT Systems*, pages 199–215, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-47560-8_13.

[18] Ivan Damgård and Nikos Triandopoulos. Supporting Non-membership Proofs with Bilinear-map Accumulators, 2008.

[19] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set, 2019.

[20] Ariel Gabizon. AuroraLight: Improved prover efficiency and SRS size in a Sonic-like system, 2019.

[21] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge, 2019.

[22] Michael T. Goodrich and Roberto Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing, October 2001. Manuscript dated October 18, 2001. URL: https://cs.brown.edu/cgc/stms/papers/hashskip.pdf.

[23] Michael T. Goodrich, Roberto Tamassia, and Jasminka Hasić. An Efficient Dynamic and Distributed Cryptographic Accumulator*. In Agnes Hui Chan and Virgil Gligor, editors, *Information Security*, pages 372–388, Berlin, Heidelberg, 2002. Springer. doi:10.1007/3-540-45811-5_29.

[24] Mahabir Prasad Jhanwar and Pratyush Ranjan Tiwari. Trading Accumulation Size for Witness Size: A Merkle Tree Based Universal Accumulator Via Subset Differences, 2019.

[25] Yan Ji and Konstantinos Chalkias. Generalized Proof of Liabilities. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3465–3486, New York, NY, USA, November 2021. Association for Computing Machinery. doi:10.1145/3460120.3484802.

[26] Ioanna Karantaidou and Foteini Baldimtsi. Efficient Constructions of Pairing Based Accumulators. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16, June 2021. doi:10.1109/CSF51468.2021.00033.

[27] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology - ASIACRYPT 2010*, volume 6477, pages 177–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-17373-8_11.

[28] John Kuszmaul. Verkle Trees. Technical report, MIT PRIMES, 2018.

[29] Chen Li, Jianting Ning, Shengmin Xu, Chao Lin, Jiguo Li, and Jian Shen. DTACB: Dynamic Threshold Anonymous Credentials With Batch-Showing. *IEEE Transactions on Information Forensics and Security*, 19:7744–7758, 2024. doi:10.1109/TIFS.2024.3443622.

[30] Jiangtao Li, Ninghui Li, and Rui Xue. Universal Accumulators with Efficient Nonmembership Proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-72738-5_17.

[31] Helger Lipmaa. Secure Accumulators from Euclidean Rings without Trusted Setup. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security*, pages 224–240, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-31284-7_14.

[32] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2111–2128, New York, NY, USA, November 2019. Association for Computing Machinery. doi:10.1145/3319535.3339817.

[33] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer. doi:10.1007/3-540-48184-2_32.

[34] Moni Naor and Kobbi Nissim. Certificate Revocation and Certificate Update. In *7th USENIX Security Symposium (USENIX Security 98)*, 1998.

[35] Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 275–292, Berlin, Heidelberg, 2005. Springer. doi:10.1007/978-3-540-30574-3_19.

[36] Charalampos Papamanthou. *Cryptography for Efficiency: New Directions in Authenticated Data Structures*. PhD thesis, Brown University, 2011.

[37] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Cryptographic Accumulators for Authenticated Hash Tables, 2009.

[38] Maksym Petkus. Efficient (Non-)Membership Tree from Multicollision-Resistance with Applications to Zero-Knowledge Proofs, 2024.

[39] Sietse Ringers. Using the RSA or RSA-B accumulator in anonymous credential schemes, 2023.

[40] Tomas Sander. Efficient Accumulators without Trapdoor Extended Abstract. In Vijay Varadharajan and Yi Mu, editors, *Information and Communication Security*, pages 252–262, Berlin, Heidelberg, 1999. Springer. doi:10.1007/978-3-540-47942-0_21.

[41] Shravan Srinivasan, Ioanna Karantaidou, Foteini Baldimtsi, and Charalampos Papamanthou. Batching, Aggregation, and Zero-Knowledge Proofs in Bilinear Accumulators. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2719–2733,

Los Angeles CA USA, November 2022. ACM. doi:10.1145/3548606.3560676.

[42] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency Dictionaries with Succinct Proofs of Correct Operation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2022. Cryptology ePrint Archive, Paper 2021/1263.

[43] Giuseppe Vitto and Alex Biryukov. Dynamic Universal Accumulator with Batch Update over Bilinear Groups. In *Topics in Cryptology – CT-RSA 2022*, pages 395–426, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-030-95312-6_17.

[44] Sijiang Xie, Rui Shi, Yang Yang, Huiqin Xie, Yingjiu Li, and Robert H. Deng. AccCred: Improved Accountable Anonymous Credentials With Dynamic Triple-Hiding Committees. *IEEE Transactions on Dependable and Secure Computing*, 22(3):1961–1975, May 2025. doi:10.1109/TDSC.2024.3474060.

[45] Jiajun Xin, Arman Haghighi, Xiangan Tian, and Dimitrios Papadopoulos. Notus: Dynamic Proofs of Liabilities from Zero-knowledge RSA Accumulators. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1453–1470, 2024.

A Key Notations Used in HACC

Table 6: Key Notations Used in HACC

Notation	Description
$\mathcal{D}$	Accumulated domain
X	Accumulated set of elements. $X \subseteq \mathcal{D}$
t	Capacity for each BP accumulator in HACC
$hacc$	HACC digest (binding scalar of the root node)
st	Internal state of the HACC tree
l	Height of the HACC tree
lvl_v	Level of node v in the tree (leaves at level 0)
idx_v	Physical index (start from 1) of node v in its level
$L_{L,v}, L_{R,v}$	Left and right interval boundaries for bucket v
$acc_{lvl,idx}$	BP accumulator digest of node at level lvl with index idx
$\mathsf{Bind}_{lvl,idx}$	Hash binding scalar of node at level lvl with index idx
$\pi_y, \overline{\pi}_y$	(Non-)Membership witness for element y
$\mathsf{aux}_{add/del}$	Witness update auxiliary message for addition/deletion
PathWit	Path witness, consists of membership witnesses of nodes along the path from a bucket to the root
BucketInfo	Information about the affected buckets after element addition or deletion
PathInfo	Information about the affected path after element addition or deletion
Hints	Additional hints for efficient witness updates
HeightDelta	Tree-height variation after addition or deletion

B Subroutines Invoked in Section 6

B.1 Path Refresh

An entry $(idx, \mathsf{Bind}^{old}, \mathsf{Bind}^{new})$ in $\Delta^{(level)}$ means that, at level $level$, the node at physical index idx changes its binding scalar from Bind^{old} to Bind^{new} (either value can be $\emptyset$ for pure addition or deletion). The subroutine Path_Refresh is formally defined in Algorithm 8. It executes three steps. First, it propagates changes level by level: for each affected parent, it deletes and adds the corresponding child binding scalar in increasing child-index order, updates the parent accumulator, and appends one elementary tuple to PathInfo per operation. The resulting parent binding scalar changes form $\Delta^{(level+1)}$. Second, once no more levels are affected, it derives $(hacc_{new}, l_{new})$ and sets $\mathsf{HeightDelta} \leftarrow l_{new} - l_{old}$. Third, it extracts the refreshed BucketInfo from $\Delta^{(0)}$, assembles

$$\mathsf{aux} = (\mathsf{op}, x, \mathsf{BucketInfo}, \mathsf{PathInfo}, \mathsf{Hints}, \mathsf{HeightDelta}),$$

and returns $(hacc_{new}, st, \mathsf{aux})$.

B.2 Sync Path

The subroutine Sync_Path (Algorithm 9) consumes the transcript PathInfo produced by Algorithm 8 to perform a bottom-

Algorithm 8 $\mathsf{Path_Refresh}(st, \Delta^{(0)}, \mathsf{op}, x, \mathsf{Hints})$

Initialize $\mathsf{PathInfo} \leftarrow \emptyset$, record the old HACC digest $hacc_{old}$ and the old tree height l_{old}.
$level \leftarrow 0$.
// Perform Bottom-Up Refresh Along Affected Paths
while $\Delta^{(level)} \neq \emptyset$ **do**
  Group the records in $\Delta^{(level)}$ by parent index $parent = \lceil idx/t \rceil$ as U_{parent}.
  Initialize $\Delta^{(level+1)} \leftarrow \emptyset$.
  for each affected $parent$ **do**
    $acc_{level+1,parent} \leftarrow \mathsf{Acc.Commit}(\emptyset)$ if $st[level+1, parent]$ is absent else retrieve it from st.
    $curr_acc \leftarrow acc_{level+1,parent}$. $\mathsf{Bind}^{old}_{level+1,parent} \leftarrow \emptyset$ if $st[level+1, parent]$ is absent else retrieve it from st.
    for each $(idx, \mathsf{Bind}^{old}, \mathsf{Bind}^{new}) \in U_{parent}$, in increasing idx order **do**
      if $\mathsf{Bind}^{old} \neq \emptyset$ **then**
        $acc^{before} \leftarrow curr_acc$. $curr_acc \leftarrow \mathsf{Acc.Del}(curr_acc, \mathsf{Bind}^{old})$.
        Append $(level+1, idx, \mathsf{del}, \mathsf{Bind}^{old}, acc^{before}, curr_acc)$ to $\mathsf{PathInfo}$.
      end if
      if $\mathsf{Bind}^{new} \neq \emptyset$ **then**
        $acc^{before} \leftarrow curr_acc$. $curr_acc \leftarrow \mathsf{Acc.Add}(curr_acc, \mathsf{Bind}^{new})$.
        Append $(level+1, idx, \mathsf{add}, \mathsf{Bind}^{new}, acc^{before}, curr_acc)$ to $\mathsf{PathInfo}$.
      end if
    end for
    if the updated parent $curr_acc$ is non-empty **then**
      Update $acc_{level+1,parent} \leftarrow curr_acc$ and compute $\mathsf{Bind}^{new}_{level+1,parent}$ by Equation (1).
      Update $st[level+1, parent]$, and append $(parent, \mathsf{Bind}^{old}_{level+1,parent}, \mathsf{Bind}^{new}_{level+1,parent})$ to $\Delta^{(level+1)}$.
    else
      Delete $st[level+1, parent]$ and append $(parent, \mathsf{Bind}^{old}_{level+1,parent}, \emptyset)$ to $\Delta^{(level+1)}$.
    end if
  end for
  $level \leftarrow level + 1$.
end while
// Record Signed Tree-Height Variation
Determine the new root binding scalar $hacc_{new}$ and the new tree height l_{new}.
$\mathsf{HeightDelta} \leftarrow l_{new} - l_{old}$.
// Assemble Witness Update Auxiliary Message
Extract the updated bucket metadata $\mathsf{BucketInfo} \leftarrow \{(idx,\ acc_{0,idx},\ K_{L,idx},\ K_{R,idx}) \mid (idx, \cdot, \cdot) \in \Delta^{(0)}\}$.
Assemble $\mathsf{aux} \leftarrow (\mathsf{op}, x, \mathsf{BucketInfo}, \mathsf{PathInfo}, \mathsf{Hints}, \mathsf{HeightDelta})$.
return $(hacc_{new}, st, \mathsf{aux})$.

up path update. At a fixed level, the witness holder only replays those elementary updates that occur under the same parent as its current node. If the holder already has an old path-witness entry for the *same child under the same parent*, then tuples concerning another child update the witness by BP witness-update rules using the recorded reference digest in that tuple ($\mathsf{Acc.MWUpd}_{\mathsf{add}}$ uses acc^{before} for add, $\mathsf{Acc.MWUpd}_{\mathsf{del}}$ uses acc^{after} for del), while tuples concerning the current child leave the witness value unchanged and only switch the active child binding from the old value to the new one after the self-add tuple is replayed. If the holder's updated path enters a parent node that did not appear on the old path, or it remains under the same parent but moves to a different child slot, then at the unique self-add tuple for the current child, the holder initializes the new path-level witness as acc^{before}; for unary parents this is exactly the trivial witness g_1, and in general it is the commitment to all siblings already present in that parent. The signed height variation $\mathsf{HeightDelta} = l_{new} - l_{old}$ is used to determine how many levels of the old path may still be reused.

C Formal Security Analysis of HACC

This appendix section provides complete proofs aligned with the main-text security model in Section 4.2. We prove correctness, dynamic membership soundness, and dynamic non-membership soundness under the assumptions below.

Algorithm 9 $\mathsf{Sync_Path}(idx^{old}, idx^{new}, \mathsf{PathWit}^{old}, \mathsf{PathInfo}, \mathsf{HeightDelta}, \mathsf{Bind}^{(0)}_{old}, \mathsf{Bind}^{(0)}_{new})$

Parse $\mathsf{PathWit}^{old}$ as $\{(\pi^{(0)}_{path}, acc_1), \ldots, (\pi^{(l-1)}_{path}, acc_l)\}$.
Initialize $\mathsf{PathWit}^{new} \leftarrow \emptyset$, $curr_idx^{old} \leftarrow idx^{old}$, $curr_idx^{new} \leftarrow idx^{new}$.
$\mathsf{Bind}^{old}_{curr} \leftarrow \mathsf{Bind}^{(0)}_{old}$, $\mathsf{Bind}^{new}_{curr} \leftarrow \mathsf{Bind}^{(0)}_{new}$.
$l_{\mathsf{reuse}} \leftarrow l + \mathsf{HeightDelta}$ if $\mathsf{HeightDelta} < 0$, else l.
for $level = 0, 1, 2, \ldots$ while either $level < l_{\mathsf{reuse}}$ or PathInfo contains tuples under the current new parent **do**
    $parent^{new} \leftarrow \lceil curr_idx^{new}/t \rceil$.
    $parent^{old} \leftarrow \lceil curr_idx^{old}/t \rceil$ if $level < l_{\mathsf{reuse}}$, else $\perp$.
    $\mathsf{reuse_old} \leftarrow (level < l_{\mathsf{reuse}}) \wedge (parent^{old} = parent^{new}) \wedge (curr_idx^{old} = curr_idx^{new})$.
    if reuse_old **then**
        Let the old level-$level$ path entry be $(\pi^{old}_{path}, acc^{old}_{level+1})$.
        Initialize $\pi_{tmp} \leftarrow \pi^{old}_{path}$, $acc_{tmp} \leftarrow acc^{old}_{level+1}$, and set $\mathsf{Bind}_{active} \leftarrow \mathsf{Bind}^{old}_{curr}$.
    else
        Initialize $\pi_{tmp} \leftarrow \perp$, $acc_{tmp} \leftarrow \perp$, and set $\mathsf{Bind}_{active} \leftarrow \mathsf{Bind}^{new}_{curr}$.
    end if
    for each $(lvl, child, \mathsf{op}, \mathsf{Bind}_\Delta, acc^{before}, acc^{after}) \in \mathsf{PathInfo}$ where $lvl = level + 1$ and $\lceil child/t \rceil = parent^{new}$ **do**
        if $\pi_{tmp} = \perp$ **then**
            if $child = curr_idx^{new}$ and $\mathsf{op} = \mathsf{add}$ **then**
                Initialize $\pi_{tmp} \leftarrow acc^{before}$, $acc_{tmp} \leftarrow acc^{after}$, and $\mathsf{Bind}_{active} \leftarrow \mathsf{Bind}_\Delta$.
            end if
            continue
        end if
        if $child \neq curr_idx^{new}$ **then**
            if $\mathsf{op} = \mathsf{add}$ **then**
                Update $\pi_{tmp} \leftarrow \mathsf{Acc.MWUpd}_{\mathsf{add}}(\mathsf{Bind}_{active}, \pi_{tmp}, acc^{before}, \mathsf{Bind}_\Delta)$.
            else
                Update $\pi_{tmp} \leftarrow \mathsf{Acc.MWUpd}_{\mathsf{del}}(\mathsf{Bind}_{active}, \pi_{tmp}, acc^{after}, \mathsf{Bind}_\Delta)$.
            end if
        else
            $\mathsf{Bind}_{active} \leftarrow \mathsf{Bind}_\Delta$ if $\mathsf{op} = \mathsf{add}$, else do nothing. **// self-**del**: keep witness unchanged**
        end if
        $acc_{tmp} \leftarrow acc^{after}$.
    end for
    if $\pi_{tmp} = \perp$ **then**
        if reuse_old **then**
            Copy the old level-$level$ path entry unchanged into $\mathsf{PathWit}^{new}$.
            Set $acc_{tmp} \leftarrow acc^{old}_{level+1}$.
        else
            break **// no more affected levels for the updated path**
        end if
    else
        Append (π_{tmp}, acc_{tmp}) to $\mathsf{PathWit}^{new}$.
    end if
    Update $\mathsf{Bind}^{new}_{curr} \leftarrow \mathcal{H}(acc_{tmp} \parallel level + 1 \parallel parent^{new})$ and $curr_idx^{new} \leftarrow parent^{new}$.
    if $level < l_{\mathsf{reuse}}$ **then**
        Update $\mathsf{Bind}^{old}_{curr} \leftarrow \mathcal{H}(acc^{old}_{level+1} \parallel level + 1 \parallel parent^{old})$ and $curr_idx^{old} \leftarrow parent^{old}$.
    end if
end for
return the updated path witness $\mathsf{PathWit}^{new}$.

Assumptions and Proof Setup. We work in the random oracle model with

$$\mathcal{H}_{\mathsf{bind}} : \{0,1\}^* \to \mathbb{Z}_p$$

for structural binding, and (for compressed-path arguments) an independent Fiat-Shamir oracle

$$\mathcal{H}_{\mathsf{FS}} : \{0,1\}^* \to \mathbb{Z}_p.$$

We use the t-SDH assumption for BP accumulators.

Let Q_H be the total number of distinct queries to $\mathcal{H}_{\mathsf{bind}}$ by challenger and adversary, and define

$$\begin{aligned}\mathsf{BadHash} = &\left[\exists m \neq m' : \mathcal{H}_{\mathsf{bind}}(m) = \mathcal{H}_{\mathsf{bind}}(m')\right] \\ &\vee \left[\exists m : \mathcal{H}_{\mathsf{bind}}(m) = -s\right].\end{aligned}$$

Since $\mathcal{H}_{\mathsf{bind}}$ is random over $\mathbb{Z}_p$,

$$\Pr[\mathsf{BadHash}] \leq \frac{Q_H(Q_H-1)}{2p} + \frac{Q_H}{p},$$

which is negligible for polynomially bounded Q_H.

Theorem 4 (Correctness of HACC)**.** *All honestly generated HACC witnesses verify correctly, and same-predicate witnesses updated by honest auxiliary messages remain valid for the updated digest.*

Proof. Membership correctness follows from BP membership correctness at the bucket plus valid parent-path membership equations up to the root. Non-membership correctness follows from BP non-membership equations using Bézout coefficients, with the same path argument. For update correctness, standard updates use BP witness-update equations; split/merge updates use published hints (A_y, Ω_y) from polynomial-remainder relations for membership and bounded local recomputation for non-membership. Path-level updates are replayed from the explicit $(acc^{before}, acc^{after})$ transcript in PathInfo, which also lets a holder initialize any newly entered parent level, while the signed height variation $\mathsf{HeightDelta} = l_{new} - l_{old}$ only determines the reusable prefix length of the old path. Therefore updated witnesses verify under the new digest. □

Lemma 1 (Local membership soundness of BP)**.** *Let $X \subseteq \mathcal{D}$ with $|X| \leq t$, and let $acc = g_1^{X(s)}$. Under t-SDH, no PPT adversary can output (y, π) such that $y \notin X$ and $\mathsf{Acc.MV}(acc, y, \pi) = 1$, except with negligible probability.*

Proof. If $\mathsf{Acc.MV}(acc, y, \pi) = 1$, then

$$e(\pi, g_2^{s+y}) = e(g_1^{X(s)}, g_2),$$

hence $\pi = g_1^{X(s)/(s+y)}$. Since $y \notin X$, polynomials $X(S)$ and $S+y$ are coprime, so there exist $\alpha \in \mathbb{Z}_p$ and polynomial $\beta(S)$ with $\deg(\beta) \leq |X| - 1 \leq t-1$ such that

$$\alpha X(S) + \beta(S)(S+y) = 1.$$

Evaluating at s and dividing by $s+y$ gives

$$\frac{1}{s+y} = \alpha \frac{X(s)}{s+y} + \beta(s).$$

From the forgery one computes $\pi^\alpha = g_1^{\alpha X(s)/(s+y)}$, and from SRS one computes $g_1^{\beta(s)}$ (degree at most $t-1$). Multiplying yields $g_1^{1/(s+y)}$, solving t-SDH, contradiction. □

Lemma 2 (Local non-membership soundness of BP)**.** *Let $X \subseteq \mathcal{D}$ with $|X| \leq t$, and let $acc = g_1^{X(s)}$. Under t-SDH, no PPT adversary can output $(y, \overline{\pi})$ such that $y \in X$ and $\mathsf{Acc.NMV}(acc, y, \overline{\pi}) = 1$, except with negligible probability.*

Proof. Write $\overline{\pi} = (\alpha, g_1^{\beta(s)})$. Acceptance implies

$$\alpha X(s) + \beta(s)(s+y) = 1.$$

Since $y \in X$, write $X(S) = (S+y)X_y(S)$ with $\deg(X_y) \leq t-1$. Then

$$(s+y)(\alpha X_y(s) + \beta(s)) = 1,$$

so

$$\alpha X_y(s) + \beta(s) = \frac{1}{s+y}.$$

From SRS and known X_y, compute $g_1^{\alpha X_y(s)}$, multiply by witness component $g_1^{\beta(s)}$, and obtain $g_1^{1/(s+y)}$, violating t-SDH. □

Lemma 3 (Path authenticity)**.** *Fix an honestly generated state $(st_i, hacc_i)$. If a standard HACC path witness verifies against $hacc_i$ and authenticates (acc_0, idx, K_L, K_R), then conditioned on $\neg\mathsf{BadHash}$ either this tuple is the canonical honest leaf tuple in st_i, or an internal BP membership forgery exists.*

Proof. Let

$$\mathsf{Bind}_0 := \mathcal{H}_{\mathsf{bind}}(acc_0 \parallel 0 \parallel idx \parallel K_L \parallel K_R).$$

For $j = 0, \ldots, l-1$, let $idx_0 := idx$, let $idx_{j+1} := \lceil idx_j / t \rceil$, and let

$$\mathsf{Bind}_{j+1} := \mathcal{H}_{\mathsf{bind}}(acc_{j+1} \parallel (j+1) \parallel idx_{j+1}).$$

Acceptance of the path witness means that, for every $j = 0, \ldots, l-1$,

$$\mathsf{Acc.MV}(acc_{j+1}, \mathsf{Bind}_j, \pi_{path}^{(j)}) = 1,$$

and that $\mathsf{Bind}_l = hacc_i$.

We prove by descending induction on j that, unless some internal BP membership witness is forged, the tuple encoded at every level of the accepted path is the honest one stored in st_i.

At the top level, let $(acc_l^{\mathsf{hon}}, l, idx_l)$ be the honest root descriptor in st_i. By definition of the honest state,

$$hacc_i = \mathcal{H}_{\mathsf{bind}}(acc_l^{\mathsf{hon}} \parallel l \parallel idx_l).$$

Since the verifier accepts, $\mathsf{Bind}_l = hacc_i$. Conditioned on $\neg\mathsf{BadHash}$, the queried input to $\mathcal{H}_{\mathsf{bind}}$ that produced Bind_l must therefore be exactly the honest root descriptor. Hence

$$acc_l = acc_l^{\mathsf{hon}} \qquad \text{and} \qquad idx_l = idx_l^{\mathsf{hon}}.$$

Now fix $j \in \{0,\dots,l-1\}$ and assume that the level-$(j+1)$ node appearing in the accepted path is the honest one. Then acc_{j+1} is the honest BP accumulator at that node, and its accumulated set consists exactly of the binding scalars of its honest children. Since the verifier accepts,

$$\mathsf{Acc.MV}(acc_{j+1}, \mathsf{Bind}_j, \pi_{path}^{(j)}) = 1.$$

If Bind_j is not one of those honest child bindings, then $\pi_{path}^{(j)}$ is a forged BP membership witness for a value not accumulated in the honest parent accumulator, and we are done. Otherwise, let $(acc_j^{\mathsf{hon}}, j, idx_j^{\mathsf{hon}})$ be the honest child descriptor at the position determined by idx_j; for $j = 0$, this descriptor also includes the interval endpoints $(K_L^{\mathsf{hon}}, K_R^{\mathsf{hon}})$. Since this honest child contributes its binding scalar to the parent accumulator,

$$\mathsf{Bind}_j^{\mathsf{hon}} = \begin{cases} \mathcal{H}_{\mathsf{bind}}(acc_j^{\mathsf{hon}} \parallel j \parallel idx_j^{\mathsf{hon}}), & j \geq 1, \\ \mathcal{H}_{\mathsf{bind}}(acc_0^{\mathsf{hon}} \parallel 0 \parallel idx_0^{\mathsf{hon}} \parallel K_L^{\mathsf{hon}} \parallel K_R^{\mathsf{hon}}), & j = 0 \end{cases}$$

is one of the accumulated child values. Since Bind_j is also such a child binding, if the accepted level-j descriptor differed from the honest one, then two distinct queried inputs to $\mathcal{H}_{\mathsf{bind}}$ would map to the same output Bind_j, contradicting $\neg\mathsf{BadHash}$. Therefore the accepted level-j descriptor is the honest one.

Applying the induction for $j = l-1, l-2, \dots, 0$, we conclude that either some internal BP membership witness is forged, or the accepted leaf tuple is exactly the honest one stored in st_i. In particular,

$$(acc_0, idx, K_L, K_R) = (acc_0^{\mathsf{hon}}, idx_0^{\mathsf{hon}}, K_L^{\mathsf{hon}}, K_R^{\mathsf{hon}}).$$

□

Lemma 4 (Ordered-bucket locality)**.** *Fix an honestly generated state $(st_i, hacc_i)$. If a membership/non-membership witness for y verifies against $hacc_i$ and its path is authentic, then the local BP check is against the unique honest bucket containing y.*

Proof. In both $\mathsf{HAcc.MemVerify}$ and $\mathsf{HAcc.NonMemVerify}$, the local BP verification is performed against the bucket accumulator digest acc_0 appearing in the leaf tuple (acc_0, idx, K_L, K_R) carried by the witness. Since the path is authentic, Lemma 3 implies that this leaf tuple is exactly the honest leaf tuple stored in st_i at index idx.

The verifier also checks the interval condition $K_L \leq y < K_R$. In an honest HACC state, leaf-bucket intervals are pairwise disjoint and form an ordered partition of the accumulated domain. Hence there exists exactly one honest bucket whose interval contains y. Because (acc_0, idx, K_L, K_R) is an honest leaf tuple and its interval contains y, it must be that unique honest bucket. Therefore the local BP check is against the unique honest bucket containing y. □

Theorem 5 (Dynamic membership soundness of HACC)**.** *Assume t-SDH and random-oracle binding hash. Then HACC satisfies Definition 2.*

Proof. Suppose a PPT adversary outputs (y^*, π^*) with $y^* \notin \mathsf{AccSet}(st_q)$ but

$$\mathsf{HAcc.MemVerify}(hacc_q, y^*, \pi^*) = 1.$$

Parse $\pi^* = (\pi_{bucket}, K_L, K_R, idx, acc_0, \mathsf{PathWit})$. If path authenticity fails, we are in $\mathsf{BadHash}$ or an internal BP membership forgery event (both negligible). Otherwise, ordered-bucket locality ties verification to the unique honest bucket containing y^*. Since $y^* \notin \mathsf{AccSet}(st_q)$, it is not in that bucket set, yet

$$\mathsf{Acc.MV}(acc_0, y^*, \pi_{bucket}) = 1,$$

contradicting local BP membership soundness. □

Theorem 6 (Dynamic non-membership soundness of HACC)**.** *Assume t-SDH and random-oracle binding hash. Then HACC satisfies Definition 3.*

Proof. Suppose a PPT adversary outputs $(y^*, \overline{\pi}^*)$ with $y^* \in \mathsf{AccSet}(st_q)$ but

$$\mathsf{HAcc.NonMemVerify}(hacc_q, y^*, \overline{\pi}^*) = 1.$$

If path authenticity fails, we reduce to $\mathsf{BadHash}$ or an internal BP membership forgery. Otherwise, ordered-bucket locality ties verification to the unique honest bucket containing y^*. Because $y^* \in \mathsf{AccSet}(st_q)$, the local acceptance

$$\mathsf{Acc.NMV}(acc_0, y^*, \overline{\pi}_{bucket}) = 1$$

contradicts local BP non-membership soundness. □

Theorem 7 (Compressed-path security preservation)**.** *If the optional KZG path compression is used, soundness is preserved assuming KZG multiproof evaluation soundness, with Fiat-Shamir challenges modeled by the random oracle: any accepting compressed-path proof implies, except with negligible probability, the existence of an accepting standard-path proof for the same leaf tuple. Hence the same membership and non-membership soundness conclusions continue to hold.*

Proof sketch. Compressed-path verification is an aggregated KZG opening statement for the same parent-polynomial evaluations checked by the standard path verifier. If an adversary makes the compressed verifier accept without any corresponding valid collection of ordinary openings, then it has produced

a false aggregate opening, contradicting KZG multiproof evaluation soundness, with the usual Fiat-Shamir loss in the random oracle model. Therefore any accepting compressed-path proof yields an accepting standard-path proof for the same leaf tuple, except with negligible probability. The soundness reduction then proceeds exactly as in Theorems 5 and 6. □

D Proofs for Amortized Complexity Results

This section proves Theorems 2 and 3 from Section 8.1.

D.1 Proof of Theorem 2

Consider any sequence of U valid global updates. For update u, write the auxiliary message size as

$$|\mathtt{aux}_u| \le c_1\, l_u + c_2 + (c_3 t + c_4)\cdot I_u,$$

where $l_u = \Theta(\log_t n_u)$ is the current tree depth, $I_u \in \{0,1\}$ indicates whether update u triggers split/merge-specific hint payload, and c_1, c_2, c_3, c_4 are constants independent of n, t.

The first two terms are always present and sum to

$$\sum_{u=1}^{U} (c_1 l_u + c_2) = O(U \log_t n_{\max}),$$

where $n_{\max} = \max_u n_u$.

It remains to bound the restructuring term. After each split/merge, every resulting bucket is rebalanced around half capacity, so before the next split/merge affecting that bucket, at least $\lfloor t/2 \rfloor$ further updates must hit that bucket lineage. Hence each restructuring event can be charged to $\Omega(t)$ ordinary updates, giving

$$R := \sum_{u=1}^{U} I_u = O(U/t).$$

Therefore

$$\sum_{u=1}^{U} (c_3 t + c_4) I_u = (c_3 t + c_4) R = O(U).$$

Combining both parts,

$$\frac{1}{U}\sum_{u=1}^{U} |\mathtt{aux}_u| = O(\log_t n_{\max}) + O(1) = O(\log_t n_{\max}),$$

which proves Theorem 2.

D.2 Proof of Theorem 3

For each update epoch u, define random variable C_u as the cost to update a uniformly sampled maintained witness before update u.

$$C_u = C_u^{\mathsf{bucket}} + C_u^{\mathsf{path}}.$$

For each fixed u, the expectation $\mathbb{E}[\cdot]$ is over the uniform random choice of the maintained witness in the current state (equivalently, over its bucket index), while the update sequence itself is fixed. C_u^{bucket} is bounded in amortized expectation, and C_u^{path} is bounded deterministically for every sampled witness.

Part I: bucket-local amortization. Fix epoch u. Let $n_u = |S_u|$, let b_u^* be the touched bucket, and let $m_u = |b_u^*| \le t$. Let I_u indicate whether the uniformly sampled witness lies in b_u^*. Then

$$\Pr[I_u = 1] = \frac{m_u}{n_u} \le \frac{t}{n_u}.$$

Conditioned on $I_u = 1$, the witness receives a local bucket update. Among updates that hit this bucket region, split/merge events cost $O(t)\mathsf{E}$ but can occur only once per $\Omega(t)$ such updates by the half-full invariant; non-restructuring local updates cost $O(1)\mathsf{E}$. Therefore

$$\mathbb{E}[C_u^{\mathsf{bucket}} \mid I_u = 1] = O(1)\mathsf{E}.$$

Hence

$$\mathbb{E}[C_u^{\mathsf{bucket}}] = \Pr[I_u = 1]\cdot \mathbb{E}[C_u^{\mathsf{bucket}} \mid I_u = 1] = O(t/n_u)\mathsf{E} \le O(1)\mathsf{E}.$$

Averaging over $u = 1, \ldots, U$ gives

$$\frac{1}{U}\sum_{u=1}^{U} \mathbb{E}[C_u^{\mathsf{bucket}}] = O(1)\mathsf{E}.$$

Part II: path-level deterministic bound. Fix epoch u and a sampled witness in bucket b. Let b_u^* be the touched bucket and let L, at level m, be the lowest common ancestor of b and b_u^*. The path entries of b fall into three regions. Below L (levels $0, \ldots, m-2$), the corresponding path nodes root subtrees disjoint from b_u^*, so their digests and child bindings are unchanged and the path entries are preserved. At L (the level-$(m-1)$ entry), one sibling of b's child changes, so acc_L changes and this single entry is refreshed by one BP witness update, costing $O(1)\mathsf{E}$. At and above L (levels $\ge m$), the path of b coincides with that of b_u^*, and the only child changing at each such node is b's own node; since a BP membership witness excludes its own accumulated binding, the witness element is unchanged and only the accompanying digest is refreshed by a copy. Therefore $C_u^{\mathsf{path}} = O(1)\mathsf{E}$ for every sampled witness and every u, the $O(\log_t n)$ levels at and above L incurring only digest reads and hashes.

Combining the two parts,

$$\frac{1}{U}\sum_{u=1}^{U} \mathbb{E}[C_u] = \frac{1}{U}\sum_{u=1}^{U} \left(\mathbb{E}[C_u^{\mathsf{bucket}}] + \mathbb{E}[C_u^{\mathsf{path}}]\right) = O(1)\mathsf{E},$$

proving Theorem 3.

Table 7: HACC witness update cost (ms/op) at t=256.

Witness	Trigger	2^{12}	2^{16}	2^{20}	2^{22}
Membership	ordinary	0.025–0.147	0.012–0.117	0.013–0.121	0.012–0.119
	split-add	0.086–0.121	0.086–0.119	0.094–0.123	0.095–0.122
	merge-delete	0.084–0.124	0.123–0.157	0.130–0.161	0.132–0.163
Non-membership	ordinary	0.086–0.118	0.061–0.092	0.062–0.095	0.059–0.093
	split-add	0.119–0.574	0.116–0.559	0.118–0.561	0.117–0.558
	merge-delete	0.115–0.562	0.153–0.595	0.157–0.592	0.158–0.609

Table 8: HACC ordinary vs. split/merge-triggering dynamic operation cost (ms/op) at $t = 256$.

n	Add (ordinary)	SplitAdd (triggered)	Del (ordinary)	MergeDelete (triggered)
2^{12}	1.46	169.0	1.36	47.4
2^{14}	2.31	169.8	2.16	48.4
2^{16}	3.34	165.9	3.14	49.4
2^{18}	3.44	163.4	3.23	48.7
2^{20}	3.59	158.6	3.35	48.8
2^{22}	4.19	156.5	3.88	51.2

E Additional HACC Measurements

This appendix collects additional HACC measurements referenced in Section 8.2: the cost of witness update and split/merge-triggering dynamic operations, and the size of auxiliary update message.

Witness update cost. After each addition or deletion, previously generated witnesses can be updated non-interactively. Table 7 reports this per-witness cost at t=256 as n grows, broken down by witness type (membership vs. non-membership) and by whether the triggering operation is an ordinary Add/Del or one that triggers a split/merge; each cell is the min–max over whether the touched leaf is *near* or *far* from the witness holder's bucket. The cost is flat in n and all type of updates are sub-millisecond.

Split/merge-triggering dynamic operations. Most additions and deletions touch a single bucket, but an operation that overflows or underflows a bucket triggers a split or merge. Table 8 contrasts the cost of ordinary Add/Del with the split/merge-triggering operations at t=256. A split or merge is dominated by an $O(t^2)$ computation preparing the auxiliary message's Hints that helps witness update. The triggering operation thus becomes essentially a constant determined by t alone, and at t=256 it stays around 160 ms for Add and around 50 ms for Del.

Auxiliary update message size. Witness update consumes an auxiliary update message published after each Add/Del. Table 9 reports the size of aux after each type of dynamic operation at t=256 as n grows. Auxiliary messages after ordinary add/delete stay small (548–904 B), growing only when the tree gains a level, consistent with the amortized $O(\log_t n)$ bound of Theorem 2. Auxiliary messages after split and merge are much larger (37–39 KB and 19–20 KB) because they carry the hints payload; incurred only once per $\Omega(t)$ operations, their amortized contribution is $O(1)$.

Table 9: HACC auxiliary update-message size (bytes) at t=256, by message type, as n grows.

n	Add (ordinary)	SplitAdd (triggered)	Del (ordinary)	MergeDelete (triggered)
2^{12}	548	551	37,345	19,154
2^{14}	548	551	37,345	19,154
2^{16}	901	904	38,130	19,988
2^{18}	901	904	38,387	20,116
2^{20}	901	904	38,387	20,116
2^{22}	901	904	38,644	20,244

F Specific Example for Dynamic Maintenance of HACC Tree

In this appendix, we provide a specific example illustrating the dynamic maintenance, especially bucket split/merge and tree height changes, of the HACC tree structure during frequent element additions and deletions. Let the accumulated domain $\mathcal{D} = \{0,1,\ldots,31\}$ and $t = 4$. After committing to $X = \{0,1,8,9,20,21,26,27,29,30\}$, the initialized HACC tree is depicted in Figure 4a, the FIP is

$$\{reusable_indices = \emptyset,\ next_new_available_index = 6\}$$

and the BMT (interval : index) is

$$[0,5)\colon 1,\quad [5,15)\colon 2,\quad [15,24)\colon 3,\quad [24,28)\colon 4,\quad [28,32)\colon 5.$$

After a sequence of additions and deletions

$$\begin{array}{lllll} \mathsf{add}(18), & \mathsf{add}(19), & \mathsf{add}(22), & \mathsf{del}(8), & \mathsf{del}(19), \\ \mathsf{add}(10), & \mathsf{add}(11), & \mathsf{del}(20), & \mathsf{del}(21), & \mathsf{add}(2), \\ \mathsf{add}(3), & \mathsf{del}(29), & \mathsf{del}(30), & & \end{array}$$

the tree is updated to the state shown in Figure 4b, with FIP

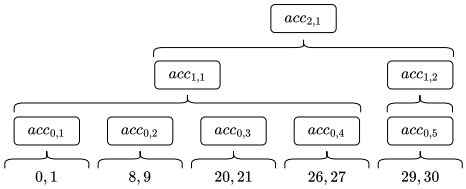
$$\{reusable_indices = \{3,5\},\ next_new_available_index = 7\}$$

and BMT

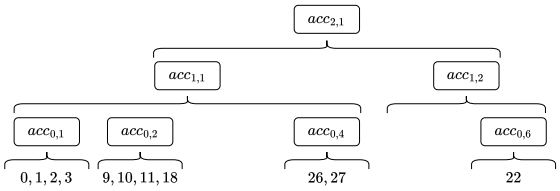
$$[0,5)\colon 1,\quad [5,20)\colon 2,\quad [20,24)\colon 6,\quad [24,32)\colon 4.$$

We show how the tree evolves as we perform a subsequent $\mathsf{add}(4)$ and $\mathsf{del}(26)$, both of which trigger bucket split/merge.

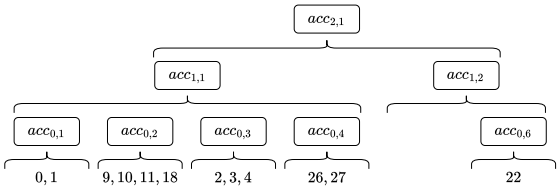
When we perform $\mathsf{add}(4)$, it falls into the interval $[0,5)$ corresponding to bucket index 1. Since bucket 1 is currently at capacity $t = 4$, it is overflow and needs to be split. Bucket 1 is splitted at the median of $\{0,1,2,3,4\}$, which is 2. The left side bucket (index 1) contains $\{0,1\}$ and is responsible for the interval $[0,2)$, while the right side bucket contains $\{2,3,4\}$



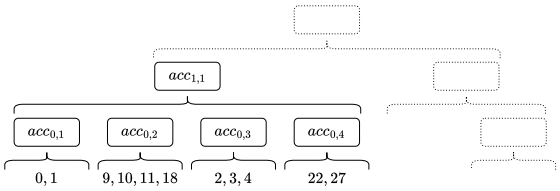
(a) The HACC tree after Commit(X)



(b) The HACC tree after the first sequence of updates



(c) The HACC tree after add(4)



(d) The HACC tree after del(26)

Figure 4: An example for the dynamic maintenance of the HACC tree during element additions and deletions

and is responsible for the interval $[2,5)$. The smallest reusable index from FIP is index 3, which is allocated to the right side bucket. The updated tree structure after add(4) is shown in Figure 4c. The FIP is updated to

$$\{reusable_indices = \{5\},\ next_new_available_index = 7\}$$

and the BMT is updated to

$$[0,2)\colon 1, \quad [2,5)\colon 3, \quad [5,20)\colon 2, \quad [20,24)\colon 6, \quad [24,32)\colon 4.$$

With the help of FIP and BMT, the bucket overflow does not move elements from bucket 1 to its logical sibling nor physical sibling bucket (bucket 2, which is also full at this time), avoiding cascading updates over many buckets.

Next, we perform del(26), which falls into the interval $[24,32)$ corresponding to bucket index 4. After deletion, bucket 4 contains $\{27\}$ and its logically adjacent bucket (index 6) contains $\{22\}$. The two buckets contain elements no more than $t/2 = 2$ in total and are thus merged. The larger index 6 is recycled back to FIP, and the merged bucket (index 4) is responsible for the interval $[20,32)$ and contains $\{22,27\}$. Note that in this case, the tree height decreases and no other buckets are affected. The updated tree structure after del(26) is shown in Figure 4d, with FIP updated to

$$\{reusable_indices = \emptyset,\ next_new_available_index = 5\}$$

and BMT updated to

$$[0,2)\colon 1, \quad [2,5)\colon 3, \quad [5,20)\colon 2, \quad [20,32)\colon 4.$$