Pruning Merkle-Tree Consistent Accumulator

Anna Mendonca[1], Hudson Shi[1], Ivan Pryvalov[2], and Amir Herzberg[1]

[1] University of Connecticut {anna.gorbenko,hudson.shi,amir.herzberg}@uconn.edu
[2] Brandenburg University of Technology ivan.pryvalov@b-tu.de

Abstract. Authenticated data structures are widely used to compute compact digests of evolving collections of elements and to support efficient verification of element inclusion. However, the authenticated collection often should not grow forever: older elements may expire and no longer require verification. Many implemented append-only approaches, for example used in Certificate Transparency (CT), do not directly support this setting, since previously accumulated elements remain part of the authenticated state indefinitely.

In this work, we introduce a *pruning accumulator*, a stateful accumulator that supports both incremental addition of new elements and pruning of an old prefix of previously accumulated elements. The resulting digest represents the unpruned sequence, while pruned elements are removed from the authenticated state. Unpruned elements continue to support proof-of-inclusion verification, proof updates, update verification, and consistency checks. This captures applications that require authenticated, incrementally maintained state over a moving window of elements.

We present two constructions of Merkle-tree-based pruning accumulators, both with efficient accumulation, prefix pruning, proof generation, proof updating, and verification. The constructions preserve the standard Merkle-tree style of verification for active elements while reducing long-term storage requirements. We provide formal definitions, correctness and security analysis, an open-source implementation, and experimental evaluation demonstrating the performance benefits of pruning.

Keywords: Merkle tree · Cryptographic accumulators · Certificate Transparency

1 Introduction

Modern transparency systems rely on compact authenticated summaries of large and evolving datasets. In these systems, a short digest commits to a collection of records, and a verifier can check that a particular record is represented by the digest using a proof-of-inclusion (PoI). Merkle trees are the most common realization of this idea, largely because they provide logarithmic-size proofs and simple hash-based verification.

The main application motivating this work is Certificate Transparency (CT) [15–17]. In CT, log operators maintain publicly auditable logs of certificates, and clients, monitors, and auditors use Merkle-tree roots, inclusion proofs, and consistency proofs to detect misbehavior. The CT model is append-only: once an entry is added, later tree heads should extend the earlier history rather than replace or rewrite it. However, append-only growth also creates a long-term maintenance problem. CT logs and related transparency

systems continue to accumulate entries over time, including entries that may eventually become less relevant for active verification, such as entries associated with expired certificates or entries in older time-bounded log periods.

We begin from the notion of a consistent accumulator, which supports repeated additions and allows verifiers to check that newer digests are consistent with earlier ones. We extend this setting by adding the option to remove old entries. The resulting primitive, which we call a *pruning accumulator*, supports both appending new elements and pruning an old prefix of previously accumulated elements. After pruning, the current digest represents the remaining unpruned sequence.

The pruning operation removes the oldest currently unpruned elements, rather than allowing arbitrary deletion from the middle of the sequence. This matches the time-ordered nature of CT-style logs, where records are appended over time and older records are the natural candidates for expiration, archival, or removal from the active authenticated state.

A pruning accumulator must preserve the verification functionality expected from Merkle-tree-based systems. Elements that remain active should still have valid PoIs. Previously issued PoIs for active elements should be updatable when the accumulator changes. Verifiers should be able to check update values and consistency proofs between digest values. At the same time, if an element has been pruned, then it should no longer be treated as included in the current digest, and a PoI for that element should not verify against the current state.

We formalize this model by defining pruning accumulators whose accumulation algorithm can either append a sequence of new elements or prune a specified number of old elements. We then present variants of the Pruning Merkle-Tree, a Merkle-tree-based pruning accumulator construction. The construction is designed for CT and CT-like transparency settings, where Merkle-tree verification is already the standard approach and where compatibility with efficient PoI verification is important. The Pruning Merkle-Tree supports incremental accumulation, prefix pruning, PoI generation, PoI updating, update verification, and consistency verification for the active sequence.

Compatibility modes The classical Merkle Tree is characterized by the single root hash. On the other hand, asynchronous accumulator [24] and Merkle Mountain Range [26], define a set of root hashes of the respective subtrees. Our pruning accumulator can be instantiated in either mode so that users have a tradeoff between a) having the short $O(1)$ digest, or b) having the asynchronism of PoI at the cost of $O(\log N)$ digests, where N is the number of accumulated non-pruned elements. The single root version is based on the multi-root version. The single root is reconstructed from all the root hashes of the subtrees of the multi-root version.

Our Contributions:

1. *Pruning accumulator model.* We introduce pruning accumulators, extending consistent accumulators with prefix pruning. The digest represents the current unpruned sequence, while pruned elements are no longer considered members of the active authenticated state.
2. *Formal definitions and security properties.* We define the syntax, correctness requirements, and security properties for pruning accumulators, including PoI

correctness for active elements, behavior of PoI updates after pruning, update verification, and consistency under legal sequences of append and prune operations.
3. *Merkle-tree construction.* We present Pruning Merkle-Tree, a Merkle-tree-based pruning accumulator construction that supports efficient appending, prefix pruning, proof generation, proof updating, and verification.
4. *Implementation and evaluation.* We provide an open-source implementation and experimental evaluation of Pruning Merkle-Tree. Our results demonstrate the storage and performance benefits of pruning for maintaining authenticated state over a moving active sequence.

1.1 Related Work

Cryptographic accumulators provide a way to commit to a collection of values using a short digest, while also allowing witnesses for membership or related properties. This idea goes back to the one-way accumulators of Benaloh and de Mare [4]. Later work, including Derler et al. [10], formalized different accumulator properties and clarified the range of accumulator definitions used in the literature.

Merkle trees are the best-known example of a hash-based accumulator. In the construction introduced by Merkle [23], a short root digest commits to a collection of values, and membership can be verified using a logarithmic-size path of sibling hashes. Camacho et al. [6] later formalized collision-resistant hashing as a strong accumulator. However, the classical Merkle-tree setting is static or append-only. It gives efficient PoIs, but it does not directly address what it means to remove an old prefix while preserving meaningful verification for the elements that remain.

Merkle-tree accumulators are also used in append-only logging systems. Certificate Transparency [16, 17], along with related work on authenticated logs [9, 11, 12], relies on the idea that later log digests should be consistent with earlier ones. In this setting, a consistency proof shows that the later tree extends the earlier tree without changing the old prefix. This is different from the setting we consider, because pruning intentionally allows an old prefix to be removed rather than preserved forever. Thus, a consistency proof would show how to get to a future digest, not necessarily that the old prefix has not been modified.

Other authenticated append-only structures provide related notions of consistency across changing states. Authenticated append-only skip lists [20], CHRONOS [5], CONIKS [21], transparency overlays [8], and append-only authenticated dictionaries [27] all consider ways to authenticate evolving logs, directories, or dictionaries over time. These works are related because they allow users to check that later states are consistent with earlier ones. However, they do not address the setting where the authenticated sequence is allowed to progress by deleting an old prefix while keeping the remaining suffix verifiable.

Asynchronous accumulators and multi-root Merkle structures are also relevant to our construction. Reyzin and Yakoubov [24] study asynchronous accumulators, and Merkle Mountain Ranges [26] use a forest of Merkle subtree roots rather than a single perfectly balanced tree. Our asynchronous construction uses a similar idea by exposing the cached forest roots, but we use the forest structure to support prefix pruning.

Prior work on Merkle-tree consistent accumulators studied the append-only setting, including efficient batch addition, proofs of consistency between digest states, and updating PoIs as new elements are added [22]. The construction in this paper changes the update model: instead of only allowing new elements to be appended, it also allows an old prefix of the accumulated sequence to be pruned.

Accumulators that support both additions and deletions have also been studied. Dynamic accumulators [7], universal accumulators [19], DDH-based dynamic universal accumulators [2], and later constructions [3, 13] provide ways to update witnesses as elements are added to or removed from the accumulated collection. These schemes consider deletion, but the object being authenticated is usually a set rather than an ordered append-only sequence. As a result, deletion in these schemes is different from pruning an old prefix of a log while preserving the order and verifiability of the remaining suffix.

Redactable blockchains [1] allow for arbitrary changes to the blockchains in a controlled way. Based on advanced cryptography, most notably chameleon hash functions [14], such techniques require a trapdoor to operate efficiently. In our approach, we require no trusted assumptions and support only the prefix-based pruning.

Proof-Infused Streams [18] authenticate sliding-window queries over streams, including through the Tumbling Merkle Tree construction. In that setting, the stream advances by adding new batches and discarding old batches from the maintained window.

The need for a separate pruning notion comes from the fact that append-only consistency does not directly apply once deletion is allowed. In an append-only log, a later digest is consistent with an earlier digest when it extends the earlier sequence without changing its prefix. Fully dynamic accumulators and authenticated dictionaries allow deletions, but they usually authenticate the current set, dictionary, or index, rather than an ordered sequence together with a consistency relation between successive digest states [2, 3, 7, 13, 19].

Outline The rest of the paper is organized as follows. § 2 defines pruning accumulator schemes and the relevant correctness and security requirements. § 3 discusses pruning accumulator constructions, where § 3.3 presents our Pruning Asynchronous Accumulator, $\mathcal{PAA}^h$. § 3.4 discusses how to extend our design into a construction compatible with a traditional single root Merkle tree, the Pruning Merkle Tree, $\mathcal{PMT}^h$. § 4 describes the evaluation and experiments under CT-like workloads. § 5 gives proof outlines for the structural, correctness, security, and complexity properties of $\mathcal{PAA}^h$ and compares its size tradeoffs with the synchronous variant and the naive baseline.§ 6 concludes.

2 Pruning Accumulator Schemes

2.1 Notations

We extend the operation-sequence notation used for consistent accumulators [22] by allowing each operation tag op_i to be either Add or Prune. Thus, the same notation can express both ordinary append-only executions and pruning executions.

Let $\langle X_i \rangle_{i=1}^{l}$ denote a finite sequence, each of whose elements X_i is itself a sequence of binary strings, i.e., $X_i \equiv \langle x_j \rangle_{j=0}^{m-1}$ where $x_j \in \{0,1\}^*$.

A pruning accumulator accepts a sequence of accumulator operations. We write $\mathsf{Ops}_l \equiv \langle (op_i, par_i) \rangle_{i=1}^{l}$, where each op_i is an operation tag in $\{\mathsf{Add}, \mathsf{Prune}\}$ and par_i is its corresponding input. If $op_i = \mathsf{Add}$, then $par_i = X_i$ is a sequence of binary strings to append. If $op_i = \mathsf{Prune}$, then $par_i = t_i \in \mathbb{N}$ specifies that the oldest t_i currently unpruned elements are removed. Both types of operations (Add and Prune) are handled by the accumulator algorithm Accum.

Let Ops_i denote the prefix $\langle (op_j, par_j) \rangle_{j=1}^{i}$ of Ops_l. Let α be a keyed accumulator scheme, let λ be a security parameter, and let $k \in \{0,1\}^\lambda$ be the accumulator key. Let $s_0 \equiv \epsilon$ denote the initial accumulator state. For $i \geq 1$, define the state after the first i operations recursively by

$$s_i \equiv \alpha_k.\mathsf{Accum}.s_{s_{i-1}}(op_i, par_i). \tag{1}$$

Using this notation, we define

$$\begin{aligned}
\alpha_k.s(\mathsf{Ops}_l) &\equiv s_l, \\
\alpha_k.D(\mathsf{Ops}_l) &\equiv \alpha_k.\mathsf{Accum}.D_{s_{l-1}}(op_l, par_l), \\
\alpha_k.u(\mathsf{Ops}_l) &\equiv \alpha_k.\mathsf{Accum}.u_{s_{l-1}}(op_l, par_l), \\
\alpha_k.PoI(\mathsf{Ops}_l) &\equiv \alpha_k.\mathsf{Accum}.PoI_{s_{l-1}}(op_l, par_l), \\
\alpha_k.PoC(\mathsf{Ops}_l) &\equiv \alpha_k.\mathsf{Accum}.PoC_{s_{l-1}}(op_l, par_l).
\end{aligned} \tag{2}$$

Equivalently, we may write $(s_l, D_l, PoI_l, u_l, PoC_l) \leftarrow \alpha_k.\mathsf{Accum}_{s_{l-1}}(op_l, par_l)$. The PoIs returned by Accum are for the elements newly added in the last operation if the last operation is an Add operation. If the last operation is a Prune operation, then no new elements are added and PoI_l may be empty. To obtain PoIs for elements added in earlier operations, one must apply UpdPoI using the appropriate update value.

We now define the current sequence represented by a sequence of operations. Let $Cur(\mathsf{Ops}_i)$ denote the ordered sequence of elements that remain unpruned after executing the first i operations. We define this sequence recursively. First, $Cur(\epsilon) \equiv \epsilon$. If the next operation is an addition, then

$$Cur(\mathsf{Ops}_{i-1} +\!\!+ \langle (\mathsf{Add}, X_i) \rangle) \equiv Cur(\mathsf{Ops}_{i-1}) +\!\!+ X_i. \tag{3}$$

If the next operation is a pruning operation, and $Cur(\mathsf{Ops}_{i-1}) = \langle x_0, \ldots, x_{r-1} \rangle$, then, for $0 \leq t_i \leq r$,

$$Cur(\mathsf{Ops}_{i-1} +\!\!+ \langle (\mathsf{Prune}, t_i) \rangle) \equiv \langle x_{t_i}, \ldots, x_{r-1} \rangle. \tag{4}$$

That is, pruning removes the oldest t_i currently unpruned elements.

It is useful to define $count$ to be the number of currently unpruned elements after a sequence of operations: $count(\mathsf{Ops}_i) \equiv |Cur(\mathsf{Ops}_i)|$.

A sequence of operations Ops_l is *legal* if no pruning operation removes more elements than are currently unpruned. Equivalently, for every $i \in \{1, \ldots, l\}$ such that $op_i = \mathsf{Prune}$, if $par_i = t_i$, then $t_i \leq count(\mathsf{Ops}_{i-1})$.

2.2 Definition

A Pruning Accumulator Scheme is an extension of a consistent accumulator that supports pruning. A consistent accumulator is stateful and supports multiple accumulation events, where each event appends a sequence of strings to the elements accumulated so far. A pruning accumulator preserves this stateful, incremental structure and additionally allows old elements to be removed from the accumulator once they are no longer relevant.

Pruning is useful in applications where the accumulator represents a growing history but where old elements eventually become irrelevant. For example, in log-based systems such as Certificate Transparency, the standard construction is append-only and does not support truncating old entries from a Merkle tree. Instead, logs may use separate trees over different time periods. A pruning accumulator provides a more direct abstraction: the manager can append new elements while also pruning an old prefix, so that the current digest represents only the elements that remain active.

More precisely, suppose a pruning accumulator receives a legal sequence of operations $\mathsf{Ops}_l = \langle (op_i, par_i) \rangle_{i=1}^{l}$. If all operations are additions, then the accumulator behaves like a consistent accumulator: the digest represents the concatenation of all added sequences, $X_1 + \ldots + X_l$. In the pruning setting, however, some of the oldest elements may have been removed by Prune operations. Therefore, after executing the legal operation sequence Ops_l, the digest represents the current unpruned sequence $Cur(\mathsf{Ops}_l)$. An element that has been pruned is no longer included in the accumulator state represented by the current digest, and a PoI for such an element should no longer verify with respect to that digest.

The accumulation operations are performed by a *stateful manager entity*, which maintains the accumulator state across calls to Accum. The verification operations remain stateless: a verifier checks proofs-of-inclusion, update values, and proofs of consistency using only the relevant digest values and proof strings. Thus, a pruning accumulator keeps the same general interface as a consistent accumulator, while changing the semantics of accumulation so that the represented sequence may evolve by both appending new elements and pruning an old prefix.

Definition 1 (Pruning accumulator schemes). *A* pruning accumulator $\alpha = (\mathsf{Accum}, \mathsf{VerPoI}, \mathsf{VerUpd}, \mathsf{VerPoC}, \mathsf{UpdPoI})$ *is a collection of PPT algorithms with integer λ, key $k \in \{0,1\}^\lambda$, binary strings s, p, D, D', x, u, and legal sequences of accumulator operations* Ops*. Let ϵ denote the initial empty accumulator state. The algorithms are defined as follows:*

- $\alpha_k.\mathsf{Accum}(s, \mathsf{Ops}) \to (s', D, PoI, u, PoC)$**:** *Given the current accumulator state s, initially ϵ, and a legal sequence of accumulator operations* Ops*, outputs an updated state s', a digest D, a set of proofs-of-inclusion PoI, an update value u, and a proof of consistency PoC. In contrast to a consistent accumulator, the digest D of a pruning accumulator must capture not only a hash commitment but also additional information about how many elements have been pruned.*
- $\alpha_k.\mathsf{VerPoI}(D, x, poi) \to \{$**True or False**$\}$**:** *Given a digest D, an element x, and its corresponding proof-of-inclusion poi, returns* True *if the proof is valid, meaning that x is included among the unpruned elements represented by D, and returns* False *otherwise.*

$\alpha_k.\mathsf{VerUpd}(D, u, D') \to \{\textbf{TRUE or FALSE}\}$**:** *Given an initial digest D, an update value u, and a new digest D', returns* TRUE *if u is a valid update value transforming D into D' under a legal sequence of append and prune operations, and returns* FALSE *otherwise.*

$\alpha_k.\mathsf{VerPoC}(D, poc, D') \to \{\textbf{TRUE or FALSE}\}$**:** *Given an initial digest D, a proof of consistency p, and a new digest D', returns* TRUE *if D' is consistent with D under a legal sequence of append and prune operations, and returns* FALSE *otherwise. In particular, consistency requires that the transition from D to D' does not modify the relative order or values of any elements that remain unpruned.*

$\alpha_k.\mathsf{UpdPoI}(poi, u) \to poi' \cup \{\perp\}$**:** *Given a current proof-of-inclusion poi for an element x and an update value u, this algorithm returns an updated proof-of-inclusion poi' for use with the new digest D', or returns $\perp$ if x has been pruned and therefore no longer has a valid proof-of-inclusion with respect to D'.*

The correctness and security requirements for pruning accumulators are natural extensions of the corresponding requirements for consistent accumulators [22]. The main difference is that pruning accumulators are interpreted with respect to the elements that remain after old prefixes have been removed, rather than with respect to all elements ever added.

We use the following correctness requirements: PoI correctness requires honestly generated PoIs for newly added elements to verify with respect to the current output digest (Def. 4). PoC and update value correctness requires honestly generated PoCs and update values to verify between the old and new output digests (Def. 5). PoI update correctness requires UpdPoI to update a valid PoI when its element remains active and to return $\perp$ once the element has been pruned (Def. 6).

2.3 Pruning Accumulator Collision Resistance

We now define collision resistance for pruning accumulators. Collision resistance is stated with respect to the current unpruned sequence represented by the digest. This differs from append-only consistent accumulators, where the digest represents all elements accumulated so far. In a pruning accumulator, two different legal operation histories may legitimately produce the same current sequence, because earlier elements may have been pruned. Such histories should not be considered collisions. A collision occurs only when two legal operation sequences produce the same non-$\perp$ digest while representing different current unpruned sequences.

Definition 2 (Pruning accumulator collisions). *Let α be a keyed pruning accumulator scheme, let $k \in \{0,1\}^\lambda$, and let Ops and Ops' be two legal sequences of accumulator operations. We say that $(\mathsf{Ops}, \mathsf{Ops}')$ is a* collision *for α_k if*

$$\alpha_k.D(\mathsf{Ops}) = \alpha_k.D(\mathsf{Ops}') \neq \perp \ \textit{ and } \ Cur(\mathsf{Ops}) \neq Cur(\mathsf{Ops}').$$

Definition 3 (Pruning accumulator any-collision resistance). *We say that the keyed pruning accumulator scheme α is* Any-Collision-Resistant (ACR) *if for every efficient*

PPT algorithm $\mathcal{A}$ the following holds:

$$\varepsilon^{ACR}_{\alpha,\mathcal{A}}(\lambda) \equiv \Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} \left[\begin{array}{c} (\mathsf{Ops}, \mathsf{Ops}') \leftarrow \mathcal{A}(k) \textit{ where:} \\ \mathsf{Ops} \textit{ and } \mathsf{Ops}' \textit{ are legal} \wedge \\ \alpha_k.D(\mathsf{Ops}) = \alpha_k.D(\mathsf{Ops}') \neq \perp \wedge \\ Cur(\mathsf{Ops}) \neq Cur(\mathsf{Ops}') \end{array} \right] \in NEGL(\lambda).$$

The key difference between pruning accumulator collision resistance and collision resistance for append-only consistent accumulators is that pruning changes which elements are represented by the current digest. In a consistent accumulator without pruning, the digest represents the full sequence of elements accumulated so far. In a pruning accumulator, the digest represents only $Cur(\mathsf{Ops})$, the sequence of elements that remain unpruned. Thus, two different legal operation histories are not considered a collision if they lead to the same current unpruned sequence. Table 1 illustrates this distinction. Examples 1–3 show pairs of operation histories that would be collisions if they produced the same non-$\perp$ digest, because their current unpruned sequences differ. Examples 4–5 show histories that differ syntactically but are not collisions, because they represent the same current unpruned sequence.

Ex.	Ops	Ops'	Collision
1	$\{\mathsf{Add}, \langle a, b\rangle\}, \{\mathsf{Prune}, 1\}$	$\{\mathsf{Add}, \langle a, b\rangle\}$	Yes
2	$\{\mathsf{Add}, \langle a, b, c\rangle\}, \{\mathsf{Prune}, 1\}$	$\{\mathsf{Add}, \langle a, b, c\rangle\}, \{\mathsf{Prune}, 2\}$	Yes
3	$\{\mathsf{Add}, \langle a, b, c\rangle\}, \{\mathsf{Prune}, 1\}$	$\{\mathsf{Add}, \langle c\rangle\}, \{\mathsf{Add}, \langle b\rangle\}$	Yes
4	$\{\mathsf{Add}, \langle a, b, c\rangle\}, \{\mathsf{Prune}, 1\}$	$\{\mathsf{Add}, \langle b, c\rangle\}$	No
5	$\{\mathsf{Add}, \langle a, b\rangle\}, \{\mathsf{Prune}, 1\}$	$\{\mathsf{Add}, \langle c, b\rangle\}, \{\mathsf{Prune}, 1\}$	No

Table 1: Examples of legal operation sequences for pruning accumulators. The table assumes that $\alpha_k.D(\mathsf{Ops}) = \alpha_k.D(\mathsf{Ops}') \neq \perp$, and indicates whether this equality would constitute a collision.

PoI unforgeability requires that no efficient adversary can produce a valid PoI for an element that is not currently represented by the digest (Def. 7). The remainder of the formal interface and definitions are given in § A.

3 Asynchronous Pruning Merkle-Tree Consistent Accumulator

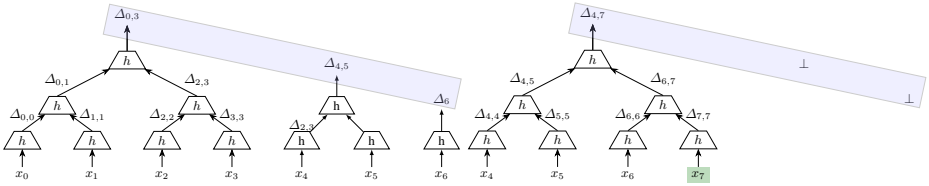
In this section, we present our pruning Merkle-tree consistent accumulator based on the multi-hash accumulator designs [24,26]. We first begin in § 3.1 with the naïve attempt, by eventually dropping the left-most hash in the multi-hash design, and show its inefficiency in terms of pruning. We then present our asynchronous construction in § 3.2 which eliminates this drawback and achieves efficient pruning.

3.1 Strawman idea for pruning

We begin by presenting our first attempt in constructing a pruning Merkle Tree accumulator based on the multi-hash design of [24]. In this design, in contrast to the classical Merkle

Tree (MT), the scheme generates *multiple* root hashes, which represent full binary subtrees with an exponential number of accumulated elements in them; each such hash is a (static) MT. When a new element is added, a new MT on level 0 is formed, initially of size $N = 1$, where N is the number of accumulated elements that the root hash represents. If there is a root hash already present with the same size, two subtrees are merged and the resulting subtree of size $2N$ is moved to the next level. This process continues until there are no more merge operations possible. The accumulator's state consists of the root hashes, which we call *subdigests*, and the total number of accumulated elements n.

Storing multiple hashes as accumulator's state suggests the following straightforward extension to support the truncation (or "pruning") of older elements from the accumulator, with a minor effect on the initial design: We maintain counters p, the number of elements requested for pruning, and p_e, the number of elements that are effectively removed from the accumulator's state; both initialized with 0. When a prune operation for ℓ elements is requested, we increase p by ℓ. As long as $p - p_e$ is at least the size of the subtree that corresponds to the left-most subdigest Δ, we purge Δ and increase p_e by its size. We illustrate this idea in Fig. 1.



(a) Before adding x_7.

(b) After adding x_7 and pruning $x_0, \ldots, x_3$.

Fig. 1: The Naïve Pruning Tree. The accumulator's state is shown in the top shaded row above each image. Note if the largest subtree is ℓ elements (4 in this example), pruning less than ℓ elements will not affect the tree structure, only increasing the prune counter.

Problem: This construct has an inefficient pruning. To show this, we characterize in Lemma 1 the pruning overhead of the binary representation, which is the ratio between the size of the data structure and the number of non-pruned elements.

Lemma 1 (Pruning overhead of the binary representation). *Let* $n = \sum_{i=0}^{k-1} 2^i \cdot a_i > 0$ *be the binary representation of* n*, where* $a_i \in \{0, 1\}$ *and* $a_{k-1} = 1$*. Let* $p^* < 2^{k-1}$*. Then* $\frac{n}{n-p^*} = O(n)$*.*

Proof. Follows immediately if we substitute p^* with $2^{k-1} - 1$ and n with 2^{k-1}. □

To demonstrate the worst-case scenario, consider the following sequence of operations: There are $n = 2^k$ elements added and subsequently $p = 2^k - 1$ elements pruned. After executing these steps, the state consists of a single binary subtree of size 2^k, which will only be removed from the state as soon as at least a total of 2^k elements are requested for

pruning. We have that, effectively, $2^k - (2^k - 1) = 1$ elements should have a valid proof of inclusion (PoI), for which the cost of PoI is $O(\log k)$. Ideally, we would like to have this cost $O(\log(n-p))$, which would be $O(1)$ in this particular instance. In § 3.2, we address this problem by introducing a construction with a more efficient pruning.

3.2 Efficient pruning

We present our asynchronous construction with efficient pruning. It is based on the strawman design § 3.1, with the following adjustment: *instead of storing up to 1 tree of size 2^i at each level i, we instead allow either 1 or 2 trees*. We illustrate this idea in Fig. 2.

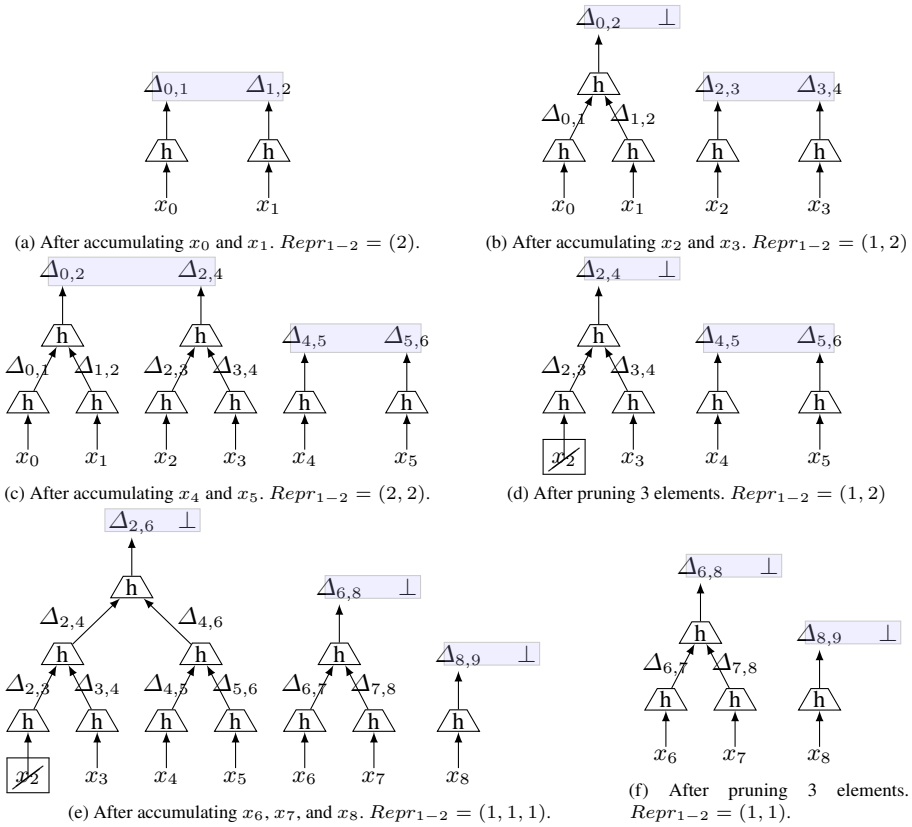


Fig. 2: Our Pruning Merkle Tree construction. Example of accumulating 9 elements and pruning 6. $\boxed{\not{x_2}}$ denotes that the element has been pruned but still remains in the data structure.

Indeed, similar to the binary representation of the levels, the 1–2 representation of n (Alg. 1) is uniquely determined by n. The 1–2 representation guarantees that each level i

is represented by at least a single full binary tree of size 2^i, which positively impacts the efficiency of the pruning functionality (Lemma 2).

Algorithm 1 $\mathsf{ConvertTo12}(n) \to a$

1 : // $n \geq 1$; **returns** the 1–2 representation $a = (a_{k-1}, \ldots, a_0)$ of n
2 : $k \leftarrow \lfloor \log_2(n+1) \rfloor$
3 : $rem \leftarrow n - (2^k - 1)$
4 : write rem **in** binary as $a_{k-1}, \ldots, a_0$ with $a_i \in \{0, 1\}$ so that $rem = \sum_{i=0}^{k-1} a_i \cdot 2^i$
5 : **for** $i = 0$ **to** $k-1$ $\quad \{a_i \leftarrow a_i + 1\}$
6 : **return** a

Lemma 2 (Pruning overhead of the 1–2 representation). *Let* $n = \sum_{i=0}^{k-1} 2^i \cdot a_i > 0$ *be the 1–2 representation of* n*, where* $a_i \in \{1, 2\}$*. Let* $p^* < 2^{k-1}$*. Then* $\frac{n}{n-p^*} = O(1)$*.*

Proof. We have $n \geq \sum_{i=0}^{k-1} 2^i = (2^k - 1)$ and $n \leq 2 \cdot \sum_{i=0}^{k-1} 2^i = 2 \cdot (2^k - 1)$. Substituting n with its lower bound and maximizing p^* as $2^{k-1} - 1$, we have $\frac{n}{n-p^*} = \frac{2^k-1}{2^k-1-(2^{k-1}-1)} = \frac{2^k-1}{2^{k-1}} < 2$. □

Lemma 2 characterizes the *constant* pruning overhead w.r.t. the irreducible state ($p^* < 2^{k-1}$). For arbitrary n and $p \leq n$, the number of elements stored effectively is $O(n - p)$ rather than $O(n)$.

In the following, we present our strawman and efficient asynchronous pruning constructions using the same algorithmic base, while introducing the parameter `level_threshold` $\in \{1, 2\}$, which denotes the maximum possible number of trees that can be stored at any level.

3.3 $\mathcal{PAA}$: Pruning Asynchronous Accumulator

Data structures The asynchronous construction maintains an array of levels, where level i stores full binary Merkle trees of size 2^i. Each tree stores its root hash, its size, and its offset, where the offset is the absolute position of the tree's leftmost leaf among all elements ever added.

The construction's digest D consists of root hashes of trees of all levels $\langle \Delta \rangle_i$ and a tuple (n, p, p_e), where n is the number of accumulated elements, p the number of pruned elements, and p_e the number of effectively pruned elements. We have that $n \geq p \geq p_e$ at all times.

A proof of inclusion (PoI) is defined w.r.t. one of the trees. It extends the standard PoI with an additional parameter `offset`, which represents the starting position of the associated Static Merkle Tree (SMT). Parameter pos is the absolute value w.r.t. all trees. Once assigned, it never changes. On the other hand, `offset` may change as more elements are added and the trees are merged. Finally, parameter val is the standard list of Merkle hashes, required to recompute the root hash.

Accumulator algorithms *Adding* a sequence of new elements to the accumulator (Alg. 2) is implemented sequentially, one by one. In this way, we maintain a data structure, which allows the effective merging of the trees and updating the associated proofs of inclusions for new elements and update values used for a proof of consistency and for updating PoIs of previously added elements. More specifically, for a newly added element, a new tree (SMT) of size 1 is created at level zero. Whenever a level contains more than `level_threshold` trees, the two leftmost trees on that level are removed and merged and the resulting tree of the doubled size is added to the next level (Alg. 5). The case $\texttt{level_threshold} = 1$ gives the strawman asynchronous construction, while $\texttt{level_threshold} = 2$ gives the efficient-pruning construction.

Pruning a number of elements (Alg. 3) is straightforward: we increase p and as long as $p - p_e$ is at least the size of the leftmost tree at the largest level, we remove that tree entirely and increase p_e by its size.

To *verify a PoI* (Alg. 6), we first identify the index of the tree based on the proof's position pos and the offset representation of all tree intervals in the accumulator. The offset representation $offsets$ maps each tree i to its boundaries $(offsets[i], offsets[i+1])$, and is uniquely determined by the tuple (n, p, p_e); the respective conversion algorithm (Alg. 4) makes use of the binary or 1–2 representation [3]. Once the tree is identified, we run the standard verification algorithm of the Merkle tree.

Verifying consistency (Alg. 10) and *updating a proof of inclusion* (Alg. 11) may require hashes that are not explicitly present in the inputs. To compute such hashes, we first use the helper algorithm (Alg. 8) to initialize a dictionary of the input hashes from the digest and the proof of consistency. We then use the standard Merkle tree hashing algorithm based on the input interval, while using the dictionary as cache (such an algorithm is called $CompDigest$ as per [22]). In Alg. 11, we make use of the fact that all trees are full binary trees, which means that a PoI can only grow as the trees are merged. New PoI hash values are collected in Alg. 9.

3.4 $\mathcal{PMT}$: Pruning Merkle Tree

The asynchronous construction in § 3.3 may expose multiple roots of Merkle Trees, while offering smaller PoIs and efficient updates at the cost of larger digests. However, in certain (legacy) applications, a traditional single-root digest is expected, e.g., in systems built around append-only consistent accumulators [22]. In this section, we present Pruning Merkle Tree, denoted $\mathcal{PMT}^h$, which is compatible with the traditional Merkle Trees by exposing a single root hash as a digest and allowing efficient pruning.

Data structures We build up $\mathcal{PMT}^h$ based on $\mathcal{PAA}^h$. $\mathcal{PMT}^h$ uses the same state as $\mathcal{PAA}^h$: the counters n, p, and p_e, along with a set of trees spanning $[p_e, n)$. The construction's digest D consists of a single root hash and a tuple (n, p, p_e). Since $\mathcal{PMT}^h$ mimics the structure of Merkle trees (MT), a proof of inclusion (PoI) is the same as in MT.

[3] From now on, we assume that the 1–2 representation is being used.

Accumulator algorithms To add a sequence of new elements to the accumulator, we first rely on the asynchronous algorithm (Alg. 2) to output a set of tree digests $\langle \Delta \rangle_i$, an array of PoIs for newly added elements and update hashes $\langle u \rangle_j$ for proofs of consistency and updating PoI for existing elements. Next, we proceed with merging trees–not necessarily full binary trees–to reach a single root tree, while updating PoIs and $\langle u \rangle_j$. Pruning elements is more involved than in $\mathcal{PAA}^h$ due to the requirement to compute the updated root digest; here, we utilize the prune algorithm of $\mathcal{PAA}^h$ as the starting point.

To prove consistency, we make use of $cache$ as a dictionary that maps input hashes to the respective intervals. $cache$ is used both for recomputing and verifying the old root digest and the new root digest. Finally, updating a PoI requires several steps. First, the old PoI is truncated to the size of the asynchronous PoI. Then, this PoI is updated via $\mathcal{PAA}^h$ to match one of the trees in the state. Lastly, the updated PoI is extended with additional hashes, computed until the new root hash is reached.

The $\mathcal{PMT}^h$ make use of $CompDigest$, a cached version of the standard hash computation in Merkle trees [22]. The $\mathcal{PMT}^h$ algorithms are presented in § C.

4 Evaluation

We evaluate the performance of our pruning accumulator constructions in a Certificate Transparency (CT)-motivated setting. The goal of the evaluation is to measure the cost of accumulation and pruning, the communication needed to keep proofs of inclusion up to date, and the size of the PoIs.

Our experiments use the following CT-style workload. We simulate a log over a longer period of time so that the system first grows, then reaches the point where certificates begin expiring, and then continues to receive new certificates while old certificates are pruned. In all experiments, the accumulator maintains hashes and auxiliary digest information; it does not store the certificates themselves.

We compare the following implementations:

1. $\mathcal{CMT}^h$: the optimized dynamic Merkle-tree consistent accumulator [22]. This construction improves append-only accumulation, but does not remove expired elements from the authenticated state.
2. $\mathcal{PMT}$: the synchronous version of our improved pruning accumulator.
3. $\mathcal{PAA}$: the asynchronous version of our improved pruning accumulator.

The Go implementation of $\mathcal{PMT}$ was derived from the implementation of $\mathcal{CMT}^h$.

All experiments were run using the same benchmarking harness across implementations. For runtime measurements, each simulated day adds $4{,}000$ new leaves; this value is $40{,}000$ for other measurements. After day 45, we prune the leaves corresponding to certificates that have expired. We report both runtime and storage-related metrics. For runtime measurements, we will report the mean, maximum, and confidence intervals across 20 repeated runs. For deterministic storage measurements, we will report the value from a single run.

The environment includes an AMD Ryzen 9 7950X3D desktop processor (x86-64), 32 GiB memory, Ubuntu 26.04 (Windows Subsystem for Linux), and Go version 1.26.4 running in single-threaded mode. No other processes were running besides those managed

by the operating system. Each leaf hash in our experiments is a SHA-256 digest using APIs from the crypto/sha256 package.

We integrate $\mathcal{PMT}$, $\mathcal{PAA}$, and $\mathcal{CMT}^h$ into the same benchmarking harness to ensure identical conditions. We hypothesize that minor unexpected trends are caused by the Go garbage collector, dynamic allocators, and a low amount of noise from background processes.

The source code, which includes the benchmarking harness, is available at [25].

Experiment 1: Accumulation and pruning cost. The first experiment measures the runtime required to process daily updates. After day 45, the daily update also includes pruning the expired prefix. For append-only baselines, the pruning step is not performed; for pruning constructions, we measure both the cost of the add operation and the cost of the prune operation.

We report the runtime for: 1) adding a daily batch of 4,000 certificates; 2) pruning the expired batch once expiration begins; 3) the combined daily maintenance cost of add plus prune. The runtime results are shown in Fig. 3.

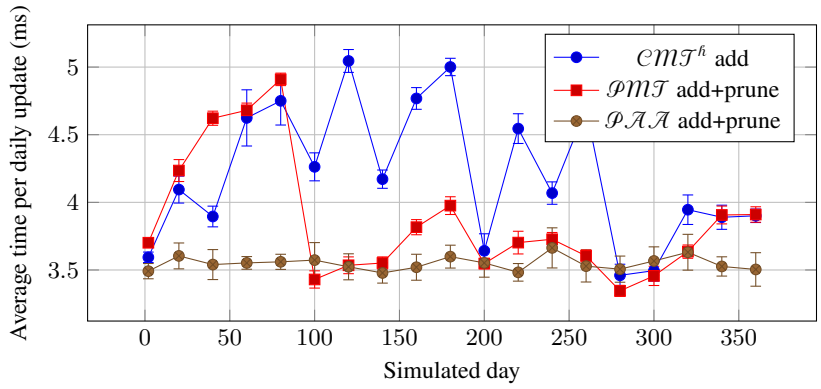
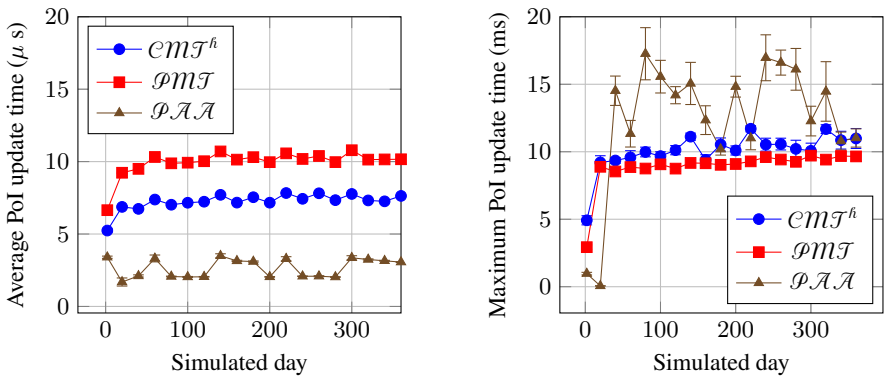


Fig. 3: Average runtime required to process daily additions and pruning under the CT-style workload. PoI generation of newly added elements is included. Error bars denote the 99% confidence interval across 20 runs.

The asynchronous construction performs the most consistently and often the fastest, due to multiple factors. One of which is that $\mathcal{PAA}$ does not use extra operations to combine the subdigests into a single root. Further, each $\mathcal{PAA}$ PoI generated during the daily accumulation process contains fewer digests than the other two constructions, cutting down further on resource usage.

Experiment 2: PoI update runtime. The second experiment measures the runtime of UpdPoI for active elements. For each implementation, we run the 365-day simulation. After each day, we measure both the average and maximum time required to update an active element's PoI from the previous day's state to the current state. For pruning constructions, expired elements were pruned from the tree before the PoIs are updated.

For pruning constructions, an update may return $\perp$ if the element was pruned before the target state. We report active-element updates separately from updates that return $\perp$. The PoI-update runtime measurements are shown in Fig. 4.



(a) Average PoI update runtime.

(b) Maximum PoI update runtime.

Fig. 4: Average and maximum runtimes required to update an active element's PoI from the previous day's state to the simulated day's state, pruning when supported. In this experiment, 4,000 certificates were added each day, with an expiration of 45 days consistent with the other experiments. Fig. 4a shows error bars denoting the 99% confidence interval, while Fig. 4b shows 95%. Not all intervals are large enough to be visible.

From the data, the asynchronous pruning tree has a clear advantage in the average case scenario, consistently able to update the PoIs in under half the time of the other constructions. This observation aligns with the fact that the synchronous and append-only constructions must generate PoI hashes along the entire path from the leaf to the root, while the asynchronous hash path ends at the local subtree root. However, when only considering the worst-case scenario, $\mathcal{PAA}$ appears to perform no better than the other constructions.

Summary of evaluation goals. Together, these experiments evaluate the main motivation for pruning: in a CT-like deployment, the log receives a large number of new certificates every day, while older certificates eventually expire. The evaluation measures whether a pruning accumulator can efficiently maintain the authenticated state over the active window, and how pruning constructions perform with proofs in terms of runtime and storage. Overall, the results show a consistent advantage in using $\mathcal{PAA}$, while $\mathcal{PMT}$ still shows advantages in many experiments over the append-only Merkle tree.

5 Analysis of $\mathcal{PAA}$ and $\mathcal{PMT}$

We now state the main properties of the pruning Merkle-tree constructions. The formal definitions of the pruning-accumulator requirements appear in § A, and the supporting

lemmas and proof details are in § D. The statements below are the properties used by the rest of the paper.

Correctness. The construction maintains enough authentication information for honestly generated PoIs, PoCs, and update values to verify against the current output digest. When an element is pruned, its updated PoI is rejected rather than being carried forward.

Theorem 1 (Correctness of $\mathcal{PAA}$). *$\mathcal{PAA}^h$ satisfies the pruning accumulator correctness requirements: PoI correctness (Def. 4), PoC and update value correctness (Def. 5), and PoI update correctness (Def. 6).*

The proof follows from the way the construction records the sibling roots needed for the local Merkle path and for transitions between digest states. The detailed argument is given in § D.

Security. The security properties reduce to collision resistance of the underlying keyed hash function. A collision in the pruning accumulator or a forged PoI would give two different Merkle computations with the same root hash.

Theorem 2 (Security of $\mathcal{PAA}$). *If h_k is any-collision resistant, then $\mathcal{PAA}^h$ satisfies pruning accumulator any-collision resistance (Def. 3) and PoI unforgeability (Def. 7).*

The proof uses the standard Merkle-tree extraction argument: if two different current sequences verify under the same digest, or if a PoI verifies for an element outside the current unpruned sequence, then some node hash must collide. The full reduction is deferred to § D.

Size. The synchronous and asynchronous variants expose different digest formats. The synchronous variant reduces the cached forest to one root, while the asynchronous variant exposes the forest roots directly.

Theorem 3 (Digest and PoI sizes). *Let $M = s.n - s.p_e$ be the number of cached leaves in a reachable state. Then:*

1. *$\mathcal{PMT}^h$ has output digest size $O(\lambda + \log s.n)$ and worst-case PoI size $\Theta(\log M)$ keyed-hash outputs.*
2. *$\mathcal{PAA}^h$ has output digest size $O((\lambda + \log s.n)\log M)$. For an active element in a subtree of size 2^i, its PoI has size $\Theta(i) \subseteq O(\log M)$ keyed-hash outputs.*
3. *For the same active element, $|D_{\mathcal{PAA}}| = O(\log M) \cdot |D_{\mathcal{PMT}}|, \quad |poi_{\mathsf{a}}| \leq |poi_{\mathsf{s}}|$.*

Thus, the asynchronous variant has a larger public digest in exchange for PoIs that stop at the local subtree root. The supporting size bounds are given in § D.

6 Conclusion

We introduced pruning accumulators, an extension of consistent accumulators that supports both appending new elements and pruning an old prefix of elements that are no longer relevant. This abstraction captures applications in which the history grows

over time but only the current unpruned sequence must remain authenticated. We formalized the syntax, correctness requirements, and security properties for pruning accumulators, including collision resistance, PoI correctness and unforgeability, and update/PoC correctness and consistency.

We then presented $\mathcal{PAA}^h$, an asynchronous pruning Merkle-tree construction that maintains a cached forest of full Merkle subtrees. By allowing up to two trees at each level, the construction supports efficient prefix pruning while keeping PoIs local to the subtree containing the proved element. The digest exposes the ordered forest roots together with counters identifying the active and cached ranges, allowing verifiers to reconstruct the relevant subtree intervals without storing them explicitly. We also described a synchronous compatibility mode for applications that require a traditional single-root digest.

Our analysis shows that the construction preserves the desired authentication properties under the collision resistance of the keyed hash function, while improving the cost of pruning and PoI maintenance compared with a naive pruning baseline. Overall, pruning accumulators provide a direct way to authenticate dynamic histories whose old prefixes can be safely removed, while retaining the Merkle-tree structure and verification mechanisms used in existing accumulator systems.

Acknowledgements

Ivan Pryvalov was supported in part by the German Federal Government, the Federal Ministry of Education and Research and the State of Brandenburg within the framework of the joint project EIZ: Energy Innovation Center (project numbers 85056897 and 03SF0693A) with funds from the Structural Development Act (Strukturstärkungsgesetz) for coal-mining regions. The other authors were partially supported by the National Science Foundation under Grants No. 2149765 and 2247810, and by the endowment to Prof. Herzberg from the Comcast Corporation.

References

1. Ateniese, G., Magri, B., Venturi, D., Andrade, E.: Redactable blockchain–or–rewriting history in bitcoin and friends. In: 2017 IEEE European symposium on security and privacy (EuroS&P). pp. 111–126. IEEE (2017)
2. Au, M.H., Tsang, P.P., Susilo, W., Mu, Y.: Dynamic universal accumulators for ddh groups and their application to attribute-based anonymous credential systems. In: Cryptographers' track at the RSA conference. pp. 295–308. Springer (2009)
3. Baldimtsi, F., Camenisch, J., Dubovitskaya, M., Lysyanskaya, A., Reyzin, L., Samelin, K., Yakoubov, S.: Accumulators with applications to anonymity-preserving revocation. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 301–315. IEEE (2017)
4. Benaloh, J., De Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. In: Workshop on the Theory and Application of of Cryptographic Techniques. pp. 274–285. Springer (1993)
5. Blibech, K., Gabillon, A.: CHRONOS: An Authenticated Dictionary Based on Skip Lists for Timestamping Systems. In: Proceedings of the 2nd ACM Workshop on Secure Web Services (SWS 2005). pp. 88–95. ACM (2005). https://doi.org/10.1145/1103022.1103037

6. Camacho, P., Hevia, A., Kiwi, M., Opazo, R.: Strong accumulators from collision-resistant hashing. In: International Conference on Information Security. pp. 471–486. Springer (2008)
7. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Annual international cryptology conference. pp. 61–76. Springer (2002)
8. Chase, M., Meiklejohn, S.: Transparency Overlays and Applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 168–179. ACM (2016)
9. Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: Proceedings of the 12th USENIX Security Symposium. pp. 29–44. USENIX (aug 2003)
10. Derler, D., Hanser, C., Slamanig, D.: Revisiting cryptographic accumulators, additional properties and relations to other primitives. In: Cryptographers' track at the rsa conference. pp. 127–144. Springer (2015)
11. Dowling, B., Günther, F., Herath, U., Stebila, D.: Secure Logging Schemes and Certificate Transparency. In: European Symposium on Research in Computer Security. pp. 140–158. Springer (2016)
12. Kales, D., Omolola, O., Ramacher, S.: Revisiting user privacy for certificate transparency. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 432–447. IEEE (2019)
13. Kemmoe, V.Y., Lysyanskaya, A.: Rsa-based dynamic accumulator without hashing into primes. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 4271–4285 (2024)
14. Krawczyk, H., Rabin, T.: Chameleon signatures. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA. The Internet Society (2000)
15. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Experimental) (Jun 2013). https://doi.org/10.17487/RFC6962, https://www.rfc-editor.org/rfc/rfc6962.txt, obsoleted by RFC 9162
16. Laurie, B., Messeri, E., Stradling, R.: Certificate Transparency Version 2.0. RFC 9162 (Experimental) (Dec 2021). https://doi.org/10.17487/RFC9162, https://www.rfc-editor.org/rfc/rfc9162.txt
17. Laurie, B.: Certificate transparency. Communications of the ACM **57**(10), 40–46 (2014)
18. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Proof-Infused Streams: Enabling Authentication of Sliding Window Queries on Streams. In: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007). pp. 147–158. VLDB Endowment (2007)
19. Li, J., Li, N., Xue, R.: Universal accumulators with efficient nonmembership proofs. In: International Conference on Applied Cryptography and Network Security. pp. 253–269. Springer (2007)
20. Maniatis, P., Baker, M.: Authenticated Append-only Skip Lists. Tech. Rep. cs/0302010, arXiv (2003), https://arxiv.org/abs/cs/0302010
21. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: Bringing Key Transparency to End Users. In: USENIX Security Symposium. pp. 383–398 (2015)
22. Mendonca, A., Shi, H., Huynh, T., Pryvalov, I., Herzberg, A.: Efficient merkle-tree consistent accumulator. In: Proceedings of the 56th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2026). Charlotte, USA (June 22–25 2026), to appear
23. Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. In: Conference on the theory and application of cryptographic techniques. pp. 369–378. Springer (1987)
24. Reyzin, L., Yakoubov, S.: Efficient asynchronous accumulators for distributed pki. In: International Conference on Security and Cryptography for Networks. pp. 292–309. Springer (2016)

25. Shi, H.: Golang implementation of apt and spt (2026), https://github.com/storchdev/opt-pruning-tree
26. Todd, P.: Merkle mountain ranges. https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md (March 2013), openTimestamps documentation
27. Tomescu, A., Bhupatiraju, V., Papadopoulos, D., Papamanthou, C., Triandopoulos, N., Devadas, S.: Transparency logs via append-only authenticated dictionaries. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1299–1316 (2019)

A Definitions for Pruning Accumulator Schemes

A.1 Correctness Requirements

We now present the requirements for pruning accumulators. As in the case of consistent accumulators [22], we first define correctness requirements, and then define collision resistance and unforgeability requirements. The main difference is that a pruning accumulator represents only the current unpruned sequence $Cur(\mathsf{Ops})$, rather than the entire sequence of elements ever added. Thus, correctness and security are stated with respect to the elements that remain unpruned.

A pruning accumulator should satisfy PoI correctness, PoC and update value correctness, and PoI update correctness. These requirements are analogous to the corresponding requirements for consistent accumulators, except that they are interpreted with respect to the current unpruned sequence.

PoI correctness. Intuitively, a pruning accumulator satisfies PoI correctness if every element newly added by an honest Add operation has a corresponding proof of inclusion that verifies with respect to the digest output by that operation. If the operation is a Prune operation, then no new element is added, and therefore no new PoI correctness requirement is imposed for newly added elements.

Definition 4 (Pruning accumulator PoI correctness). *We say that a keyed pruning accumulator scheme α ensures* Proof-of-Inclusion (PoI) correctness *if the following holds for every binary string key k, every legal operation sequence Ops_{i-1}, and every sequence of binary strings $X = \langle x_0, \ldots, x_{m-1} \rangle$:*

$$\begin{pmatrix} s_{i-1} \leftarrow \alpha_k.s(\mathsf{Ops}_{i-1}) \wedge \\ D_i \leftarrow \alpha_k.\mathsf{Accum}_{s_{i-1}}(\mathsf{Add}, X).D \wedge \\ PoI_i \leftarrow \alpha_k.\mathsf{Accum}_{s_{i-1}}(\mathsf{Add}, X).PoI \end{pmatrix} \Rightarrow \begin{pmatrix} (\forall x \in X)(\exists poi \in PoI_i) \\ \alpha_k.\mathsf{VerPoI}(D_i, x, poi) = \textsc{True} \end{pmatrix}.$$

PoC and update value correctness A pruning accumulator satisfies PoC and update value correctness if every honest state transition produces a proof of consistency and an update value that verify between the old digest and the new digest. The state transition may be either an addition operation or a pruning operation, provided that the resulting operation sequence is legal.

Definition 5 (Pruning accumulator PoC and update value correctness). *We say that a keyed pruning accumulator scheme α ensures* PoC and update value correctness *if the*

following holds for every binary string key k, every legal operation sequence Ops_{i-1}, and every operation (op_i, par_i) such that $\mathsf{Ops}_i \equiv \mathsf{Ops}_{i-1} +\!\!+ \langle (op_i, par_i) \rangle$ is legal:

$$\left(\begin{array}{c} s_{i-1} \leftarrow \alpha_k.s(\mathsf{Ops}_{i-1}) \wedge \\ D_{i-1} \leftarrow \alpha_k.D(\mathsf{Ops}_{i-1}) \wedge \\ D_i \leftarrow \alpha_k.\mathsf{Accum}_{s_{i-1}}(op_i, par_i).D \wedge \\ PoC_i \leftarrow \alpha_k.\mathsf{Accum}_{s_{i-1}}(op_i, par_i).PoC \wedge \\ u_i \leftarrow \alpha_k.\mathsf{Accum}_{s_{i-1}}(op_i, par_i).u \end{array}\right) \Rightarrow \left(\begin{array}{c} \alpha_k.\mathsf{VerPoC}(D_{i-1}, PoC_i, D_i) = \textsc{True} \wedge \\ \alpha_k.\mathsf{VerUpd}(D_{i-1}, u_i, D_i) = \textsc{True}. \end{array}\right)$$

PoI update correctness A pruning accumulator satisfies PoI update correctness if updating a valid PoI with an honestly generated update value gives the correct result for the new digest. If the element remains unpruned, the updated PoI should verify with respect to the new digest. If the element is pruned by the update, then UpdPoI should return $\perp$.

Definition 6 (Pruning accumulator PoI update correctness). *We say that a keyed pruning accumulator scheme α ensures* PoI update correctness *if the following holds for every binary string key k, every legal operation sequence Ops_{i-1}, every element $x \in Cur(\mathsf{Ops}_{i-1})$, every valid PoI poi, and every operation (op_i, par_i) such that $\mathsf{Ops}_i \equiv \mathsf{Ops}_{i-1} +\!\!+ \langle (op_i, par_i) \rangle$ is legal:*

$$\left(\begin{array}{c} s_{i-1} \leftarrow \alpha_k.s(\mathsf{Ops}_{i-1}) \wedge \\ D_{i-1} \leftarrow \alpha_k.D(\mathsf{Ops}_{i-1}) \wedge \\ \alpha_k.\mathsf{VerPoI}(D_{i-1}, x, poi) = \textsc{True} \wedge \\ D_i \leftarrow \alpha_k.\mathsf{Accum}_{s_{i-1}}(op_i, par_i).D \wedge \\ u_i \leftarrow \alpha_k.\mathsf{Accum}_{s_{i-1}}(op_i, par_i).u \wedge \\ poi' \leftarrow \alpha_k.\mathsf{UpdPoI}(poi, u_i) \end{array}\right) \Rightarrow \left(\begin{array}{c} (x \in Cur(\mathsf{Ops}_i) \Rightarrow \alpha_k.\mathsf{VerPoI}(D_i, x, poi') = \textsc{True}) \wedge \\ (x \notin Cur(\mathsf{Ops}_i) \Rightarrow poi' = \perp). \end{array}\right)$$

A.2 Pruning Accumulator Proof-of-Inclusion (PoI) Unforgeability Property

We now define the pruning accumulator PoI unforgeability requirement.

Definition 7 (Pruning Accumulator PoI Unforgeability). *Let α be a keyed pruning accumulator (Pruning Accumulator Scheme) scheme as defined in Def. 1.*

PoI Unforgeability. *We say that α ensures* Proof-of-Inclusion (PoI) Unforgeability *if for every probabilistic polynomial-time adversary $\mathcal{A}$, the following probability is negligible in λ:*

$$\Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} \left[\begin{array}{c} \mathsf{Ops} \leftarrow \mathcal{A}(k); \\ D \leftarrow \alpha_k.D(\mathsf{Ops}); \\ (x, poi) \leftarrow \mathcal{A}(\mathsf{Ops}, D, k) : \\ \mathsf{Ops} \textit{ is legal} \wedge \\ \alpha_k.\mathsf{VerPoI}(D, x, poi) = \textsc{True} \wedge \\ x \notin Cur(\mathsf{Ops}) \end{array}\right] \in NEGL(\lambda).$$

Notice that in a pruning accumulator, the digest value changes in subsequent Accum events; therefore, we should also update the PoI accordingly, using the UpdPoI function. The above definition does not account for that as we define UpdPoI requirements separately in Def. 6.

B A note on definitions

B.1 PoC and update value consistency

One might also try to define an adversarial consistency requirement for proofs of consistency and update values, analogous to PoI unforgeability. However, this does not give a meaningful additional security notion for pruning accumulators as defined here. The reason is that pruning accumulators allow both appending new elements and pruning an old prefix. Therefore, for any two legal output digests that represent valid accumulator states, there is always some legal continuation from the first current sequence to the second: prune all elements currently represented by the first digest, and then append the current sequence represented by the second digest. Thus, a definition that only asks whether the new digest could arise from the old digest under some legal sequence of operations is vacuous.

For this reason, we do not define a separate PoC or update-value "unforgeability" requirement. Instead, PoCs and update values are required to satisfy correctness: honestly generated PoCs and update values must verify between the old and new digests produced by the accumulator. Security against changing the represented sequence is captured by collision resistance of the output digest, while security against proving membership of an absent or pruned element is captured by PoI unforgeability.

C Algorithms

In this section, we first present the algorithms of $\mathcal{PAA}_k$: Add (Alg. 2), Prune (Alg. 3), $joinTrees$ (Alg. 5), VerPoI (Alg. 6), VerPoC (Alg. 10), and UpdPoI (Alg. 11), along with helper algorithms ConvertToOffsets (Alg. 4), $poc_offsets$ (Alg. 7), $initCache$ (Alg. 8), and $collect_poi$ (Alg. 9). We then present the algorithms of $\mathcal{PMT}_k$: Add (Alg. 12), Prune (Alg. 13), $updates_to_cache$ (Alg. 14), VerPoC (Alg. 15), and UpdPoI (Alg. 16), except for the PoI verification, which is the standard Merkle tree.

D Proof Details for the Analysis

This appendix gives the supporting lemmas and proof details for the main analysis statements in § 5. The structural lemmas below are used only to prove the correctness, security, and size theorems; the main text states the resulting accumulator properties.

Throughout this section, let a reachable state s have counters $s.n$, $s.p$, and $s.p_e$, where $s.n$ is the number of elements ever added, $s.p$ is the number of elements pruned, and $s.p_e$ is the number of elements effectively pruned from the cached forest. We write

$$N \equiv s.n - s.p \qquad \text{and} \qquad M \equiv s.n - s.p_e$$

Algorithm 2 $\mathcal{PAA}_k.\mathsf{Add}_s(X) \to (s', D, PoI, PoC)$

```
1 : if s ≡ ϵ  { s.levels ← ⟨⟩ ;  s.n ← 0;  s.p ← 0;  s.p_e ← 0 }
2 : s' ← s
3 : for x ∈ X
4 :   t ← SMT();  t.Δ ← h_k(0x00||x);  t.n ← 1;  t.offset ← s'.n;  t.u ← ⟨⟩
5 :   t.PoI[0].n ← 1;  t.PoI[0].offset ← s'.n;  t.PoI[0].pos ← s'.n
6 :   t.PoI[0].val ← ⟨⟩
7 :   s'.levels[0].push_back(t);  s'.n ← s'.n + 1
8 :   for i ∈ 0..|s'.levels| − 1
9 :     if |s'.levels[i]| > level_threshold
10 :      t_0 ← s'.levels[i].pop_front();      t_1 ← s'.levels[i].pop_front()
11 :      t ← joinTrees(s.n, t_0, t_1)
12 :      s'.levels[i + 1].push_back(t)
13 : Δ ← ⟨⟩ ;  PoI ← ⟨⟩ ;  PoC ← ∅
14 : for i ∈ |s'.levels| − 1..0;  for t ∈ s'.levels[i]
15 :     Δ.push_back(t.Δ)
16 :     for poi ∈ t.PoI;  PoI.push_back(poi)
17 :     for upd ∈ t.u;  PoC.u.push_back(upd)
18 :     t.PoI ← ∅;  t.u ← ∅
19 : D ← (Δ, s'.n, s'.p, s'.p_e)
20 : PoC.n_old ← s.n;  PoC.p_old ← s.p;  PoC.(p_e)_old ← s.p_e
21 : PoC.n_new ← s'.n;  PoC.p_new ← s'.p;  PoC.(p_e)_new ← s'.p_e
22 : return (s', D, PoI, PoC)
```

for the number of currently active elements and the number of cached leaves, respectively. For a digest

$$D = (\boldsymbol{\Delta}, n, p, p_e),$$

let $\mathsf{Intervals}(D)$ denote the ordered list of intervals obtained by running $\mathsf{ConvertToOffsets}(n, p, p_e)$. The j-th component of $\boldsymbol{\Delta}$ is intended to be the root hash of the j-th interval in $\mathsf{Intervals}(D)$.

D.1 Main theorem proofs

Theorem 1 (Correctness of $\mathcal{PAA}$). *$\mathcal{PAA}^h$ satisfies the pruning accumulator correctness requirements: PoI correctness (Def. 4), PoC and update value correctness (Def. 5), and PoI update correctness (Def. 6).*

Proof. PoI correctness follows from Lemma 5. PoC and update value correctness follows from Lemma 6. PoI update correctness follows from Lemma 7. Together these are exactly the correctness requirements listed in the theorem. □

Theorem 2 (Security of $\mathcal{PAA}$). *If h_k is any-collision resistant, then $\mathcal{PAA}^h$ satisfies pruning accumulator any-collision resistance (Def. 3) and PoI unforgeability (Def. 7).*

Proof. Any-collision resistance follows from Lemma 8. PoI unforgeability follows from Lemma 9. Both reductions use any-collision resistance of the keyed hash function through the standard Merkle-tree extraction argument. □

Algorithm 3 $\mathcal{PAA}_k.\mathsf{Prune}(s, count) \to (s', D, PoC)$

$s' \leftarrow s$; $s'.p \leftarrow s'p + count$
while $|s'.\texttt{levels}| > 0$ and $s'.p - s'.p_e \geq s'.\texttt{levels}[last][0].n$
  $t_{pruned} \leftarrow s'.\texttt{levels}[last].pop_front()$
  $s'.p_e \leftarrow s'.p_e + t_{pruned}.n$
  while $|s'.\texttt{levels}| > 0$ and $|s'.\texttt{levels}[last]| = 0$; $s'.\texttt{levels}.pop_back()$
$\Delta \leftarrow \langle\rangle$
for $i \in |s'.\texttt{levels}| - 1..0$; **for** $t \in s'.\texttt{levels}[i]$; $\Delta.push_back(t.\Delta)$
$D \leftarrow (\Delta, s'.n, s'.p, s'.p_e)$
$PoC \leftarrow \emptyset$; $PoC.u \leftarrow \langle\rangle$
$PoC.n_{old} \leftarrow s.n$; $PoC.p_{old} \leftarrow s.p$; $PoC.(p_e)_{old} \leftarrow s.p_e$
$PoC.n_{\mathbf{new}} \leftarrow s'.n$; $PoC.p_{\mathbf{new}} \leftarrow s'.p$; $PoC.(p_e)_{\mathbf{new}} \leftarrow s'.p_e$
return (s', D, PoC)

Algorithm 4 $\mathsf{ConvertToOffsets}(n, p, p_e) \to offsets$

$(a_{k-1}, ..., a_0) \leftarrow \mathsf{ConvertTo12}(n, p, p_e)$
$offsets = \langle p_e \rangle$
for $level \in k - 1...0$
  $level_tree_size \leftarrow 2^{level}$
  for $i \in 0..a[level] - 1$
    $start \leftarrow offsets[last]$
    $offsets.push_back(start + level_tree_size)$
return $offsets$

Theorem 3 (Digest and PoI sizes). *Let $M = s.n - s.p_e$ be the number of cached leaves in a reachable state. Then:*

1. *$\mathcal{PMT}^h$ has output digest size $O(\lambda + \log s.n)$ and worst-case PoI size $\Theta(\log M)$ keyed-hash outputs.*
2. *$\mathcal{PAA}^h$ has output digest size $O((\lambda + \log s.n) \log M)$. For an active element in a subtree of size 2^i, its PoI has size $\Theta(i) \subseteq O(\log M)$ keyed-hash outputs.*
3. *For the same active element, $|D_{\mathcal{PAA}}| = O(\log M) \cdot |D_{\mathcal{PMT}}|$, $|poi_{\mathsf{a}}| \leq |poi_{\mathsf{s}}|$.*

Proof. The synchronous size bound is the single-root digest bound of Thm. 4. The asynchronous digest and PoI bounds follow from Thm. 5. The comparison follows by combining these two bounds: the asynchronous digest exposes one tuple per cached subtree, while the synchronous digest exposes one merged root and one count. For PoIs, the asynchronous PoI stops at the local subtree root, while the synchronous PoI continues to the merged root, so the asynchronous PoI is no larger. □

D.2 Structural invariants

Lemma 3 (Asynchronous forest invariant). *For every reachable state s of $\mathcal{PAA}^h$, the trees stored in $s.\texttt{levels}$, read in digest order, form a left-to-right partition of the cached range*

$$[s.p_e, s.n).$$

Algorithm 5 $\mathcal{PAA}_k.joinTrees(n_{old}, t_0, t_1) \to t$

$t \leftarrow SMT()$
$t.n \leftarrow t_0.n + t_1.n$; $t.\texttt{offset} \leftarrow t_0.\texttt{offset}$; $t.PoI \leftarrow \langle\rangle$
$t.\Delta \leftarrow h_k(0x01||t_0.\Delta||t_1.\Delta)$
for $poi \in t_0.PoI$; $poi.val.push_back(t_1.\Delta)$
for $poi \in t_1.PoI$; $poi.val.push_back(t_0.\Delta)$
for $poi \in t_0.PoI$; $t.PoI.push_back(poi)$
for $poi \in t_1.PoI$; $t.PoI.push_back(poi)$
$t.u \leftarrow t_0.u$
if $t_0.\texttt{offset} < n_{old}$ and $t_1.\texttt{offset} + t_1.n > n_{old}$
  if $t_1.\texttt{offset} < n_{old}$
    for $u \in t_1.u$; $t.u.push_back(u)$
  else
    $t.u.push_back(t_1.\Delta)$
for $poi \in t.PoI$; $\{poi.\texttt{offset} \leftarrow t.\texttt{offset}$; $poi.n \leftarrow t.n\}$
return t

Algorithm 6 $\mathcal{PAA}_k.\mathsf{VerPoI}(D, x, poi) \to \{\textsc{True}$ or $\textsc{False}\}$

parse D as $(\langle\Delta_D\rangle_i, n, p, p_e)$
parse poi as $(pos, n_{poi}, offset, \langle\Delta_{poi}\rangle_i)$
if $pos < p$: **return** $\textsc{False}$
$offsets = \mathsf{ConvertToOffsets}(n, p, p_e)$
find $index$ s.t. $offsets[index] \le pos < offsets[index + 1]$
return $SMT.\mathsf{VerPoI}(\Delta_D[index], x, (\langle\Delta_{poi}\rangle_i, n_{poi}, pos - offsets[index]))$

Each tree at level i is a full binary Merkle tree with 2^i leaves, its `offset` *field is the left endpoint of its interval, and its root field Δ is the Merkle root of the leaves in that interval. Moreover, for the efficient pruning setting* `level_threshold` $= 2$*, there are at most two trees at each level.*

Proof. The invariant is immediate for the empty state. An addition first creates a singleton tree whose interval is $[s.n, s.n + 1)$. Whenever a level contains too many trees, joinTrees merges the two oldest adjacent trees of the same size into a full tree of twice the size. The merged tree keeps the left child's offset and hashes the two child roots with the internal-node tag, so both the interval and the root are correct. Because merges are performed in FIFO order within each level, they preserve the left-to-right order of the forest. A pruning operation only removes whole trees from the left of the forest and advances $s.p_e$ by exactly the size of each removed tree. Thus the remaining trees continue to partition $[s.p_e, s.n)$, and the level bound is preserved by the merging rule. □

Lemma 4 (Digest and offset reconstruction). *Let $D = (\boldsymbol{\Delta}, n, p, p_e)$ be an honestly generated digest of $\mathcal{PAA}^h$. Then* $\mathsf{Intervals}(D)$ *is exactly the ordered list of intervals represented by the trees in the cached forest, and each entry $\boldsymbol{\Delta}[j]$ is the Merkle root of the corresponding interval.*

Algorithm 7 $poc_offsets(n_{old}, left, right) \to \langle offsets \rangle_i$

```
1 : offsets ← ⟨right⟩
2 : while right − left > 1
3 :   mid ← (left + right)/2
4 :   if mid ≥ n_old
5 :     offsets.push_front(mid)
6 :     right ← mid
7 :   else
8 :     left ← mid
9 : return offsets
```

Algorithm 8 $initCache(offsets, \langle \Delta_D \rangle_i, offsets', poc) \to cache$

```
1 : cache ← {}
2 : for each i;   cache ← cache ∪ {[offsets[i], offsets[i + 1]) ⟹ Δ_D[i]}
3 : find index s.t. offsets'[index] ≤ poc.n_old − 1 < offsets'[index + 1]
4 : if offsets'[index + 1] > poc.n_old
5 :   addPocToCache(cache, offsets'[index], offsets'[index + 1], poc)
6 : return cache
7 :
8 : addPocToCache(cache, left_t, right_t, poc){
9 :   offsets ← poc_offsets(poc.n_old, left_t, right_t)
10 :  for each i
11 :    cache ← cache ∪ {[offsets[i], offsets[i + 1]) ⟹ poc.u[i]}
12 : }
```

Proof. By Lemma 3, the cached forest is determined by n, p_e, and the 1–2 representation of the cached length $n - p_e$. The procedure ConvertToOffsets reconstructs the interval boundaries by starting at p_e and walking these tree sizes in digest order. The digest construction emits exactly the root fields of the same trees in the same order, so the reconstructed intervals and the digest roots agree entry-by-entry. □

D.3 Correctness

Lemma 5 ($\mathcal{PAA}$ PoI correctness). *$\mathcal{PAA}^h$ satisfies pruning accumulator PoI correctness (Def. 4).*

Proof. When a new element is added, the construction first creates a singleton tree and an empty Merkle path for that element. Each time joinTrees merges the element's current tree with a sibling tree, it appends the sibling root to the element's PoI and updates the PoI's tree size and offset. Therefore, after all merges finish, the PoI contains exactly the sibling roots on the path from the element's leaf to the root of the final tree containing it. By Lemma 3, VerPoI reconstructs the same tree interval from the digest, selects the matching root by the element's absolute position, and then runs the standard static Merkle-tree verifier on the local position. Hence every honestly generated PoI for a newly added element verifies against the digest returned by that operation. □

Algorithm 9 $collect_poi(collected, cache, left_{small}, right_{small}, left_{big}, right_{big})$

```
1 : if right_big − left_big > right_small − left_small
2 :   m ← largest power of 2 < (right_big − left_big)
3 :   mid ← left_big + m
4 :   if right_small ≤ mid
5 :     collect_poi(collected, cache, left_small, right_small, left_big, mid)
6 :     collected.push_back(CompDigest^cache_{mid,right_big}(∅))
7 :   else
8 :     collect_poi(collected, cache, left_small, right_small, mid, right_big)
9 :     collected.push_back(CompDigest^cache_{left_big,mid}(∅))
```

Algorithm 10 $\mathcal{PAA}_k.\mathsf{VerPoC}(D, poc, D') \rightarrow \{\textsc{True}$ or $\textsc{False}\}$

```
1 : parse D as (⟨Δ_D⟩_i , n, p, p_e)
2 : parse D' as (⟨Δ'_D⟩_i , n', p', p'_e)
3 : offsets = ConvertToOffsets(n, p, p_e)
4 : if p' ≠ p
5 :   map each Δ_D[i] to [offsets[i], offsets[i + 1])
6 :   return verify ⟨Δ'_D⟩_i is a subset of ⟨Δ_D⟩_i by pruning elements outside [p'_e, n')
7 : else
8 :   if n − p'_e = 0   { return TRUE }
9 :   offsets' = ConvertToOffsets(n', p', p'_e)
10 :  cache ← initCache(offsets, ⟨Δ_D⟩_i , offsets', poc)
11 :  for each i s.t. offsets'[i] < n
12 :    Δ_ver ← CompDigest^cache_{offsets'[i],offsets'[i+1]}(∅)
13 :    if Δ_ver ≠ Δ'_D[i]   { return FALSE }
14 :  return TRUE
```

Lemma 6 ($\mathcal{PAA}$ PoC and update value correctness). *$\mathcal{PAA}^h$ satisfies pruning accumulator PoC and update value correctness (Def. 5), with the update value and PoC carrying the same authentication information.*

Proof. Consider an honest transition from D to D'. If the transition only prunes, then the new cached forest is obtained by deleting a prefix of whole trees from the old cached forest and advancing p_e; the verifier can check this directly from the old and new offset lists. If the transition appends elements, then the new forest may contain trees that combine old cached roots with newly added roots. The PoC records exactly the sibling roots needed to rebuild every new root whose interval intersects the old boundary. The verifier initializes a cache with the roots from D, adds the roots supplied in the PoC, and recomputes the roots of the intervals in D'. Each recomputed root matches because it follows the same merge sequence as the honest update. Thus $\mathsf{VerPoC}(D, PoC, D')$ accepts; the same argument applies to VerUpd when the update value contains the same information. □

Lemma 7 ($\mathcal{PAA}$ PoI update correctness). *$\mathcal{PAA}^h$ satisfies pruning accumulator PoI update correctness (Def. 6).*

Algorithm 11 $\mathcal{PAA}_k.\mathsf{UpdPoI}(poi, \langle \Delta_D \rangle_i, u) \to poi' \cup \{\bot\}$

```
1 : if poi.pos < u.p_new;  return
2 : offsets = ConvertToOffsets(u.n_old, p, p_e)
3 : offsets' = ConvertToOffsets(u.n_new, p, p_e)
4 : find index s.t. offsets[index] ≤ poi.pos < offsets[index + 1]
5 : find index' s.t. offsets'[index] ≤ poi.pos < offsets'[index + 1]
6 : collected ← ⟨⟩
7 : cache ← initCache(offsets, ⟨Δ_D⟩_i, offsets', u)
8 : left_small ← offsets[index];   right_small ← offsets[index + 1]
9 : left_big ← offsets'[index'];   right_big ← offsets'[index' + 1]
10 : collect_poi(collected, cache, left_small, right_small, left_big, right_big)
11 : poi' ← poi
12 : for extra_hash ∈ collected   { poi'.val.push_back(extra_hash) }
13 : poi'.n ← right_big − left_big
14 : return poi'
```

Proof. Let poi be a valid PoI for an element at absolute position $poi.\mathsf{pos}$ before an honest update. If the update prunes that position, then the updated active range begins after $poi.\mathsf{pos}$, and UpdPoI returns $\bot$. Otherwise, the element remains active. Its local Merkle path remains valid inside the old tree, and the only possible change is that the tree containing the element may be merged into larger trees. For each such merge, the update value supplies the sibling root needed to extend the old path by one level. Appending these roots gives a path from the same leaf to the new tree root. By Lemma 3, the verifier selects that new root from D', so the updated PoI verifies. □

D.4 Security

Lemma 8 ($\mathcal{PAA}$ any-collision resistance). *If h_k is any-collision resistant, then $\mathcal{PAA}^h$ satisfies pruning accumulator any-collision resistance (Def. 3).*

Proof. Suppose two legal operation sequences produce the same non-$\bot$ digest $D = (\boldsymbol{\Delta}, n, p, p_e)$ but different current unpruned sequences. Equality of the digest fixes n, p, p_e, and the full ordered list of cached subtree roots. By Lemma 3, the two executions have the same cached interval decomposition. If any corresponding cached interval contained different leaves, then equality of the two Merkle roots for that interval would yield a collision in h_k by the standard Merkle-tree extraction argument. Except with negligible probability, all cached leaves are therefore equal interval-by-interval. Since equality of p fixes the active suffix $[p, n)$, the current unpruned sequences are equal, contradicting the assumed collision. □

Lemma 9 ($\mathcal{PAA}$ PoI unforgeability). *If h_k is any-collision resistant, then $\mathcal{PAA}^h$ satisfies pruning accumulator PoI unforgeability.*

Proof. A successful forgery consists of a digest D, an element x, and a PoI poi such that $x \notin Cur(\mathsf{Ops})$ but $\mathsf{VerPoI}(D, x, poi) = \textsc{True}$. The verifier first checks that the claimed

Algorithm 12 $\mathcal{PMT}_k.\mathsf{Add}_s(X) \to (s', D, PoI, PoC)$

collect digests $\langle \Delta_{old} \rangle_i$ from s
$(s', D_a, PoI_a, PoC_a) \leftarrow \mathcal{PAA}_k.\mathsf{Add}_s(X)$
parse D_a as $(\langle \Delta_a \rangle_i)$
$offsets = \mathsf{ConvertToOffsets}(n, p, p_e)$
$cache = \emptyset$
for each i; $\quad cache \leftarrow cache \cup \{[offsets[i], offsets[i+1]) \implies \Delta_a[i]\}$
$\Delta_{root} \leftarrow CompDigest^{cache}_{p_e,n}(\emptyset)$
$PoC \leftarrow PoC_a$; $\quad PoC.u \leftarrow \langle\rangle$
if $n_{old} - (p_e)_{old}$
  for each $i \in \langle \Delta_{old} \rangle_i$; $\quad PoC.u.push_back(\Delta_{old}[i])$
  for each $upd \in PoC_a.u$; $\quad PoC.u.push_back(upd)$
  find $index$ s.t. $offsets[index] \le pos < offsets[index+1]$
  for each i; $\quad$ **if** $offsets[i] \ge offsets[index+1]$
    $PoC.u.push_back(\Delta_a[i])$
$PoI \leftarrow PoI_a$
for poi **in** PoI
  find $index$ s.t. $offsets[index] \le poi.\mathsf{pos} < offsets[index+1]$
  $left_{small} \leftarrow offsets[index]$; $\quad right_{small} \leftarrow offsets[index+1]$
  $collected_poi = \langle\rangle$
  $collect_poi(collected_poi, cache, offsets[index], offsets[index+1], p_e, n)$
  for $\Delta \in colected_poi$; $\quad poi.val.push_back(\Delta)$
  $poi.n = n - p_e$; $\quad poi.offset = p_e$
$D \leftarrow (\Delta_{root}, s'.n, s'.p, s'.p_e)$
return (s', D, PoI, PoC)

position is active and lies in one of the intervals reconstructed from D. It then verifies a static Merkle path from x to the root of that interval. Since x is not the genuine active element at that position, the forged path and the honest tree for the same interval differ at some leaf. Walking upward from the forged and honest leaves, the first node at which the two computations agree after having different children gives two distinct inputs to h_k with the same output. This is an any-collision for h_k, except with negligible probability. □

D.5 Digest size and PoI complexity

We now compare the digest and PoI sizes of the two constructions to the naive baseline $\mathcal{NP}p\mathcal{T}$ and to the synchronous variant. All sizes are counted in keyed-hash outputs of length λ, plus integer counts of bit-length $O(\log s.n)$.

Lemma 10 (Naive PoI complexity). *The worst-case number of sibling hashes in a PoI of an active element under $\mathcal{NP}p\mathcal{T}$ is $\Theta(\log s.n)$.*

Proof. $\mathcal{NP}p\mathcal{T}$ retains the dyadic cover of $[0, s.n)$ regardless of $s.p$, so a PoI traverses the binary tree of depth $\lceil \log_2 s.n \rceil$. The worst case is realized by an active leaf in the largest cached subtree. □

Algorithm 13 $\mathcal{PMT}_k.\mathsf{Prune}(s, count) \to (s', D, PoC)$

collect digests $\langle \Delta_{old} \rangle_i$ from s
$(s', D_a, PoC_a) \leftarrow \mathcal{PAA}_k.\mathsf{Prune}(s, count)$
parse D_a as $(\langle \Delta_a \rangle_i, s'.n, s'.p, s'.p_e)$
$\Delta \leftarrow \langle\rangle$
if $n - p_e > 0$
  $offsets = \mathsf{ConvertToOffsets}(n, p, p_e)$
  $cache = \{\}$
  for each i; $cache \leftarrow cache \cup \{[offsets[i], offsets[i+1]) \implies \Delta_a[i]\}$
  $\Delta_{root} \leftarrow CompDigest^{cache}_{p_e,n}(\emptyset)$; $\Delta.push_back(\Delta_{root})$
$D \leftarrow (\Delta, s'.n, s'.p, s'.p_e)$
$PoC \leftarrow PoC_a$
return (s', D, PoC)

Algorithm 14 $\mathcal{PMT}_k.updates_to_cache(cache, u)$

$offsets = \mathsf{ConvertToOffsets}(u.n_{old}, u.p_{old}, u.(p_e)_{old})$
parse $u.val$ as $(\langle \Delta_{old} \rangle_i, \langle upd \rangle_i, \langle \Delta_{\mathbf{new}} \rangle_i)$
$cache \leftarrow \{\}$
for each i; $cache \leftarrow cache \cup \{[offsets[i], offsets[i+1]) \implies \Delta_{old}[i]\}$
if $u.n_{old} < u.n_{\mathbf{new}}$
  $offsets' = \mathsf{ConvertToOffsets}(u.n_{\mathbf{new}}, u.p_{\mathbf{new}}, u.(p_e)_{\mathbf{new}})$
  find $index$ s.t. $offsets'[index] \leq u.n_{old} - 1 < offsets'[index+1]$
  $poc_offsets_a \leftarrow poc_offsets(n_{old}, offsets'[index], offsets'[index+1])$
  $cache \leftarrow cache \cup \{[poc_offsets_a[i], poc_offsets_a[i+1]) \implies upd[i]\}$
  find smallest $skip$ s.t. $offsets'[skip] \geq offsets'[index+1]$
  for each $i \in |\Delta_{\mathbf{new}}|$
    $cache \leftarrow cache \cup \{[offsets'[skip+i], offsets'[skip+i+1]) \implies \Delta_{\mathbf{new}}[i]\}$

Theorem 4 (**$\mathcal{PMT}$ digest and PoI complexity**). *For every reachable state of $\mathcal{PMT}^h$:*

1. *the output digest D has size $O(\lambda + \log s.n)$, namely one root hash and one integer count;*
2. *the worst-case PoI size for an active element is $\Theta(\log M)$ keyed-hash outputs, where $M = s.n - s.p_e$ is the number of cached leaves.*

Proof. (1) Direct from the digest format (Δ, c). (2) By the 1-2 forest invariant the cached tree has depth $\lceil \log_2 M \rceil$ and a PoI from leaf to root emits one sibling per level; the upper bound holds always, and the lower bound is realised by leaves in the deepest cached subtree. □

Theorem 5 (**$\mathcal{PAA}$ digest and PoI complexity**). *For every reachable state of $\mathcal{PAA}^h$:*

1. *the output digest D has size $O\big((\lambda + \log s.n) \cdot \log M\big)$, namely one tuple per subtree of the 1-2 forest;*
2. *the worst-case PoI size for an active element in a subtree of size 2^i is $\Theta(i) \subseteq O(\log M)$ keyed-hash outputs;*

Algorithm 15 $\mathcal{PMT}_k.\mathsf{VerPoC}(D, poc, D') \to \{\textsc{True}$ or $\textsc{False}\}$

```
1 : parse D as (⟨Δ_D⟩_i, n, p, p_e)
2 : parse D' as (⟨Δ'_D⟩_i, n', p', p'_e)
3 : if n' = p'_e;   return TRUE
4 : if n = p_e;   return TRUE
5 : PMT_k.updates_to_cache(cache, poc)
6 : Δ_root ← CompDigest^cache_{p_e,n}(∅)
7 : if Δ_root ≠ Δ_D[0];   return FALSE
8 : Δ'_root ← CompDigest^cache_{p'_e,n'}(∅)
9 : if Δ'_root ≠ Δ'_D[0];   return FALSE
10 : return TRUE
```

3. *an honest PoC or PoI update contains at most one additional sibling root per level crossed by the update, hence has size $O(\log M')$, where M' is the cached length after the update.*

Proof. (1) The 1-2 forest of M elements has at most $2\lfloor \log_2 M \rfloor + 2$ trees, each contributing one $((i,j), \Delta_{i,j}, c_{i,j})$ tuple of size $O(\lambda + \log s.n)$. (2) The local subtree is full, so if the local subtree interval is (a, b), then the path length is $\log_2(b-a)$. (3) For updates, a PoC or PoI update only needs roots for merges that cross the old/new boundary or enlarge the tree containing the proved element; at most one such merge occurs per level, giving $O(\log M')$ hashes. □

Theorem 6 (Sync versus async size tradeoff). *For a reachable state with cached count M, let poi_{s} and poi_{a} be the PoIs that $\mathcal{PMT}^h$ and $\mathcal{PAA}^h$ emit for the same active element. Then*

$$|D_{\mathcal{PAA}}| = O(\log M) \cdot |D_{\mathcal{PMT}}|, \qquad |poi_{\mathsf{a}}| \le |poi_{\mathsf{s}}|.$$

Proof. The digest comparison is immediate from the preceding two size bounds. For the PoI comparison, the synchronous PoI hashes a leaf all the way to the merged root over $[s.p_e, s.n)$, while the asynchronous PoI stops at the local subtree root. Both paths agree inside the subtree of size 2^i; the synchronous PoI additionally emits one sibling hash for each of the remaining 1-2 forest roots above the local subtree. □

Lemma 11 ($\mathcal{PAA}$ pruning work). *A prune operation in $\mathcal{PAA}^h$ removes at most $O(\log M)$ cached roots and therefore updates the digest in $O(\log M)$ root-level operations, where M is the cached length before pruning.*

Proof. By Lemma 3, before pruning there are at most two cached trees at each level. The prune procedure removes only whole trees from the left of the forest and stops as soon as the remaining leftmost tree contains at least one still-cached inactive or active element. Thus it can remove no more than the total number of cached trees, which is $O(\log M)$ for the 1–2 forest. □

Algorithm 16 $\mathcal{PMT}_k.\mathsf{UpdPoI}(poi, \langle \Delta_D \rangle_i, u) \to poi' \cup \{\bot\}$

$u.n_{\mathbf{new}} = u.n_{old}$ and $u.(p_e)_{\mathbf{new}} = u.(p_e)_{old}$; **return** poi
if $poi.pos < u.p_{\mathbf{new}}$; **return** $\bot$
$offsets \leftarrow \mathsf{ConvertToOffsets}(u.n_{old}, u.p_{old}, u.(p_e)_{old})$
find $index$ s.t. $offsets[index] \leq poi.pos < offsets[index+1]$
$poi_size_a \leftarrow \log_2(offsets[index+1] - offsets[index])$
$poi'.u \leftarrow$ **first** poi_size_a elements of $poi.u$
$offsets' \leftarrow \mathsf{ConvertToOffsets}(u.n_{\mathbf{new}}, u.p_{\mathbf{new}}, u.(p_e)_{\mathbf{new}})$
parse $u.val$ as $(\langle \Delta_{old} \rangle_i, \langle upd \rangle_i, \langle \Delta_{\mathbf{new}} \rangle_i)$
$u_a \leftarrow u$; $u_a.val \leftarrow \langle upd \rangle_i$
$poi'_a \leftarrow \mathcal{PAA}_k.\mathsf{UpdPoI}(poi', \langle \Delta_{old} \rangle_i, u_a)$
$poi' \leftarrow poi'_a$
$cache \leftarrow \{\}$
$\mathcal{PMT}_k.updates_to_cache(cache, poc)$
find $index'$ s.t. $offsets'[index] \leq poi.pos < offsets'[index+1]$
$collected \leftarrow \langle\rangle$
$collect_poi(collected, cache, offsets'[index], offsets'[index+1], u.(p_e)_{\mathbf{new}}, u.n_{\mathbf{new}})$
for $\Delta \in collected$; $poi'.u.push_back(\Delta)$
$poi'.offset \leftarrow u.(p_e)_{\mathbf{new}}$
$poi'.n \leftarrow u.n_{\mathbf{new}} - poi'.offset$
return poi'

E Postponed Evaluations

In § 4, we presented figures comparing the daily update runtime and PoI update runtime across 3 implementations of a Merkle tree accumulator. Here, we additionally provide figures comparing PoI size in digests and PoI verification runtime.

Experiment 3: PoI size. The third experiment measures the size of the PoIs. We run the same 365-day simulation, adding 40,000 elements per day and pruning any expired elements in the pruning constructions. After each day, we report the average digest length of the active elements' PoIs.

The data show that the asynchronous pruning construction consistently holds fewer digests in an average proof. This observation aligns with Thm. 6 which states that the PoI size for the synchronous construction must be no less than the asynchronous over the same active range.

Experiment 4: PoI verification runtime. The fourth experiment measures the time required for a verifier to call VerPoI. After each day's appending and pruning routine, we compute and update PoIs before calling VerPoI on each one and reporting both the average and maximum runtime.

The data show that the pruning constructions exhibit a slight improvement in average performance compared to the append-only construction, becoming dramatic after day 50. This is explained by the cached range of the pruning construction staying a fixed size at day 45 when pruning first occurs. Compare this to the append-only construction's cached

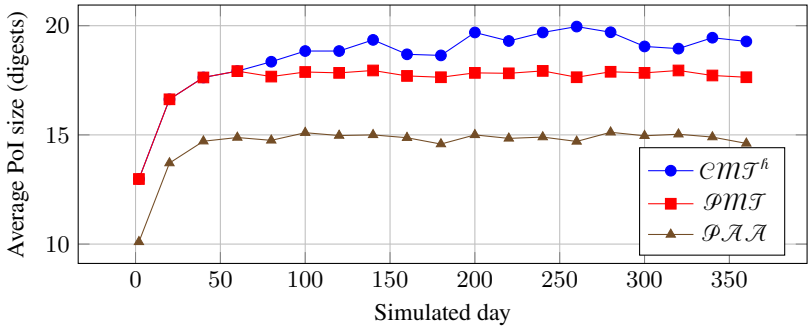
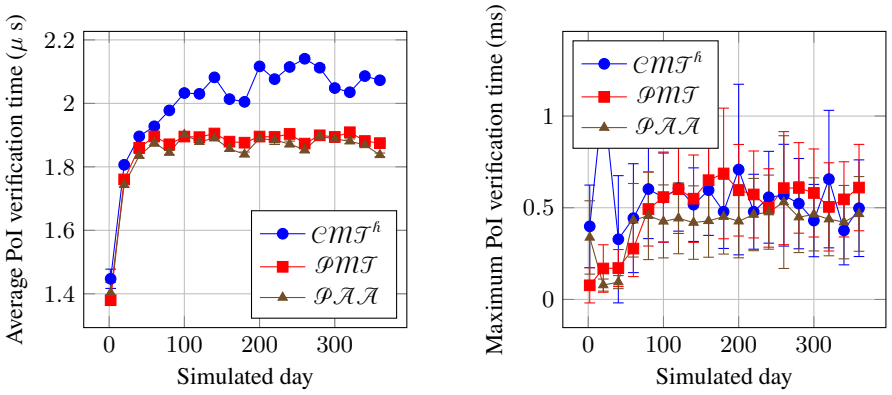


Fig. 5: Average PoI size of the active elements on a given day. 40,000 certificates were added each day, with an expiration of 45 days.



(a) Average PoI verification time.

(b) Maximum PoI verification time.

Fig. 6: Average and maximum runtime to verify a PoI of the active elements on a given day. 4,000 elements were added each day with an expiration of 45 days. In Fig. 6a, the average runtime measurement of the active elements was repeated 20 times; the average of these 20 runs is what is recorded. The same method was used to compute the maximum data in Fig. 6b, but averaging the repeated maximums instead of averages. Fig. 6a shows error bars denoting the 99% confidence interval, while Fig. 6b shows 95%. Not all intervals are large enough to be visible.

range, which perpetually grows and requires the verifier to hash more times to step from the leaf to the root.