Efficient Merkle-Tree Consistent Accumulator

Anna Mendonca[†*], Hudson Shi[†], Triet Huynh[†], Ivan Pryvalov[‡¶], Amir Herzberg[†]
[†] {*anna.gorbenko,hudson.shi,triet.huynh,amir.herzberg*}*@uconn.edu* University of Connecticut
[‡] *ivan.pryvalov@b-tu.de* Brandenburg University of Technology
[¶]University of Luxembourg

***Abstract*—A *consistent accumulator* computes a digest for a dynamically-growing set of elements, with a *proof of consistency* of the new digest with the previous digests. Consistent accumulators are in wide use, in particular, by Certificate Transparency (CT), which is part of the Web PKI, and in blockchains.**

We present a significantly more efficient design for a consistent accumulator. Our design is compatible with the CT specifications; similarly to the widely-used, open-source CT implementation, it uses a Merkle tree, but much more efficiently. We provide open source implementation, security analysis and experimental evaluation showing the performance improvements.

***Index Terms*—Merkle tree, Cryptographic accumulators, Certificate Transparency, Proof-of-inclusion, Keyed accumulators**

I. Introduction

Merkle trees and other *cryptographic accumulators* are widely used to accumulate and ensure integrity of large collections of data. A short *digest* authenticates an entire dataset and a compact *Proof-of-Inclusion* (PoI), also called a membership witness, for each data element, to allow verification that this data element belongs to the collection. Accumulators are crucial to append-only logs, blockchains, and any storage services that require a tamper-evident history with efficient verification.

In many real-world deployments, this collection of data elements grows over time. For example, in Certificate Transparency (CT) [1], a log operator must continuously add new certificate entries, and clients and monitors are able to track the log's growth, verifying *consistency* between the old and new digests. It is also desirable to be able to efficiently update previously issued Proofs-of-Inclusions (PoIs).

Another example is blockchains, where Merkle trees allow for short proofs of inclusion of a particular block in the global state. While such compact proofs enable light (stateless) clients to verify the transactions without knowing the full state, it is also desired to efficiently maintain the consistency of such proofs as the blockchain grows.

In this work, we present the Efficient Merkle-tree Consistent Accumulator ($\mathcal{CMT}^h$), an efficient construction of a consistent accumulator, based on Merkle trees. $\mathcal{CMT}^h$ can be keyless, using a keyless hash h, or keyed, using a keyed hash function h; when using a key, the key is chosen randomly and made public (non-secret). $\mathcal{CMT}^h$ is significantly more efficient than current constructions for consistent accumulators, such as the *transparency-dev* repository used by CT implementations [2], while being compatible with the CT specifications.

$\mathcal{CMT}^h$ is also compatible with other existing Merkle-tree implementations of an accumulator. It produces the same digests and the same PoIs for the same sequences of elements, so existing verifiers do not need to change their code or trust model. This design choice keeps deployment costs low, and makes the benefits directly applicable to systems already built around Merkle trees.

To achieve this high efficiency, $\mathcal{CMT}^h$ implements several functionalities and satisfies requirements that are beyond the classical definitions of cryptographic accumulators. Therefore, we define *consistent accumulator schemes* and their requirements, including definitions for correctness requirements and for security requirements, and unforgeability requirements. This allows us to prove that $\mathcal{CMT}^h$ satisfies these requirements. Our definition of consistent accumulators allows us to expose the efficiency gain of our construction. In particular, unlike previous definitions where each new element is accumulated individually, in our definition, every accumulation operation receives a sequence of elements, allowing better efficiency than adding one element at a time. Also, accumulation operations output not only proofs-of-consistency (PoCs) but also update values, which allow verifiers to efficiently update proofs-of-inclusion (PoIs), and a separate function allows efficient joining of these update values. The accumulator also supports efficient retrieval of PoCs that prove consistency between any pair of accumulation operations. These definitions of a consistent accumulator and its requirements allow the creation of other efficient designs, e.g., designs not related to the Merkle tree.

Our design is compatible with real-world systems, in particular and primarily, Certificate Transparency (CT) [3], [1], where Merkle trees are used to audit and verify large append-only logs. In such settings, clients must continuously validate updated digests and reuse existing proofs efficiently. While CT is widely adopted, $\mathcal{CMT}^h$ can be used for many other applications that require efficient accumulation with proofs of consistency, e.g., blockchains.

Contributions:

1) *Efficient Merkle-tree Consistent Accumulator*: We present $\mathcal{CMT}^h$, an efficient Merkle-tree Consistent Accumulator, constructed using a hash function h. The design is compatible with the 'classical' (non-consistent and/or non-optimized) Merkle tree implementations, making it easy to use in existing systems.
2) We provide open-source implementation of $\mathcal{CMT}^h$ in

* Corresponding author.

Go [4], and show its significant performance advantages in computation time compared to Google's open-source transparency.dev implementation [2], also implemented in Go.

3) *Definitions for Consistent Accumulators*: We define *consistent accumulator schemes* and *extended consistent accumulator schemes*, including security properties tailored to the dynamic setting. In our definitions, a consistent accumulator supports adding a *batch* of elements in one accumulation event, returns PoIs for the newly added elements, and returns a compact update value and proof of consistency for moving from the old digest to the new one. This is important in practice, since systems such as CT and blockchains naturally add many elements at once rather than one by one, and treating a batch as a first-class operation avoids unnecessary recomputation.
4) *Security Proofs*: We provide proofs showing that our design meets the security requirements of consistent accumulators.

A. High Level Overview of $\mathcal{CMT}^h$

Our design is centered around the *dyadic cover* of the Merkle tree, i.e. the minimum set of full subtrees that completely cover the full state of the log. The roots of these subtrees are what $\mathcal{CMT}^h$ stores as its state digests. As will be discussed in a later paragraph, the proofs of consistency generated by $\mathcal{CMT}^h$ are also based on the dyadic cover.

A crucial part of $\mathcal{CMT}^h$'s space savings comes from the fact its state digest set, the dyadic cover, grows $\log(n)$ with respect to the size of the tree. This fact is apparent in Fig. 1, where the dyadic cover is shown by the brown digests, with each such digest being the root of a full subtree. In the worst case, when the tree size is $n = 2^k - 1$, the size of this dyadic cover, and therefore the state digests, is $\log(n + 1)$.

$\mathcal{CMT}^h$ returns the proof of consistency (PoC), as well as an *update*, for the input sequence batch. To support maximally-compact dyadic covers for the PoC and update, we add an extended Definition 9 to support retrieving the most compact PoC and update between any two batch boundaries in the state history. The most compact PoC or update is the dyadic cover of the leaf difference, the leaf nodes that are present in the final state but not in the initial. This is useful as using our main (unextended) definition set, the PoC or update across multiple accumulations would include redundant digests because the digests are directly taken from the single batch accumulations.

$\mathcal{CMT}^h$ optimizes for space efficiency by allowing additions of batches instead of singular elements. This means only supporting PoCs between these batches, specifically, the states produced by a call to Accum, rather than for every individual element addition. This choice limits the granularity of verifiable states, but it enables the log to remain performant and scalable under high-throughput conditions.

B. Related Work

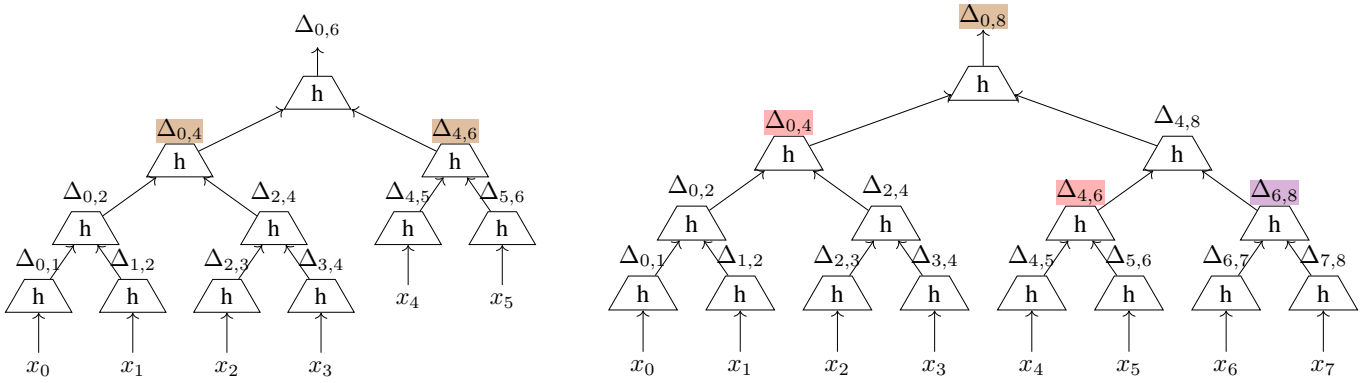
Cryptographic accumulators [5], [6] accumulate multiple data elements into a compact digest and allow compact *proofs of inclusion (PoIs)*. Accumulators are widely used in practice, most commonly, using designs based on Merkle trees [5], e.g., used in Certificate Transparency (CT) [3], [1], [7]. CT, and other applications, allow multiple accumulation events, using *proofs of consistency (PoCs)* between two digests. Informally, proofs of consistency guarantee that data elements accumulated in the older digest are also present in the newer digest. However, in spite of the extensive use of CT and other Merkle-tree system supporting multiple accumulation events, we are not aware of efforts to improve its performance, or, in fact, of analysis showing its security properties.

There are many works extending accumulators to allow for multiple accumulation events [8], [9], [10], [11], [12], [13], [14], [15]. A smaller number of works also ensure *proofs of consistency (PoCs)*. Crosby and Wallach [16] use Merkle trees as a basis for constructing *tamper evident logging*. Dowling et al. [17] use Merkle trees in append-only *secure logging schemes*, while Kales et al. [18] extend this setting for privacy-preserving retrieval. Tomescu et al. [19] introduce *append-only authenticated dictionaries* and use them for building transparency logs, allowing key-search.

CONIKS [20] uses prefix Merkle trees to implement transparent name-to-key bindings, which facilitate a key-search and both proofs of inclusion and non-inclusion. Extensions of CT to support *key transparency* have recently received attention [21], [22], [23]; such protocols allow individual updates by replacing previously accumulated elements. While offering additional functionality and supporting different system models, these data structures require at least a linear state.

We believe no prior Merkle-tree design allowed multiple accumulations with a *compact state*, or provided (efficient) *proofs of consistency (PoCs)*. The asynchronous accumulator by Reyzin and Yakoubov [24], where the proof of inclusion can be valid after a certain number of modifications to the accumulator (update frequency is logarithmic in the number of elements), has a compact state, however no proofs of consistency. The closest work to ours is that of Crosby and Wallach [16]. In both [16] and our work, standard Merkle trees are used as a basis and essential nodes are used for proving updated membership and consistency. However, our design is much more efficient in storage and computations (e.g., $O(\log n)$ vs. $O(n)$), in addition, it can return an update value between any two states, thereby reducing the overall verification complexity for users. We refer to Table I for a comparison of our work to related work.

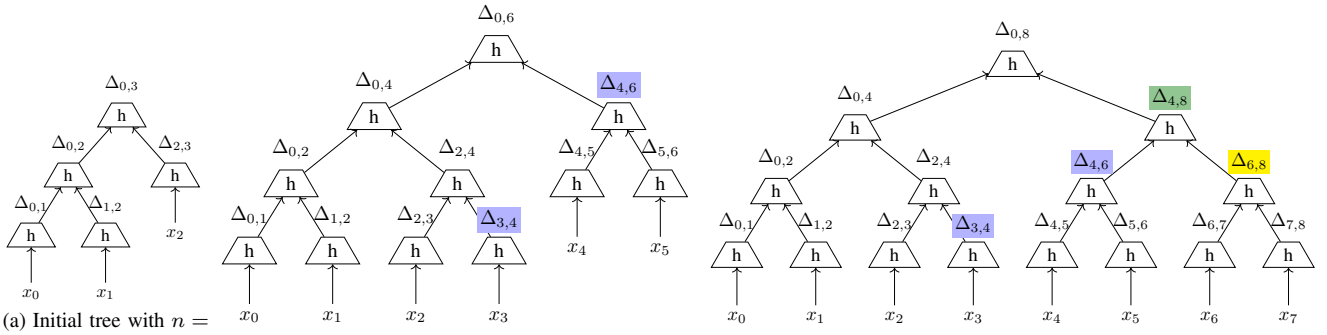
Our technique is similar to *Merkle Mountain Ranges* [26] used by Reyzin and Yakoubov [24] to describe the accumulator by splitting a single tree into multiple full binary subtrees, whose size is a power of two. When adding a new element, two full subtrees of equal size are merged into a new subtree by connecting their root nodes. While we use the roots of the subtrees to describe the state of the accumulator, our design requires only a single root node as digest, allowing it to be compatible with other Merkle tree designs. The idea of caching the roots of full subtrees to avoid unnecessary recomputations was suggested in [16]; such nodes are called "frozen". Our consistency proofs between two accumulator states follow [16],



(a) Before addition of x_6 and x_7.

(b) After addition of x_6 and x_7.

Fig. 1: An Efficient Merkle-tree Consistent Accumulator ($\mathcal{CMT}^h$) accumulator update adding elements x_6 and x_7. The state's digest set transitions from $\{\Delta_{0,4}, \Delta_{4,6}\}$ (brown in Fig. 1a) to $\{\Delta_{0,8}\}$ (Fig. 1b). The old state digests are shown in red, and the update value u (which also serves as the PoC) is $\Delta_{6,8}$ (violet).



(a) Initial tree with $n = 3$.

(b) After addition of $x_3..x_5$.

(c) After addition of x_6 and x_7.

Fig. 2: An Efficient Merkle-tree Consistent Accumulator ($\mathcal{CMT}^h$) update adding a batch of 3 elements x_3, x_4, x_5, followed by another batch of 2 elements x_6, x_7. For 2b and 2c, their corresponding update or proof-of-consistency nodes from the previous state are highlighted in blue ($\{\Delta_{3,4}, \Delta_{4,6}\}$) and yellow ($\{\Delta_{6,8}\}$), respectively. These nodes are added to $s.UM$, as well as $\Delta_{4,8}$ (shown in green) to hold the invariant that a parent must exist in $s.UM$ if both children also are present. As a result, a client can request any of the three possible PoCs $\{(a) \to (b), (b) \to (c), (a) \to (c)\}$. The batch addition $(a) \to (b)$ corresponds to the PoC digests $\{\Delta_{3,4}, \Delta_{4,6}\}$ (blue, blue); $(a) \to (c)$ corresponds to $\{\Delta_{3,4}, \Delta_{4,8}\}$ (blue, green); $(b) \to (c)$ corresponds to $\{\Delta_{6,8}\}$ (yellow).

Work	**State**	**PoI**	**PoC**	**Accum** (runtime, digest)	**Update PoI**	**Trustless setup**
[16], [17], [25]	n	$\log n$	$\log n$	$\log n$, digest 1	*not supported*	**Yes**
[19]	λn	$\log n$	$\log n$	$\lambda \log^3 n$, digest 1	*not supported*	No
$\mathcal{CMT}^h$	$\mathbf{\log n}$	$\log n$	$\log n$	$\log n$, digest 1	$\boldsymbol{m + \log n}$	**Yes**

TABLE I: Comparison of our Merkle-tree consistent accumulator $\mathcal{CMT}^h$ with related publications; better values in bold. All numbers are in $O(\cdot)$ notation; n is the number of elements in tree, λ the security parameter. *Accum* numbers are given for accumulating a single element, which is supported by all constructions. All publications feature a Proof-of-Inclusion (PoI) and Proof-of-Consistency (PoC). *Update PoI* refers to updating an outdated Proof-of-Inclusion to be compatible with a newer tree having m more elements. All works use a Merkle tree-like design, except for [19], which is based on bilinear maps.

[17], adapted to require only *small state*.

The dynamic addition of *multiple* elements is known as *batched updates*, e.g., in the context of blockchains [27], [28]. Here, the batched proof aggregates multiple individual proofs of inclusion. Such aggregated proofs are more compact than individual ones, due to the tree structure and the fact that neighboring elements may share intermediate nodes [27]. In our design, we mainly focus on reducing the *server* state when adding multiple elements at once and techniques such as [27] are orthogonal to our approach. In the context of definitions for *accumulators* and their security properties, the functionality of adding multiple elements is defined for a single element [8], [24], [13], [15], [18], thus we start by formalizing a system which allows us to define, instantiate and analyze a consistent accumulator that supports efficient batched additions. We extend the definitions in [24], to create a unified formal model that allows for *batched* accumulation of new elements. In particular, this model suits real implemented systems, for example, in both CT logs and blockchains.

Related to accumulators notions are polynomial commit-

ments [29] and vector commitments (VCs) [30]. In the former notion, a single value represents a commitment to a polynomial and allows proofs of correct polynomial evaluations. The latter is a fixed-length primitive, a more generic than Merkle Trees, allowing constant size proofs of membership and efficient update of committed elements. Aggregate proofs are possible as well (e.g., [31]). In our work, we diverge from VCs by considering the append-only data structure with the consistency property, ensuring the correct accumulation of new elements.

Specific to the blockchain applications, Coded Merkle Trees [32] address the data availability problem. Their essential part is the ability to generate the so-called fraud proofs w.r.t. the ledger state and the semantic correctness of block transactions. This is beyond the scope of our paper.

II. Consistent Accumulators

Notations. We use $\langle X_i \rangle_{i=1}^{l}$ to 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 $x_j \in \{0,1\}^*$. The union operation, when applied to two or more sequences, outputs the (unordered) set containing the elements that appear in any of the sequences, i.e.,

$$\bigcup \langle X_i \rangle_{i=1}^{l} \equiv X_1 \cup \ldots \cup X_l \equiv \{x \in X_i \text{ for some } i \text{ s.t. } 1 \leq i \leq l\}$$

To refer to a particular output parameter of a function of the accumulator, we append the parameter name to the function name, separated by a dot. For example, let algorithm Alg take as input par and output (a, b), namely $(a, b) \leftarrow \mathsf{Alg}(par)$. Then $\mathsf{Alg}(par).a$ denotes the first element of the output pair, $\mathsf{Alg}(par).b$ the second element.

Accumulators compute the digest of a set of elements; the digest can be used to ensure the integrity of the set, and the fact that a particular element was included in the set resulting in the given digest. We define *Consistent accumulators*, i.e., (stateful) accumulators that support multiple accumulation events, in each of which, a set of elements is added to the elements accumulated so far. Consistent accumulators provide *Proof of Consistency* (PoC), allowing validation that a new digest is the result of accumulating additional elements to the set of elements accumulated into a previous digest. This can be important for different applications; e.g., a specific consistent accumulator, based on the *Merkle Tree*, is used by the *Certificate Transparency (CT)* standards [25], [1].

To key or not to key. Applications such as *CT* usually use keyless accumulators. However, we present keyed definitions and constructions, using keyed hash functions and a random string as a key; this is since keyless accumulators, like keyless hash functions, cannot ensure collision resistance (under the standard security model). When using a keyless hash, our constructions are secure under the Random Oracle Model (ROM).

a) Why new definitions: In contrast to previous definitions where new elements had to be accumulated one at a time [8], [24], [13], [15], [18], the consistent accumulator accumulates, each time, a whole set of strings. It also includes functions for updating PoIs (proofs of inclusion) and for generating and validating updates and PoCs (proofs of consistency) between the digests and PoIs produced in arbitrary accumulations. The support for accumulating sets of strings in each accumulation, and the new functions, allow the consistent accumulator to provide significantly more efficient accumulation service to applications.

To support accumulation of multiple sets of strings (in multiple accumulation events), the accumulation operations are done by a *stateful manager entity*. The verification operations are still stateless, and include verification algorithms such as VerPoI, VerUpd, and VerPoC. The last two algorithms are used to check the validity of PoC and update values, respectively.

Definition 1 (Consistent accumulator schemes)**.** A *consistent accumulator* $\alpha = (\mathsf{Accum}, \mathsf{VerPoI}, \mathsf{VerUpd}, \mathsf{VerPoC}, \mathsf{UpdPoI})$ is a collection of PPT (Probabilistic Polynomial Time) algorithms, where, for every integer λ, key $k \in \{0,1\}^\lambda$, binary strings $s, p, \Delta, \Delta', x, u$ and sequences of binary strings X, X', we have $\alpha.\mathsf{Accum}(X||X', \emptyset) = \alpha.\mathsf{Accum}(X', s)$ where $s = \alpha.\mathsf{Accum}(X, \emptyset).s$, and:

- $\alpha_k.\mathsf{Accum}(X, s) \rightarrow (s', \Delta, PoI, u, PoC)$**:** given the current accumulator state[1] s and a sequence of strings $X = \langle x_i \rangle_{i=0}^{n-1} \in (\{0,1\}^*)^n$, outputs an updated state s', a digest $\Delta \in \{0,1\}^\lambda \cup \{\perp\}$, a set of proofs-of-inclusion (PoIs) PoI, an update value u and a proof of consistency PoC.
- $\alpha_k.\mathsf{VerPoI}(\Delta, x, p) \rightarrow \{\textbf{TRUE or FALSE}\}$**:** given a digest Δ, an element x, and its corresponding proof-of-inclusion p, where $0 \leq i \leq |PoI|$, return TRUE if the proof is valid (i.e., x was included in the set represented by Δ), and FALSE otherwise.
- $\alpha_k.\mathsf{VerUpd}(\Delta, u, \Delta') \rightarrow \{\textbf{TRUE or FALSE}\}$**:** given an initial digest Δ, an update value u, and a new digest Δ', return TRUE if u is a valid update value from digest Δ' to Δ, and FALSE otherwise.
- $\alpha_k.\mathsf{VerPoC}(\Delta, p, \Delta') \rightarrow \{\textbf{TRUE or FALSE}\}$**:** given an initial digest Δ, a proof of consistency p, and a new digest Δ', return TRUE if Δ' is consistent with Δ, and FALSE otherwise.
- $\alpha_k.\mathsf{UpdPoI}(p, u) \rightarrow p'$**:** given a current proof-of-inclusion p for input string x, and an update value u, this algorithm returns an updated p' for x for use with the new digest Δ'.

We now present the requirements for consistent accumulators.

A. Correctness Requirements: definitions

Intuitively, a consistent accumulator ensures *PoI correctness* if for every string x accumulated into a digest Δ, there exist a corresponding Proof of Inclusion (PoI) such that $\mathsf{VerPoI}(\Delta, x, p) = \text{TRUE}$.

Definition 2 (PoI correctness)**.** We say that a consistent (keyed) accumulator α scheme ensures *Proof-of-Inclusion (PoI)*

[1] In the first call, we set the state s to the empty string ϵ.

Correctness if the following holds for every binary string key k, every binary string state s, and any finite sequence of binary string X:

$$\left(\begin{array}{c}\Delta \leftarrow \alpha_k.\mathsf{Accum}(X,s).\Delta \wedge \\ PoI \leftarrow \alpha_k.\mathsf{Accum}(X,s).PoI\end{array}\right) \Rightarrow \left(\begin{array}{c}(\forall x \in X)(\exists p \in PoI) \\ \alpha_k.\mathsf{VerPoI}(\Delta,x,p) = \textsc{True}\end{array}\right)$$

Similarly, a consistent accumulator ensures *PoC and Update value correctness* if for any two digests Δ and Δ', where Δ was the result of an accumulation that happened before Δ', there must exist an update value u and a Proof of Consistency (PoC) such that $\mathsf{VerUpd}(\Delta,u,\Delta') = \mathsf{VerPoC}(\Delta,PoC,\Delta') = \textsc{True}$.

Definition 3 (PoC and Update value correctness)**.** We say that a consistent (keyed) accumulator α ensures PoC and Update value correctness if the following holds for every binary string key k, every initial binary string state s,and any finite sequence of binary strings: X and X':

$$\left(\begin{array}{c}\Delta \leftarrow \alpha_k.\mathsf{Accum}(X,s_0).\Delta \wedge \\ s_1 \leftarrow \alpha_k.\mathsf{Accum}(X,s_0).s \wedge \\ \Delta' \leftarrow \alpha_k.\mathsf{Accum}(X',s_1).\Delta \wedge \\ PoC \leftarrow \alpha_k.\mathsf{Accum}(X',s_1).PoC \wedge \\ u \leftarrow \alpha_k.\mathsf{Accum}(X',s_1).u\end{array}\right) \Rightarrow \left(\begin{array}{c}\alpha_k.\mathsf{VerPoC}(\Delta,PoC,\Delta') = \textsc{True} \wedge \\ \alpha_k.\mathsf{VerUpd}(\Delta,u,\Delta') = \textsc{True}.\end{array}\right)$$

Let u be the result of GetU or Accum from digest Δ to Δ'. A consistent accumulator ensures *updating PoI correctness* if, when updating a PoI p for string x to p' using a valid update value u, the new PoI for x would be valid, i.e., $\mathsf{VerPoI}(\Delta',x,p') = \textsc{True}$.

Definition 4 (Updating PoI Correctness)**.** We say that a consistent (keyed) accumulator α ensures PoI Update Correctness if the following holds for every binary string key k, every initial binary string state s_0, and any finite sequence of binary strings X and X':

$$\left(\begin{array}{c}s_1 \leftarrow \alpha_k.\mathsf{Accum}(X,s_0).s \wedge \\ PoI \leftarrow \alpha_k.\mathsf{Accum}(X,s_0).PoI \wedge \\ \Delta' \leftarrow \alpha_k.\mathsf{Accum}(X',s_1).\Delta \wedge \\ u' \leftarrow \alpha_k.\mathsf{Accum}(X',s_1).u\end{array}\right) \Rightarrow \left(\begin{array}{c}(\forall x \in X)(\exists p \in PoI) \\ p' \leftarrow \alpha_k.\mathsf{UpdPoI}(p,u') \wedge \\ \alpha_k.\mathsf{VerPoI}(\Delta',x,p') = \textsc{True}\end{array}\right)$$

B. Unforgeability requirements

Next, we define three unforgeability requirements: *PoI unforgeability*, *update unforgeability* and *PoC unforgeability*. Note also that an accumulator is collision-resistant if for any PPT adversary the probability of finding two distinct sets of elements that produce the same digest is negligible. PoI correctness together with PoI unforgeability implies accumulator ACR (see Section II-C).

PoI unforgeability can be understood as the following. Consider any string x that was never the input to the accumulation that generated Δ. Then the adversary cannot produce a proof of inclusion p that will verify as if x was one of the input strings for the accumulation operations that produces Δ, i.e. $\alpha_k.\mathsf{VerPoI}(\Delta,x,p) = \textsc{True}$.

Definition 5 (PoI Unforgeability)**.** Let α be a consistent accumulator scheme with key k. We say that α ensures *Consistent Accumulator Proof-of-Inclusion (PoI) Unforgeability* if for every PPT adversary $\mathcal{A}$, the following holds:

$$\Pr_{k \xleftarrow{\$} \{0,1\}^\lambda}\left[\begin{array}{c}\alpha_k.\mathsf{VerPoI}(\Delta,x,p) = \textsc{True} \wedge \\ \forall x \notin \bigcup(\xi), \text{ where} \\ \xi,x,p, \text{and } \Delta \text{ were generated by:} \\ \left\{\begin{array}{c}\xi \leftarrow \mathcal{A}(k); \\ \Delta \leftarrow \alpha_k(\xi,\epsilon).\Delta; \\ (x,p) \leftarrow \mathcal{A}(\xi,\alpha_k(\xi,\epsilon),k)\end{array}\right\}\end{array}\right] \in Negl(\lambda)$$

In the following definitions, we use the shorthand $\xi \equiv \langle X_j \rangle_j^l$ to denote a sequence of l sequences of binary strings, each $X_j \equiv \langle x_i \rangle_{i=0}^{m-1}$, as defined in Section II. In addition, the symbol ϵ is used to denote the empty string, which is also the initial state.

Update unforgeability requires that once a PoI is valid, applying any valid update value preserves validity; an attacker cannot use a valid update value to invalidate an existing PoI. If proof-of-inclusion p shows that string x was accumulated into digest Δ_0, and u is a valid update from Δ_0 to Δ_1, then the updated proof $p' \leftarrow \alpha_k.\mathsf{UpdPoI}(p,u)$ must also shows that x was accumulated into Δ_1.

Definition 6 (Update unforgeability)**.** A (keyed) consistent accumulator scheme α ensures *Consistent Accumulator Update value u unforgeability* if for every PPT adversary $\mathcal{A}$, the following holds:

$$\Pr_{k \xleftarrow{\$} \{0,1\}^\lambda}\left[\begin{array}{c}(\Delta_0,\Delta_1,u,x,p) \leftarrow \mathcal{A}(k) \text{ where:} \\ \alpha_k.\mathsf{VerUpd}(\Delta_0,u,\Delta_1) = \textsc{True} \wedge \\ \alpha_k.\mathsf{VerPoI}(\Delta_0,x,p) = \textsc{True} \wedge \\ p' = \alpha_k.\mathsf{UpdPoI}\,(p,u) \wedge \\ \alpha_k.\mathsf{VerPoI}(\Delta_1,x,p') = \textsc{False}\end{array}\right] \in Negl(\lambda)$$

Proof-of-Consistency (PoC) unforgeability requires that an adversary cannot produce a valid PoC showing that a new digest Digest_1 is consistent with a previous digest Digest_0, given there is some string included in Digest_0 but not in Digest_1. For a given state s and two sets $X_0 \not\subseteq X_1$, denote their corresponding digests as: $\Delta_0 \leftarrow \alpha_k.\mathsf{Accum}(X_0,s).\Delta$ and $\Delta_1 \leftarrow \alpha_k.\mathsf{Accum}(X_1,s).\Delta$. The consistent accumulator α ensures PoC unforgeability, if the adversary cannot find such s, X_0, X_1 and a valid *proof of consistency (PoC)* p from Δ_0 to Δ_1, namely, $\alpha_k.\mathsf{VerPoC}(\Delta_0,p,\Delta_1) = \textsc{True}$.

Definition 7 (Proof-of-Consistency (PoC) unforgeability)**.** We say that the (keyed) accumulator scheme α ensures Proof-of-Consistency (PoC) and update value Unforgeability if for

every PPT adversary $\mathcal{A}$ interacting with the accumulator α, the following holds:

$$\Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} \left[\begin{array}{c} (X, X', PoC, u) \leftarrow \mathcal{A}(k) \text{ where:} \\ (X \not\subseteq X') \wedge (|X| < |X'|) \wedge \\ \Delta \leftarrow \alpha_k.\mathsf{Accum}(X, s).\Delta \wedge \\ \Delta' \leftarrow \alpha_k.\mathsf{Accum}(X', s).\Delta \wedge \\ \alpha_k.\mathsf{VerPoC}(\Delta, PoC, \Delta') = \textsc{True} \end{array} \right] \in Negl(\lambda)$$

C. Collision resistance requirements

Two notions of collision resistance exist, target-collision resistance and any-collision resistance, as defined in [33]. We define and focus on the stronger notion, specifically *any-collision-resistant* (ACR) accumulators.

In an ACR accumulator, the attacker is given a uniformly random key k and can then produce two arbitrary sequences of inputs to the accumulation operation, i.e., two sequences of sequences ξ, ξ'. The attacker 'wins' if the strings in ξ differ from the strings in ξ', yet the digests of the ξ and ξ' are identical. The accumulator is ACR if this occurs with (at most) negligible probability as a function of λ, which is the key length $|k| = \lambda$.

Definition 8 (Consistent Accumulator Any-Collision-Resistant)**.** We say that the (keyed) consistent accumulator scheme α is *Any-Collision-Resistant (ACR)* if for every efficient PPT algorithm $\mathcal{A}$ the following holds:

$$\varepsilon^{ACR}_{\alpha,\mathcal{A}}(\lambda) \equiv \Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} \left[\begin{array}{c} (\xi, \xi') \leftarrow \mathcal{A}(k) \text{ where:} \\ \alpha_k(\xi, \emptyset).\Delta = \alpha_k(\xi', \emptyset).\Delta \neq \perp \wedge \\ \bigcup \xi \neq \bigcup \xi' \end{array} \right] \in Negl(\lambda)$$

PoI correctness together with PoI unforgeability implies ACR of the scheme.

Lemma 1 (PoI correctness plus unforgeability implies ACR)**.** If a (keyed) consistent accumulator scheme α satisfies PoI correctness (Def. 2) and PoI unforgeability (Def. 5), then it satisfies ACR (Def. 8).

Proof. Assume to the contrary that α is not ACR, i.e., there exists a PPT adversary A_{ACR} such that with non-negligible probability over $k \leftarrow \{0,1\}^\lambda$, $A_{\mathsf{ACR}}(k)$ outputs (ξ, ξ') such that

$$\Delta \equiv \alpha_k(\xi, \emptyset).\Delta = \alpha_k(\xi', \emptyset).\Delta \neq \perp$$

and

$$\bigcup(\xi) \neq \bigcup(\xi').$$

Without loss of generality, assume $\exists x \in \bigcup(\xi)$ s.t. $x \notin \bigcup(\xi')$.

We now build a PPT adversary A_{PoI} against Def. 5. On input k, adversary A_{PoI} runs $A_{\mathsf{ACR}}(k)$ and obtains (ξ, ξ') as above. Then A_{PoI} computes

$$(s', \Delta, PoI, u, PoC) \leftarrow \alpha_k.\mathsf{Accum}(\xi, \emptyset).$$

Since $x \in \bigcup(\xi)$, PoI correctness (Def. 2) implies that there exists $p \in PoI$ such that

$$\alpha_k.\mathsf{VerPoI}(\Delta, x, p) = \textsc{True}.$$

Now observe that the same digest Δ is also obtained from ξ', i.e.,

$$\Delta = \alpha_k(\xi', \emptyset).\Delta.$$

Recall that $x \notin \bigcup(\xi')$. Hence the pair (x, p) is a successful forgery against Def. 5: it verifies under a digest corresponding to an accumulation in which x was not accumulated. Therefore, $\varepsilon^{ACR}_{\alpha,\mathcal{A}_{\mathsf{ACR}}}(\lambda) = \varepsilon^{PoI}_{\alpha,\mathcal{A}_{\mathsf{PoI}}}(\lambda)$.

□

D. Extended Consistent Accumulator Schemes

Every consistent accumulator must provide the basic functionality in Def. 1: add elements, verify PoIs, verify updates, verify consistency, and update PoIs. This baseline is enough to capture correctness and security, and it keeps the abstraction clean. In practice, however, many applications benefit from helper procedures that avoid redundant work across extensions. The extended variant in Def. 9 formalizes this with the algorithms such as GetU and GetPoC, which let verifiers and monitors compute update values and consistency proofs directly, and combine multiple updates into a single object.

Definition 9 (Extended consistent accumulator schemes)**.** An *extended consistent accumulator* extends Def. 1 with the following PPT algorithms:

- $\alpha_k.\mathsf{GetU}(ctr_0, ctr_1, s) \rightarrow u$**:** computes an update value that transforms PoIs valid after ctr_0 accumulations into PoIs valid after $ctr_1 > ctr_0$ accumulations, relative to the public state s.
- $\alpha_k.\mathsf{GetPoC}(ctr_0, ctr_1, s) \rightarrow PoC$**:** computes a proof of consistency between the digest after ctr_0 accumulations and the digest after $ctr_1 > ctr_0$ accumulations, relative to s.

E. Extended Consistent Accumulator Correctness Requirements:

Definition 10 (GetU correctness)**.** We say that a consistent (keyed) accumulator α ensures GetU *correctness* if the following holds for every binary string key k, every initial binary string state s_0, and any finite sequences of binary strings X and sequence of sequences ξ:

$$\left(\begin{array}{c} s_1 \leftarrow \alpha_k.\mathsf{Accum}(X, s_0).s \wedge \\ \Delta_{ctr_0} \leftarrow \alpha_k.\mathsf{Accum}(X, s_0).\Delta \wedge \\ s_2 \leftarrow \alpha_k.\mathsf{Accum}(\xi, s_1).s \wedge \\ \Delta_{ctr_1} \leftarrow \alpha_k.\mathsf{Accum}(\xi, s_1).\Delta \wedge \\ ctr_0 \leftarrow s_1.n \ \wedge \ ctr_1 \leftarrow s_2.n \ \wedge \ ctr_0 < ctr_1 \wedge \\ u \leftarrow \alpha_k.\mathsf{GetU}(ctr_0, ctr_1, s_1) \end{array} \right) \Rightarrow \left(\begin{array}{c} \alpha_k.\mathsf{VerUpd}(\Delta_{ctr_0}, u, \Delta_{ctr_1}) = \textsc{True} \wedge \\ \big(\forall x, p: \ \alpha_k.\mathsf{VerPoI}(\Delta_{ctr_0}, x, p) = \textsc{True} \wedge \\ p' \leftarrow \alpha_k.\mathsf{UpdPoI}(p, u) \\ \wedge \alpha_k.\mathsf{VerPoI}(\Delta_{ctr_1}, x, p') = \textsc{True}\big) \end{array} \right)$$

Definition 11 (GetPoC correctness)**.** We say that a consistent (keyed) accumulator α ensures GetPoC *correctness* if the

following holds for every binary string key k, every initial binary string state s_0, and any finite sequences of binary strings X and sequence of sequences ξ:

$$\left(\begin{array}{c} s_1 \leftarrow \alpha_k.\mathsf{Accum}(X, s_0).s \wedge \\ \Delta_{ctr_0} \leftarrow \alpha_k.\mathsf{Accum}(X, s_0).\Delta \wedge \\ s_2 \leftarrow \alpha_k.\mathsf{Accum}(\xi, s_1).s \wedge \\ \Delta_{ctr_1} \leftarrow \alpha_k.\mathsf{Accum}(\xi, s_1).\Delta \wedge \\ ctr_0 \leftarrow s_1.n \ \wedge \ ctr_1 \leftarrow s_2.n \ \wedge \ ctr_0 < ctr_1 \wedge \\ PoC \leftarrow \alpha_k.\mathsf{GetPoC}(ctr_0, ctr_1, s_1) \end{array}\right) \Rightarrow (\alpha_k.\mathsf{VerPoC}(\Delta_{ctr_0}, PoC, \Delta_{ctr_1}) = \textsc{True}) .$$

III. $\mathcal{CMT}$: Efficient Merkle-tree Consistent Accumulator

A. The $\mathcal{CMT}^h$ construction

In this section, we present $\mathcal{CMT}^h$, an efficient consistent accumulator construction based on the Merkle tree. Here, the superscript h denotes the underlying keyed hash function used in the construction, which we assume to be any-collision resistant (ACR). $\mathcal{CMT}^h$ maintains the same digest and PoI format as of a Merkle tree accumulator, for example, the Merkle tree implemented in the *Transparency.dev* library [2], also written in GoLang, used by Google's Certificate Transparency PKI system.

However, in contrast to *Transparency.dev*, $\mathcal{CMT}^h$ supports dynamic updates with compact recomputation, by caching the roots of previously computed subtrees and reusing them when possible. The result is an efficient accumulator whose PoI length and number of hashes required for verification is only $O(\log(n))$, where n is the (total) number of accumulated elements, and whose update operations require only $O(|X_{l+1}|+\log(n))$ hash operations when adding a new sequence X_{l+1}.

In our construction, the Proof of Consistency (PoC) and the update value (u) are identical, making them interchangeable. For example, the UpdPoI function can accept either a u or a PoC as its input. Although conceptually distinct, this identity means they share the same logic for generation (Accum, GetPoC, GetU) and verification algorithms (VerPoC, VerUpd).

$\mathcal{CMT}^h$ is an efficient consistent accumulator, whose state s is defined as a tuple of four components (n, DM, UM, ctr):

- $s.n$: The total number of strings accumulated.
- $s.DM$: A map from an index pair (i, j) to the corresponding digest $\Delta_{i,j}$. We write entries as $(i, j) \mapsto \Delta_{i,j}$ and use $\Delta_{i,j} \leftarrow s.DM[(i, j)]$ for lookups.
- s.ctr: The number of accumulation events that have occurred.
- $s.UM$: A map used for efficient generation of update values, which is detailed in Alg. 5.
- $s.stateHist$: A list of references to previous states, used to retrieve the starting states for updates and PoCs.

In the event that we receive ϵ as the state, we initialize the state with $s.n = s.\text{ctr} = 0$ and $s.DM = s.UM = \emptyset$.

In the construction, the state s essentially acts as a cache, storing the root digests of all maximal full subtrees within the accumulator. This structure is the key to efficiency, as it allows for incremental updates without needing to re-access all historical elements. When new strings are added, $\mathcal{CMT}^h$ simply computes the new digests and updates the cache with the new set of maximal subtree roots.

For example, Fig. 1a shows the state after accumulating six elements $(x_0, \ldots, x_5)$. The cache holds the two digests corresponding to the maximal subtrees: $\{\Delta_{0,4}, \Delta_{4,6}\}$. After adding $\{x_6, x_7\}$, the structure forms a single maximal full subtree, and the new state becomes just $\{\Delta_{0,8}\}$, as shown in Fig. 1b. In our implementation, this cache is populated by the $\mathsf{CompCache}^s_{i,j}(X)$ algorithm. We now proceed with describing the pseudocode algorithms in the $\mathcal{CMT}^h$ design.

a) Accumulating element sequences (Alg. 1): The Accum algorithm uses four functions to compute its output: CompUDM, ComPoIAll, CompCache, and CompDigest.

In every call to Accum from s_i to s_{i+1}, we update the global $s.UM$ with the nodes needed for update/PoC between s_i and s_{i+1}. We also hold the invariant that if there exists two sibling nodes in $s.UM$, their parent must also exist in $s.UM$ Holding this invariant ensures that GetPoC is efficient and does not need to compoute new digests. Merging siblings is performed by MergeUpdDig, which computes the parent of sibling digests and recurses on the new parent.

Algorithm 1 $\mathcal{CMT}^h_k.\mathsf{Accum}(X, s) \rightarrow (s', \Delta, PoI, u, PoC)$

```
1 : if s = ε
2 :    s.n = 0
3 :    s.DM = ∅
4 :    s.UM = ∅
5 :    s.stateHist = ∅
6 : s'.n = s.n + |X|   / new total number of elements accumulated
7 : u ← (s, CompUDM^s_{s'.n}(X), s'.n)
8 : if s.n > 0
9 :    s'.stateHist ← s.stateHist.append({s.n, s.DM})
10 :   s'.UM = s.UM
11 :   for each((i, j) ↦ Δ_u) ∈ u.DM
12 :      s.UM ← s.UM ∪ {(i, j) ↦ Δ_u}
13 :      s.UM ← MergeUpdDig(s', (i, j) ↦ Δ_u)
14 :   / modifies UM to make it s_temp.DM = u.DM ∪ s.DM
15 : s_temp.n = s'.n
16 : s'.DM ← CompCache^{s_temp}_{0,s'.n}(∅)
17 :   / update state using old state plus u.DM
18 : s'.Δ ← CompDigest^{s'}_{0,s'.n}(∅)
19 :   / compute root digest using elements in s'
20 : PoI ← ComPoIAll^s_{0, s.n}(X)
21 :   / PoI of new elements, for each new element in X
22 : return (s', s'.Δ, PoI, u, u)
23 :   / PoC is the same as update value u in our construction.
```

b) Computing digests (Alg. 2): The CompDigest algorithm recursively computes the digest value of a given root node.

To compute the digest of a given range, CompDigest recursively hashes the concatenation of its left and right children's digests. It determines the split point between these children using the $lP_2(d)$ function , which finds the largest power of two less than or equal to the range size, d. This value, m, defines the midpoint between two child subtrees.

For example, to compute the root digest $\Delta_{0,6}$ in Fig. 1a, the algorithm first finds the midpoint $m = lP_2(6) = 4$. This splits the task into two sub-problems: computing the digest for the left subtree on indices $[0, 4)$, which yields $\Delta_{0,4}$, and the right subtree on indices $[4, 6)$, which yields $\Delta_{4,6}$. The final result, $\Delta_{0,6}$, is the hash of these two sub-trees' digests.

Algorithm 2 $\mathsf{CompDigest}^s_{i,j}(X) \rightarrow \Delta$

```
/ inputs should satisfy i < j ≤ s.n
if ∃Δ such that s.DM[(i, j)] = Δ

    return Δ   / digest (Δ) of element was previously computed
else
if j − i = 1
    return h_k(0x00||X[i − s.n])   / hash of single element
else
    m ← largest power of 2 ≤ j − i
    return h_k (0x01||CompDigest^s_{i,i+m}(X)
                    ||CompDigest^s_{i+m,j}(X))
      / digest of a sub-tree containing a left and right sub-trees
```

c) Populating the state digests (Alg. 3): For each batch added, CompCache computes $s'.DM$, the minimum set of left-aligned full subtrees that span every leaf in the new state s'. CompCache achieves this by recursively calling CompDigest on root nodes of smaller subtrees. We avoid recomputing digests at this point by first computing the update value u and using this value to update the old state by creating a temporary cache s_{temp}, which contains digests in the old state and update values computed from CompUDM.

Algorithm 3 $\mathsf{CompCache}^s_{i,j}(X) \rightarrow DM$

```
if i ≥ j
    return ∅
else
    m ← largest power of 2 ≤ j − i
    return {(i, i + m) ↦ CompDigest^s_{i,i+m}(X)}
            ||CompCache^s_{i+m,j}(X)
      / add the digest of the 2^log2(m) elements to the DM,
      / and contact the digests of the full trees from i + m to j − 1
```

d) Computing update digests (Alg. 4): CompUDM computes the dyadic cover, i.e., the minimal set of full subtrees with leaves that span the range from the old state to the new state. The set of root digests of all these ranges is computed with calls to CompDigest and used later in Accum to efficiently compute the new state's root as well as to update PoI in the future.

e) Optimizing Update Storage (Alg. 5): The function MergeUpdDig computes the update map $s.UM$ in a compact, canonical form by merging together adjacent dyadic ranges into their parent node whenever both siblings are present. The parent is inserted into $s.UM$ and the procedure continues the recursion. If one sibling node is not present, the process stops. This guarantees the storage of maximal full subtrees.

f) Retrieving updates and PoCs (Alg. 6): When a client calls $\mathsf{GetU}(\mathsf{ctr}_0, \mathsf{ctr}_1, s)$ or $\mathsf{GetPoC}(\mathsf{ctr}_0, \mathsf{ctr}_1, s)$ (Alg. 6), they are requesting a joined update *from* the tree after ctr_0 calls of Accum *to* the tree after ctr_1 calls of Accum. This algorithm computes the locations of the dyadic cover nodes of the leaves between states ctr_0 and ctr_1. Then it fetches the digests from $s.UM$, which must exist because of the procedure outlined in Accum and MergeUpdDig.

Algorithm 4 $\mathsf{CompUDM}^s_n(X) \rightarrow \{(i, j) \mapsto \Delta\}$

```
DM ← ∅
i ← s.n
while i < n
    m ← the largest power of 2 which divides i
      / compute the largest possible subtree (i, i + m) with a leftmost leaf at i
    if m > n − i
        m ← largest power of 2 ≤ j − i
    / if we accumulated less than m items,
    / then compute largest perfect subtree that fits
    DM ← DM ∪ {(i, i + m) ↦ CompDigest^s_{i,i+m}(X)}
    i ← i + m
return DM
```

Algorithm 5 $\mathsf{MergeUpdDig}(s, (i, j) \mapsto \Delta)$

```
span ← j − i
if i ≡ 0  mod (span ∗ 2)
    i_sib ← j
    j_sib ← j + span
else
    j_sib ← i
    i_sib ← i − span
if ∃Δ_sib  s.t.  s.UM[(i_sib, j_sib)] = Δ_sib
    i_prt ← min(i, j, i_sib, j_sib)
    j_prt ← max(i, j, i_sib, j_sib)
    if i_sib < i
        Δ_prt = h_k (0x01||Δ_sib||Δ)
    else
        Δ_prt = h_k (0x01||Δ||Δ_sib)
    s.UM ← s.UM ∪ {(i_prt, j_prt) ↦ Δ_prt}
    MergeUpdDig(s, (i_prt, j_prt) ↦ Δ_prt)
```

Algorithm 6 $\mathcal{CMT}^h_k.\mathsf{GetPoC}(ctr_0, ctr_1, s) \rightarrow u$ and $\mathcal{CMT}^h_k.\mathsf{GetU}(ctr_0, ctr_1, s) \rightarrow u$. Comment: In $\mathcal{CMT}$, both functions have the same implementation.

```
s_i ← s.stateHist[ctr_0]   / state corresponding to ctr_0
s_f.n ← s.stateHist[ctr_1]   / # of elms. in state corresponding to ctr_1
u.s ← s_i
u.n ← s_f.n
u.DM ← ∅
i ← s_i.n
while i < s_f.n
    m ← largest power of 2 which divides i
    if m = 0 ∨ m > s_f.n − i
        m ← largest power of 2 ≤ s_f.n − i
    Δ ← s.UM[(i, i + m)]
    u.DM ← u.DM ∪ {(i, i + m) ↦ Δ}
    i ← i + m
return u
```

g) Computing and updating PoIs (Algorithms 9 and 10): The PoI is the proof of inclusion that verifies membership of a specific element in the tree. For example, in Fig. 1a, the PoI p of element x_1 will contain the following digests: $\{\Delta_{0,1}, \Delta_{2,4}, \Delta_{4,6}\}$. ComPoIDM computes this set of digests

for a particular element with index ID by recursively traversing down either the left or right side of the current subtree.

The old state's digests and new digest sets in our PoCs also serve as the update value u, which, in our construction, can be used to update an old PoI. Specifically, $u.DM$ contains exactly the additional full-subtree digests needed to construct the changed digests in a new PoI. Concretely, for x_1, the new PoI is obtained by replacing the top sibling $\Delta_{4,6}$ with $\Delta_{4,8} = h\big(0x01 \parallel \Delta_{4,6} \parallel \Delta_{6,8}\big)$, while keeping the lower siblings $\Delta_{0,1}$ and $\Delta_{2,4}$ unchanged.

Algorithm 7 $\mathsf{CompPoIState}^{s,s_{poi}}_{i,j}(X) \rightarrow \Delta$

```
/ inputs should satisfy i < j ≤ s.n
if ∃Δ such that s.DM[(i, j)] = Δ
    / digest (Δ) of element was previously computed
else if j − i = 1
    Δ ← h_k(0x00||X[i − s.n])   / digest of single element
else
    m ← largest power of 2 ≤ j − i
    Δ ← h_k (0x01||CompDigest^s_{i,i+m}(X)
                ||CompDigest^s_{i+m,j}(X))
    / digest of a sub-tree containing a left and right sub-trees
s_poi.DM ← s_poi.DM ∪ {(i, j) ↦ Δ}
return Δ
```

Algorithm 8 $\mathsf{ComPoIAll}^{s}_{i,j}(X) \rightarrow PoIs$

```
/ inputs should satisfy 0 ≤ i < j
PoIs ← ∅
s_poi.n = s.n + |X|
s_poi.DM = s.DM
CompPoIState^{s,s_poi}_{i,j}(X)   / populate DM with all new digests
for ID ∈ {j, j + |X|}
    ComPoIDM^{s_poi}_{i,j}(X)
    / compute every PoI using shared state to avoid recomputing new digests
return PoIs
```

Algorithm 9 $\mathsf{ComPoIDM}[ID]^{s}_{i,j}(X) \rightarrow DM$

```
/ inputs should satisfy 0 ≤ i < j
m ← largest power of 2 < j − i
if ID < m   / target is in left subtree; include right sibling
    return (i + m, j, CompDigest^s_{i+m,j}(X))∪
                ComPoIDM[ID]^s_{i,i+m}(X)
else   / target is in right subtree; include left sibling
    return (i, i + m, CompDigest^s_{i,i+m}(X))∪
                ComPoIDM[ID − m]^s_{i+m,j}(X)
```

Algorithm 10 $\mathcal{CMT}^{h}_{k}.\mathsf{UpdPoI}(p, u) \rightarrow p'$

```
/ u.n is num elements after update
s'.n ← u.n   / update length
s'.DM ← u.s.DM ∪ p.DM ∪ u.DM
    / u.s.DM, p, u.DM provides all necessary values to compute new PoI
p'.i ← p.i
p'.n ← s'.n
p'.DM ← (ComPoIDM[p.i]^{s'}_{0,s'.n}(∅))
    / p includes new vals for PoI
return p'
```

h) Verifying PoI, PoC, and updates (Algorithms 11 and 12): Both verification algorithms reconstruct the root digest of the latest tree. Verification succeeds if the reconstructed root digest matches the one passed by the client; otherwise, it fails. VerPoI reconstructs the root starting from the leaf by recursively hashing the current digest with its sibling (the next-highest digest in the PoI digest set).

Alg. 12, which implements both VerPoC and VerUpd, uses CompDigest to compute the old and new roots using the old state $PoC.s$/$u.s$. The old state's digests combined with the digests in $PoC.DM$/$u.DM$ are enough to construct the new root.

Algorithm 11 $\mathcal{CMT}^{h}_{k}.\mathsf{VerPoI}(\Delta, x, p) \rightarrow \{\text{TRUE or FALSE}\}$

```
return Δ = PoIDigest(x, p)

PoIDigest(x, p) → Δ {
    if p.n ≤ 1 then return h_k(0x00||x)
    if |π.DM| = 0 then return ⊥
    ((i, j) ↦ Δ') ← find map entry with the smallest j − i value
                    in p.DM
    p'.DM ← p.DM \ {(i, j) ↦ Δ'}
    m ← lP_2(p.n − 1)
    if p.i < n   / PoI target is in the left subtree
        p'.n ← n
        p'.i ← p.i
        return h_k(0x01||PoIDigest(x, p')||Δ')
    else   / PoI target is in the right subtree
        p'.n ← p.n − n
        p'.i ← p.i − n
        return h_k(0x01||Δ'||PoIDigest(x, p'))
}
```

Algorithm 12 Algorithm of $\mathcal{CMT}^{h}_{k}$.VerPoC (Δ, PoC, Δ') and $\mathcal{CMT}^{h}_{k}$.VerUpd (Δ, u, Δ') where both return either TRUE or FALSE.

```
VerPoC(Δ, PoC, Δ') → {TRUE or FALSE}{
    return VerUpdnPoC(Δ, PoC, Δ')
}
VerUpd(Δ, u, Δ') → {TRUE or FALSE}{
    return VerUpdnPoC(Δ, u, Δ')
}
VerUpdnPoC(Δ, o, Δ') → {TRUE or FALSE}{
    / Here o is the tuple of (s, DM, n) representing either PoC or update value u
    s'.n ← o.n
    s'.DM ← o.s.DM ∪ o.DM
    / Δ's covering range of all elements accumulated
    if CompDigest^{o.s}_{0,o.s.n}(∅) ≡ Δ and
        CompDigest^{s'}_{0,s'.n}(∅) ≡ Δ'
        return True   / original Δ is consistent with original state and
    / new Δ' is consistent with PoC values
    else return False
}
```

IV. Evaluation

We implement our $\mathcal{CMT}^{h}$ design in Go, enabling direct comparison with Google's open-source *Transparency.dev* implementation [2], which is written in Go and deployed in many production CT-like logs. This provides a realistic benchmark

against an industry-standard CT codebase under the same language and runtime environment.

All experiments were executed using an AMD Ryzen 9 7950X3D desktop processor, one CPU core, 64 GiB memory, AMD64 Linux, and Go version 1.25.4. Each leaf hash in our experiments is a SHA-256 digest of 100 randomly generated bytes, using APIs from the crypto/sha256 package. For each experiment, we integrate $\mathcal{CMT}^h$ and Google's implementation into the same benchmarking harness to ensure identical conditions, isolating the performance differences attributable to our accumulator design. Values were chosen that span multiple orders of magnitude for algorithms that exhibited logarithmic complexity. Further experimentation showed that values that are exact powers of 2 and near powers of 2 exhibit very low and high memory usage, respectively. A similar pattern was noticed for runtime. Therefore, as a compromise, we chose values as $1.5 \cdot 2^k, k \in \mathbb{N}$, i.e. halfway between powers of 2. For each runtime data point, we express the mean with error bars to denote the 95% confidence interval across 50 runs. Because our memory values do not fluctuate across runs, each memory data point is the value of a single run. We made efforts to limit background processes in user space while running tests, but some figures consistently fail to exhibit clear monotonic trends. We hypothesize the reason is a combination of the Go garbage collector, dynamic allocators, and memory locality.

a) Accumulation Performance: Fig. 3 and Fig. 4 show the time and space required to:

1) Add elements to an initially empty log (starting with 0 elements).
2) Append an additional batch of elements to an existing log (incremental update).

While Google's implementation stores all digests of the current tree, recomputing internal nodes as required for each addition, $\mathcal{CMT}^h$ uses cached maximal subtrees to avoid recomputation for unchanged portions of the tree. As seen in Fig. 3, the memory allocation of $\mathcal{CMT}^h$ significantly outperforms that of Transparency.dev. The second experiment measuring the time it takes to accumulate elements, shown in Fig. 4, demonstrates how incremental accumulation with caching improves throughput in large CT logs. We can see that the storage of a large state as in Transparency.dev negatively influences performance, causing the addition time to vary widely between accumulation events.

In Fig. 4 we show accumulation time vs. number of elements, comparing $\mathcal{CMT}^h$ to Google's *Transparency.dev* with fixed $|X| = 1{,}000$ as n increases (we also measured varying $|X|$ at fixed $n = 100{,}000$, and do not present these results as they showed similar results).

b) Proof-of-Consistency and Update Verification Performance: We do not present PoC and update verification performance as both $\mathcal{CMT}^h$ and Transparency.dev compute the same PoC from pre-generated digests, just returning the PoC from stored metadata. Note that $\mathcal{CMT}^h$'s reliance on batch accumulations means only PoCs between batch boundaries are supported. The advantage is needing to store less digests while still achieving the same $O(\log m)$ generation time. We do not present update verification performance data as updates are not supported by Transparency.dev.

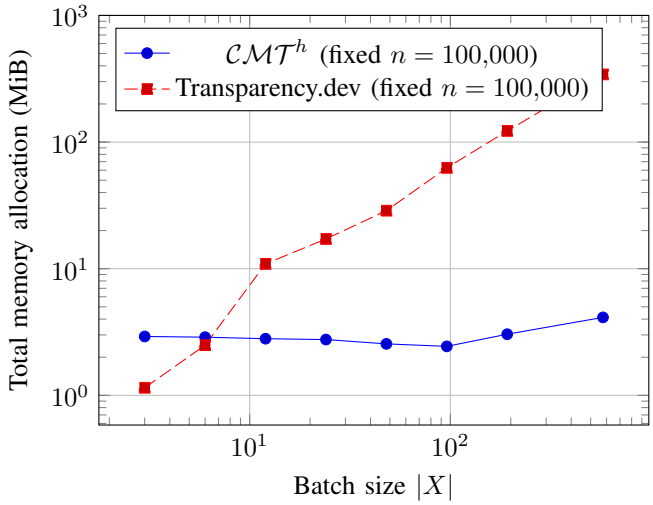


Fig. 3: Appending various batch sizes $|X|$ to a log of fixed size $n = 100{,}000$. For each experiment, the log has a starting size of 100,000 elements. We measure the total memory allocated as a result of appending 1000 batches, each of size $|X|$.

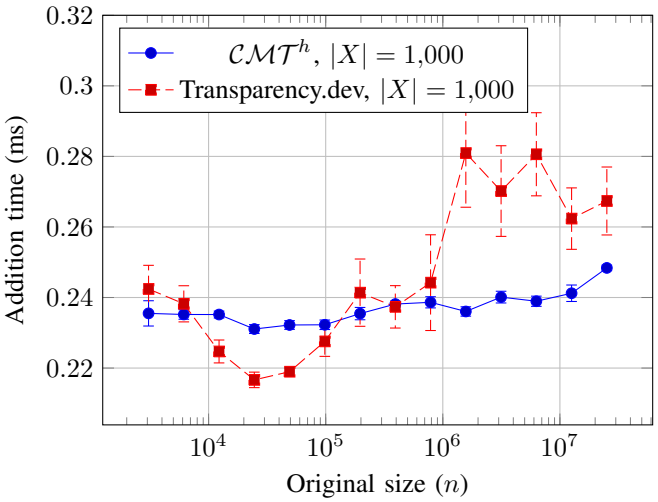


Fig. 4: Incremental append of a fixed batch $|X| = 1{,}000$ as n grows. In each experiment, a varied n elements are added first and we measure the time it takes to add $1,000$ elements.

c) Proof-of-Consistency Generation: We now measure the time required to generate a proof of consistency between two tree sizes. We ran experiments varying both:

1) The size n of the original tree.
2) The size of each element batch, $|X|$.

Due to the graphs showing the same trend, we only present here Fig. 5, varying tree size, but the results from varying batch size were similar, with $\mathcal{CMT}^h$ showing a similar but faster trend compared to Transparency.dev.

For each scenario, we begin with a log of size n and then append 1000 batches of size $|X|$. After every batch b_i is added, $i \in [1, 1000]$, let t_i be the new state of the log. When all

1000 batches are added, the final log t_{1000} has size $n + 1000 \cdot |X|$ elements. We measure the time it takes to generate 1000 proofs of consistency, from a random starting state $t_r, r \in \text{Uniform}\{1, ..., 999\}$, to t_{1000}.

Since these tests compute significantly more per data point, we run the tests 7 times per point instead of 50.

Google's *Transparency.dev* generates PoCs after accumulation as defined in the CT standard [25]. $\mathcal{CMT}^h$ stores only the digests needed to generate a PoC between any two batch boundaries, i.e. states that result from calling Accum on a batch of elements. As a result, $\mathcal{CMT}^h$ saves space by storing only the necessary digests need to compute specific PoCs, but losing the ability to generate the fine-grained proofs to and from within batch appends. This tradeoff was described in more detail in Section I-A.

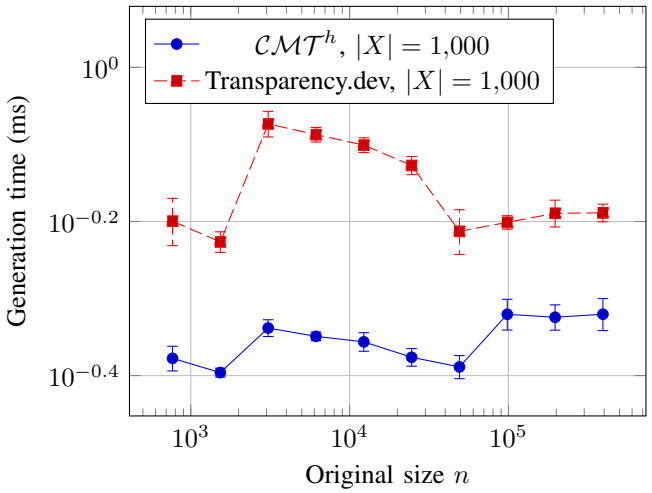


Fig. 5: Time to generate a PoC from 1000 random starting states to the most recent state. There are 1000 possible starting states, each resulting from appending a constant 1000 elements to the previous state. The earliest of these states has size n, which we vary in this test.

Fig. 5: Consistency proof generation time vs. number of elements, comparing $\mathcal{CMT}^h$ to Google's *Transparency.dev* varying $|X|$ at fixed $n = 100{,}000$ (we do not present, but we also measured fixed $|X| = 1{,}000$ as n increases, seeing similar results).

d) Proof-of-Inclusion Generation: We do not compare PoI generation time, as the implementations are fundamentally different. In contrast to Transparency.dev where PoIs are generated on demand when requested, we don't generate PoIs retroactively in $\mathcal{CMT}^h$, instead generating them during an accumulation event. In real systems objects and their corresponding PoIs are stored together. In a CT-like system, the PoI is sent with the certificate; in blockchains, the PoI should again be kept with the transaction. Retroactively requesting the log or block producer for a PoI is backwards (and expensive). The current CT practice of retroactive PoI generation creates a huge performance tax; our design avoids that by producing updateable PoIs upfront, which matches the real-world use case for these systems.

A. Summary of Results

Across all experiments, $\mathcal{CMT}^h$ consistently outperforms Google's *Transparency.dev* implementation. In the CT setting, this translates directly into reduced CPU load on log servers, faster proof generation, and lower latency for monitors and auditors when synchronizing with the log.

V. Analysis of $\mathcal{CMT}^h$

In this section we analyze $\mathcal{CMT}^h$, proving that $\mathcal{CMT}^h$ satisfies the consistent accumulator requirements presented in Section II-A and Section II-B and the extended consistent accumulator requirements presented in Section II-D.

a) Correctness requirements: We first note that $\mathcal{CMT}^h$ ensures, unconditionally, the correctness requirements.

Lemma 2. $\mathcal{CMT}^h$ ensures the consistent accumulator correctness requirements: *PoI correctness*, *updating PoI correctness*, and *PoC and Update value correctness* (Defs. 2, 3, 4).

Proof. (1) **PoI Correctness.** By construction, each call to Accum replaces the state's digest map with the dyadic cover of $[0, s'.n)$ and sets $\Delta' = \mathsf{CompDigest}^{s'}_{0,s'.n}(\emptyset)$. Then the recomputed root from the full subtrees in $s'.DM$ is exactly the returned digest, giving PoI correctness for all newly added leaves, as their ComPoIDM paths are assembled from these subtrees.

(2) **Updating PoI Correctness.** For updating PoIs, $u = (s, u.DM, s'.n)$ contains precisely the full subtrees that cover the appended range $[s.n, s'.n)$; UpdPoI only swaps siblings that cross into this new range and leaves the original path before the update the same. Hence the updated proof recomputes to $\mathsf{CompDigest}^{\tilde{s}}_{0,s'.n}(\emptyset)$ where $\tilde{s}.DM = s.DM \cup u.DM$, which is computationally equivalent to $s'.DM$, so verification under Δ' holds.

(3) **PoC and Update value Correctness.** Finally, VerUpd and VerPoC each reconstruct a state from $(s, u.DM, s'.n)$ and check that the roots before and after match Δ and Δ'. Since the dyadic cover is maximal and aligned, the reconstructed map yields the same root as $s'.DM$, hence both verifiers accept. □

b) Unforgeability requirements: Next, we show that $\mathcal{CMT}^h$ ensures the three unforgeability requirements, *PoI Unforgeability*, *update unforgeability*, and *PoC unforgeability*. *PoI (proof of inclusion) unforgeability* means that an adversary who sees any number of digests, states, and update values still cannot produce a valid PoI for an element that was never accumulated except with negligible probability. *Update unforgeability* means that once a PoI is valid, applying any valid update value preserves validity. *PoC (proof of consistency) unforgeability* means that the adversary cannot produce a valid PoC showing that a new digest Δ_1 is consistent with a previous digest Δ_0, if there is some string included in Δ_0 but not in Δ_1 except with negligible probability.

Lemma 3. If the hash used by $\mathcal{CMT}^h$ is any-collision resistant (ACR), then $\mathcal{CMT}^h$ satisfies *PoI unforgeability*, *update unforgeability*, and *PoC unforgeability* (Defs. 5, 6, 7).

Proof. All three properties reduce to the fact that $\mathcal{CMT}^h$'s verifiers (VerPoI, VerUpd, VerPoC) accept if and only if the recomputed root equals the claimed digest, where roots are computed by concatenating child digests over the dyadic cover.

(1) **PoI unforgeability.** A forged PoI for an element $x \notin \bigcup(\xi)$ means that there are two different leaf assignments that hash to the same root, which is a collision in the Merkle hashing.

(2) **Update unforgeability.** If a valid update u could turn a valid PoI (x, π) under Δ_0 into an invalid PoI under Δ_1, or turn an invalid PoI into a valid one, then at some point in the tree hashing, a sibling node given (or omitted) by u would have to change the recomputed root without changing any of the siblings from the pre-update range. That would mean that two inputs map to the same node digest, i.e., a hash collision.

(3) **PoC unforgeability.** If an adversary produced a PoC accepted between Δ_0 and Δ_1 where Δ_1 is not an append-only extension, the verifier's reconstructed state would necessarily contain a subtree whose leaf contents do not match between the two histories yet yield the same root digest. This again implies a hash collision.

In all cases, any-collision resistant (ACR) of the hashing function rules out these events except with negligible probability. □

A. The $\mathcal{CMT}^h$ is ACR

Collision resistance for a consistent accumulator is analogous to collision resistance for static accumulators: no efficient adversary can produce two different element sets that yield the same digest. In $\mathcal{CMT}^h$, regardless of if elements are added all at once or in batches, digests are Merkle roots computed over a fixed ordering of leaves. The cache only keeps subtrees of previously added elements and its use does not change the root function. Hence, under an ACR hash, distinct accumulated sets yield distinct roots, which gives accumulator ACR.

Lemma 4. Assume the underlying hash function used by $\mathcal{CMT}^h$ is any-collision resistant (ACR). Then $\mathcal{CMT}^h$ is an ACR consistent accumulator (Def. 8).

Proof. By Lemma 3, under ACR of the underlying hash, $\mathcal{CMT}^h$ satisfies PoI unforgeability. By Lemma 2, $\mathcal{CMT}^h$ satisfies PoI correctness. Therefore, by Lemma 1, $\mathcal{CMT}^h$ is ACR. □

Lemma 5. $\mathcal{CMT}^h$ satisfies extended accumulator correctness properties (Defs. 10 and 11).

Proof. (1) GetU **correctness.** Given $ctr_0 < ctr_1$, $\mathsf{GetU}(ctr_0, ctr_1, s_1)$ returns $u = (s_1, u.DM, ctr_1)$ where $u.DM$ is the dyadic cover of $[ctr_0, ctr_1)$; $s_1.DM$ already covers $[0, ctr_0)$. The verifier forms $\tilde{s}$ with $\tilde{s}.DM = s_1.DM \cup u.DM$ and $\tilde{s}.n = ctr_1$. This map is *computationally equivalent* to the real map we get after the update, therefore $\mathsf{CompDigest}^{\tilde{s}}_{0,ctr_1}(\emptyset) = \Delta_{ctr_1}$ and $\mathsf{VerUpd}(\Delta_{ctr_0}, u, \Delta_{ctr_1}) =$ TRUE. For any PoI π valid under Δ_{ctr_0}, $\mathsf{UpdPoI}(\pi, u)$ preserves siblings in $[0, ctr_0)$ and replaces only those crossing into $[ctr_0, ctr_1)$ from $u.DM$; the recomputed root is Δ_{ctr_1}, so verification holds.

(2) GetPoC **correctness.** $\mathsf{GetPoC}(ctr_0, ctr_1, s_1)$ returns $PoC = (s_1, u.DM, ctr_1)$ with $u.DM$ the dyadic cover of $[ctr_0, ctr_1)$. The verifier reconstructs $\tilde{s}$ as above and checks the first root equals Δ_{ctr_0} (by definition) and the second root equals Δ_{ctr_1}. Since $\tilde{s}.DM = s_1.DM \cup u.DM$ is computationally equivalent to the honest state's map over $[0, ctr_1)$, the post-root matches and $\mathsf{VerPoC}(\Delta_{ctr_0}, PoC, \Delta_{ctr_1}) =$ TRUE. □

VI. Conclusion

In this work, we introduced a precise definition for consistent accumulators, specifying a uniform interface and the exact security properties that a consistent accumulator should satisfy.

Building on this framework, we presented the Efficient Merkle-tree Consistent Accumulator ($\mathcal{CMT}^h$), a new Merkle-compatible consistent accumulator that preserves the standard digest and proof formats while enabling efficient dynamic updates. By caching the roots of maximal power-of-two subtrees, $\mathcal{CMT}^h$ performs incremental digest computation and PoI updates without ever re-hashing the entire dataset.

We proved that $\mathcal{CMT}^h$ meets the required security and consistency properties, and our run results as compared to *Transparency.dev* showed significant improvements as compared to the widely used CT standard.

Since $\mathcal{CMT}^h$ is a drop-in replacement for existing Merkle-based logs (e.g. Certificate Transparency) yet improves both on computation and storage overheads, it offers a practical upgrade path for systems that rely on append-only transparency logs, secure audit trails, or other authenticated data structures demanding scalability.

Acknowledgements

Ivan Pryvalov was supported in part by the Luxembourg National Research Fund (FNR), under the CORE Junior project FP2 (C20/IS/14698166/FP2/Mueller) and 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] B. Laurie, E. Messeri, and R. Stradling, "Certificate Transparency Version 2.0," RFC 9162 (Experimental), RFC Editor, Fremont, CA, USA, Dec. 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9162.txt

[2] Google, "Transparency.dev: Trust your data with a tamper-evident log," https://transparency.dev/, 2025.

[3] B. Laurie, "Certificate transparency," *Communications of the ACM*, vol. 57, no. 10, pp. 40–46, 2014.

[4] H. Shi, "Golang implementation of cmt," 2025. [Online]. Available: https://github.com/storchdev/optimized-dynamic-mt-go

[5] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[6] J. Benaloh and M. De Mare, "One-way accumulators: A decentralized alternative to digital signatures," in *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 1993, pp. 274–285.

[7] B. Laurie, A. Aas, D. Belcher, A. C. DeCarlo, M. Engert, E. Kasper, E. Messeri, M. Seigler, and R. Sleevi, "Trillian: General transparency," in *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 2019, pp. 1–7.

[8] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Annual international cryptology conference*. Springer, 2002, pp. 61–76.

[9] J. Li, N. Li, and R. Xue, "Universal accumulators with efficient nonmembership proofs," in *International Conference on Applied Cryptography and Network Security*. Springer, 2007, pp. 253–269.

[10] P. Camacho, A. Hevia, M. Kiwi, and R. Opazo, "Strong accumulators from collision-resistant hashing," in *International Conference on Information Security*. Springer, 2008, pp. 471–486.

[11] M. H. Au, P. P. Tsang, W. Susilo, and Y. Mu, "Dynamic universal accumulators for ddh groups and their application to attribute-based anonymous credential systems," in *Cryptographers' track at the RSA conference*. Springer, 2009, pp. 295–308.

[12] H. de Meer, M. Liedel, H. C. Pöhls, J. Posegga, and K. Samelin, "Indistinguishability of one-way accumulators," *Faculty Comput. Sci. Math., Univ. Passau, Passau, Germany, Tech. Rep. MIP-1210*, 2012.

[13] D. Derler, C. Hanser, and D. Slamanig, "Revisiting cryptographic accumulators, additional properties and relations to other primitives," in *Cryptographers' track at the rsa conference*. Springer, 2015, pp. 127–144.

[14] E. Ghosh, O. Ohrimenko, D. Papadopoulos, R. Tamassia, and N. Triandopoulos, "Zero-knowledge accumulators and set algebra," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 67–100.

[15] F. Baldimtsi, J. Camenisch, M. Dubovitskaya, A. Lysyanskaya, L. Reyzin, K. Samelin, and S. Yakoubov, "Accumulators with applications to anonymity-preserving revocation," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 301–315.

[16] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging." in *USENIX Security Symposium*, 2009, pp. 317–334.

[17] B. Dowling, F. Günther, U. Herath, and D. Stebila, "Secure Logging Schemes and Certificate Transparency," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 140–158.

[18] D. Kales, O. Omolola, and S. Ramacher, "Revisiting user privacy for certificate transparency," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 432–447.

[19] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas, "Transparency logs via append-only authenticated dictionaries," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1299–1316.

[20] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing Key Transparency to End Users," in *USENIX Security Symposium*, 2015, pp. 383–398.

[21] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa, "Merklê 2: A low-latency transparency log system," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 285–303.

[22] D. Reijsbergen, A. Maw, Z. Yang, T. T. A. Dinh, and J. Zhou, "{TAP}: Transparent and {Privacy-Preserving} data services," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6489–6506.

[23] J. Len, M. Chase, E. Ghosh, K. Laine, and R. C. Moreno, "{OPTIKS}: An optimized key transparency system," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4355–4372.

[24] L. Reyzin and S. Yakoubov, "Efficient asynchronous accumulators for distributed pki," in *International Conference on Security and Cryptography for Networks*. Springer, 2016, pp. 292–309.

[25] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," RFC 6962 (Experimental), RFC Editor, Fremont, CA, USA, Jun. 2013, obsoleted by RFC 9162. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6962.txt

[26] P. Todd, "Merkle mountain ranges," https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md, March 2013, openTimestamps documentation.

[27] B. Bailey and S. Sankagiri, "Merkle trees optimized for stateless clients in bitcoin," in *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 2021, pp. 451–466.

[28] C. Papamanthou, S. Srinivasan, N. Gailly, I. Hishon-Rezaizadeh, A. Salumets, and S. Golemac, "Reckle trees: Updatable merkle batch proofs with applications," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1538–1551.

[29] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *International conference on the theory and application of cryptology and information security*. Springer, 2010, pp. 177–194.

[30] D. Catalano and D. Fiore, "Vector commitments and their applications," in *International Workshop on Public Key Cryptography*. Springer, 2013, pp. 55–72.

[31] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang, "Hyperproofs: Aggregating and maintaining proofs in vector commitments," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3001–3018.

[32] M. Yu, S. Sahraei, S. Li, S. Avestimehr, S. Kannan, and P. Viswanath, "Coded merkle tree: Solving data availability attacks in blockchains," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 114–134.

[33] M. Bellare and P. Rogaway, "Collision-resistant hashing: Towards making UOWHFs practical," in *Annual International Cryptology Conference*. Springer, 1997, pp. 470–484.