Efficient Asynchronous Accumulators for Distributed PKI

Leonid Reyzin and Sophia Yakoubov

Boston University
{reyzin, sonka}@bu.edu

Abstract. Cryptographic accumulators are a tool for compact set representation and secure set membership proofs. When an element is added to a set by means of an accumulator, a membership witness is generated. This witness can later be used to prove the membership of the element. Typically, the membership witness has to be synchronized with the accumulator value: it has to be updated every time another element is added to the accumulator, and it cannot be used with outdated accumulator values. However, in many distributed applications (such as blockchain-based public key infrastructures), requiring strict synchronization is prohibitive. We define *low update frequency*, which means that a witness only needs to be updated a small number of times, and *old-accumulator compatibility*, which means that a witness can be used with outdated accumulator values. Finally, we propose an accumulator that achieves both of those properties.

Keywords: cryptographic accumulators

1 Introduction

Cryptographic accumulators, first introduced by Benaloh and DeMare [BdM94], are compact binding (but not necessarily hiding) set commitments. Given an accumulator, an element, and a *membership witness* (or *proof*), the element's presence in the accumulated set can be verified. Membership witnesses are generated upon the addition of the element in question to the accumulator, and are typically updated as the set changes. Membership witnesses for elements not in the accumulator are computationally hard to find.

There are many applications of cryptographic accumulators. These can be divided into *localized* applications, where a single entity is responsible for proving the membership of all the elements, and *distributed* applications, where many entities participate and each entity has interest in (or responsibility for) some small number of elements. An example of a localized application is an authenticated outsourced database, where the database owner outsources responsibility for the database to an untrusted third party. When responding to a query, that party can then use an accumulator to prove the presence of returned records in the database record set. An example of a distributed application is a credential system; different parties can prove the validity of their credentials by showing

that they are in the accumulated set. In this paper, we focus on distributed applications, which were the original motivation for accumulators [BdM94].

A trivial accumulator construction simply uses digital signatures. That is, when an element is added to the accumulator, it is signed by some trusted central authority, and that signature then functions as the witness for that element. The public verification key of the central authority functions as the accumulator value. However, this solution is very limited, since it requires trust in the central authority who holds the secret signing key. Many distributed applications have no central authority that can be trusted, particularly if they are executed by a number of mutually distrusting peers. In this paper, we are interested in so-called *strong* accumulators (defined formally in Section 2), which require no secrets.

A classic example of a strong accumulator construction is a Merkle tree [Mer89]. A set element is a leaf of the tree, and the corresponding witness is its authenticating path (that is, the sequence of the element's ancestors' siblings). The Merkle tree root is the accumulator value. Unfortunately, like all existing strong accumulator constructions, this construction has the significant drawback of requiring strict synchronization: membership witnesses need to be updated every time a new element is added to the accumulated set, and can then only be verified against the current accumulator. If elements are added at a high rate, having to perform work linear in the number of new elements in order to retain the ability to prove membership can be prohibitively expensive for the witness holders. Additionally, because of the high update rate, the verifier might have trouble maintaining the most current accumulator to use in verifications.

To address these issues, we introduce *asynchronous accumulators*, which have two additional properties. *Low update frequency* allows witnesses to be updated only a sub-linear number of times (in the number of element additions), which in particular means that it is possible to verify a witness that is somewhat older than the current accumulator value. Conversely, *old-accumulator compatibility* allows membership verification against an old accumulator, as long as the old accumulator already contains the element whose membership is being verified.

Section 3 describes these properties in more detail.[1]

Our New Accumulator In this work, we introduce the first strong asynchronous accumulator construction. It leverages Merkle hash trees, but maintains multiple Merkle tree roots as part of the accumulator value, not just one. Our construction has a low update frequency; it requires only a logarithmic amount of work (in the number of subsequent element additions) in order to keep a witness up to date. It is also old-accumulator compatible; unlike any prior construction, it supports the verification of an up-to-date witness against an outdated accumulator, enabling verification by parties who are offline and without access to the most current accumulator. Our construction is made even

[1] The question of whether accumulators updates can be batched, as in our scheme, was first posed by Fazio and Nicolosi [FN03] in the context of dynamic accumulators, which support deletions. It was answered in the negative by Camacho [Cam09], but only in the context of deletions, and only in the centralized case (when all witnesses are updated by the same entity).

more well suited for distributed applications by the fact that it does not require knowledge of the accumulated set (or any other information linear in the number of elements) for the execution of element additions. Section 4 describes our construction in detail, and provides comparisons to prior constructions. Figure 1 describes the asynchrony of our construction.

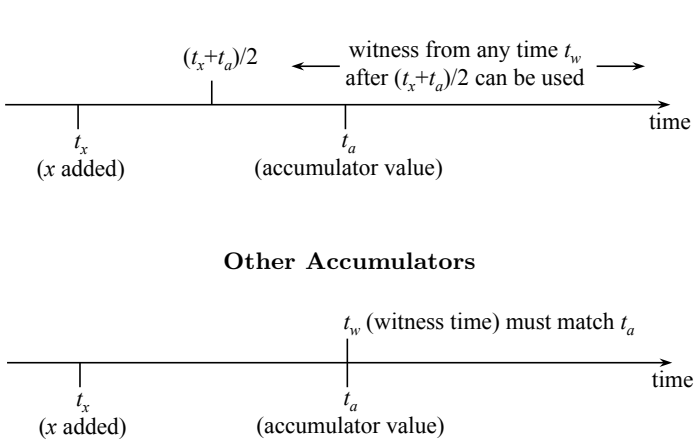

Fig. 1. A membership witness w can either be outdated, or up-to-date. Our accumulator construction is *asynchronous* because even if w is older or newer than the accumulator, verification can still work. This table illustrates the constraints on how outdated w can be. Note that in all other strong accumulator schemes, w must be perfectly synchronized with the accumulator.

1.1 Application: Distributed PKI

The original distributed application proposed by Benaloh and DeMare [BdM94] involved a canonical common state, but did not specify how to maintain it. Public append-only bulletin boards, such as the ones implemented by Bitcoin [Nak08] and its alternatives (altcoins, such as Namecoin [Namrg]), provide a place for this common state. Bitcoin and altcoins implement this public bulletin board by means of blockchains; in Bitcoin they are used primarily as transaction ledgers, while altcoins extend their use to public storage of arbitrary data.

Altcoins such as Namecoin can be used for storing identity information in a publicly accessible way. For instance, they can be used to store (IP address, domain) pairs, enabling DNS authentication [Sle13]. They can also be used to store (identity id, public key pk) pairs, providing a distributed alternative to certificate authorities for public key infrastructure (PKI) [YFV14].

Elaborating on the PKI example, when a user Bob registers a public key pk_{Bob}, he adds the pair ("Bob", pk_{Bob}) to the bulletin board. When the bulletin

board is implemented as a blockchain, it falls to the blockchain miners, who act as the conduit by means of whom content is posted, to verify the validity of this registration. They must, for instance, check that there is not already a public key registered to Bob. Details of the verification process are described by Yakoubov et al. [YFV14]. If the registration is invalid, the miners do not post it to the bulletin board. As a result of the miners' verifications, the bulletin board does not contain any invalid entries.[2]

When Alice needs to verify Bob's public key, she can look through the bulletin board to find this pair. However, when executed naively, this procedure requires Alice to read the entire bulletin board—i.e., a linear amount of data. Bob can save Alice some work by sending her a pointer to the bulletin board location where ("Bob", pk_{Bob}) is posted, which Alice will then follow to check that it points to Bob's registration, and retrieve pk_{Bob}. However, that still requires that Alice *have access* to a linear amount of data during verification. What if Alice doesn't have access to the bulletin board at the time of verification at all, or wants to reduce latency by avoiding on-line access to the bulletin board during verification?

Adding our accumulator to the bulletin board can free Alice from the need for on-line random access to the bulletin board [YFV14] (see also [GGM14] for a similar use of accumulators). The accumulator would contain all of the (id, pk) pairs on the bulletin board, with responsibility for the witnesses distributed among the interested individuals. When Bob posts ("Bob", pk_{Bob}) to the bulletin board, he also adds ("Bob", pk_{Bob}) to the accumulator, and stores his witness w_{Bob}. He posts the updated accumulator to the bulletin board. The validity of this new accumulator needs to be checked, by the same parties who check the validity of Bob's registration. In the blockchain setting, this check is performed by all of the miners. Since our accumulator construction is strong (meaning trapdoor-free, as explained in Section 2) and deterministic, the validity of the new accumulator can be checked simply by re-adding ("Bob", pk_{Bob}) to the old accumulator and comparing the result to the new accumulator.

With our accumulator in place, Alice can simply download the latest accumulator from the end of the bulletin board at pre-determined (perhaps infrequent) intervals. When Alice wants to verify that pk_{Bob} is indeed the public key belonging to Bob, all she needs is w_{Bob} and her locally cached accumulator. As long as Bob's registration pre-dates Alice's locally cached accumulator, Alice can use that accumulator and w_{Bob} to verify that ("Bob", pk_{Bob}) has been posted to the bulletin board. She does not need to refer to any of the new bulletin board contents because our scheme is old-accumulator compatible.

Our construction also reduces the work for Bob, as compared to other accumulator constructions. In a typical accumulator construction, Bob needs to update w_{Bob} every time a new (id, pk) pair is added to the accumulator. However, in a large-scale PKI, the number of entries on the bulletin board and the frequency of element additions can be high. Thus, it is vital to spare Bob the

[2] Note that we do not address public key updates; see Yakoubov et al. [YFV14] for a discussion of such updates.

need to be continuously updating his witness. Because it has a low update frequency, our accumulator reduces Bob's burden: Bob needs to update his witness only a logarithmic number of times. Moreover, Bob can update his witness on-demand—for instance, when he needs to prove membership—by looking at a logarithmic number of bulletin board entries (see Section 5 for details).

2 Background

As described in the introduction, informally, a cryptographic accumulator is a compact representation of a set of elements which supports proofs of membership.[3] In this section, we provide a more thorough description of accumulators, their algorithms and their security definitions. In Section 3, we introduce new properties for asynchronous accumulators.

2.1 Accumulator Algorithms

A basic accumulator construction consists of four polynomial-time algorithms: Gen, Add, MemWitUpOnAdd and VerMem, described below. They were first introduced in Baric and Pritzmann's [BP97] formalization of Benaloh and DeMare's [BdM94] seminal work on accumulators, and a more general version was provided by Derler, Hanser and Slamanig [DHS15]. For convenience, we enumerate and explain all of the input and output parameters of the accumulator algorithms in Figure 2.

k:	The security parameter.
t:	A discrete time / operation counter.
a_t:	The accumulator at time t.
x, y:	Elements which might be added to the accumulator.
w_t^x:	The witness that element x is in accumulator a_t at time t.
upmsg_t:	A broadcast message sent (by the accumulator manager, if one exists) at time t to all witness holders immediately after the accumulator has been updated. This message is meant to enable all witness holders to update the witnesses they hold for consistency with the new accumulator. It will often contain the new accumulator a_t, and the nature of the update itself (e.g. "x has been added and witness w_t^x has been produced"). It may also contain other information.

Fig. 2. Accumulator algorithm input and output parameters.

We separate the accumulator algorithms into (1) those performed by the accumulator manager if one exists, (2) those performed by any entity responsible

[3] There also exist *universal* accumulators [LLX07] which additionally support proofs of non-membership; however, we only consider proofs of membership in this paper.

for an element and its corresponding witness (from hereon-out referred to as *witness holder*), and (3) those performed by any third party.

Algorithms performed by the accumulator manager:

- $\mathsf{Gen}(1^k) \to a_0$ instantiates the accumulator a_0 (representing the empty set). In some accumulator constructions, a secret key sk and auxiliary storage m are additionally generated for the accumulator manager if these are needed to perform additions. However, this is not the case in our scheme.
- $\mathsf{Add}(a_t, x) \to (a_{t+1}, w^x_{t+1}, \mathsf{upmsg}_{t+1})$ adds the element x to the accumulator, producing the updated accumulator value a_{t+1}, and the membership witness w^x_{t+1} for x. Additionally, an update message upmsg_{t+1} is generated, which can then be used by all other witness holders to update their witnesses.

Note that accumulator constructions where Gen and Add are deterministic and publicly executable are also *strong* (as defined by Camacho, Hevia, Kiwi and Opazo [CHKO08]), meaning that the accumulator manager does not need to be trusted. An execution of Gen or Add can then be carried out by an untrusted accumulator manager, and verified by any third party in possession of the inputs simply by re-executing the algorithm and checking that the outputs match. In fact, an accumulator manager is not necessary at all, since Gen and Add can be executed by the (possibly untrusted) witness holders themselves and verified as needed.

Algorithms performed by a witness holder:

- $\mathsf{MemWitUpOnAdd}(x, w^x_t, \mathsf{upmsg}_{t+1}) \to w^x_{t+1}$ updates the witness for element x after another element y is added to the accumulator. The update message upmsg_{t+1} might contain any subset of $\{w^y_{t+1}, a_t, a_{t+1}, y\}$, as well as other parameters.

Algorithms performed by any third party:

- $\mathsf{VerMem}(a_t, x, w^x_t) \to b \in \{0, 1\}$ verifies the membership of x in the accumulator using its witness.

Accumulator Size Space-efficiency is an important benefit of using an accumulator. A trivial accumulator construction would eschew witnesses entirely, and have the accumulator consist of a list of all elements it contains. However, this is not at all space-efficient; any party performing membership verification would need to hold all of the elements in the set. Ideally, accumulators (as well as their witnesses and update messages) should remain small no matter how many items are added to them. In the construction presented in this work, the accumulator and its witnesses and updated messages grow only logarithmically.

2.2 Accumulator Security Properties

Now that we have defined the basic functionality of an accumulator, we can describe the security properties an accumulator is expected to have. Informally, the *correctness* property requires that for every element in the accumulator it should be easy to prove membership, and the *soundness* (also referred to as *security*) property requires that for every element not in the accumulator it should be infeasible to prove membership.

Definition 1 (Correctness). *A strong accumulator is* correct *if an up-to-date witness w^x corresponding to value x can always be used to verify the membership of x in an up-to-date accumulator a.*

More formally, for all security parameters k, all values x and additional sets of values $[y_1, \ldots, y_{t_x-1}]$, $[y_{t_x+1}, \ldots, y_t]$:

$$\Pr\left[\begin{array}{l} a_0 \leftarrow \mathsf{Gen}(1^k); \\ (a_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Add}(a_{i-1}, y_i) \text{ for } i \in [1, \ldots, t_x - 1]; \\ (a_{t_x}, w_{t_x}^x, \mathsf{upmsg}_{t_x}) \leftarrow \mathsf{Add}(a_{t_x-1}, x); \\ (a_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Add}(a_{i-1}, y_i) \text{ for } i \in [t_x + 1, \ldots, t]; \\ w_i^x \leftarrow \mathsf{MemWitUpOnAdd}(x, w_{i-1}^x, \mathsf{upmsg}_i) \text{ for } i \in [t_x + 1, \ldots, t] : \\ \mathsf{VerMem}(a_t, x, w_t^x) = 1 \end{array}\right] = 1$$

In Section 3, we modify the correctness definition for asynchronous accumulators. In asynchronous accumulators, the witness w^x does not always need to be up-to-date in order for verification to work (as described in Definition 4), and the accumulator itself can be outdated (as described in Definition 6).

Definition 2 (Soundness). *A strong accumulator is* sound *(or* secure*) if it is hard to fabricate a witness w for a value x that has not been added to the accumulator.*

More formally, for any probabilistic polynomial-time stateful adversary $\mathcal{A}$, there exists a negligible function $negl$ in the security parameter k such that:

$$\Pr\left[\begin{array}{l} a_0 \leftarrow \mathsf{Gen}(1^k); t = 1; x_1 \leftarrow \mathcal{A}(1^k, a_0); \\ \textit{while } x_t \neq \perp \\ \quad (a_t, w_t^{x_t}, \mathsf{upmsg}_t) \leftarrow \mathsf{Add}(a_{t-1}, x_t); \\ \quad t = t + 1; \\ \quad x_t \leftarrow \mathcal{A}(a_{t-1}, w_{t-1}^{x_{t-1}}, \mathsf{upmsg}_{t-1}); \\ (x, w) \leftarrow \mathcal{A} : \\ x \notin \{x_1, \ldots, x_t\} \textit{ and } \mathsf{VerMem}(a_{t-1}, x, w) = 1 \end{array}\right] \leq negl(k)$$

3 New Definitions: Asynchronous Accumulators

An accumulator is *asynchronous* if the accumulator value and the membership witnesses can be out of sync, and verification still works. An accumulator and witness can be out of sync in two ways. First, the witness can be "old" relative to

the accumulator, meaning that more values have been added to the accumulator since the witness was last updated. Second, the accumulator can be "old" relative to the witness, meaning that the witness has been brought up to date relative to a newer accumulator value, but an old accumulator value is now being used for verification. We describe correctness definitions for these two scenarios in Sections 3.1 and 3.2, respectively. The soundness definition doesn't change.

3.1 Low Update Frequency

We consider an accumulator to have a *low update frequency* if the frequency with which a witness for element x needs to be updated is sub-linear in the number of elements which are added after x. The fact that witnesses do not need to be updated with every addition naturally implies that they can be "old" relative to the accumulator, and still verify correctly. Of course, the fact that updates are needed at all implies that witnesses can't be *arbitrarily* old without verification failing.

Low update frequency requires a change in the correctness definition (but not in the soundness definition). In the new correctness definition, we introduce a function $\mathsf{UpdateTimes}(t, t_w, t_x)$ which describes when the witness needs to be updated. It returns a set T of times between t_w and t at which the witness w last updated at time t_w for an element x added at time t_x needs to be updated. Note that $\mathsf{UpdateTimes}(t_w, t_w, t_x) = \emptyset$ and $\mathsf{UpdateTimes}(t_1, t_w, t_x) \subseteq \mathsf{UpdateTimes}(t_2, t_w, t_x)$ if $t_1 < t_2$.

Definition 3 (Low Update Frequency (LUF)). *An accumulator has* low update frequency *if there exists a function* $\mathsf{UpdateTimes}(t, t_w, t_x)$ *such that (a)* $|\mathsf{UpdateTimes}(t, t_w, t_x)|$ *is sublinear in* t *for all fixed* t_w, t_x *s.t.* $t_w \geq t_x$*, and (b) the accumulator is* $\mathsf{UpdateTimes}(t, t_w, t_x)$-LUF-correct*, as described in Definition 4.*

Definition 4 (Low Update Frequency (LUF) Correctness). *An accumulator is* $\mathsf{UpdateTimes}(t, t_w, t_x)$-LUF-correct *if an outdated witness* w^x *from time* t_w *corresponding to value* x *added at time* t_x *can be used to verify the membership of* x *in an up-to-date accumulator* a *at time* t *as long as* $\mathsf{UpdateTimes}(t, t_w, t_x)$ *is empty.*

More formally, for all security parameters k*, for all values* x *and additional sets of values* $[y_1, \ldots, y_{t_x-1}]$*,* $[y_{t_x+1}, \ldots, y_t]$*, the following probability is equal to 1:*

$$\Pr\left[\begin{array}{l} a_0 \leftarrow \mathsf{Gen}(1^k); \\ (a_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Add}(a_{i-1}, y_i) \text{ for } i \in [1, \ldots, t_x - 1]; \\ (a_{t_x}, w_{t_x}^{x}, \mathsf{upmsg}_{t_x}) \leftarrow \mathsf{Add}(a_{t_x-1}, x); \\ (a_i, w_i^{y_t}, \mathsf{upmsg}_i) \leftarrow \mathsf{Add}(a_{i-1}, y_i) \text{ for } i \in [t_x + 1, \ldots, t]; \\ w_i^x \leftarrow \mathsf{MemWitUpOnAdd}(x, w_{i-1}^x, \mathsf{upmsg}_i) \text{ for } i \in \mathsf{UpdateTimes}(t, t_x, t_x) : \\ \mathsf{VerMem}(a_t, x, w_i^x) = 1 \text{ for } i = \max(\mathsf{UpdateTimes}(t, t_x, t_x)) \end{array}\right]$$

3.2 Old-Accumulator Compatibility

We consider an accumulator to be *old-accumulator compatible* if up-to-date witnesses w_t^x can be verified even against an outdated accumulator a_{t_a} where $t_a < t$, as long as x was added to the accumulator before (or at) t_a.[4] Old-accumulator compatibility allows the verifier to be offline and out of sync with the latest accumulator state.

Like low update frequency, old-accumulator compatibility requires a change in the correctness definition (but not in the soundness definition). Note that unlike Definition 4, Definition 6 is not parametrized by a function; we expect a witness to be compatible with an old accumulator no matter how out of sync they are, as long as the accumulator already contains the element in question.

Definition 5 (Old-Accumulator Compatibility (OAC)). *An accumulator is* old-accumulator compatible *if the accumulator is* OAC-correct*, as described in Definition 6.*

Definition 6 (Old-Accumulator Compatiblity (OAC) Correctness). *An accumulator is* OAC-correct *if an up-to-date witness w^x corresponding to value x can always be used to verify the membership of x in* any out-of date accumulator a which already contains x.

More formally, for all security parameters k, all values x and additional sets of values $[y_1, \ldots, y_{t_x-1}]$, $[y_{t_x+1}, \ldots, y_t]$:

$$\Pr\left[\begin{array}{l} a_0 \leftarrow \mathsf{Gen}(1^k); \\ (a_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Add}(a_{i-1}, y_i) \text{ for } i \in [1, \ldots, t_x - 1]; \\ (a_{t_x}, w_{t_x}^x, \mathsf{upmsg}_{t_x}) \leftarrow \mathsf{Add}(a_{t_x-1}, x); \\ (a_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Add}(a_{i-1}, y_i) \text{ for } i \in [t_x + 1, \ldots, t]; \\ w_i^x \leftarrow \mathsf{MemWitUpOnAdd}(x, w_{i-1}^x, \mathsf{upmsg}_i) \text{ for } i \in [t_x + 1, \ldots, t] : \\ \forall j \in \{t_x, \ldots, t\}, \mathsf{VerMem}(a_j, x, w_t^x) = 1 \end{array}\right] = 1$$

4 Our New Scheme

There are several known accumulator constructions, including the RSA construction [BdM94,CL02,LLX07], the Bilinear Map construction [Ngu05,DT08,ATSM09], and the Merkle tree construction [CHKO08]. (Other similar Merkle tree constructions are described in [BLL00] and [BLL02].) Their properties are described in Figure 3. None of these constructions have low update frequency or old-accumulator compatibility.

We present a different Merkle tree construction which, unlike the constructions given in [CW09,CHKO08,BLL00,BLL02], is asynchronous: that is, it has low update frequency and old-accumulator compatibility. However, unlike some of those Merkle tree constructions, it is not universal (meaning that it does not support proofs of non-membership).

[4] Note that this does not compromise the soundness property of the accumulator, because if x was not a member of the accumulator at t_a, w_t^x does not verify with a_{t_a}.

Accumulator Protocol Runtimes and Storage Requirements					
Accumulator	**Signatures**	**RSA**	**Bilinear Map**	**Merkle**	**This Work**
Add runtime	1	1	1 w/ trapdoor, n without	$\log(n)$	$\log(n)$
Add storage	1	1	1 w/ trapdoor, n without	$\log(n)$	$\log(n)$
MemWitUpOnAdd runtime	0	1	1	$\log(n)$	$\log(n)$
MemWitUpOnAdd storage	0	1	1	$\log(n)$	$\log(n)$
Accumulator Properties					
Accumulator	**Signatures**	**RSA**	**Bilinear Map**	**Merkle**	**This Work**
Accumulator size	1	1	1	1	$\log(n)$
Witness size	1	1	1	$\log(n)$	$\log(n)$
Strong?	no	no[5]	no	yes	yes
Update frequency [6]	0	n	n	n	$\log(n)$
Old-accumulator compatibile?	yes	no	no	no	yes

Fig. 3. Various accumulator constructions and their protocol runtimes, storage requirements, and properties. For each of Add and MemWitUpOnAdd, this table gives the algorithm runtime, and the storage required for the algorithm execution. Additionally, the table describes other accumulator properties, such as accumulator size, witness size, strength, update frequency and old-accumulator compatibility. We let n denote the total number of elements in the accumulator. The RSA Construction is due to [BdM94,CL02,LLX07]. The Bilinear Map construction is due to [Ngu05,DT08,ATSM09]. The Merkle tree construction is due to [CW09], though it is not described as an accumulator construction. (Other Merkle tree constructions are given in [CHKO08,BLL00,BLL02].) Big-O notation is omitted from this table in the interest of brevity, but it is implicit everywhere.

4.1 Construction

Let n be the number of elements in our accumulator, and let h be a collision-resistant hash function. When h is applied to pairs or elements, we encode the pair in such a way that it can never be confused with a single element x – e.g., a pair is prefaced with a '2', and a single element with '1'. Additionally, we encode accumulated elements in such a way that they can never be mistaken for the output of h. For instance, we might preface each element with 'elt', and the output of the hash function with 'hash'.

Our accumulator maintains a list of $D = \lceil \log(n+1) \rceil$ Merkle tree roots $r_{D-1}, \ldots, r_0$ (as opposed to just one Merkle tree root). The leaves of these Merkle trees are the accumulated elements. r_d is the root of a complete Merkle tree with 2^d leaves if and only if the dth least significant bit of the binary expansion of n is 1. Otherwise, $r_d = \perp$. Note that this accumulator is very similar to a binary counter of elements, but instead of having a 1 in the dth least significant

[5] Sander [San99] shows a way to make the RSA accumulator strong by choosing the RSA modulus in such a way that its factorization is never revealed.

[6] Here n refers to the number of elements added and deleted *after* the addition of the element whose witness updates are being discussed.

position representing the presence of 2^d elements, the accumulator has root of the complete Merkle tree containing those elements.

A witness w^x for x is the authenticating path for x in the Merkle tree that contains x. That is, if x is in the Merkle tree with root r_d, then $w^x = ((z_1, \mathsf{dir}_1), \ldots, (z_{d-1}, \mathsf{dir}_{d-1}))$, where each z_i is in the range of the hash function h, and each dir is either right or left. These are the (right / left) sibling elements of all of the nodes along the path from element x to the Merkle tree root of depth d. These siblings, together with the element x, can be used to reconstruct the root of the Merkle tree. Note that, if x is in the Merkle tree of depth 0, then the witness is empty. An illustration of an accumulator a and a witness w is given in Figure 4.

Membership Verification Verification is done by using the authenticating path w^x and the element x in question to recompute the Merkle tree root and check that it indeed matches the accumulator root r_d, where d is the length of w^x. In more detail, this is done by recomputing the *ancestors* of the element x using the authenticating path w^x as described in Algorithm 1, where the ancestors are the nodes along the path from x to its root, as defined by x and by elements in w^x. If the accumulator is up to date, the last ancestor should correspond to the appropriate accumulator root r_d.

If the accumulator is outdated but still contains x, one of the recomputed ancestors should still correspond to one of the accumulator roots. This is because, as described in Algorithm 4 of Appendix C, witnesses (and thus the ancestors they are used to compute) are append-only.

If the witness is outdated (that is, it is from time $t_w < t$), verification can be done at time t as long as $t < t_w + 2^d$. In Figure 1, we lower-bound d as $\log_2(t_w - t_x)$, where t_x is the time at which the element in question was added. This results in the condition $t < 2t_w - t_x$.

Verification is described in full detail in Algorithm 5 of Appendix C.

Element Addition Element addition is done by merging Merkle trees to create deeper ones. Specifically, when the nth element x is added to $a = [r_{D-1}, \ldots, r_0]$, if $r_0 = \perp$, we set $r_0 = h(x)$. If, however, $r_0 \neq \perp$, we "carry" exactly as we would in a binary counter: we create a depth-one Merkle tree root $z = h(r_0, h(x))$, set $r_0 = \perp$, and try our luck with r_1. If $r_1 = \perp$, we can set $r_1 = z$. If $r_1 \neq \perp$, we must continue merging Merkle trees and "carrying" further up the chain. Element addition is described in full detail in Algorithm 3 of Appendix C, and is illustrated in Figures 5 and 6 of Appendix B.

Membership Witness Updates Membership witness updates need to be performed only when the root of the Merkle tree containing the element in question is merged, or "carried", during a subsequent element addition. This occurs at most D times. Membership updates use the update message $\mathsf{upmsg}_{t+1} = (y, w^y_{t+1})$ (where y is the element being added and w^y_{t+1} is the witness generated for y) in order to bring the witness w^x_t for the element x up to date, as described in Algorithm 4 of Appendix C.

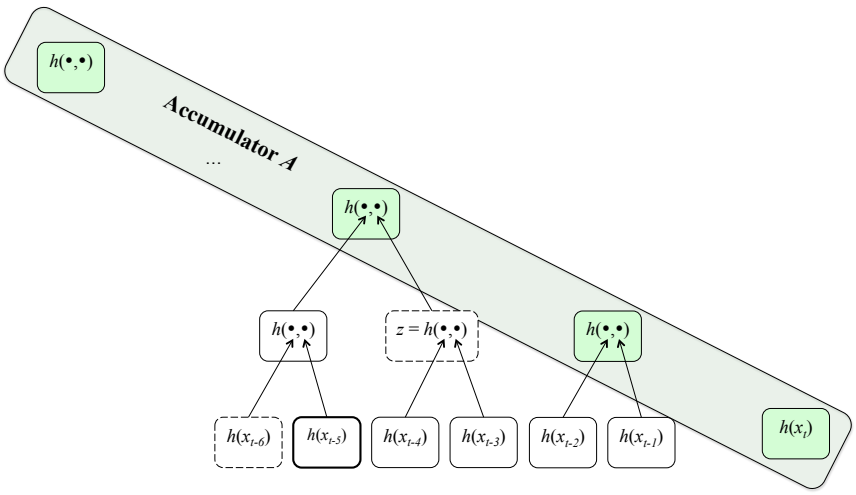

Fig. 4. An illustration of our accumulator. The accumulator itself is shaded; the unshaded elements are elements of the Merkle trees which are not actually a part of the accumulator. The elements with dashed outlines belong to the authenticating path for x_{t-5} (which itself has a bold outline). So, the witness for x_{t-5} would be $w^{x_{t-5}} = ((h(x_{t-6}), \mathsf{left}), (z, \mathsf{right}))$.

4.2 Properties

The accumulator construction presented in this paper is sound (Theorem 1), has low update frequency (Theorem 2) and is old-accumulator compatible (Theorem 3).

Theorem 1. *The construction presented in this paper is* sound *as described in Definition 2 as long as h is collision resistant.*

Proof. We prove soundness using the classical technique for Merkle trees. Say we have an adversary $\mathcal{A}_a$ who can break soundness (i.e., who can find a witness for an element that has not been added to the accumulator). We construct the accumulator Merkle forest using the elements $x_1, \ldots, x_{t-1}$ that $\mathcal{A}_a$ requests be added to the accumulator. $\mathcal{A}_a$ then gives us an authenticating path for an element $x \notin \{x_1, \ldots, x_{t-1}\}$, and therefore not in any of the accumulator Merkle trees. Using that path, we reconstruct the ancestors of that element. The first ancestor that actually appears in any of our accumulator Merkle trees is a collision, since we have two values that hash to it: the value or pair of values that legitimately appears in our accumulator Merkle tree, and the value or pair of values provided by the adversary.

Theorem 2. *The construction presented in this paper has* low update frequency *as described in Definition 3.*

Proof. A witness for element x need only be updated when the Merkle tree in which x resides is merged (or "carried"), which happens at most $D = \lceil \log(n+1) \rceil$ times.

Theorem 3. *The construction presented in this paper is* old-accumulator compatible *as described in Definition 5.*

Proof. Correctness (as described in Definition 1) is self-evident; a Merkle tree root can be correctly reconstructed given its authenticating path. Old accumulator compatibility (OAC) correctness (as described in Definition 6) follows from the fact that witnesses are append-only; whenever they are modified, the entire prior witness state remains and new information is tacked on at the end. Thus, for every past accumulator (as long as it already contains the element x in question), there is a subset of the current witness w_t^x which can be used for verification against that outdated accumulator.

Strength Additionally, the construction presented in this paper is *strong*, meaning that it does not rely on a trusted accumulator manager. Since every operation is deterministic and publicly verifiable, the accumulator manager would have no more luck breaking soundness than a witness holder would. This is important for distributed use-cases, where there might not be a central trusted party to execute element additions.

Distributed Storage The construction presented in this paper has fully distributed storage; all storage requirements are logarithmic in the number of elements. The accumulator manager does not need to store the accumulated elements, or any other additional data, to perform additions. This, too, is important for distributed use-cases, because storing a lot of data might be too burdensome for the users of the distributed system, and there might not be a central manager willing to store the necessary data.

5 Taking Advantage of Infrequent Membership Witness Updates in a Distributed PKI

We now return to the PKI application of our accumulator described in Section 1.1, where we have a membership witness holder who may not be able to make a witness update whenever a new element is added. As highlighted in Section 4.2, our accumulator scheme requires that the witness for a given element x be updated at most $D = \lceil \log(n+1) \rceil$ times, where n is the number of elements added to the accumulator after x. However, one might observe that having to *check* whether the witness needs updating each time a new element addition occurs renders this point moot, since this check itself must be done a linear number of times. We solve this problem by giving our witness holders the ability to "go back in time" to observe past accumulator updates. If they can ignore updates when they occur, and go back to the relevant ones when they need to bring their witness up to date (e.g. at when they need to show it to a verifying third party), they can avoid looking at the irrelevant ones altogether.

"Going back in time" is possible in the public bulletin board setting of our PKI application, in which our accumulator is maintained as part of a public bulletin board. The bulletin board is append-only, so it contains a history of all of the accumulator states. Along with these states, we will include the update message, and a counter indicating how many additions have taken place to date. Additionally, we will include pointers to a selection of other accumulator states, so as to allow the bulletin board user to move amongst them efficiently. The pointers from accumulator state t would be to accumulator states $t - 2^i$ for all i such that $0 < 2^i < t$ (somewhat similarly to what is done in a skip-list). These pointers can be constructed in logarithmic time: there is a logarithmic number of them, and each of them can be found in constant time by using the previous one, since $t - 2^i = t - 2^{i-1} - 2^{i-1}$. Note that storing these pointers is not a problem, since we are already storing a logarithmic amount of data in the form of the accumulator and witness.

Our witness holder can then ignore update messages altogether, performing no checks or work at all. Instead, he updates his witness only when he needs to produce a proof. When this happens, he checks the counter of the most recently posted accumulator state. The counter alone is sufficient to deduce whether his witness needs updating. If his witness does not need updating, he has merely performed a small additional constant amount of work for the verification at hand. If, as happens a logarithmic number of times, his witness does need updating, the pointers and counters allow him to locate in logarithmic time the (at most logarithmic number of) bulletin board entries he needs to access in order to bring his witness up to date, as described in Algorithm 9 of Appendix C. Thus, the total work performed by our witness holder will remain logarithmic in the number of future element additions.

Acknowledgements

This research is supported, in part, by US NSF grants CNS-1012910, CNS-1012798, and CNS-1422965. Leonid Reyzin gratefully acknowledges the hospitality of IST Austria and École normale supérieure, where part of this work was performed.

The authors would like to thank Dimitris Papadopoulos and Foteini Baldimtsi for their insightful feedback.

References

ATSM09. Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In Marc Fischlin, editor, *Topics in Cryptology — CT-RSA 2009*, volume 5473 of *Lecture Notes in Computer Science*, pages 295–308. Springer Berlin Heidelberg, 2009.

BdM94. Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application*

of Cryptographic Techniques on Advances in Cryptology, EUROCRYPT '93, pages 274–285, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

BLL00. Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 9–17, New York, NY, USA, 2000. ACM.

BLL02. Ahto Buldas, Peeter Laud, and Helger Lipmaa. Eliminating counterevidence with applications to accountable certificate management. *J. Comput. Secur.*, 10(3):273–296, September 2002.

BP97. Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 480–494, 1997.

Cam09. Philippe Camacho. On the impossibility of batch update for cryptographic accumulators. Cryptology ePrint Archive, Report 2009/612, 2009.

CHKO08. Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security*, volume 5222 of *Lecture Notes in Computer Science*, pages 471–486. Springer Berlin Heidelberg, 2008.

CL02. Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2002.

CW09. Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 317–334, Berkeley, CA, USA, 2009. USENIX Association.

DHS15. David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In Kaisa Nyberg, editor, *Topics in Cryptology — CT-RSA 2015*, volume 9048 of *Lecture Notes in Computer Science*, pages 127–144. Springer International Publishing, 2015.

DT08. Ivan Damgrd and Nikos Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. Cryptology ePrint Archive, Report 2008/538, 2008.

FN03. Nelly Fazio and Antonio Nicolosi. Cryptographic accumulators: Definitions, constructions and applications, 2003.

GGM14. Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

LLX07. Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, volume 4521 of *Lecture Notes in Computer Science*, pages 253–269. Springer Berlin Heidelberg, 2007.

Mer89. Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 218–238, 1989.

Nak08. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

Namrg. Namecoin, https://www.namecoin.org/.

Ngu05. Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *Topics in Cryptology — CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 275–292. Springer Berlin Heidelberg, 2005.

San99. Tomas Sander. Efficient accumulators without trapdoor extended abstracts. In *Information and Communication Security, Second International Conference, ICICS'99, Sydney, Australia, November 9-11, 1999, Proceedings*, pages 252–262, 1999.

Sle13. Greg Slepak. Dnschain + okturtles. `http://okturtles.com/other/dnschain_okturtles_overview.pdf`, 2013.

YFV14. Sophia Yakoubov, Conner Fromknecht, and Dragos Velicanu. Certcoin: A namecoin based decentralized authentication system, 2014.

A Limited Dynamism

In 2002, Camenisch and Lysyanskaya [CL02] introduced the notion of *dynamic accumulators*, which support the deletion of elements from the accumulator. Deletions can, of course, be performed simply be generating a new accumulator and re-adding all the elements which have not been deleted. This takes a polynomial amount of time in the number of elements, and so is, strictly speaking, efficient. However, a dynamic accumulator should support deletions in time which is either independent of the number of elements altogether, or is sublinear in the number of elements. A dynamic accumulator has the following additional algorithms:

- $\mathsf{Del}(a_t, x) \to (a_{t+1}, \mathsf{upmsg}_{t+1})$ deletes the element x from the accumulator.
- $\mathsf{MemWitUpOnDel}(x, w_t^x, \mathsf{upmsg}_{t+1}) \to w_{t+1}^x$ updates the membership witness for element x after y is deleted from the accumulator.

We can make our accumulator construction dynamic by giving the accumulator manager auxiliary storage m consisting of the leaves of the Merkle trees (i.e., the set of elements in the accumulator). Then, to perform a deletion Del, the manager replaces the leaf in the tree corresponding to x with $\perp$, updates the ancestors of this leaf, and broadcasts the updated ancestors of $\perp$ as the update message upmsg. To perform a witness update MemWitUpOnDel (upon receipt of upmsg), each witness holder whose value x is in the same Merkle tree replaces one node on its path (namely, the child node of the lowest common ancestor of the deleted value and x) with the corresponding value from upmsg.

This modification degrades the space efficiency of the manager by adding auxiliary linear storage on top of the very short (logarithmic) accumulator. (We note that this extra storage can be avoided if the witness holder, or perhaps several other cooperating witness holders, can provide the necessary portions of the Merkle tree to the manager when needed. However, this would only truly work if withdrawing an element from the accumulator was a voluntary act—for instance, this would not work in the application of credential revocation.) This modification would also degrade the low update frequency property, and old-accumulator compatibility.

To keep both distributed storage and low update frequency, we can limit deletions to newer elements (e.g. an element can only be deleted within a constant number of turns of being added), since newer elements are in the shallow Merkle trees. While this appears to be limiting, it should be noted that in many applications, deletions of older elements can be avoided altogether by wrapping "time to live stamps" or "expiration dates" into the elements themselves.

B Element Addition

In Figures 5 and 6, we illustrate a single element addition. Element x_{t+1} is being added to the accumulator. The depth 0 and depth 1 Merkle trees are both present in the accumulator, so two "carries" occur before x_{t+1} is successfully added into the Merkle tree of depth 2.

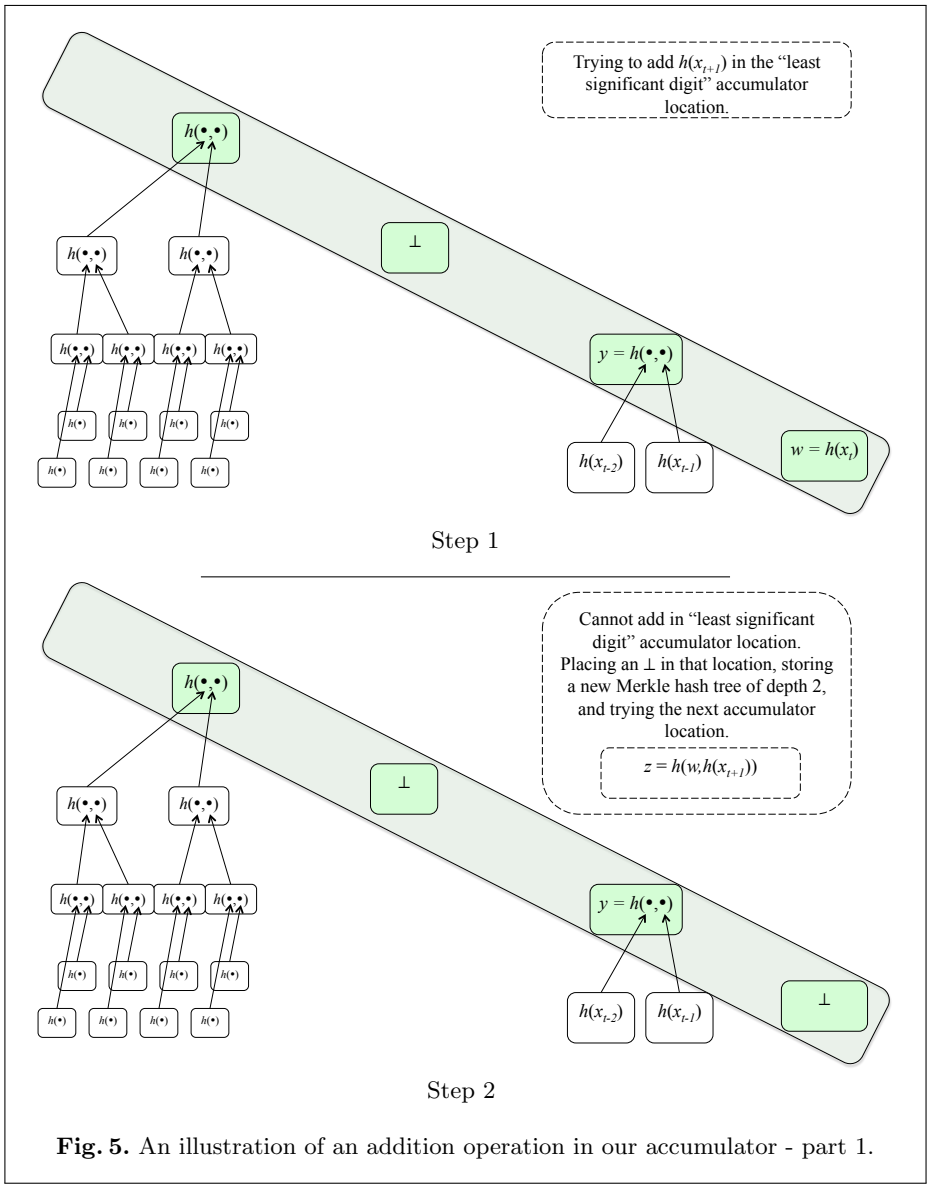

Fig. 5. An illustration of an addition operation in our accumulator - part 1.

C Algorithms

In this appendix, we give the pseudocode for all of the algorithms used in our accumulator scheme. A Python implementation of these algorithms is available upon request.

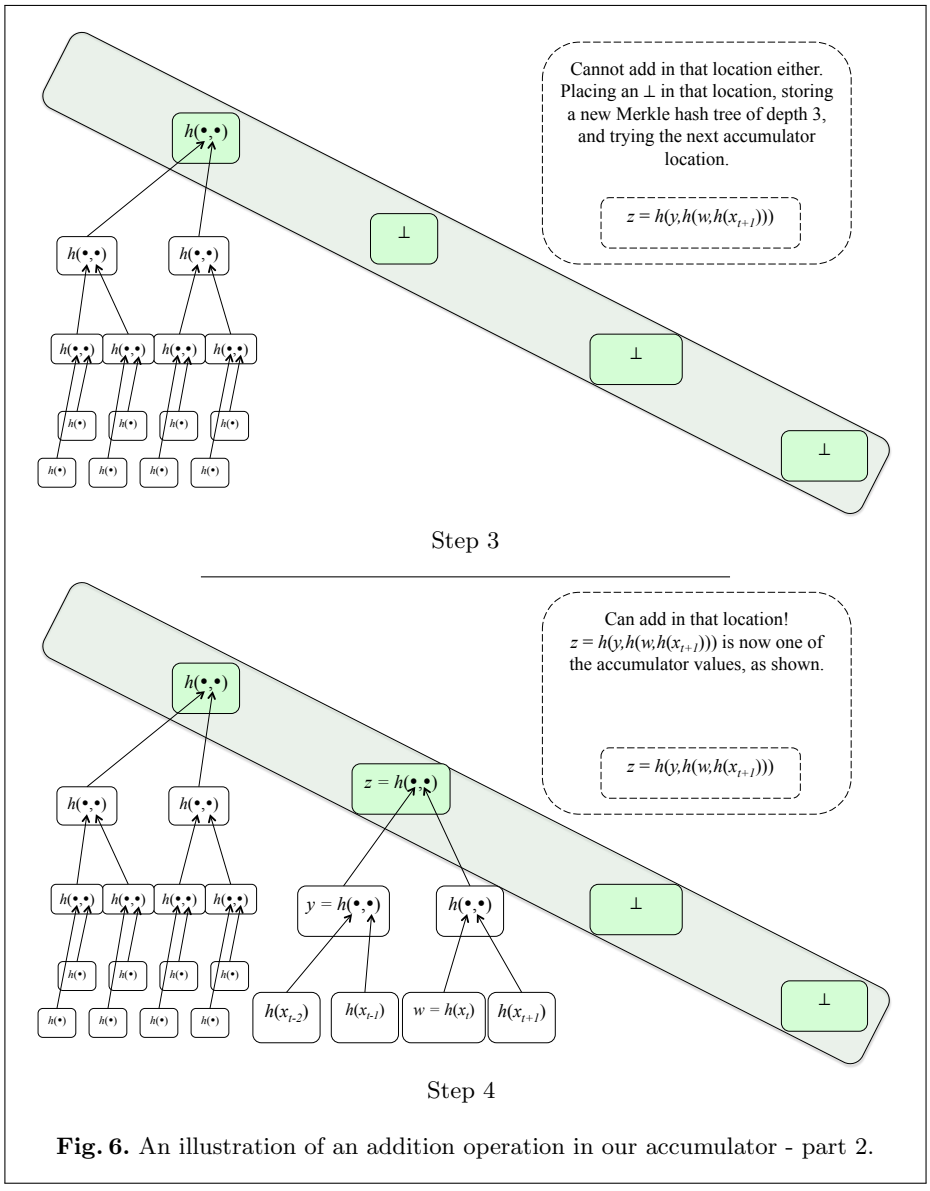

Fig. 6. An illustration of an addition operation in our accumulator - part 2.

C.1 Accumulator Algorithms

In this section, we give the pseudocode for the basic accumulator algorithms, such as Gen (Algorithm 2), Add (Algorithm 3), MemWitUpOnAdd (Algorithm 4) and VerMem (Algorithm 5). $L[i]$ denotes the ith element of a list. For convenience, in these algorithms the accumulator is represented in reverse order. Recall that h is a hash function.

Algorithm 1 GetAncestors: a helper function for MemWitUpOnAdd (Algorithm 4) and VerMem (Algorithm 5).

Require: p, x
$c = h(x)$
$\overline{p} = [c]$
for (z, dir) in p **do**
  if $\mathsf{dir} = \mathsf{right}$ **then**
    $c = h(c||z)$
  else if $\mathsf{dir} = \mathsf{left}$ **then**
    $c = h(z||c)$
  append c to $\overline{p}$
return $\overline{p}$

Algorithm 2 Gen

Require: 1^k
return $a_0 = \perp$

Algorithm 3 Add

Require: a_t, x
$a_{t+1} = a_t$ (the new accumulator starts out as a copy of the old one)
$w^x_{t+1} = []$ (the witness starts out as an empty list)
$d = 0$ (the depth of the witness starts out as 0)
$z = h(x)$
while $a_{t+1}[d] \neq \perp$ **do**
  if the length of $a_{t+1} < d + 2$ **then**
    append $\perp$ to a_{t+1}
  $z = h(a_{t+1}[d]||z)$
  append $(a_{t+1}[d], \mathsf{left})$ to w^x_{t+1}
  $a_{t+1}[d] = \perp$
  $d = d + 1$
$a_{t+1}[d] = z$
return $a_{t+1}, w^x_{t+1}, \mathsf{upmsg} = (x, w^x_{t+1})$

Algorithm 4 MemWitUpOnAdd

Require: y, w^y_{t+1}, w^x_t
let d^x_t be the length of w^x_t
let d^y_{t+1} be the length of w^y_{t+1}
if $d^y_{t+1} < d^x_t$ **then**
  return w^x_t (the witness has not changed)
else
  $d^x_{t+1} = d^y_{t+1}$
  $w^x_{t+1} = w^x_t$ (the new authenticating path starts out as a copy of the old one)
  $\overline{w}^y_{t+1} = \mathsf{GetAncestors}(w^y_{t+1}, y)$
  append $(\overline{w}^y_{t+1}[d^x_t], \mathsf{right})$ to w^x_{t+1}
  append $w^y_{t+1}[d^x_t + 1, \dots]$ to w^x_{t+1}
  return w^x_{t+1}

Algorithm 5 VerMem

Require: a_t, x, w^x
1: $\overline{p} = \mathsf{GetAncestors}(w^x, x)$
2: **if** a_t and r have any elements in common (computable in linear time) **then**
3: **return** TRUE
4: **else**
5: **return** FALSE

C.2 Batch Witness Updates

In this section, we give the pseudocode for the algorithms which allow our witness holder to avoid reading to every update message, and instead do only a logarithmic amount of work upon every verification in order to bring the witness up to date. In the following algorithms, we assume the existence of a public append-only random access bulletinboard. bulletinboard[ptr] gives the ptrth entry of bulletinboard. However, since bulletinboard may be used for things other than accumulator entries, the ptrth entry of bulletinboard is not guaranteed to correspond to the ptrth accumulator update. Instead, we let $t' = \mathsf{bulletinboard}[\mathsf{ptr}].t$ denote the timestep t' such that the ptrth entry of bulletinboard corresponds to the t'th accumulator update. GetPointers (Algorithm 6) and GetPointer (Algorithm 7) are helper algorithms for creating the pointers and using them to move amongst the entries of bulletinboard which are relevant to the accumulator.

Let i be the number of irrelevant entries on bulletinboard after the last relevant entry, and let n be the number of elements which have been added to the accumulator. GetPointers (Algorithm 6), which finds the pointer to include in a new bulletin board entry, takes $O(i) + O(\log(n))$ time. (The $O(i)$ is present because GetPointers finds the newest accumulator bulletinboard entry by iterating over the entries of bulletinboard backwards.) Similarly, GetPointer (Algorithm 7), which finds a pointer to a desired bulletin board entry given another pointer at which to start, takes $O(\log(n))$ time.

BatchMemWitUpOnAdd (Algorithm 9), the batch witness update algorithm itself, also takes $O(\log(n))$ time. BatchMemWitUpOnAdd reverses the list of relevant indices before finding the pointers to them for reasons of efficiency. This way, the total number pointers followed is $O(\log(n))$, and not $O(\log(n)^2)$. The list of relevant pointers is then reversed again, so as to perform the actual membership witness updates in order.

Algorithm 6 GetPointers: finds all the pointers needed for a new accumulator update bulletin board entry. If the accumulator update happens at timestep t, the pointers should be to all accumulator updates at timesteps $t - 2^i$ for i such that $0 < 2^i < t$. This is a helper function for BatchMemWitUpOnAdd (Algorithm 9).

Require: the bulletin board bulletinboard

ptrs = []
find the last occurring addition entry $(\mathsf{lt}, a_{\mathsf{lt}}, y, w_{\mathsf{lt}}^y)$ on bulletinboard.
if one does not exist **then**
  return ptrs
let lptr be the pointer to the last entry
append lptr to ptrs
exp = 1
stuck = FALSE
while not stuck **do**
  lptr = ptrs[−1] (the last element of ptrs)
  let numPointersAtLastPointer be the number of pointers stored at bulletinboard[lptr]
  if numPointersAtLastPointer < exp **then**
    stuck = TRUE
  else
    ptr = bulletinboard[lptr].ptrs[exp − 1]
    append ptr to ptrs
  exp = exp + 1
return ptrs

Algorithm 7 GetPointer: finds a pointer to the bulletin board entry corresponding to the t^*th accumulator update. This is a helper function for BatchMemWitUpOnAdd (Algorithm 9).

Require: the bulletin board bulletinboard, the timestep t^*, and a pointer ptr to a bulletin board entry corresponding to $t' \geq t^*$

t' = bulletinboard[ptr].t
while $t' \neq t^*$ **do**
  difference = $t' - t^*$
  let exp be the largest exponent such that 2^{exp} is smaller than difference
  ptr = bulletinboard[ptr].ptrs[exp]
  t' = bulletinboard[ptr].t
return ptr

Algorithm 8 GetUpdateTimeSteps: finds the timesteps at which a witness needs to be updated. This is a helper function for BatchMemWitUpOnAdd (Algorithm 9).

Require: the bulletin board bulletinboard, the timestep lupt at which the last witness update occurred, the timestep lt at which the last accumulator update occurred, and the depth d of the witness in question

relevantTimeSteps = []
power = 2^d
t = lupt + power
while $t \leq$ lt **do**
    append t to relevantTimeSteps
    while $t \mod \text{power} \times 2 = 0$ **do**
        power = power $*$ 2
    $t = t +$ power
return relevantTimeSteps

Algorithm 9 BatchMemWitUpOnAdd

Require: the bulletin board bulletinboard, the witness w^x, and the timestep lupt at which w^x was last updated

let d be the length of w^x
find the last occurring addition entry $(\text{lt}, a_{\text{lt}}, \text{upmsg} = (y, w^y_{\text{lt}}))$ on bulletinboard, and let ptr be the pointer to this entry
relevantTimeSteps = GetUpdateTimeSteps(lupt, lt, d)
reverse the order of relevantTimeSteps
relevantPointers = []
for $t \in$ relevantTimeSteps **do**
    ptr = GetPointer(bulletinboard, t, ptr)
    append ptr to relevantPointers
reverse the order of relevantPointers
for ptr $\in$ relevantPointers **do**
    get $(t, a_t, \text{upmsg} = (y, w^y_t))$ using ptr from bulletinboard
    w^x = MemWitUpOnAdd(y, w^y_t, w^x)
return w^x