# Efficient Asynchronous Accumulators for Distributed PKI

Leonid Reyzin[1] and Sophia Yakoubov[1,2]

[1] Boston University
{reyzin, sonka}@bu.edu
[2] MIT Lincoln Laboratory

**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, and to be updated every time another element is added to the accumulator. In this work we propose an accumulator that, unlike any prior scheme, does not require strict synchronization.

In our construction a membership witness needs to be updated only a logarithmic number times in the number of subsequent element additions. Thus, an out-of-date witness can be easily made current. Vice versa, a verifier with an out-of-date accumulator value can still verify a current membership witness. These properties make our accumulator construction uniquely suited for use in distributed applications, such as blockchain-based public key infrastructures.

**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 with every subsequent addition. Membership witnesses for elements not in the accumulator are computationally hard to find.

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. However, this solution is very limited, since it requires trust in a central authority (in other words, it is not *strong* as defined in Section 2).

There are many applications of cryptographic accumulators. These can mostly 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. In this paper, we focus on distributed applications, which were the original motivation for accumulators [BdM94].

**Our Contribution** One significant problem with accumulators in the context of distributed applications is that all existing constructions require that membership witnesses be updated every time a new element is added into the accumulator. If elements are added to the accumulator 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.

In this work, we introduce a new strong accumulator construction which requires only a logarithmic amount of work (in the number of subsequent element additions) in order to keep a witness up to date. In our accumulator construction, it is also the case that an up-to-date witness can be verified even against an outdated accumulator, enabling verification by parties who are offline and without access to the most recent accumulator. Our construction is made even more well suited for distributed applications by the fact that, unlike the accumulator construction in [CHKO08], it does not require any additional storage in order to perform accumulator updates. Section 3 describes our construction in detail.

**Applications** The original distributed applications 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 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 block chains; 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 $pk_{\mathsf{Bob}}$, he adds the pair ("Bob", $pk_{\mathsf{Bob}}$) to the bulletin board. When Alice needs to verify Bob's public key, she could look through the bulletin board to find this pair. However, when executed naively, this procedure would require 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", $pk_{\mathsf{Bob}}$) is posted; however, that would still require 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?

Our accumulator construction can be used in this setting to free Alice from the need for on-line random access to the bulletin board [YFV14] (see also [GGM14] for a similar use of accumulators). It allows her to instead simply download a small amount of data from the end of the bulletin board at predetermined (perhaps infrequent) intervals. 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", $pk_{\mathsf{Bob}}$) to the bulletin board, he also adds ("Bob", $pk_{\mathsf{Bob}}$) to the accumulator, and stores his witness $w_{\mathsf{Bob}}$. He posts the updated accumulator to the bulletin board, and since our accumulator construction is trapdoor-free and deterministic, the validity of the new accumulator can be checked by all parties simply by re-adding ("Bob", $pk_{\mathsf{Bob}}$) to the old accumulator. Details of how such posts are monitored and validated can be found in [YFV14].

Then, when Alice wants to verify that $pk_{\mathsf{Bob}}$ is indeed the public key belonging to Bob, all she needs is $w_{\mathsf{Bob}}$ and a locally cached accumulator value. As long as Bob's bulletin board post pre-dates Alice's locally cached accumulator value, Alice can use that accumulator value and $w_{\mathsf{Bob}}$ to verify that ("Bob", $pk_{\mathsf{Bob}}$) has been posted to the bulletin board. She does not need to refer to any of the new bulletin board contents, because in our scheme (as opposed to other accumulator schemes), an up-to-date witness can be used for verification even against an outdated accumulator (as long as the addition of the element in question pre-dates the accumulator).

Our construction also reduces the work for Bob, as compared to other accumulator constructions. In a typical accumulator construction, Bob needs to update $w_{\mathsf{Bob}}$ 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 need to be continuously updating his witness. 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 3.2 for details).[3]

## 2 Definitions

As described in the introduction, informally, a cryptographic accumulator is a compact representation of a set of elements which supports proofs of membership. In this section, we provide a more thorough description of accumulators, their algorithms and some properties which may be desired of them.

A basic accumulator construction consists of four polynomial-time algorithms Gen, Add, MemWitUpOnAdd and VerMem, described below. Various flavors of these algorithms have been restated in literature a number of times. 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]. We present them slightly

[3] 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 (i.e., accumulators that support deletions as described in Section 2). 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).

differently: we model all potential input and output parameters more explicitly, and we categorize the algorithms by their intended executor.

For convenience, we enumerate and explain all of the input and output parameters here:

- $k$: the security parameter.
- $sk$: the accumulator manager's secret key or trapdoor. (The corresponding public key, if one exists, is not modeled here as it can be considered to be a part of the accumulator itself.)
- $t$: a discrete time / operation counter.
- $a_t$: the accumulator at time $t$.
- $m_t$: any auxiliary values which might be necessary for the maintenance of the accumulator. These are typically held by the accumulator manager. Note that while the accumulator itself should be constant (or at least sub-linear) in size, $m$ may be larger. Note also that unlike previous Merkle tree constructions, our construction, described in Section 3, does not require $m$.
- $x, y$: elements which might be added to the accumulator.
- $w_t^x$: the witness that element $x$ is in accumulator $a_t$ at time $t$.
- $\mathsf{upmsg}_t$: a broadcast message sent (by the accumulator manager, if one exists) at time $t$ immediately after the accumulator has been updated. This message is meant to enable all entities 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.

We separate the accumulator algorithms into (1) those performed by the accumulator manager if one exists, (2) those performed by any entity responsible for an element and its corresponding witness (from hereon-out referred to as *witness holder*), and (3) those performed by any third party. Parameters which are omitted in some schemes are in grey.

**Algorithms performed by the accumulator manager:**

- $\mathsf{Gen}(1^k) \rightarrow (a_0, m_0, sk)$ instantiates the accumulator $a_0$ (representing the empty set), the auxiliary value $m_0$ necessary for the maintenance of the accumulator, and the accumulator manager's secret key $sk$.
- $\mathsf{Add}(a_t, m_t, x, sk) \rightarrow (a_{t+1}, m_{t+1}, w_{t+1}^x, \mathsf{upmsg}_{t+1})$ adds the element $x$ to the accumulator.

Note that accumulator constructions where Gen and Add are deterministic and do not use $sk$ are also *verifiable*, meaning that an execution of Gen or Add can be carried out by anyone, and verified by any third party in possession of the inputs simply by re-executing the algorithm and checking that the outputs match. In such a case, an accumulator manager is not necessary, since Gen and Add can be executed by the (possibly untrusted) witness holders and verified as needed.

**Algorithms performed by a witness holder:**

– $\mathsf{MemWitUpOnAdd}(x, w_t^x, \mathsf{upmsg}_{t+1}) \rightarrow w_{t+1}^x$ updates the witness for element $x$ after an element $y$ is added to the accumulator. $\mathsf{upmsg}_{t+1}$ might contain any subset of $\{w_{t+1}^y, a_t, a_{t+1}, y\}$, as well as other parameters.

**Algorithms performed by any third party:**

– $\mathsf{VerMem}(a_t, x, w_t^x) \rightarrow b \in \{0, 1\}$ verifies the membership of $x$ in the accumulator using its witness.

Now that we have defined the basic functionality of an accumulator, we can describe the properties an accumulator is expected to have. Informally, the *correctness* property requires that for every element in the accumulator $a$ 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 $a$ it should be infeasible to prove membership.

**Correctness** An 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 $[y_0, \ldots, y_{i-1}]$, $[y_{i+1}, \ldots, y_n]$:

$$\Pr\left[\begin{array}{l}(a_0, m_0, sk) \leftarrow \mathsf{Gen}(1^k);\\ (a_{t+1}, m_{t+1}, w_{t+1}^{y_t}, \mathsf{upmsg}_{t+1}) \leftarrow \mathsf{Add}(a_t, m_t, y_t, sk) \text{ for } t \in [0, \ldots, i-1];\\ (a_{i+1}, m_{i+1}, w_{i+1}^{x}, \mathsf{upmsg}_{i+1}) \leftarrow \mathsf{Add}(a_i, m_i, x, sk);\\ (a_{t+1}, m_{t+1}, w_{t+1}^{y_t}, \mathsf{upmsg}_{t+1}) \leftarrow \mathsf{Add}(a_t, m_t, y_t, sk) \text{ for } t \in [i+1, \ldots, n];\\ w_{t+1}^x \leftarrow \mathsf{MemWitUpOnAdd}(x, w_t^x, \mathsf{upmsg}_{t+1}) \text{ for } t \in [i+1, \ldots, n] :\\ \mathsf{VerMem}(a_{n+1}, x, w_{n+1}^x) = 1\end{array}\right] = 1$$

**Soundness** An 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 all security parameters $k$, for any probabilistic polynomial-time adversary $\mathcal{A}$ with black-box access to a $\mathsf{Add}$ oracle on accumulator $a$, it holds that:

$$\Pr\left[\begin{array}{l}(a_0, m_0, sk) \leftarrow \mathsf{Gen}(1^k);\\ (x, w) \leftarrow \mathcal{A}^{\mathsf{Add}}(1^k, a_0, m_0);\\ \mathsf{Add} \text{ has not been called on } x :\\ \mathsf{VerMem}(a, x, w) = 1\end{array}\right] \leq negl(k)$$

Where *negl* is a negligible function in the security parameter, $x$ is an element $\mathcal{A}$ has not called $\mathsf{Add}$ on, and $a$ is the accumulator after the adversary made all of his calls to $\mathsf{Add}$.

In addition to correctness and soundness, there are a number of properties that might be needed from an accumulator, depending on the application it is being used for. Some of these (e.g. dynamism, universality and strength) have been introduced over the years as interesting additional properties, while others

(e.g. full distribution, low update frequency and old-accumulator compatibility) are new in this paper.

**Constant Size** The trivial accumulator construction would have the accumulator consist of a list of all elements it contains. However, this is not at all space-efficient. Ideally, accumulators should remain small no matter how many items are added to them. An accumulator is *constant-size* if its size (as well as the size of the witnesses created for it) is independent of the number of elements it contains.

It should be noted that there are solutions (e.g. [CHKO08], as well as the construction presented in this work) which grow logarithmically. While these are not constant size, they are still interesting in many applications.

**Dynamism** In 2002, Camenisch and Lysyanskaya [CL02] introduced the notion of *dynamic accumulators*, which support 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:

- $\mathsf{Del}(a_t, m_t, x, sk) \rightarrow (a_{t+1}, m_{t+1}, \mathsf{upmsg}_{t+1})$ (executed by the accumulator manager, if one exists) deletes the element $x$ from the accumulator.
- $\mathsf{MemWitUpOnDel}(x, w_t^x, \mathsf{upmsg}_{t+1}) \rightarrow w_{t+1}^x$ (executed by a witness holder) updates the witness for element $x$ after $y$ is deleted from the accumulator.

Unlike Fazio and Nicolosi [FN03], we present MemWitUpOnDel and MemWitUpOnAdd as two separate algorithms, because in all of the existing accumulator constructions the mechanism by which the update is performed is very different for deletions and additions.

**Universality** In 2007, Li, Li and Xue [LLX07] introduced the notion of *universal accumulators*, which are accumulators that support non-membership proofs as well as membership proofs. For distinction, we let $w$ denote a membership witnesses and $u$ denote a non-membership witness. A universal accumulator has the following additional algorithms:

- $\mathsf{VerNonMem}(a_t, x, u_t^x) \rightarrow \{0, 1\}$ (executed by any third party) verifies the non-membership of $x$ in the accumulator using its non-membership witness $u_t^x$.
- $\mathsf{NonMemWitUpOnAdd}(x, u_t^x, \mathsf{upmsg}_{t+1}) \rightarrow u_{t+1}^x$ (executed by a witness holder) updates the non-membership witness for element $x$ after $y$ is added to the accumulator.

If the accumulator is dynamic as well as universal, it also has the following algorithm:

– $\mathsf{NonMemWitUpOnDel}(x, u_t^x, \mathsf{upmsg}_{t+1}) \to u_{t+1}^x$ (executed by a witness holder) updates the non-membership witness for element $x$ after $y$ is removed from the accumulator.

A universal accumulator should additionally be *undeniable* (as named by Lipmaa, [Lip12]), meaning that it should be infeasible to prove the membership and non-membership of the same element.

**Strength** In 2008, Camacho, Hevia, Kiwi and Opazo [CHKO08] introduced the notion of *strong accumulators*, which are accumulators that do not assume that the accumulator manager is trusted. Strong accumulators cannot use trapdoor information in the creation or maintenance of the accumulator; that is, the parameter $sk$ should be absent from all accumulator algorithms. The construction presented in this work is strong.

**Full Distribution** We consider an accumulator to be *fully distributed* if there is no party (including the accumulator manager, if one exists) which must store an amount of information that is linear or super-linear in the number of elements in the accumulator. That is, the parameter $m$ (if it exists) must be sub-linear in size. (Note that all other parameters are already assumed to be sub-linear.)

**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 $x$.

**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 $a_{t'}$ where $t' < t$, as long as $x$ was added to the accumulator before $t'$. Note that this does not compromise the soundness property of the accumulator, because if $x$ was not a member of the accumulator at $t'$, $w_t^x$ does not verify with $a_{t'}$. Old accumulator compatibility allows the verifier to be offline and out of synch with the latest accumulator state.

## 3 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]. Their properties are described in Figure 2. None of these constructions have low update frequency or old-accumulator compatibility. The construction given in [CHKO08], which is similar to ours in that both are based on Merkle trees, is made more complicated and somewhat less efficient by the fact that it is designed it to be universal. We present a different Merkle tree construction which is fully distributed, old-accumulator compatible and saves on update frequency, but is not universal.

| **Accumulator Protocol Runtimes and Storage Requirements** | | | | | |
|---|---|---|---|---|---|
| **Accumulator** | **Signatures** | **RSA** | **Bilinear Map** | **Merkle** | **This Work** |
| Add runtime | 1 | 1 | 1 w/ trapdoor, $n$ without | $\log(n)$ | $\log(a)$ |
| Add storage | 1 | 1 | 1 w/ trapdoor, $n$ without | $n$ | $\log(a)$ |
| MemWitUpOnAdd runtime | 0 | 1 | 1 | $\log(n)$ | $\log(a)$ |
| MemWitUpOnAdd storage | 0 | 1 | 1 | $\log(n)$ | $\log(a)$ |
| NonMemWitUpOnAdd runtime | – | 1 | 1 | $\log(n)$ | – |
| NonMemWitUpOnAdd storage | – | 1 | 1 | $\log(n)$ | – |
| Del runtime | – | 1 | 1 | $\log(n)$ | $\log(a)$ |
| Del storage | – | 1 | 1 | $n$ | $\log(a)$ (with additional inputs) [4] |
| MemWitUpOnDel runtime | – | 1 | 1 | $\log(n)$ | $\log(a)$ |
| MemWitUpOnDel storage | – | 1 | 1 | $\log(n)$ | $\log(a)$ |
| NonMemWitUpOnDel runtime | – | 1 | 1 | $\log(n)$ | – |
| NonMemWitUpOnDel storage | – | 1 | 1 | $\log(n)$ | – |
| **Accumulator Properties** | | | | | |
| **Accumulator** | **Signatures** | **RSA** | **Bilinear Map** | **Merkle** | **This Work** |
| Accumulator size | 1 | 1 | 1 | 1 | $\log(a)$ |
| Witness size | 1 | 1 | 1 | $\log(n)$ | $\log(a)$ |
| Dynamic? | no | yes | yes | yes | yes (with additional inputs) |
| Universal? | no | yes | yes | yes | no |
| Strong? | no | no[5] | no | yes | yes |
| Fully distributed? | yes | yes | yes | no | yes |
| Update frequency [6] | 0 | $a+d$ | $a+d$ | $a+d$ | $\log(a)+d$ |
| Old accumulator compatibile? | yes | no | no | no | yes |

**Fig. 1.** Various accumulator constructions and their protocol runtimes, storage requirements, and properties. We let $n$ denote the total number of elements in the accumulator, $a$ denote the number of elements added to the accumulator, and $d$ denote the number of elements deleted from 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 [CHKO08]. Big-O notation is omitted from this table in the interest of brevity, but it is implicit everywhere.

### 3.1 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, the pair is encoded in such a way that it can never be confused with a single element $x$ – e.g., a pair is prefaced with a 1, and a single element with a 0.) The accumulator

[4] Refer to Section 3.3 for details.

[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 $a$ and $d$ refer to the number of elements added and deleted *after* the addition of the element whose witness updates are being discussed.

maintains a list of $D = \lceil \log(n+1) \rceil$ elements $r_0, \ldots, r_D$ (as opposed to just one Merkle tree root). The element $r_i$ is the root of a complete Merkle tree with $2^i$ leaves if and only if the $i$th least significant bit of the binary expansion of $n$ is 1. Otherwise, $r_i = \perp$. A witness $w^x$ for $x$ is the authenticating path for $x$ in the Merkle tree that contains $x$. That is, $w^x = ((z_0, \mathsf{dir}_0), \ldots, (z_d, \mathsf{dir}_{d-1}))$, where each $z_i$ is in the range of the hash function $h$, and each $\mathsf{dir}$ is either $\mathsf{right}$ or $\mathsf{left}$. These are the (right / left) sibling elements of all of the nodes along the path from element $x$ to the accumulator root of depth $d$. An illustration of an accumulator $a$ and a witness $w$ is given in Figure 2.

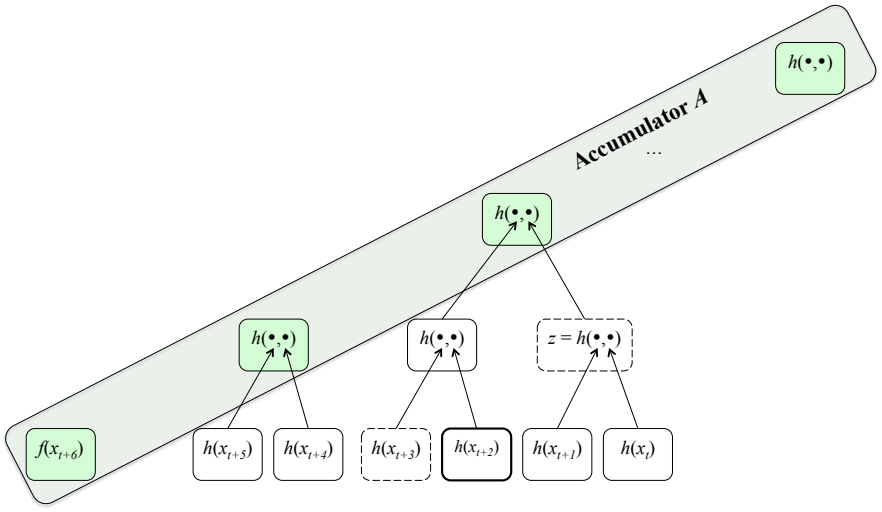


**Fig. 2.** An illustration of our accumulator. The accumulator itself is shaded; the unshaded elements are elements of the Merkle trees which are not actually a part of the accumulator. The elements with dashed outlines belong to the authenticating path for $x_{t+2}$ (which itself has a bold outline). So, the witness for $x_{t+2}$ would be $w^{x_{t+2}} = ((h(x_{t+3}), \mathsf{left}), (z, \mathsf{right}))$.

Verification is done by using the authenticating path $w$ and the element $x$ in question to recompute the Merkle tree root and check that it indeed matches the accumulator element $r_d$, where $d$ is the length of $w$. 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 $p$. If the accumulator is up to date, the last ancestor should correspond to the appropriate accumulator element $r_d$. If the accumulator is outdated but still contains $x$, one of the recomputed

ancestors should still correspond to one of the accumulator elements. Verification is described in full detail in Algorithm 5 of Appendix A.

Element addition is done by merging Merkle trees to create deeper ones. Specifically, when the $n$th element $x$ is added to $a = [r_0, \ldots, r_D]$, 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 $z = h(h(x), r_0)$, 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 A.

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 A.

### 3.2 Infrequent Membership Witness Updates

As highlighted in Section 3.1, this 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 $x$. 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. However, we can get around this 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 many distributed applications. Recall the application from Section 1, 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 A. Thus, the total work performed by our witness holder will remain logarithmic in the number of future element additions.

### 3.3 Other Properties

This construction is trivially correct. It is sound as long as $h$ is collision resistant. Soundness can be proven using the classical technique for Merkle trees: if an adversary $\mathcal{A}$ can find a witness for an element that has not been added to the accumulator, then $\mathcal{A}$ can be used to find a collision for $h$.

This construction is strong, since every operation is deterministic and publicly verifiable. It is also fully distributed; all storage requirements are logarithmic in the number of elements. No auxiliary storage $m$ (as described in Section 2) is necessary for accumulator updates.

**Limited Dynamism** 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, thus compromising full distribution. (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 will also degrade the low update frequency property.

To keep both full distribution 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 small trees. While this appears to be limiting, it should be noted that in many applications, deletions of older elements may be avoided altogether by wrapping "time to live stamps" or "expiration dates" into the elements themselves.

## Acknowledgements

This research is supported, in part, by US NSF grants CNS-1012910, CNS-1012798, and CNS-1422965.

The authors would like to thank Dimitris Papadopoulos for his insightful feedback.

## References

ATSM09. ManHo Au, PatrickP. 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.

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. `http://eprint.iacr.org/`.

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.

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.

Lip12. Helger Lipmaa. Secure accumulators from euclidean rings without trusted setup. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security*, volume 7341 of *Lecture Notes in Computer Science*, pages 224–240. Springer Berlin Heidelberg, 2012.

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.

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 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.

## A.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). Recall that $h$ is a hash function.

## A.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

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

**Require:** $p$, $x$
$c = h(x)$
$\overline{p} = [c]$
**for** $(z, \mathsf{dir})$ 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

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 $2^{\mathsf{exp}}$ 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** $w^x$