Fast hash-based additive accumulators

[DRAFT]

Salvatore Ingala

August 10, 2020

Abstract

Cryptographic accumulators allow to succinctly represent a large set of values by providing a membership witness that can be used to verify that the element was indeed part of a committed set. Accumulators could be used in a public blockchain to create accountable protocols, in which a party could prove to a smart contract that certain events have happened without having to store an expensive audit log. In this work we introduce the first append-only hash-based accumulators with sub-logarithmic insertion cost and poly-logarithmic witness size.

1 Introduction

Cryptographic accumulators are short commitments to a set S in a way that allows one to produce compact proofs of membership that can be verified efficiently by anyone with the knowledge of the accumulator's value.

Introduced in [BDM93] with an application to timestamping, accumulators find a wide range of applications, for example in accountable certificate management [NN98, BLL00], authenticated dictionaries [GTH02], anonymous credentials [CL02], computations on authenticated data [ABC+15], anonymous E-cash [STS99, MGGR13, SALY17], data outsourcing [Sla12], updateable signatures [CJ10, PS14], and decentralized bulletin boards [FVY14, GGM14].

Accumulators find natural applications in blockchains. They are among the building blocks of Zerocoin [MGGR13], a protocol for anonymous E-cash.

Cryptocurrencies like Bitcoin [Nak08] need all network nodes to reach consensus on the set of *unspent transaction outputs* (UTXOs). The memory size of the UTXO set has grown to several gigabytes over time, which increases the cost of running a node, therefore affecting the decentralization of such networks. Similar concerns equally apply to cryptocurrencies in the account model like Ethereum. [BBF19] builds constant-sized accumulators using groups of unknown order, and uses them to create *stateless blockchains* with lightweight nodes. While this construction is theoretically the best possible and it has other useful properties like batching, it has the downside of requiring additional cryptographic assumptions compared to hash-based accumulators. Moreover, hash functions are in practice orders of magnitude faster than operations in number-theoretic groups. In fact, [Dry19] builds on the dynamic hash-based accumulators of [RY16] to show how to create efficient, hash-based dynamic accumulators for the UTXO set of Bitcoin.

1.1 Append-only audit logs

In some protocols, there is an asymmetry in the operational costs of updates, proof generations and proof verification. For example, in Pisa [MBB+18], an audit log of the actions executed by the operator is used in order to later be able to prove that such actions were indeed executed. Thus, if the operator was hired for a task and it did not complete it, the user can issue a challenge, and the operator will lose it and be slashed; vice versa, if the user issues a challenge despite the operator has done the task, then there will be an entry in the audit log that the operator can use to win the challenge.

The downside of the approach above is that the size of the state that grows linearly with the number of tasks. This can be solved by using an accumulator.

Note that in the above protocol, it is not expected to actually have to generate or verify proofs frequently. In fact, if the operator did complete the task, a rational user will *never* issue a challenge that will lose. Thus, while it is important that the costs of are small enough that a proof *could* be created and verified, it is game-theoretically not expected to happen. On the contrary, the accumulator needs to be updated with a new item every time a new task is performed. This justifies the search for accumulators whose update cost is as small as possible, even at the cost of slightly larger proof sizes.

We remark that there is no requirement of ever *deleting* elements that

were added to the accumulator. That is, using the nomenclature in [BCD+17], an *additive* accumulator suffices, rather than a fully dynamic accumulator. It is reasonable to expect that one can obtain an improvement on proof size and update costs by reducing the set of allowed operations on the accumulator, and this is the topic of this paper.

1.2 Our contribution

In this work, we propose a hash-based *additive* accumulator with extremely efficient element insertion time, while retaining poly-logarithmic proof size and verification times. Unlike static accumulators, our constructions allow to add new elements to the committed set, but fall short of being fully dynamic in that we do not allow to remove elements.

Our first construction has a $O(1)$ insertion time (notably, only one hash and the counter need to be written at each insertion), while the proof size is $O(\log^2 n)$, where n is the number of elements added to S from the beginning.

By storing the state of our first construction in a Merkle tree, we obtain an accumulator with $O(\log \log n)$ insertion time, and proofs of size $O(\log n \log \log n)$. In both cases, the space occupation for Add is $O(\log n)$, while the accumulator is a single hash.

To the best of our knowledge, previous hash-based accumulator constructions are based on Merkle trees ([Mer80]) or forests of Merkle trees ([RY16], [Dry19]), and have $O(\log n)$ insertion time and proof size.

2 Definitions

We assume that H is a collision-resistant hash function, and we denote with $||$ the concatenation operator.

For any positive integer n, we define $d(n)$ as the largest power of 2 that divides n. For example, $d(17) = 1$, $d(6) = 2$ and $d(24) = 8$. When n is expressed in binary, $d(n)$ is the number obtained by zeroing all 1 digits, except the least significant one, for example $d(88) = d(1011000_2) = 1000_2 = 8$.

For a positive integer n, we further define $zeros(n) = \log_2(d(n))$ the number of trailing zeros of n, and we define the *predecessor* $pred(n) := n - d(n)$. It is easy to see that $pred(n)$ is the number obtained by zeroing the rightmost 1 bit in the binary representation of n. For example $pred(88) = pred(1011000_2) = 1010000_2 = 80$.

For convenience, we also define $pred(0) = zeros(0) = 0$.

We also define $pred_1(n) := pred(n)$ and, for $t \geq 2$, we define $pred_t(n) := pred(pred_{t-1}(n))$; that is, $pred_t(n)$ is the result obtained by starting from n and applying t times the transform $n \mapsto pred(n)$.

Finally, for any $i \leq n$, we define $rpred(i, n) = pred_t(n)$ for the largest t such that $pred_t(i) \leq n$

3 Intuition

In this section we present the main idea behind our new construction for an additive accumulator.

Let R_k be the value of the accumulator after the k-th element x_k is added. We consider schemes where R_k is defined as a commitment to the value of x_k, together with the value of some subset of previous accumulators $\{R_l\}_{l \in P_k}$, where $P_k \subseteq [1, k-1]$. Therefore, once R_k is known, a witness for some element x_j with $j < k$ can be produced by opening the commitment for R_k itself to prove the values of x_k and $R_{k'}$ for an appropriately chosen $k' \in P_k$, and concatenating it with the witness for x_j starting from $R_{k'}$.

In the graph that has a node for each R_j, and a directed arch from R_j to R_i if $i \in P_j$, building a witness for some element x_j amounts to finding a path from the current accumulator's node R_k to the node corresponding to R_j, and revealing all that is necessary to verify the chain of commitments; by rebuilding the same path, one can verify such a witness. Of course, it is best to find a path that is as short as possible.

The most trivial such scheme is obtained by setting $P_k := \{k-1\}$ for each $k > 1$, obtaining a chain of commitments. This is not an interesting accumulator, though, as the length of the witness is $O(k-j)$.

A more interesting construction can be obtained by choosing

$$P_k := \{R_{k-1}, R_{k-2}, R_{k-4}, \ldots, R_{k-2^i} \ldots\}$$

As $|P_k| = \lceil \log k \rceil$, it is straightforward to obtain a construction with poly-logarithmic witness size, by always revealing R_{k-2^i} where i is chosen so that $k - 2^i$ is the smallest number that is bigger than or equal to the target j. This construction presents a major drawback: the size of the accumulator's state grows linearly in k, and any party that wishes to verify that updates are correct needs to replicate the same state. In the following sections we present our constructions where we address this problem by carefully choosing P_k.

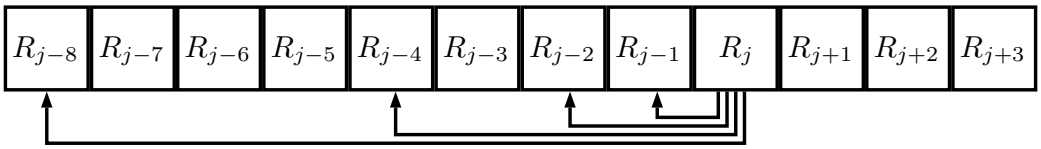

Figure 1: A simple accumulator where each new value R_j commits to previous accumulators at geometrically increasing distance. Each arrow represents one of the values of previous accumulators that R_j commits to.

A crucial idea in all our construction is the following observation: for any fixed integer $d > 1$, the largest multiple of d not bigger than $k+1$ is equal to the highest multiple of d not larger than k, *unless* $k+1$ itself is a multiple of d.

Therefore, we use as 'hook' to the past history of the accumulator the largest $k' \leq k$ that is divisible by 2^i but not divisible by 2^{i+1}. We do this for each i such that $2^i \leq k$, obtaining a set of $\lfloor \log n \rfloor$ hashes that are stored in the state of the accumulator. Crucially, when a new element x_{k+1} is added, only one of these hooks need to be updated, corresponding to the highest power of two that divides $k+1$.

4 First scheme: smart back-links

In our first construction, we choose $P_k = \{k-1, pred(k)\}$, using the notation of Section 1[1].

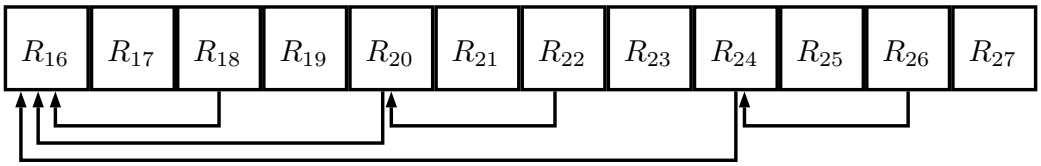

Figure 2: Representation of the commitments of our construction for the accumulator's values R_{16} to R_{27}. Each R_i also commits to the R_{i-1}, but the arrow is omitted for simplicity; for even values of i, R_i also commits to $R_{pred(i)}$.

[1]If k is odd, $pred(k) = k-1$, hence it is redundant to commit to both values; we ignore such optimizations for the sake of simplicity.

The public state of our accumulator contains a counter k, initially set to 0, and an array S. We assume that $S[i] = 0$ by default for every integer i, until $S[i]$ is assigned a different value. The value of k is the number of elements that were added to the accumulator from the beginning.

At any point in time, we maintain the following invariant: $S[i]$ for a non-negative i is the value of the accumulator when the largest $k' \leq k$ was inserted with the property that k' is divisible by 2^i, but not divisible by 2^{i+1}; $S[i] = 0$ if no such number exists.

In particular, the current accumulator R_k after x_k was added is always the value stored in $S[zeros(k)]$; we call R_k the value of the accumulator after the k-th element is added. In order to simplify the notation, we also define $R_0 = NIL$.

Only the first $O(\lfloor \log k \rfloor)$ elements of S can be non-zero, so no other element needs to be stored as part of the public state of the accumulator.

The accumulator's operator (or anyone that wishes to produce witnesses) needs to also store the value R_k of the accumulators for each k.

Element insertion In order to add the k-th element x_k, the accumulator's operator computes R_k as produces a new commitment (by hashing) to the concatenation of $H(x_k)$, R_{k-1} and $R_{pred(k)}$.

The resulting hash is stored in $S[zeros(k)]$, which is the new accumulator's root.

Algorithm 1: Inserting element x

ADD(x)
- $k \leftarrow$ k + 1
- $prev \leftarrow$ S[$zeros(k-1)$]
- $parent \leftarrow$ S[$zeros(pred(k)$]
- S[$zeros(k)$] $\leftarrow$ H($x||prev||parent$)

Once the k-th element x is inserted in the accumulator, we assume that the operator stores R_k and x_k forever if it needs to be able to produce proofs of membership. Note that storing R_k is only needed to produce proofs of membership.

Observe that the value of $prev$ in the algorithm above is equal to R_{k-1}, while the value of $parent$ equals $R_{pred(k)}$. Therefore it is always the case that:

$$R_k = H(x_k||R_{k-1}||R_{pred(k)})$$

Witness generation Given the value of R_i for some i, the operator can generate a membership witness of the value of x_j, for any $0 < j \leq i$.

We define $\pi_i = (x_i, R_{i-1}, R_{pred(i)})$. Note that π is the opening of the commitment R_k.

If $j = i$, the witness consists of the values of R_{i-1} and $R_{pred(i)}$, necessary to open the commitment for the accumulator R_i.

If $j < i$, then we recursively generate the witness starting from either R_{i-1} or $R_{pred(i)}$, whichever is "most convenient". That is, we use always use the witness for $R_{pred(i)}$, unless $pred(i) < j$; in that case, we concatenate π_i with the witness for R_{i-1}.

Algorithm 2: Generating a witness for x_j starting from i

```
MemWitCreate(i, j)
    π ←(x_i, R_{i-1}, R_{pred(i)})
    if i = j then
        return π
    else
        if pred(i) < j then return π || MemWitCreate(i − 1, j)
        else return π || MemWitCreate(pred(i), j)
    end
```

Remark: the above algorithm generates the witness for element x_j, starting from the accumulator at time i. The witness for element x_j for the current accumulator is therefore generated by calling MemWitCreate(k, j).

Witness verification Given a witness π for the value of x_j given the value of R_i, for some positive $j \leq i$, the verifier first checks that the hash of the concatenation of the first three elements of π is indeed R_i. Let π' be equal to π after removing the first 3 elements of π, and let x, P, Q the first three elements of π, respectively. Then:

- If $j = i$, it checks that the first element of π is indeed x_j.
- If $j < i$ and $pred(i) < j$, it recursively verifies that π' is a valid proof of the value of x_j given that the value of R_{i-1} is P.

- Otherwise, if $j < i$ and $pred(i) \geq j$, it recursively verifies that π' is a valid proof of the value of x_j given that the value of $R_{pred(i)}$ is Q.

Algorithm 3: Verifying that π is a valid proof that $x_j = x$, given R_i.

```
VerMem(R_i, i, j, π, x)
    Parse w as x', P, Q, w', where x', P, Q are the first three elements of
     π, and π' is the rest of the list.
    if H(x'||P||Q) ≠ R_i then
        return 0
    end
    if i = j then
        if x = x' then return 1 else return 0
    else
        if pred(i) < j then return VerMem(P, i − 1, j, π', x)
        else return VerMem(Q, pred(i), j, π', x)
    end
```

Similarly to the witness generation function, the VerMem verifies that π is a valid proof that the given element x is indeed element the j-th element added to the accumulator, given that the accumulator's value for some $i \geq j$ is R_i. To verify a proof against the current accumulator after k elements have been added, one calls the VerMem function with first parameters R_k and k, respectively.

Witness update As there is no trapdoor and the execution of the Add function is public, any party that was observing the accumulator can generate the witness for a specific x_k; in that case, it does not need more than a number of elements proportional to the size of the proof itself.

With a more careful algorithm, the running time of the witness update can be slightly improved. In fact, the old witness and the new witness are identical, except for a prefix of length at most $O(\log n)$ that is replaced from the old witness.

4.1 Analysis

Theorem 1. *The construction defined above is an additive accumulator with* $O(1)$ *insertion cost, size of the accumulator state* $O(\log n)$*, and length of the*

proof $O(\log^2 n)$. *The cost of a witness update is* $O(\log n)$.

Remark. *More precisely, the size of the proof for element* $n-k$ *is* $O(\log n \log k)$.

5 Second scheme: Merkleized back-links

In this section we sketch the modifications to the above scheme to obtain a different trade-off in running time; namely, we modify the commitment for the k-th element to include the roots for $O(\log n)$ previous elements instead of one. This will allow to obtain proof size $O(\log n \log\log n)$, at the cost of a slightly worse insertion cost $O(\log\log n)$.

The main observation is that since $pred(k)$ is the number obtained by zero-ing the least significant 1 of k, in the above construction we end up revealing a sequence of past commitments corresponding to the sequence $pred(k), pred(pred(k)), \ldots$ and so on, until we reach a number k' such that $pred(k')$ is too small. Moreover, observe that all the commitments in the sequence $R_k, R_{pred(k)}, \ldots, R_{k'}$ are indeed available as elements of the array S.

Based on these observations, we modify the above construction as follows:

- The accumulator's operator stores and maintains in its state the entire Merkle tree that has the elements of the array S as leafs; let M_k be the value in the root of this Merkle tree after k elements have been added (we also define $M_0 = NIL$).
- We define $R_k := H(x_k||M_{k-1})$, in order to commit to both the element x_k and the state of the whole array S prior to the insertion of x_k.

As the size of the array S is $\lfloor \log k \rfloor$, it follows that it is possible to reveal a specific leaf of the Merkle tree by revealing $O(\log\log k)$ hashes. This allows to modify the witness generation (and similarly the witness verification) algorithm as follows: when $pred(i) \geq j$, instead of recurring over $pred(i)$, we recur over $pred_t(i)$ for the largest t such that $pred_t(i) \geq j$.

The cost of updating the Merkle tree when an element of the array S is modified (or appended) is $O(\log\log k)$ as well, since only the hashes from a leaf to the root need to be recomputed by the accumulator operator.

The accumulator manager keeps an additive Merkle Tree that is initially empty. See Section A in the Appendix for details on the additive Merkle trees.

Element insertion Adding an element is very similar to algorithm 1, the only substantial difference being that the array S is replaced with an additive Merkle tree that contains as leafs exactly the same elements that the array S stored in the first construction. The value of the accumulator is computed as $accValue$ and is always equal to the leaf of the MerkleTree with index $zeros(k)$.

Algorithm 4: Inserting element x

ADD(x)
  $M \leftarrow merkleTree$.root
  $k \leftarrow$ k + 1
  $accValue \leftarrow \mathrm{H}(x||M)$
  merkleTree.set($zeros(k), accValue$)

Witness generation We assume that the algorithm for witness generation has access to each past Merkle tree used by the structure; namely, $merkleTree_i$ is the Merkle Tree after i-th element was added. Note that if all the values of R_i for each i are stored, then $merkleTree_i$ can be computed in logarithmic time. This can be obtained without storing any additional value other than each R_i.[2]

Similarly to Algorithm 2, the witness for x_j starting from R_i is obtained by concatenating the opening of the commitment for R_i with the Merkle proof of the appropriate leaf of $merkleTree_{i-1}$ equal to a value i' appropriately chosen, and then (recursively) the witness for such value i'. The recursion terminates if $i = j$.

Remark. *A prover that only wishes to provide proof for an element i does not need to store each of those Merkle trees, but only $O(\log n)$ of them.*

[2]We believe this is efficient enough for many applications. At the cost of increased memory occupation of $O(\log \log n)$ hashes per element and a more complex implementation of the additive Merkle trees it is possible to memorize each of those Merkle trees and avoid such recomputation.

Algorithm 5: Generating a witness for x_j starting from i

```
MemWitCreate(i, j)
    π ← (x_i, merkleTree_{i−1}.root)
    if i > j then
        i' ← rpred(j, i − 1)
        merkleProof ← merkleTree_{i−1}.proveLeaf(zeros(i'))
        w ← w||merkleProof||MemWitCreate(i', j)
    end
    return π
```

Witness verification The algorithm for a witness verification mirrors Algorithm 5: it first verifies that the first two elements of the witness correctly hash to the expected value. The second element of the witness must be the Merkle root of the tree after x_{i-1} was inserted; therefore, if $i \neq j$, the Merkle proof for the appropriate leaf which contains $R_{i'}$ to the $i' = (i-1, j)$ is verified; the remaining part of the witness is a must be a correct witness for x_j starting from the value of $R_{i'}$, which is then verified recursively.

The pseudocode in Algorithm 6 describes the idea in more detail.

Algorithm 6: Verifying that w is a valid proof that $x_j = x$, given R_i.

```
VerMem(R_i, i, j, π, x)
    Parse π as x', M, π', where x', M are the first two elements of π,
     and π' is the rest of the list.
    if H(x'||M) ≠ R_i then
        return 0
    end
    if i = j then
        if x = x' then return 1 else return 0
    else
        i' ← rpred(i − 1, j)
        leafIndex ← zeros(i')
        Parse π' as L, π_M, π'', where L is one hash, π_M is a Merkle
         proof for L, and π'' is the rest of the list.
        if merkleProofVerify(M, L, leafIndex, π_M) = 1 then
            return VerMem(L, i', j, π'', x)
        else
            return 0
        end
    end
```

Theorem 2. *There is an additive accumulator that has with insertion cost $O(\log \log n)$ and proof size $O(\log n \log \log n)$ for an accumulator with n elements. The cost of a witness update is $O(\log \log n)$.*

Remark. *More precisely, the size of the proof for element $n-k$ is $O(\log k \log \log n)$.*

6 Remarks and future extensions

The following ideas are still underdeveloped.

- Both the above schemes can be modified to support simultaneous insertion of a batch of multiple elements at a cost significantly lower than what would be paid by inserting the elements one by one.
- If the whole array S is considered as accumulator's value (instead of a single hash), the construction also has a property analogous to *low update frequency* as defined in [RY16].

7 Acknowledgments

The author would like to thank Chris Buckland, Sergi Delgado Segura and Patrick McCorry for many valuable comments and discussions on this work.

References

[ABC+15] Jae Hyun Ahn, Dan Boneh, Jan Camenisch, Susan Hohenberger, Abhi Shelat, and Brent Waters. Computing on authenticated data. *Journal of Cryptology*, 28(2):351–395, 2015.

[AGT01] Aris Anagnostopoulos, Michael T Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security*, pages 379–393. Springer, 2001.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[BCD+17] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 301–315. IEEE, 2017.

[BDM93] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[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*, pages 9–17, 2000.

[CJ10] Sébastien Canard and Amandine Jambert. On extended sanitizable signature schemes. In *Cryptographers Track at the RSA Conference*, pages 179–194. Springer, 2010.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual International Cryptology Conference*, pages 61–76. Springer, 2002.

[CW11] Scott A Crosby and Dan S Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):1–30, 2011.

[Dry19] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the bitcoin utxo set. 2019.

[FVY14] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A decentralized public key infrastructure with identity retention. *IACR Cryptology ePrint Archive*, 2014:803, 2014.

[GGM14] Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. In *NDSS*. Citeseer, 2014.

[GTH02] Michael T Goodrich, Roberto Tamassia, and Jasminka Hasić. An efficient dynamic and distributed cryptographic accumulator. In *International Conference on Information Security*, pages 372–388. Springer, 2002.

[MBB+18] Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. Pisa: Arbitration outsourcing for state channels. *IACR Cryptology ePrint Archive*, 2018:582, 2018.

[Mer80] Ralph C Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122. IEEE, 1980.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE, 2013.

[MHKS14] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. *ACM SIGPLAN Notices*, 49(1):411–423, 2014.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2008.

[NN98] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In *USENIX Security Symposium*. Citeseer, 1998.

[NN00] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on selected areas in communications*, 18(4):561–570, 2000.

[PS14] Henrich C Pöhls and Kai Samelin. On updatable redactable signatures. In *International Conference on Applied Cryptography and Network Security*, pages 457–475. Springer, 2014.

[RY16] Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed pki. In *International Conference on Security and Cryptography for Networks*, pages 292–309. Springer, 2016.

[SALY17] Shi-Feng Sun, Man Ho Au, Joseph K Liu, and Tsz Hon Yuen. Ringct 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency monero. In *European Symposium on Research in Computer Security*, pages 456–474. Springer, 2017.

[Sla12] Daniel Slamanig. Dynamic accumulator based discretionary access control for outsourced storage with unlinkable access. In *International Conference on Financial Cryptography and Data Security*, pages 215–222. Springer, 2012.

[STS99] Tomas Sander and Amnon Ta-Shma. Flow control: a new approach for anonymity control in electronic cash systems. In *International Conference on Financial Cryptography*, pages 46–61. Springer, 1999.

A Dynamic Merkle Trees

Originally proposed in [Mer80], Merkle trees (also called *hash trees*) allow to succinctly commit to a large set S while still being able to produce short

proofs of membership (with size logarithmic in $|S|$) that can be readily verified with only the knowledge of a single hash (the *Merkle root*). While the initial construction was for a static set S, it is a versatile construction that has been adapted to a number of dynamic settings as well, where the set S can be modified over time. See for example the works of [NN00], [AGT01], [CW11], [MHKS14] and many others.

For the sake of completeness, here we sketch a simple variation of dynamic Merkle trees that fits the requirements of the constructions in this paper.

Our trees commit to an ordered zero-indexed vector $S = \{x_0, \ldots, x_{k-1}\}$, and allows the following operations on S:

- Change the value of the i-th element of S, where $0 \leq i < |S|$
- Append a new element at index $|S|$ (therefore, the size of S is increased by one).

At any time, a prover (that holds S in memory, together with any necessary additional value) can produce a witness w of size $\log |S|$ that the value of element i is x_i. A verifier can check the correctness of the proof by the knowledge of w, the Merkle root, and the size of S.

More formally, the $merkleTree$ used in the construction in section 5 is defined as follows:

- $merkleTree.root$ is the value of the current Merkle root (or a special value NIL if the tree is empty)
- $merkleTree.set(i, x)$, where $0 \leq i \leq |S|$ modifies the set S: if $i < |S|$, then the value of the element of S with index i is set to x. If $i = |S|$, then a new element with value x is appended to S.
- $merkleTree.proveLeaf(i)$, where $0 \leq i < |S|$ returns the witness for the element with index i.
- MerkleProofVerify(M, x, i, w): given a hash M and an element x, returns 1 if w is a valid witness that the tree with Merkle root M has x as element with index i; returns 0 otherwise.

Note that, for notational convenience, adding a new element at index $|S|$ or changing the value of an existing element are combined into the $merkleTree.set$ operation.

We store a Merkle tree for a vector $S = \{x_0, \ldots, x_{k-1}\}$ as a complete binary tree of depth $D := \lceil \log k \rceil$. Such a tree has $2^D \geq k$ leaf nodes, and we store each of the k nodes of as the value of the first k leafs of the tree, in order. Any other leaf has the special value NIL.

Each internal node that has left child with value l and right child with value r contains $H(l|r)$, that is, the hash of the concatenation of the value of the two children. See Figure 3 for an example of a Merkle tree corresponding to a vector S with 6 leafs.

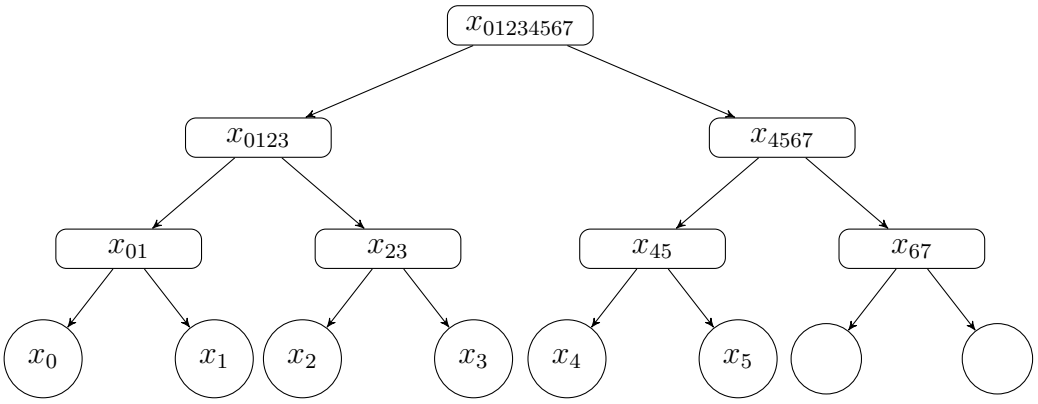

Figure 3: A depth 3 Merkle tree that contains 6 leaf nodes with values x_0, x_5 and two empty nodes containing NIL.

When $merkleTree.set(i, x)$ is called for some $i < 2^D$, then the value of the i-th leaf (zero-indexed) is set to x; then the value of each of the D ancestor of that leaf are recomputed.

When $merkleTree.set(i, x)$ is called for some $i = 2^D$, then the tree is extended by adding additional 2^D leafs, and then we proceed as before.

$merkleTree.proveLeaf(i)$ and MerkleProofVerify(M, x, i, w) work as in normal Merkle trees: $merkleTree.proveLeaf(i)$ produces the list of all the siblings of nodes in the path that goes from the i-th leaf (zero-indexed) to the root (excluding the root that has no sibling); symmetrically, MerkleProofVerify(M, x, i, w) performs the same computation in reverse to check if the root hash M matches the one produced with the Merkle proof.

Theorem 3. *The construction above has the following properties:*

- *the value of* $merkleTree.root$ *and the witness produced by executing* $merkleTree.proveLeaf(i)$ *for a fixed* i *are deterministic and only depend on the current set* S*, regardless of the past changes.*
- *the length of the witness produced by* $merkleTree.proveLeaf(i)$ *is at most* $\lfloor \log |S| \rfloor$ *hashes.*
- *the computational cost of* $merkleTree.set$ *is* $O(\log |S|)$.