Merkle Trees Optimized for Stateless Clients in Bitcoin

Bolton Bailey and Suryanarayana Sankagiri

University of Illinois Urbana-Champaign, Champaign IL, USA

Abstract. The ever-growing size of the Bitcoin UTXO state is a factor preventing nodes with limited storage capacity from validating transactions. Cryptographic accumulators, such as Merkle trees, offer a viable solution to the problem. Full nodes create a Merkle tree from the UTXO set, while stateless nodes merely store the root of the Merkle tree. When provided with a proof, stateless nodes can verify that a transaction's inputs belong to the UTXO set. In this work, we present a systematic study of Merkle tree based accumulators, with a focus on factors that reduce the proof size. Based on the observation that UTXOs typically have a short lifetime, we propose that recent UTXOs be co-located in the tree. When proofs for different transactions are batched, such a design reduces the per-transaction proof size. We provide details of our implementation of this idea, describing certain optimizations that further reduce the proof size in practice. On Bitcoin data before August 2019, we show that our design achieves a 4.6x reduction in proof size vis-a-vis UTREEXO [10], which is a different Merkle-tree based system designed to support stateless nodes.

1 Introduction

Bitcoin and other cryptocurrencies are peer-to-peer systems, designed to maintain an ordered ledger of transactions. Peers participate in a blockchain protocol to come to a consensus on the *state* of the ledger. Roughly speaking, the state specifies the amount of currency each public key has in the system. A peer that stores the state can validate a transaction, i.e., check whether it double spends a coin, or whether it leads to some account balance going negative. In Bitcoin, the state of the system consists of the set of unspent transaction outputs (UTXOs) at any given time. Every transaction spends some UTXOs (except coinbase transactions), and generates new ones in turn. The state is updated after each block by deleting the spent UTXOs and adding the newly generated ones.

The size of the state in any cryptocurrency system can be quite large, making it difficult for peers to store the entire state. For example, in Bitcoin, the state currently contains about 70 million UTXOs, which requires about four gigabytes (GB) to store, and is expected to keep growing with time. This is a major scalability issue for cryptocurrencies. A *stateless cryptocurrency* system, built around a *cryptographic accumulator scheme*, is a promising solution to this issue. It works on the principle that storing the entire state is not a necessity for a peer to verify transactions; it can do so if it is provided with a *proof* that the transaction is consistent with the current state. In such a system, a *stateless client* merely stores an *accumulator*, which is a compact representation of the state. When provided with a *witness* of a particular UTXO, it can prove for itself that the UTXO is part of the state. A computationally bounded adversary cannot generate a witness for a UTXO that is not part of the state.

A cryptographic accumulator scheme can be thought of as a primitive that is a generalization of a hash function. Like a hash, it provides a compact representation of a set of values, which is also binding. In addition, it also provides the means of generating short witnesses for each element in the set (or a subset of elements); a hash function does not have this feature. The witnesses and the accumulator together act as a proof that the element is part of the set. For a formal definition of cryptographic accumulators and a survey of different designs, see [8]. An accumulator scheme is *dynamic* if it supports additions and deletions from the accumulated set. Specifically, given only the witness for a particular element and the accumulator, a party can compute the accumulator of the new set with the element deleted. Moreover, given the accumulator, a party can compute the accumulator of the new set with any element added. To enable a stateless cryptocurrency, we need dynamic accumulators.

In this work, we study the design of Merkle tree based accumulator schemes (henceforth, simply referred to as Merkle trees) for stateless clients in Bitcoin. Here, the accumulated set is the set of UTXOs, i.e., the state of the system. A Merkle tree is (typically) a binary tree, with each tree node containing hash pointers to its children. The leaf nodes contain hash pointers to the UTXOs. The hash of the root node of the tree is the accumulator, which stateless peers store. Any change in the UTXO set is reflected in the accumulator. The witness (or proof) of a UTXO consists of the branch of tree nodes from the corresponding leaf to the root node. A stateless peer verifies the proof of a UTXO by checking that the leaf node has a hash pointer to the UTXO, the root node's hash equals the accumulator, and the internal hashes in the proof are consistent. More generally, the proof for a subset of UTXOs is a sub-tree of the entire Merkle tree. We refer the reader to Chapter 1 of [16] for a detailed description of Merkle trees.

The use of Merkle trees as accumulators is well known, and the idea of using them to enable stateless cryptocurrencies has also been around since at least 2010 [15]. Recently, the UTREEXO project [10] applied this idea to real Bitcoin data. Their proposed system consists of *bridge servers* and *stateless clients*. The bridge servers store the whole Merkle tree, while stateless clients store a small number of hashes of the tree nodes. For every block that arrives in the system, a bridge server provides stateless clients the required sub-tree to prove that the UTXOs consumed by transactions in the block are part of the state. An important aspect of UTREEXO's design is that the size of the witnesses is small enough for it to be viable in practice. This is remarkable given that theoretically speaking, proof sizes for Merkle trees can be quite large; in fact, this is cited as a major drawback of such schemes [3]. The average proof size required per block is an important metric to minimize, as it is a communication and storage overhead for a stateless cryptocurrency system.

1.1 Our Contributions

In this work, we explore whether it is possible to further reduce the proof size required in a stateless Bitcoin system, by altering the construction of the Merkle tree. We identify two factors that influence the batched proof size. The first factor is the height of the Merkle tree. The height of a balanced (binary) tree with n elements is $\Theta(\log n)$. The total proof size for k elements is thus $O(k \log n)$. The design of UTREEXO keeps the tree balanced [10]. The second factor is the location of the UTXOs for which a batch proof is sought. When considering multiple elements, their individual witnesses (proofs) may have some overlapping tree nodes. These nodes need not be repeated in the proof data provided for a single block. The batched proof size is minimized if the leaves whose proof we require are co-located in the tree.

While it is possible to design the Merkle tree such that it is balanced with certainty, one cannot say the same when it comes to co-locating UTXOs that will be spent together. Indeed, when a new UTXO is added to the tree, it may be spent at any time in the future. Thus, the worst-case proof size for proving a batch of k among n UTXOs cannot be better than $\Theta(k \log n)$. The main insight of our work is that if we analyze *average-case proof sizes* instead of worst-case proof sizes, we can build a system that does much better in practice.

Empirical analysis of Bitcoin data reveals that most UTXOs have very short lifetimes (see Figure 2 in [10]). Stated differently, a large fraction of the UTXOs being spent in a block are likely to be recently added. By arranging UTXOs in the Merkle tree in the same order in which they arrive in the system, it is likely that a large fraction of the UTXOs being spent are co-located. This simple design leads to a significant reduction in the batch proof size, on average.

To elaborate, our contributions are twofold:

– We develop a probabilistic model where the lifetimes of UTXOs have a power-law distribution with index α, consistent with their empirically observed statistics. Under this model, we prove that the batch proof size required per block is $O(d + k^{\alpha})$, where d is the depth of the tree, and k is the number of UTXOs added per block. This is proven in Section 2.
– We build a system implementation of our idea in Go, based on the open-source UTREEXO project [10]. In particular, we implement a Merkle Trie, wherein UTXOs are kept in the order in which they arrive. We demonstrate that our design significantly outperforms UTREEXO in terms of the average proof-size required per block (see Figure 1). The implementation details of our system are given in Section 3.

We now take a closer look at the extent to which co-location helps in reducing the batch proof size. Consider the examples given in Figure 2. Both images highlight the branches of the Merkle tree required to prove the existence of

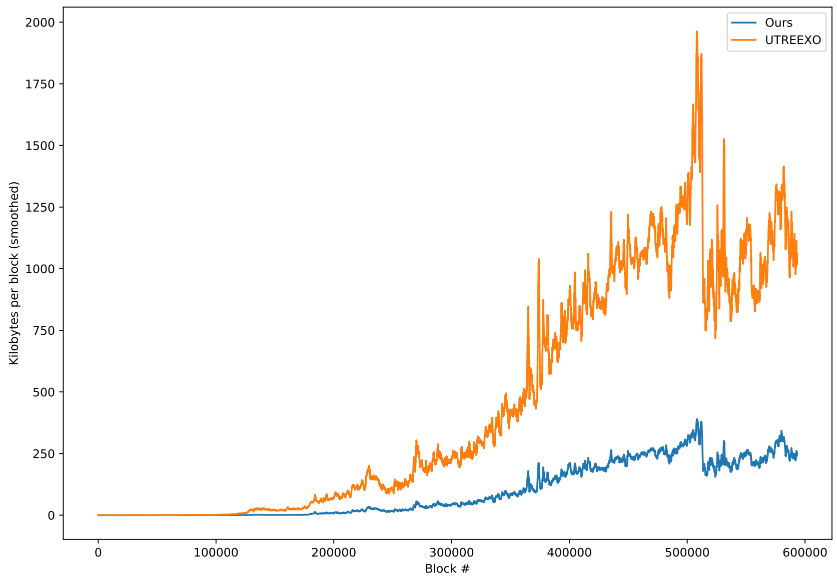

Fig. 1. The size of the proof data required to validate each block in Bitcoin, shown for our implementation and UTREEXO. The plot is smoothed by taking a rolling average over 1000 blocks. The plot is shown for blocks up to height 591, 000, which corresponds to August 2019. Totaled over this duration, the UTREEXO proof size is 269.3GB, whereas ours is 58.2GB, about a 4.6 factor smaller than UTREEXO. Note that this data does not include hashes of TXOs included in blocks.

three UTXOs. In the first, the relevant UTXOs are spaced apart, while in the second, they are co-located. One can observe that the total number of nodes to be included in the proof is larger when the UTXOs are spaced farther apart. Indeed, when all k UTXOs being proven are adjacent and the tree is balanced with depth d, the proof size reduces to $O(d+k)$. This is because the smallest sub-tree containing the k pertinent leaves has size $O(k)$, and the branch leading down to the sub-tree is of length at most d. The remarkable and somewhat surprising result of Section 2 is that we get nearly the same order complexity even when the UTXOs being proven are randomly chosen (with appropriate assumptions on the distribution).

Remark 1. The term 'node' is reserved for a vertex in the Merkle tree, which contains hash pointers to its children. For individuals participating in the system, we use the terms 'peer', 'server', 'client'.

1.2 Related Work

The concept of a stateless cryptocurrency seems to originate in the bitcointalk.org forum [1]. Miller was the first to suggest red-black Merkle trees as an accumulator for bitcoin state in [2].

The main point of comparison for this paper is UTREEXO [10], the project off of which our code is based. The UTREEXO paper considers a few techniques that we did not, including client-side caching of Merkle tree data to reduce proof size and including the block hash in the leaves to harden against collision attacks. An interesting direction for future work might be to see if these concepts can be ported to our system.

Non-Hash Based Accumulators Other forms of cryptographic accumulators exist that are not based on Merkle trees. These include hidden order group accumulators, which depend on the RSA assumption or groups of unknown order. This line of work originated with [1]. Progress has been made on these types of accumulators on several fronts: [4] shows how to create dynamic accumulators under this scheme. [13] allowed non-membership proofs. [3] showed how to batch groups of operations on these accumulators to make the proofs smaller. This work, as well as [5], expanded these ideas to vector commitments, which allow for commitments to key-value stores. A downside of many of these constructions is that to instantiate them over an RSA group requires a trusted setup. To avoid this, [14] and [9] show how constructions of class groups can be used that do not require this setup - although operations in class groups use more expensive number-theoretic operations. [6] avoid this in a different way - they show how a multi-party computation to generate an RSA modulus which remains secure if any of the participants are honest.

[1] `https://bitcointalk.org/index.php?topic=505.0`
[2] `https://bitcointalk.org/index.php?topic=101734.0`

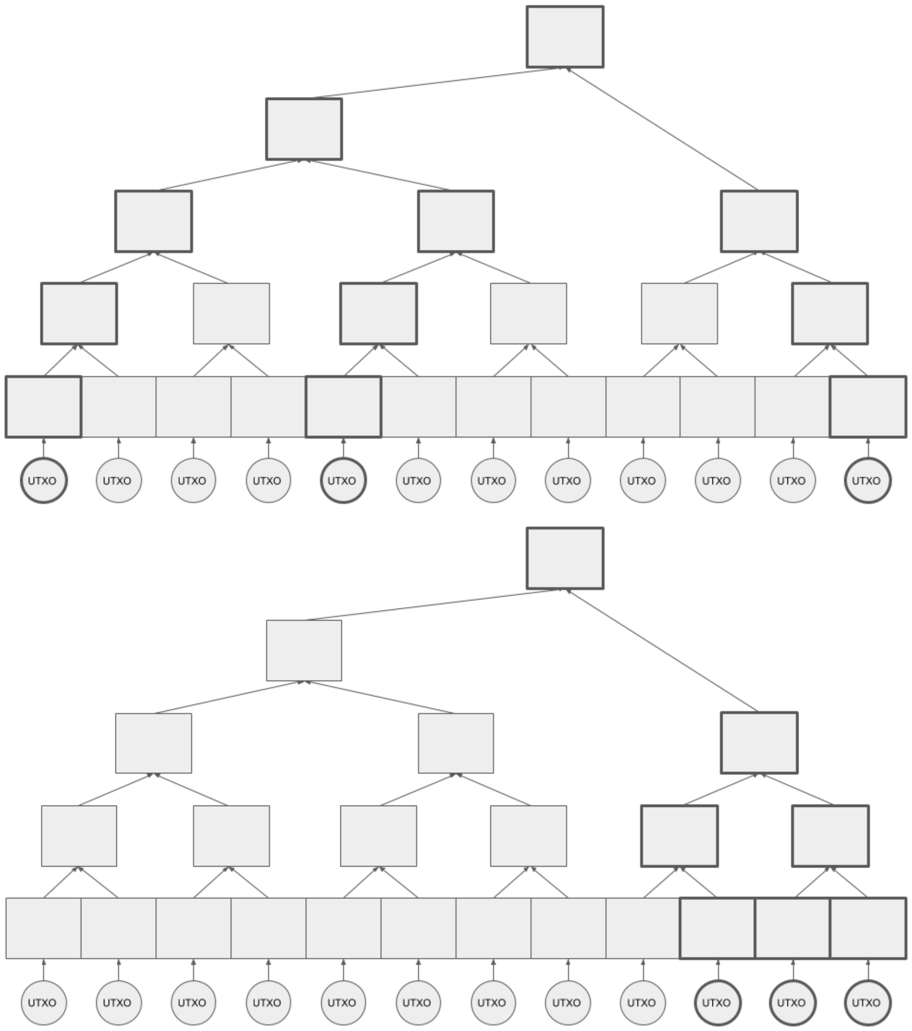

Fig. 2. Two examples, one where we require the proof of three elements that are spread out over the Merkle tree, one where the three elements are close together. The figure demonstrates the value of having proved elements close together in the tree: The total proof size is less for the bottom tree.

Other accumulators are based on bilinear curve pairings [11] [17]. These types of accumulators also have very small proof sizes, but they also rely on trusted setups.

Ultimately, these accumulators have benefits over Merkle trees in that they have constant proof sizes, but they have some drawbacks: They require big-integer arithmetic operations that may take time, they are based on more recent cryptographic assumptions, and they are not post-quantum secure.

2 Average-Case Hash Accumulator Complexity

In this section, we will describe constructions for accumulators. We will then introduce an idealized probabilistic model for arrival of and sending of transactions. We will then prove some theorems about the average case performance of various accumulators under this model.

2.1 Accumulators

As stated in the introduction, our approach to constructing practically efficient Merkle tree accumulators is to, as much as possible, cause UTXOs that are spent in the same block to be co-located in the tree. One observation made by UTREEXO was that UTXO durations in the blockchain tend to follow a power-law pattern, which results in the most recently added UTXOs being the most likely to be spent in any block. With this is mind, it is prudent to consider designs that will keep all UTXOs in the tree in the order in which they were inserted, so that the most recently added UTXOs will all be co-located in the right side of the tree. We discuss a few designs for Merkle trees and the extent to which they accomplish this.

UTREEXO The UTREEXO accumulator maintains a list of at most $\log_2 n$ perfect binary Merkle trees containing the UTXOs. When a batch of elements is removed, each binary tree decomposes into a set of smaller trees, which are then recombined until there is at most one tree of each depth.

UTREEXO has the benefit that there is no overhead in the form of tree nodes containing further data beyond the hashes of the children. However, it does not keep the hashes of UTXOs in the order in which they are inserted, since the recombination algorithm will necessarily sometimes switch the order of subtrees. It is still the case that the new UTXOs in a block will initially be co-located immediately after being added by virtue of being added in the same tree - but as time goes on, the recombination process may take them apart. How fast this happens depends on the details of the recombination algorithm, which can potentially be implemented in multiple ways, and which is described in the appendix of [10].

Red-Black Merkle Trees By red-black Merkle tree, we mean a red-black tree [12] in which all references in internal nodes to other nodes are replaced by cryptographic hashes of the contents of those other nodes. The elements of the accumulated set are stored in the leaves of the tree. To use a red-black Merkle tree as an accumulator for the Bitcoin UTXO set, a counter is maintained that tracks the total number of TXOs ever added to the state, and each new TXO is keyed with the next number. To perform a batch addition of k UTXOs to the tree, one first creates a red-black tree out of the k new leaves, then joins this with the existing tree. Note that the time to add a new collection of k elements to the right of an n leaf red-black tree is $O(\log(n))$ (see [2] Lemma 2) and the proof size is therefore also $O(\log(n))$.

As we have mentioned, Miller introduced the idea of using a red-black Merkle tree as an accumulator for Bitcoin UTXOs in a forum post. The benefit of red-black Merkle trees is that they are self-balancing and therefore have depth that is logarithmic in their size. They therefore achieve the worst case bound on batch proof size of $O(k \log n)$, where n is the state size. However, they have some drawbacks. One is that the set of nodes required to prove a collection of members of the accumulator is sometimes less than the set of nodes required to delete those members - the deletion process may require “rotations” which necessitate additional data. Another drawback is that, while the depth of the red-black tree is $O(\log n)$, the number of nodes in a tree of depth d can range from $2^{d/2}$ to 2^d, which makes analysis of the efficiency of the accumulator hard. Merkleizations of other self-balancing trees such as AVL trees could also be considered, but they have many of the same properties we will just focus on red-black trees here. An implementation of red-black Merkle trees exists [3].

Insertion-Order Indexed Merkle Tries A Merkle trie, or Merkle Patricia tree, is another cryptographic accumulator. Analogous to red-black Merkle trees, a Merkle trie is a trie [7] in which references in internal nodes to other nodes are replaced by hashes.

Merkle tries are used in the Ethereum protocol [18], which, unlike Bitcoin, uses an account model rather than a UTXO model. Nevertheless, we can adapt the data structure for use in tracking UTXOs in the same way as we do for the red-black Merkle tree: We maintain a counter of the number of UTXOs that have been added to the trie and assign numbers to each new UTXO using the counter.

A drawback of this scheme is that we do not have the same guarantee that the depth of the trie will be $O(\log(n))$ where n is the number of elements of the accumulated set as we do for the balanced trees. Instead, we have a max depth of $O(\log(N))$ where N is the total number of elements that have ever passed through the accumulator - for Bitcoin, the difference in these numbers is only about a factor of 10, so there is not a big difference in the logarithms of these values. We will show, under some assumptions, that this difference does not affect the asymptotics of the proof size in the average case. Nevertheless, the

[3] `https://github.com/amiller/redblackmerkle`

trie algorithm is less complex, and the trie seems to be more efficient in practice (see Section 3 for details). Tries also have the useful property that the collection of proofs of a batch can be used by a stateless client to delete that batch.

2.2 A Model for Transaction Durations

The theoretical work for this paper will frame a blockchain in terms of a random process of transaction outputs which enter and leave the blockchain at specific block numbers. Specifically, let $k_{i,t}$ be the number of t-duration-TXOs introduced in block i, (that is, the number of TXOs that were created in block i and spent in block $i+t$. Our theorems will depend on some idealized assumptions about the distribution of $k_{i,t}$:

Assumption 1 *The total number of TXOs entering the blockchain in a given block $k = \sum_{t=1}^{\infty} k_{i,t}$ is constant.*

This assumption reflects that fact that Bitcoin and other cryptocurrencies, by virtue of having a maximum block size, have a fixed cap on the number of TXOs that can be introduced in a block.

Assumption 2 *The duration for an individual TXO within a block is independent of other TXOs and is zeta-distributed. That is, the probability that a TXO will last t blocks is*

$$t^{-\alpha}/\zeta(\alpha)$$

for some $\alpha > 1$ fixed across all TXOs and blocks.

This assumption reflects the observation that there appears to be a power-law effect in the number of t-duration-TXOs [10].

Note that under these assumptions, taking $1 < \alpha < 2$, the state size of the blockchain will grow without bound (but sublinearly).

Theorem 1. *Under assumptions 1 and 2, the expected size of the state at block height B is $\approx \frac{k}{\zeta(\alpha)(2-\alpha)} \cdot B^{2-\alpha}$*

Proof. The expected size of the state equals the sum of the expected numbers of UTXOs introduced in each block which remain at block B. Equivalently, it is the sum from 1 to B of the number of i duration UTXOs introduced in the last i blocks

$$\sum_{i=1}^{B} i \cdot k \cdot i^{-\alpha}/\zeta(\alpha) = k \cdot \sum_{i=1}^{B} i^{1-\alpha}/\zeta(\alpha)$$

And from Riemann sum approximations, we have

$$\frac{(B+1)^{2-\alpha}-1}{2-\alpha} = \int_{1}^{B+1} i^{1-\alpha} di \leq \sum_{i=1}^{B} i^{1-\alpha} \leq 1 + \int_{1}^{B} i^{1-\alpha} di = 1 + \frac{B^{2-\alpha}-1}{2-\alpha}$$

□

2.3 Average-Case Asymptotics for Insertion-Order Indexed Merkle Tries

We will first introduce a lemma about the probability of a TXO being spent based on its rank among all TXOs ever added to the blockchain.

Lemma 1. *Consider the blockchain under assumptions 1 and 2 at some block b. Let x_i be the ith most recently added TXO (including those added then deleted). Then the probability x_i is spent in the current block is $\leq \frac{k^\alpha}{i^\alpha \zeta(\alpha)}$.*

Proof. By Assumption 1, k TXOs are added per block, so x_i was necessarily introduced $\lceil i/k \rceil$ blocks before the current block. By Assumption 2, the probability of such a TXO being spent in the current block is

$$\Pr[x \text{ is spent}] = \frac{\lceil i/k \rceil^{-\alpha}}{\zeta(\alpha)} \leq \frac{(i/k)^{-\alpha}}{\zeta(\alpha)}$$

□

We will now state our main result, which shows that in the limit as the state becomes large, the expected proof size is essentially the same that of a proof of a single element:

Theorem 2. *Let T be the insertion-order indexed Merkle trie (of depth d) of UTXOs in the blockchain just before block B. Then under assumptions 1 and 2, the expected size of the subtree S consisting of the branches to all TXOs that are spent in block B is $\leq d + O(k^\alpha)$, where the O hides factors that depend on α.*

Proof. As a simplification, consider a trie T' consisting of all UTXOs ever added to the accumulator, without removal. Since every node in S appears in T' at the same location, it suffices to prove the bound on the size of S', the union of branches to spent TXOs in this complete trie.

For $1 \leq i \leq d$ define n_i to be the rightmost node in T' such that the subtree rooted at that node has depth i. Note that there exists such a subtree for every i: By induction on the depth of the tree, n_d is the root of the tree, and either the left or right subtree is depth $d-1$. The parent of every n_i other than the root is an n_i, since if the subtree rooted at this node has depth j, there cannot be a subtree to the right of it with depth j, or else there would be a subtree of that with depth i, contradicting the fact that n_i is the root of the rightmost depth i subtree.

We can therefore bound the size of S' by splitting T' into disjoint subsets: Consider the partition of T consisting of the set $\{n_i, 1 \leq i \leq d\}$ and then $T_1, T_2, \ldots, T_d$, where T_i is the set of nodes that are in the subtree rooted at n_i, but not in any subtree rooted at n_j for $j < i$.

$$\begin{aligned}
\mathbb{E}[|S|] &\leq \mathbb{E}[|S'|] \\
&\leq d + \sum_{i=1}^{d} \mathbb{E}[|S' \cap T_i|].
\end{aligned}$$

Since the branches in T_i in the subtree are of height at most the depth of the subtree, we get

$$\leq d + \sum_{i=1}^{d} i \cdot \mathbb{E}[\# \text{ spent leaves in } T_i|].$$

We can loosen this to put it in terms of the maximum number of leaves in T_i and maximum probability of leaf inclusion

$$\leq d + \sum_{i=1}^{d} i \cdot 2^i \cdot \max_{x \in T_i} \Pr[x \text{ is spent in current block}]$$

The leaves of T_i must be entirely to the right of each element of the subtree at n_{i-1}. The left subtree of n_{i-1} has at least 2^{i-2} leaves, since for the node to exist in the complete trie, its left child must be a complete perfect binary tree of depth 2^{i-2}. Therefore, there are at least $\frac{1}{4} \cdot 2^i$ leaves to the right of each leaf in T_i. We can therefore bound the probability of one of these leaves being spent using lemma 1.

$$\begin{aligned}
\mathbb{E}[|S|] &\leq d + \sum_{i=1}^{d} i \cdot 2^i \cdot \max_{x \in T_i} \Pr[x \text{ is spent in current block}] \\
&\leq d + \sum_{i=1}^{d} i \cdot 2^i \cdot \frac{k^\alpha}{(\frac{1}{4} \cdot 2^i)^\alpha \zeta(\alpha)} \\
&= d + \frac{4^\alpha k^\alpha}{\zeta(\alpha)} \sum_{i=1}^{d} i \cdot (2^{1-\alpha})^i \\
&\leq d + \frac{4^\alpha k^\alpha}{\zeta(\alpha)} \sum_{i=1}^{\infty} i \cdot (2^{1-\alpha})^i \\
&= d + \frac{4^\alpha k^\alpha}{\zeta(\alpha)} \cdot \frac{2^{\alpha+1}}{(2^\alpha - 2)^2}
\end{aligned}$$

2.4 Mixed Average-case Adversarial Setting

It is worth asking what happens if the Merkle Trie accumulator is attacked by an adversary who wishes to increase the size of the proofs by spamming the blockchain. To consider this case, we consider a modification of our previous assumptions:

Assumption 3 *In any block, the number of TXOs introduced is $k = k_a + k_r$, where k_r are chosen according to a zeta distribution, and the other k_a are chosen adversarially.*

We see that even if some of the transactions are adversarial, this only hurts the performance of the algorithm by an amount proportional to the amount of adversarial power.

Theorem 3. *Let T be the insertion-order indexed Merkle trie (of depth d) of UTXOs in the blockchain just before block B. Then under assumption 3, the expected size of the subtree S consisting of the branches to all TXOs that are spent in block B is $\leq d + O(k^\alpha) + dk_a$, where the O hides factors that depend on α.*

Proof. Following the proof of theorem 2, we see that for the expected size of the subset of S associated with the zeta-distributed TXOs, the same bound of $d + \frac{4^\alpha k^\alpha}{\zeta(\alpha)} \cdot \frac{2^{\alpha+1}}{(2^\alpha-2)^2}$ applies. The only remaining nodes to account for are in the branches associated with adversarial TXOs, which can number at most dk_a, the height of the tree times the number of such branches.

3 Practical Implementation

In this section, we describe the details of our Merkle trie accumulator, to be used in a stateless Bitcoin system. We first specify the exact construction of the trie, and what parts are stored by bridge servers and stateless clients respectively. We then specify the operations performed by servers and clients upon receiving a block. In particular, we specify how the batch proof for a block is constructed.

Construction of a Merkle Trie In our design, all UTXOs are assigned a unique index, which is a 64-bit unsigned integer. The numbers are assigned to the UTXOs in the order in which they appear in the blockchain starting from 0. Given that the UTXOs are indexed, it is possible to assemble them into a Merkle trie. There are a variety of ways to create the node data structure [4] - our approach is as follows: The Merkle trie is a collection of *tree nodes*, each of which contains a left hash pointer, a right hash pointer, and a *prefix*, which represents the range of indices of all UTXOs below the node. A hash pointer is simply the SHA-256 hash of a piece of data, which serves as a means of recovering the data provided it is stored in a hash map. Tree nodes are either *internal nodes* or *leaf nodes*. For internal nodes, the hash pointers refer to other tree nodes, while for leaf nodes, both the left and right hash pointer refer to the same UTXO. Thus, a leaf node can be distinguished from an internal node by checking if its hash pointers are equal or not. There is a single root node, which does not have a parent. Thus, the collection of tree nodes forms a single binary tree. The total size of a tree node is 72 bytes, with 32 bytes for each of the hash pointers, and 8 bytes for the prefix.

We now elaborate on the role of the prefix, and how it is constructed. The prefix is represented as a 64-bit unsigned integer. For a leaf node, the prefix

[4] `https://ethresear.ch/t/binary-trie-format/7621/6`

represents the index of the corresponding UTXO. For an internal node, the prefix represents a range of the form $[k \cdot 2^i, (k+1) \cdot 2^i)$, for any non-negative integer k and positive integer i. The value of the prefix itself is $(2k+1) \cdot 2^i$. For example, the prefix 8 denotes the range $[0, 8)$, while the prefix 10 denotes the range $[4, 6)$. Note that any positive integer n can be written in the form $(2k+1) \cdot 2^i$ for a unique k, i. Furthermore, an internal node must have at least two unique UTXOs among its descendants, and thus covers a range whose width is at least two. Thus, the range of values below an internal node is represented compactly by a single integer, of 8 bytes.

A bridge server stores two hash maps: one that maps the hash of a UTXO to its index, and the other that maps the hash of a tree node to the tree node itself. In order to retrieve the branch of the tree leading to a target UTXO, a bridge server first retrieves the index of the target, and then descends the Merkle trie starting from the root node. At each node, it checks whether the index it is seeking is in the lesser half of the prefix range or the greater half. If the former, it retrieves the left child of the node; else, it retrieves the right child. In our implementation, we use a combination of RAM and disk memory to store both the hash maps. The reading and writing operations from RAM are much faster, but the total amount of RAM available is sometimes insufficient to store the entire tree. We populate the RAM according to a 'least recently used' rule. Whenever a new entry is created, it is written into the RAM portion of the hash map; if space is limited, the least recently used key-value pair is moved to the disk. A stateless client merely needs to store the hashes of the nodes along the right most branch of the Merkle trie, i.e., the branch from the root node to the UTXO with the highest index.

Construction and Verification of a Proof In order to verify that a particular UTXO is part of the state, a sufficient proof consists of all the tree nodes from the root node to the leaf node corresponding to the target UTXO. A stateless client, which holds the target UTXO and the accumulator, can verify that the leaf node has a hash pointer to the target, and each subsequent node points to the one preceding it, with the hash of the root node matching the accumulator. However, this is overkill; a stateless client does not need to know the hashes of the nodes along the path. It suffices for it to know the hashes of the siblings of all the nodes in the path, along with the prefix lengths of the nodes in the path. The nodes along the path can then be reconstructed based on this information. In particular, to reconstruct the leaf node, the stateless client only needs to know the index of the UTXO from the bridge server. Therefore, in our implementation, the proof consists of the index of the targeted UTXO, the prefix lengths of all nodes in the Merkle tree leading up to the target, and the hashes of the siblings of those nodes. Note that the number of prefixes equals the number of locations, and that they are provided in the same order in the proof. This suffices for a stateless client to reconstruct the whole branch of tree nodes and verify the proof.

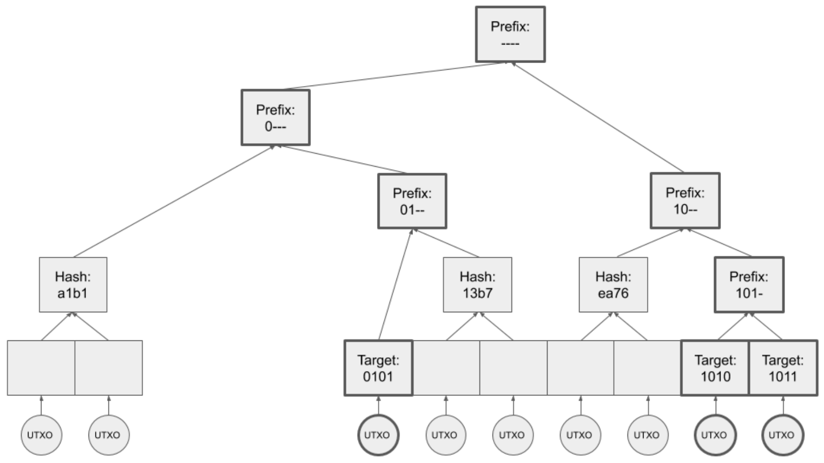

Fig. 3. An example of a batch proof in our implementation. The nodes in the branches leading down to the targets are shown in bold; the prefix lengths and targets from these nodes are given in the proof. In addition, the hashes of the relevant sibling nodes are specified. These nodes are labeled in the figure with their hash. In the real implementation, the hashes are 32 bytes long, and the prefixes/targets are 64 bits long.

Batching the proof for multiple targets provides even greater savings in the proof size. A tree node, which is a sibling of one of the nodes in the proof of a certain target, may lie on the proof branch of another target. In this case, the hash of this node needn't be included as part of the proof. This is best illustrated via the example shown in Figure 3. If we were only concerned with the proof for target 0101, we would have to include the hash of the right child of the root node. When batched together with additional proofs, this node will be reconstructed by the stateless client, and hence its hash needn't be included. In the example of Figure 3, the proof needs to include three targets, five prefix lengths and three hashes; without batching, nine prefix lengths and nine hashes would be required (three for each branch). We specify the hashes and the prefix lengths in the order they would be encountered in a depth-first search (DFS) for the targets. Having a specific order allows the stateless clients to reconstruct the tree nodes correctly.

In our design, a further reduction in proof size is achieved by using a standard compression function (we use zlib) to compress the batch proof data. In practice, we observe that this helps compress our proofs by $\sim 20 - 40\%$. This reduction is due to the list of targets and prefix lengths in the proof, which are fairly structured strings of bits. In contrast, the hashes in the proof are random strings of bits and cannot be compressed effectively.

Modifying the Merkle Trie In processing each block, the Merkle trie held by the bridge server is modified by first removing all UTXOs spent during the block, then adding all new UTXOs. Nodes are removed by descending the tree in DFS order, identifying all nodes that will be removed from the tree, removing them, then recomputing the hashes of all nodes whose descendants have changed. Thus, deletion is best implemented as a recursive function. Note that the tree nodes that are modified in the deletion process are exactly the set of tree nodes that are (implicitly) included in the proof. This allows the stateless clients to perform the same set of computations as a bridge server, and compute the intermediate accumulator of the state when UTXOs are deleted.

Nodes are added in a batch by composing new nodes into trees, then adding those trees to the existing trie as subtrees of new nodes. Our technique of indexing UTXOs in an increasing order and ordering them into a trie implies that all new UTXOs get added to the right side of the tree. In fact, while adding UTXOs, only the nodes on the right most branch of the tree are modified. For a stateless client to correctly compute the updated accumulator, it must keep track of the hashes of the right most branch. Thus, strictly speaking, the accumulator is not a single hash, but a set of hashes along a branch. The number of hashes grows as $\log N$, where N is the total number of UTXOs seen until the present moment.

4 Conclusion

This paper presents a new technique for constructing Merkle trees. We have shown, both in a theoretical model and in a practical implementation, that this approach has benefits over existing Merkle accumulator techniques. The code for our project is available on Github [5].

5 Acknowledgements

This material is based upon work supported by the National Science Foundation under the Graduate Research Fellowship Program with Grant No. DGE – 1746047 and under Grant No. CCF 19-00636. The authors would like to thank Andrew Miller for his advice on the project.

References

1. Benaloh, J., De Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. In: Workshop on the Theory and Application of of Cryptographic Techniques. pp. 274–285. Springer (1993)
2. Blelloch, G., Ferizovic, D., Sun, Y.: Parallel ordered sets using join. arXiv preprint arXiv:1602.02120 (2016)
3. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to iops and stateless blockchains. In: Annual International Cryptology Conference. pp. 561–586. Springer (2019)

[5] `https://github.com/surya-sankagiri/utreexo`

4. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Annual International Cryptology Conference. pp. 61–76. Springer (2002)
5. Campanelli, M., Fiore, D., Greco, N., Kolonelos, D., Nizzardo, L.: Incrementally aggregatable vector commitments and applications to verifiable decentralized storage (2020)
6. Chen, M., Hazay, C., Ishai, Y., Kashnikov, Y., Micciancio, D., Riviere, T.: Diogenes: Lightweight scalable rsa modulus generation with a dishonest majority.
7. De La Briandais, R.: File searching using variable length keys. In: March 3-5, 1959, Western Joint Computer Conference. p. 295–298. IRE-AIEE-ACM '59 (Western), Association for Computing Machinery, New York, NY, USA (1959)
8. Derler, D., Hanser, C., Slamanig, D.: Revisiting cryptographic accumulators, additional properties and relations to other primitives. In: Cryptographers' track at the RSA conference. pp. 127–144. Springer (2015)
9. Dobson, S., Galbraith, S.D., Smith, B.: Trustless groups of unknown order with hyperelliptic curves. (2020)
10. Dryja, T.: Utreexo: A dynamic hash-based accumulator optimized for the bitcoin utxo set. IACR Cryptol. ePrint Arch. **2019**, 611 (2019)
11. Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: Aggregating proofs for multiple vector commitments. Cryptology ePrint Archive, Report 2020/419 (2020), `https://eprint.iacr.org/2020/419`
12. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: 19th Annual Symposium on Foundations of Computer Science. pp. 8–21 (1978)
13. Li, J., Li, N., Xue, R.: Universal accumulators with efficient nonmembership proofs. In: Katz, J., Yung, M. (eds.) Applied Cryptography and Network Security. pp. 253–269. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
14. Lipmaa, H.: Secure accumulators from euclidean rings without trusted setup. In: International Conference on Applied Cryptography and Network Security. pp. 224–240. Springer (2012)
15. Miller, A., Hicks, M., Katz, J., Shi, E.: Authenticated data structures, generically. ACM SIGPLAN Notices **49**(1), 411–423 (2014)
16. Narayanan, A., Bonneau, J., Felten, E., Miller, A., Goldfeder, S.: Bitcoin and cryptocurrency technologies: a comprehensive introduction. Princeton University Press (2016)
17. Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable subvector commitments for stateless cryptocurrencies. IACR Cryptol. ePrint Arch. **2020**, 527 (2020)
18. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger