Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies

Leonid Reyzin* Dmitry Meshkov† Alexander Chepurnoy‡ Sasha Ivanov§

November 10, 2016

Abstract

We improve the design and implementation of two-party and three-party authenticated dynamic dictionaries and apply these dictionaries to cryptocurrency ledgers.

A public ledger (blockchain) in a cryptocurrency needs to be easily verifiable. However, maintaining a data structure of all account balances, in order to verify whether a transaction is valid, can be quite burdensome: a verifier who does not have the large amount of RAM required for the data structure will perform slowly because of the need to continually access secondary storage. We demonstrate experimentally that authenticated dynamic dictionaries can considerably reduce verifier load. On the other hand, per-transaction proofs generated by authenticated dictionaries increase the size of the blockchain, which motivates us to find a solution with most compact proofs.

Our improvements to the design of authenticated dictionaries reduce proof size and speed up verification by 1.4–2.5 times, making them better suited for the cryptocurrency application. We further show that proofs for multiple transactions in a single block can compressed together, reducing their total length by approximately an additional factor of 2.

We simulate blockchain verification, and show that our verifier can be about 20 times faster than a disk-bound verifier under a realistic transaction load.

1 Introduction

The Motivating Application. A variety of cryptocurrencies, starting with Bitcoin [Nak08], are based on a public ledger of the entire sequence of all transactions that have ever taken place. Transactions are verified and added to this ledger by nodes called *miners*. Multiple transactions are grouped into blocks before being added, and the ledger becomes a chain of such blocks, commonly known as a *blockchain*.

If a miner adds a block of transactions to the blockchain, other miners verify that every transaction is valid and correctly recorded before accepting the new block. (Miners also perform other work to ensure universal agreement on the blockchain, which we do not address here.) However, not only miners participate in a cryptocurrency; others watch the blockchain and/or perform partial

*Boston University, `http://www.cs.bu.edu/faculty/reyzin`. Research supported by the Waves platform.

†IOHK Research and N. N. Semenov Institute of Chemical Physics, Russian Academy of Sciences, `dmitry.meshkov@iohk.io`

‡IOHK Research, `alex.chepurnoy@iohk.io`

§Waves platform, `sasha@wavesplatform.com`

verification (e.g., so-called light nodes, such as Bitcoin's SPV nodes [Nak08, Section 8]). It is desirable that these other participants are able to check a blockchain with full security guarantees on commodity hardware, both for their own benefit and because maintaining a large number of nodes performing full validation is important for the health of the cryptocurrency [Par15]. To verify each transactions, they need to know the balance of the payer's account.

The simple solution is to have every verifier maintain a dynamic dictionary data structure of (key, value) pairs, where keys are account addresses (typically, public keys) and values are account balances. Unfortunately, as this data structure grows, verifiers need to invest into more RAM (and thus can no longer operate with commodity hardware), or accept significant slowdowns that come with storing data structures in secondary storage. These slowdowns (especially the ones caused by long disk seek times in an adversarially crafted set of transactions) have been exploited by denial of service attacks against Bitcoin [Wik13] and Ethereum [But16].

Authenticated Dictionaries to the Rescue. We propose using cryptographically authenticated data structures to make *verifying* transactions in the blockchain much cheaper than *adding* them to the blockchain. Cheaper verification benefits not only verifiers, but also miners: in a multi-token blockchain system (where tokens may represent, for example, different currencies or commodities), such as Nxt [nxt], miners may choose to process transactions only for some types of tokens, but still need to verify all transactions.

Specifically, we propose storing balance information in a *dynamic authenticated dictionary*. In such a data structure, *provers* (who are, in our case, miners) hold the entire data structure and modify it as transactions are processed, publishing *proofs* that each transaction resulted in the correct modification of the data structure (these proofs will be included with the block that records the transaction). In contrast, *verifiers*, who hold only a short *digest* of the data structure, verify a proof and compute the new digest that corresponds to the new state of the data structure, without ever having to store the structure itself. We emphasize that with authenticated data structures, the verifier can perform these checks and updates without trusting the prover: the verification algorithm will reject any attempt by a malicious prover or man-in-the-middle who tries to fool the verifier into accepting incorrect results or making incorrect modifications. In contrast to the unauthenticated case discussed above, where the verifier must store the entire data structure, here verifier storage is minimal: 32 bytes suffice for a digest (at 128-bit security level), while each proof is only a few hundred bytes long and can be discarded immediately upon verification.

1.1 Our Contributions

A Better Authenticated Dictionary Data Structure. Because reducing block size a central concern for blockchain systems [CDE$^+$16, DW13], we focus on reducing the length of a modification proof, which must be included into the block for each transaction. Moreover, because there is no central arbiter in a blockchain network, we require an authenticated data structure that can work without any assumptions about the existence of a trusted author or setup and without any secret keys (unlike, for example, [PTT16, BGV11, CF13, CLH$^+$15, MWMS16, CLW$^+$16]). And, because miners may have incentives to make verification more time-consuming for others, we prefer data structures whose performance is independent of the choices made by provers.

We design and implement an authenticated dictionary data structure requiring no trusted setup or authorship whose proofs are, on average, 1.4 times shorter than authenticated skip lists of [PT07] and 2.5 times shorter than the red-black trees of [CW11]. Moreover, our prover and verifier times are faster by the same factor than corresponding times for authenticated skip lists, and, unlike the work of [PT07], our data structure is deterministic, not permitting the prover to bias supposedly random choices in order to make performance worse for the verifier. In fact, our data structure's *worst-case* performance is comparable to the *expected-case* performance of [PT07]. Our work was inspired by the dynamic Merkle [Mer89] trees of [NN00, AGT01, CW11, MHKS14] in combination with the classic tree-balancing algorithm of [AVL62].

Our improvement to authenticated data structures improves not only the aforementioned so-called two-party case (where there is no trusted author to make modifications), but also the three-party case addressed in multiple works [Mer89, NN00, GT00, GTS01, AGT01, MND$^+$04, GPT07, CW11]. Our design can be used as a replacement for authenticated skip lists of [GTS01] in both two-party and three-party applications that rely on them (e.g., [BP07, GPTT08, HPPT08, EK13] and many others), improving performance and removing the need for randomization. We further reduce proof length per operation when putting together proofs for multiple operations. For example, when proofs for 1000 operations on a 1 000 000-entry dictionary are put together, our proof length is about 390 bytes per operation.

Application to Blockchains. We consider a multi-token blockchain system (unlike Bitcoin, which has bitcoins as the only tokens) with accounts in which balances can grow or shrink over time (again, unlike Bitcoin, in which a transaction output must be spent all at once). One example of such a system is Nxt [nxt]. For each token type t, there is an authenticated data structure S_t maintaining balances of all accounts, locally stored by miners who are interested in the ability to add transactions for that token type. All miners, regardless of interest, maintain a local copy of the short digest of S_t.

In order to publish a block with a number of transactions, a miner adds to the block the proof of validity of these transactions, including the proofs of correct updates to S_t, and also includes the new digest of S_t into the block header. All miners, as well as verifiers, verify the proof with respect to the digest they know and check that the new digest in the block header is correct. (It is important to note that verification of transactions includes other steps that have nothing to do with the data structure, such as verifying the signature of the payer on the transaction; these steps do not change.) In contrast to simple payment verification nodes [Nak08, Section 8] in Bitcoin, who cannot fully verify the validity of a new block because they do not store all unspent outputs, our verifiers can do so without storing any balance information.

While there have been many proposals to use authenticated data structures for blockchains (see, e.g., [Mil12] and references therein), not many have suggested publishing proofs for modifications to the data structure. At a high level, our approach is similar to (but considerably more efficient than) the proposal by White [Whi15], who suggests building a trie-based authenticated data structure for Bitcoin (although he does not use those terms).

Because of our improved authenticated data structure, provers[1] and verifiers are more efficient, and proofs are shorter, than they would be with previous solutions. We show that whenever a block includes multiple transactions for a given token, their proofs can be combined, further reducing the amount of space used per transaction, by about a factor of 2 for realistic conditions. We benchmark block generation verification and demonstrate that verifying the authenticated data structure can

[1] How much efficiency of proof generation matters depends on the cryptocurrency design. In those cryptocurrencies for which every miner attempts to generate a block (such as Bitcoin), it matters a lot, because every miner has to run the proof generation procedure. On the other hand, in those cryptocurrencies for which the miner wins a right to generate a block before the block is produced (such as ones based on proof of stake [BGM16, KKR$^+$16]), only one miner per block will generate proofs.

be about 20 times faster than maintaining a full on-disk unauthenticated data structure, while generating proofs does not add much to a miner's total clost.

Reducing the Cost of a Miner's Initial Setup. A new miner Molly wishing to join the network has to download the entire blockchain and verify the validity of every block starting from the first (so-called "genesis") block. It is not necessary to verify the validity of every transaction, because the presence of the block in the blockchain assures Molly that each transaction was verified by other miners when the block was added. However, without authenticated data structures, Molly still needs to download and replay all the transactions in order to establish the up-to-date amount held in each account and be able to validate future transactions.

Our solution allows Molly to reduce communication, computation, and memory costs of joining the network, by permitting her to download not entire blocks with their long lists of transactions, but only the block headers, which, in addition to demonstrating that the block has been correctly generated and linked to the chain, contain the digest of all the transactions processed and digests of every authenticated data structure S_t that has changed since the previous block. This information is enough to start validating future transactions. If Molly wants to not only validate, but also process transactions for tokens of type t, she needs to obtain the full S_t; importantly, however, she does not need a trusted source for this data, because she can verify the correctness of S_t against the digest.[2]

2 The Model for Two-Party Authenticated Dictionaries

Given the variety of security models for authenticated data structures, let us briefly explain ours (to the best of our knowledge, it was first implicitly introduced in [BEG⁺91] and more explicitly in [GSTW03, PT07]; it is commonly called the *two-party* model; see [Pap11] for an overview of the relevant literature).

Each state of the data structure is associated with an efficiently computable *digest*; it is computationally infeasible to find two different states of the data structure that correspond to the same digest. There are two types of parties: *provers* and *verifiers*. The provers possess the data structure, perform operations on it, and send *proofs* of these operations to verifiers, who, possessing only the digest of the current state of the data structure, can use a proof to obtain the result of the operation and update their digests when the data structure is modified. The security goal is to ensure that malicious provers can never fool verifiers into accepting incorrect results or computing incorrect digests. Importantly, neither side generates or possesses any secrets.

A secondary security goal (to prevent denial of service attacks by provers who may have more computing resources than verifiers) is to ensure that a malicious prover cannot create a valid state of the data structure that causes operations to take longer than a prespecified upper bound (for instance, by creating a highly unbalanced trees).

Importantly, the model assumes that the verifiers and the provers agree on which data structure operations need to be performed (in our cryptocurrency application, whether performing the operation is a valid choice will be verified separately, for example by checking the signature of the

[2]Ethereum [Woo14] adds the digest of the current state of the system to each block, but, because it does not implement proofs for data structure modifications, this digest cannot be used unless the miner downloads the entire state of the system—although, importantly, this state may be downloaded from an untrusted source and verified against the digest. Miller et al. [MHKS14, Appendix A] suggested using authenticated data structures to improve memory usage, but not communication or computation time, of Bitcoin's initial setup.

payer). The model also assumes that the verifier initially has the correct digest (for example, by maintaining it continuously starting with the initial empty state of the data structure).

The specific data structure we wish to implement is a dictionary (also known as a map): it allows insertion of (key, value) pairs (for a new key), lookup of a value by key, modification of a value for a given key, and deletion by key.

We provide formal security definitions in Appedix A.

3 Our Construction

Despite a large body of work on authenticated data structures, to the best of our knowledge, only two prior constructions—those of [PT07] (based on skip lists) and [MHKS14] (based on skip lists and red-black trees)—address our exact setting. As mentioned in the introduction, many other works address the three-party setting in which modifications are performed by a trusted author and only lookups are performed by the provers (which we also improve), and/or propose solutions requiring a secret key that remains unknown to the prover.

We will explain our construction from the viewpoint of unifying prior work and applying a number of optimizations to existing ideas.

Starting Point: Merkle Tree. We start with the classic Merkle tree [Mer89]. Let H be a collision-resistant hash function. The leaves of the tree store the data we wish to authenticate—in our case, (key, value) pairs. The label of each leaf is defined as the hash of its content (preceded by a special symbol—for example, a 0 byte—indicating that it's a leaf), and the value of each internal node defined (recursively) as the hash of the labels of its two children (preceded by a special symbol—for example, a 1 byte—indicating that it's an internal node). The digest is the label of the root. The proof that a given key is in the data structure and has a given value consists of the labels of siblings of nodes on the path from the root to the leaf, together with information on whether the path goes left or right at each step. The proof can be verified by recomputing the alleged root label and checking that it matches the digest. This proof is known as the *authenticating path*.

Incorporating a Binary Search Tree. To make searches possible, we turn the tree into a slight variant of the standard binary search tree, the same way as in [NN00, AGT01, MHKS14]. First, we sort the leaves by key. Each internal node stores a key y that is the minimum of its right subtree: that way, the left subtree contains exactly the leaves with keys less than y. (This description breaks ties in the opposite way of [AGT01] and [MHKS14], but is more intuitive given our improvements described below.) Unlike the standard binary search tree, this binary search tree has internal nodes only for helping the search rather than for storing values, which are only at the leaves. The proof is still the same authenticating path. (We note that the approach based on standard binary search trees, where internal nodes also store keys and values, is explored in [CW11]; as we demonstrate below in Section 4, it results in longer proofs, because the slight savings in tree height are more than negated by the fact that internal nodes must also include their keys and values into the computation of their label and therefore into the proof.)

Furthermore, we make sure that every non-leaf node has exactly two children. To insert a new (key value) pair, go down to the correct leaf ℓ like in the standard binary search tree, and replace ℓ with a new internal node that has ℓ and a new leaf containing the new (key, value) pair as its two children. To simplify insertions, we can make sure that the key being inserted always goes to the right of ℓ, and the new internal node gets the same key as the one being inserted, by simply initializing the empty tree with a single leaf containing $-\infty$ as the key (when keys are long random values, such as public keys, setting $-\infty$ to an all 0s string is reasonable). (It is easy to prove that then every insertion goes to the right at the last step: if a search for an insertion never took a step to the right, then it reached $-\infty$; and if it did take a step to the right, then consider the key of the last node that caused the search to take a right step, and observe that the key in ℓ is the same and therefore less than the key being inserted). This approach saves us from having special cases in the code and reduces by one the number of comparisons needed during the insert operation.

Proving Absence of a Key. There are two approaches for proving nonmembership of a key k (which is needed, in particular, during insertion). The first approach (used in [NN00, AGT01, CW11, MHKS14]) is to show proofs for two neighboring leaves with keys $k_1 < k < k_2$. The second approach (used in [GT00] and other works based on skip lists, such as [PT07]) is to add a `next` pointer to every leaf and modify the way a label of a leaf is computed, by hashing not only the key and the value stored at the leaf, but also the key of the next leaf (and $+\infty$ when there is no next leaf).

We adopt the second approach for its simplicity: it unifies the code for successful and unsuccessful lookups, in both cases giving us a proof that consists of a single authenticating path. (While this second approach lengthens the proof of every successful lookup by the length of a key, it slightly shortens the proof of an average unsuccessful lookup by about the length of a label.). Moreover, our creation of a $-\infty$ sentinel, which makes sure that insertions always go to the right of an existing leaf, makes maintaining the pointer to the next leaf trivial: when a leaf ℓ_{new} is inserted to the right of a leaf ℓ_{old}, just set $\ell_{\text{new}}.\texttt{next} = \ell_{\text{old}}.\texttt{next}$ and $\ell_{\text{old}}.\texttt{next} = \ell_{\text{new}}$.

Modifying the Value for an Existing Key. If the prover modifies the value stored at a leaf (for example, subtracting from it money used for a transaction), the label of that leaf and all the nodes above it need to be recomputed, but no other information in the tree changes. Observe that this recomputation of labels needs only the new value at the leaf and information that is already present in the authenticating path. Therefore, the verifier has all the information needed to compute the new digest after checking that the authenticating path is correct. Thus, the proof for a modification is the same as the proof for a lookup.

Simple Insertions. Insertions into our Merkle binary search tree, like insertions into ordinary binary search tree, may require some rebalancing in order to make sure that the paths to the leaves do not grow too long, increasing the computation and communication time per operation. However, we will discuss rebalancing in the next section. For now, consider an insertion without rebalancing. Such an insertion simply replaces an old leaf (that was found by performing a search for the key being inserted) with a new internal node, linked to two leaves. Therefore, knowledge of the contents of these two leaves and the authenticating path is enough to be able to compute the new digest. Thus, the proof for such a simple insertion is the same as before: the authenticating path to the leaf that is found during the search for the key that was being inserted. This proof is enough for the verifier to check that the key does not exist and to perform insertion.

3.1 Our Improvements

Observation 1: For Insertions, Use Tree-Balancing Operations that Stay on Path. A variety of algorithms for balancing binary search trees exist. Here we focus on AVL trees [AVL62], red-black trees [GS78] and their left-leaning variant [Sed08], treaps [SA96] (and their equivalent randomly-balanced binary search trees [MR98]). They all maintain some extra information in the nodes that enables the insertion algorithm to make a decision as to whether, and how, to perform tree rotations in order to maintain a reasonably balanced tree. They can be easily adapted to work with our slightly modified trees that have values only at the leaves (simply don't apply any of the balancing procedures to the leaves), and all maintain our invariant that the key of an internal node is the minimum of its right subtree.

The extra information they maintain for balancing is usually not large (just one bit for "color" per node for red-black trees; one trit for "balance" per node for AVL trees; and roughly $\log n$ bits for "priority" per node for treaps, where n is the number of nodes). This information should be added as an input to the hash computation for the label of each internal node. This information, for each node on the path from the root to the leaf, should also be included into the proof (as it has to be input by the verifier into the hash function).

We observe that if the tree balancing operation rotates only ancestors of the newly inserted leaf, and does not use or modify information in any other nodes, then the proof we already provide for a search has sufficient information for the verifier to perform the tree-balancing operation. This is the case for AVL trees[3] and treaps. Red-black trees, depending on the variant, may access information in children and grandchildren of ancestors in order to decide on rotations, and are therefore less well suited for our application, because the contents of those children and grandchildren will need to be proven, lengthening the proofs. (It may be possible to modify red-black trees by storing colors of nodes with the parents and/or grandparents, but we do not explore this option, because we find a better solution.)

However, of these options, only red-black trees have been implemented in our setting [MHKS14], and this implementation sometimes must access the color of a node that is not an ancestor of the newly inserted leaf. Therefore, the insertion proofs of [MHKS14] must be longer, in order to include authenticating paths to additional nodes (according to Miller [Mil16], proofs for insertions in the red-black trees of [MHKS14] are approximately three times longer than proofs for lookups). Thus, the balanced trees that are better suited for our setting have not been implemented before (we should note that treaps were implemented in the three-party setting of [CW11]; see our comparison in Section 4).

Observation 2: Do Not Hash Internal Keys. To verify that a particular leaf is present (which is all we need for both positive and negative answers), the verifier does not need to know how the leaf was found—only that it is connected to the root via an appropriate hash chain. Therefore, like the authors of [PT07] (and many works in the three-party setting), we do not add the keys of internal nodes into the hash input, and do not put them into the proof. This is in contrast to the work of [MHKS14], whose general approach requires the label to depend on the entire contents of a node, and therefore requires keys of internal nodes to be sent to the verifier, so that the verifier can compute the labels. When keys do not take up much space (as in [MHKS14]), the difference between sending the key of an internal node and sending the direction (left or right) that the search path took is small. However, when keys are comparable in length to labels (as in the cryptocurrency application, because they are account identifiers, computed as hash function outputs or public keys), this difference can mean nearly a factor of two in the proof length.

Observation 3: Skip Lists are Just a Variant of Treaps. Dean and Jones [DJ07] observed

[3] For those familiar with AVL trees, we note that this is the case when AVL trees are implemented with every node maintaining the difference of heights between right and left children, rather than its own height, because if a node maintains height, then it needs to compute its balance in order to decide whether to rotate, and this computation requires the heights of both children, while our proof contains only the height of one child.

that a binary search tree can be built on the tops of towers of a skip list [Pug90], and the nodes encountered in a search will be the same in both cases, as long as the tree follows the rule that a child's level is less than (or equal to, if the child is right) than the parent's. They show that tree insertions can be accomplished by inserting at the bottom and then rotating with the parent until the above rule is satisfied.

We observe that skip lists viewed this way are just a variant of treaps, with "level" in a tree-based skip list corresponding to "priority" in a treap. Heights in a skip list are sampled so that value h has probability $1/2^{h+1}$, while priorities in a treap are sampled uniformly, but otherwise they are equivalent. Of course, we further convert this treap-based view of skips lists to have values only at leaves, as already described above. This view enables us to test the performance of skip lists and treaps with essentially the same implementation.

In prior work, in order to make them authenticated, skip lists were essentially converted to binary trees by [GT00]; this conversion was made explicit in [CW11]. Our binary tree, which results in combining the observation of [DJ07] with the transformation of putting values only at leaves, ends up being almost exactly the same, with the following main difference: each internal node in our data structure stores the minimum of its right subtree, while each internal node in the data structure of [GT00] stores the minimum of its entire subtree. (To see the equivalence, note that our data structure can be obtained from the data structure of [PT07] by having every parent replace its key with the key of its right child; the only remaining difference is nonstandard chaining of leaves in skip lists.) No prior implementation, however, treated skip lists the same way as other binary trees.

Observation 4: Deterministic is Better. Treaps and skip lists perform well in expectation (and also with very high probability), when the priorities (for treaps) and levels (for skip lists) are chosen at random, independently of the keys in the data structure. However, if an adversary is able to influence or predict the random choices, performance guarantees no longer hold. In our setting, the problem is that the provers and verifiers need to somehow agree on the randomness used. (This is not a problem for the three-party setting, where the randomness can be supplied by the trusted author.)

Prior work in the three-party model suggested choosing priorities and levels by applying hash functions to the keys [CW11, Section 3.1.1]. However, since inserted keys may be influenced by the adversary, this method of generating randomness may give an attacker the ability to make the data structure very slow and the proofs very long, effectively enabling a denial of service attack. To eliminate this attack by an external adversary, we could salt the hash function after the transactions are chosen for incorporation into the data structure (for example, including a fresh random salt into each the block header). However, an internal adversary still presents a problem: the prover choosing this salt and transactions would have the power to make the data structure less efficient for everyone by choosing a bad salt, violating our secondary security goal stated in Section 2.

Observation 5: AVL Trees Outperform on the Most Relevant Parameters. Regardless of the tree balancing method (as long as it satisfies observations 1 and 2), costs of lookups, modifications, and insertions are determined simply by the depth of the relevant leaf, because the amount of nodes traversed, the size of the proof, and the number of hashes performed by both provers and verifiers is directly proportional to this depth. Of course, different tree balancing methods may use slightly different logic and cause a different number of rotations, but the amount of time spent on those is negligible compared to the cost of hash function evaluation (note that a tree rotation changes only two pointers and does not change the number of hashes that need to be computed).

Therefore, our goal is to choose a tree in which leaves are as close as possible to the root. We are concerned mostly with the average case but also, because operations can be adversarially chosen, with the worst case.

The average-case distance between the root and a random leaf for both AVL and red-black trees after the insertion of n random keys is very close to the optimal $\log_2 n$ [Knu98, p. 468], [Sed08]. The worst-case distance for red-black trees is twice the optimal [Sed08], while the worst-case distance for AVL trees is 1.44 times the optimal [Knu98, p. 460]. In contrast, the expected (not worst-case!) distance for treaps and skip lists is 1.5 times the optimal [Pug90]. Thus, AVL trees, even the *worst case*, are better than treaps and skip lists *in expectation*.

Observation 6: Proofs for Multiple Operations Can Be Compressed. When multiple operations on the data structure are processed together, their proofs can be compressed. A verifier will not need the label of any node more than once. Moreover, the verifier will not need the label of any node that lies on the path to a leaf in another proof (because it will be computed during the verification of that proof). Nor will the verifier need the label of any node that is created by the verifier (for example, if there is an insertion into the right subtree of the root, then the verifier will replace the right child of the root with a new node and will thus know its label when the label is needed for a proof about some subsequent operation on the left subtree).

Performing this compression is nontrivial (generic compression algorithms, as used in [MHKS14] and reported to us by [Mil16], can take care of repeated labels, but will not perform the other optimizations). We propose the following approach to compressing a batch of operations, which conceptually separates the tree nodes from the operations that touch them.

Let T be the starting tree. Every node of T that is visited when performing the batch of operations is marked as "visited." New and modified nodes do not replace nodes of T, but are created afresh. Thus, T is preserved, and new nodes may point to nodes of T. At the end of the batch, there is a subtree of T (staring at the root) that is marked "visited".

The proof contains the contents of the nodes of this "visited" subtree of T (excluding keys for internal nodes but including both keys and next-leaf keys for leaves), as well as labels of nodes that are one step away from subtree. Such a proof is easy to obtain and serialize by performing a post-order traversal of the "visited" nodes and their children, and writing down appropriate information about each node reached during the traversal. In addition to this tree, the proof also contains a sequence of single-bit (left or right) "directions" (see Observation 2) that the prover's algorithm took when executing the batch of operations. Once the prover constructs the proof, the "visited" flags are reset for the next batch, and nodes of T that are not reachable from the new tree root can be garbage-collected.

The verifier simply reconstructs this visited portion of T by using the proof and computes the label of the root of the reconstructed tree to make sure it is equal to the digest. Then the verifier runs essentially the same algorithm as the prover in order to perform the batch of modifications. The only difference is that the verifier replaces key comparisons on internal nodes with following the left-or-right directions from the proof, and adds the check that the key sought either equal to the key in the leaf, or between the key in the leaf and the key of the next leaf (this check ensures those directions in the proof were honest).

Putting these observations together, we obtain the data structure to implement: an AVL tree with values stored only at the leaves, sometimes known as an AVL+ tree. We implement this data structure and compare it against other options in the next section. We prove its security in Appendix B.

4 Implementation and Evaluation

We implemented our AVL+ trees, as well as treaps and our tree-based skip lists, in the Scala [sca] programming language using the Blake2b [ANWOW13] hash function with 256-bit (32-byte) outputs. Our implementation is available at [cod]. For the AVL+ implementation, we used the textbook description [Wei06] with the same balance computation procedure as in [Pfa02, Chapter 5.4.4]. We ran experiments by measuring the cost of 1000 random insertions (with 26-byte keys and 8-byte values), into the data structure that already had size $n = 0, 1000, 2000, \ldots, 999000$ keys in it.

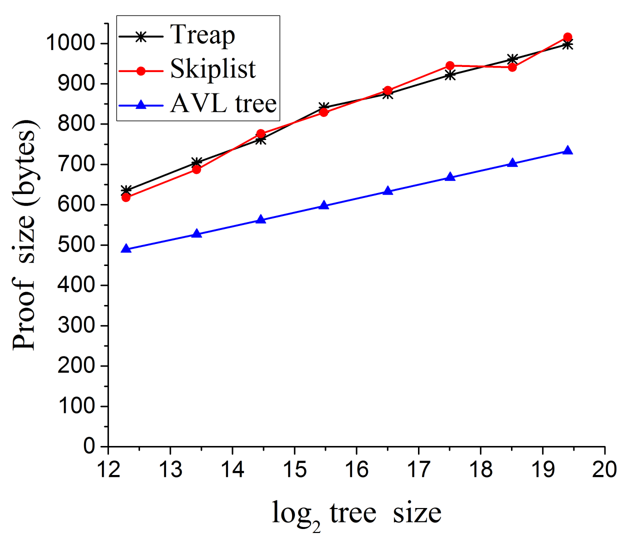
As expected, the length of the path from the root to a random leaf in the n-leaf AVL+ tree was only $2-3\%$ worse than the optimal $\log_2 n$. In contrast, the length of the path in a skip list was typically about 44% worse than optimal, and in a treap about 32% worse than optimal.

Proof length for a single operation. The average length of our proof for inserting a new key into a 1 000 000-node tree with 32-byte hashes, 26-byte keys, and 8-byte values, is 753 bytes. We now explain this number and compare it to prior work.

Note that for a path of length k, the proof consists of:

- k labels (which are hash values),
- $k+1$ symbols indicating whether the next step is right or left, or we are at a leaf with no next step (these fit into two bits each),
- k pieces of balance or level information (these fit into two bits for an AVL+ tree, but require a byte for skip lists and three or four bytes for treaps),
- the leaf key, the next leaf key, and the value stored in the leaf node (the leaf key is not needed in the proof for lookups and modifications of an existing key, although our compression technique of Observation 6 will include it anyway, because it does not keep track of why a leaf was reached)

Thus, the proof length is almost directly proportional to the path length: with the 32-byte hashes, 26-byte keys, and 8-byte values, the proof takes 34k+61 bytes assuming we don't optimize at bit level, or about k bytes fewer if we do (our implementation currently does not). Note that the value of k for $n = 1\,000\,000$ is about 20 for AVL+ trees and about 29 for skip lists, which means that AVL-tree-based proofs are about 1.4 times shorter than skip-list-based ones. Treap proofs have slightly smaller k, but this advantage is completely negated in our experiments by the extra bytes needed to write down the level.



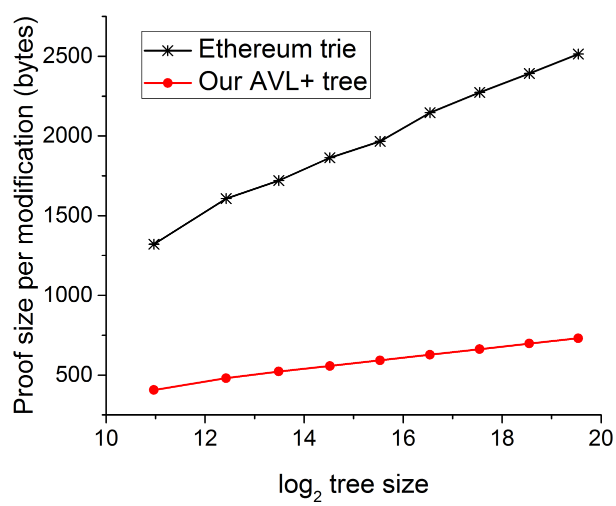
Proof Length Comparison with Existing Work. Our numbers are consistent with those reported by Papamanthou and Tamassia [PT07, Section 4], who also report paths of length 30 for skip lists with 1 000 000 entries. (They use a less secure hash function whose output length is half of ours, resulting in shorter proofs; if they transitioned to a more secure hash function, their proofs would be about the same length as our skip-list-based proofs, thus 1.4 times longer than our AVL+-based proofs).

Direct comparison with the work of [MHKS14] is harder, because information on proof length for a single insertion in red-black trees is not provided in [MHKS14] (what is reported in [MHKS14] is the result of off-the-shelf data compression by gzip [GA] of the concatenation of proofs for 100 000 lookup operations). However, because keys of internal nodes are included in the proofs of [MHKS14], the proofs for lookups should be about 1.7 longer than in our AVL+ trees (for our hash and key lengths). According to [Mil16], the proofs for insertions for the red-black trees of [MHKS14] are about 3 times longer than for lookups (and thus about 5 times longer than proofs for insertions in our AVL+ trees). Of course, the work [MHKS14] has the advantage of being generic, allowing implementation of any data structure, including AVL+ trees, which should reduce the cost of insertions to that of lookups; but, being generic, it cannot avoid sending internal keys, so the cost of lookups will remain.

We can also compare our work with work on three-party authenticated data structures, because our data structure also works in the three-party model (and some three-party data structures will also work in our model). Work based on skip lists, such as [AGT01, GTS01, GPT07], has proof sizes that are the same as the already-mentioned [PT07], and therefore our improvement is about the same factor of 1.4.

For three-party work based on red-black trees, there are two variants. The variant that stores values only at leaves, like we do, was implemented by Anagnostopoulos et al. [AGT01], who do not report proof length; however, we can deduce it approximately from the number of hashes reported in [AGT01, Figure 6, "hashes per insertion"] and conclude that it is about 10-20% worse than ours. The variant that uses a standard binary search tree, with keys and values in every node, was implemented by [CW11] and had the shortest proofs among the data structures tested in [CW11]. The average proof length (for a positive answer) in [CW11] is about 1500 bytes when searching for a random key in a tree that starts empty and grows to 10^5 nodes, with 28-byte keys, values, and hashes. In contrast, our average proof size in such a scenario is only 593 bytes (an improvement of 2.5 times), justifying our decision to put all the values in the leaves.

Finally, Ethereum implements a Merkle patricia trie [Woo14, Appendix D] in a model similar to the three-party model (because it does not implement proofs for changes to the trie). In our experiments (which used the code from [Tea16, trie/proof.go] to generate proofs for the same parameter lengths as ours) using for n ranging from 2000 to 1 000 000, Ethereum's proofs for lookups were consistently over 3 times longer than our AVL+-based ones.



Proof Length for Multiple Operations. Compressing together proofs for a batch of B operations at once (using Observation 6 in Section 3) reduces the proof length per operation by approximately $36 \cdot \log_2 B$ bytes (where the operations are lookups or insertions). This improvement is considerably greater than what we could achieve by concatenating individual proofs and

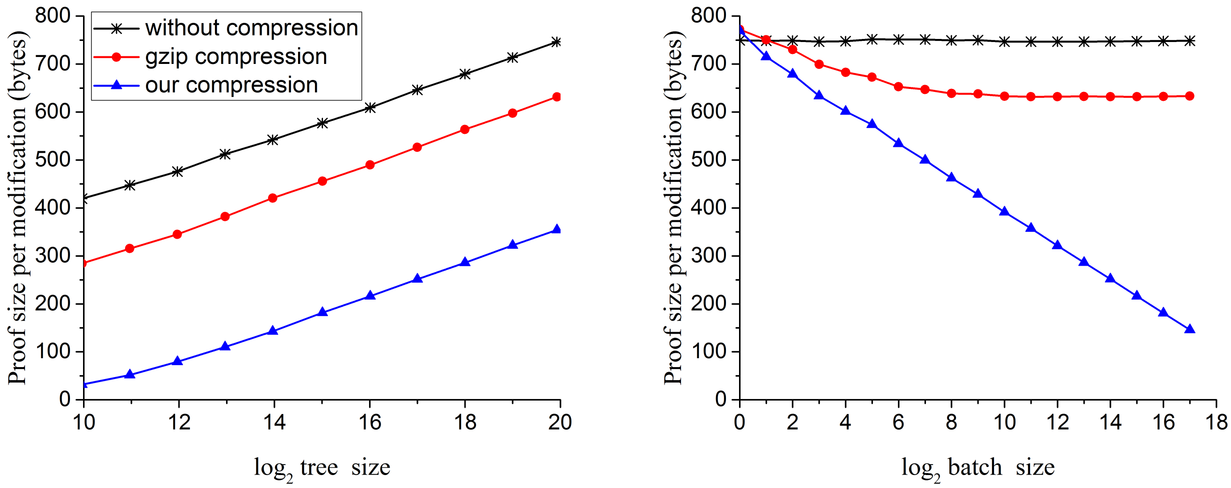


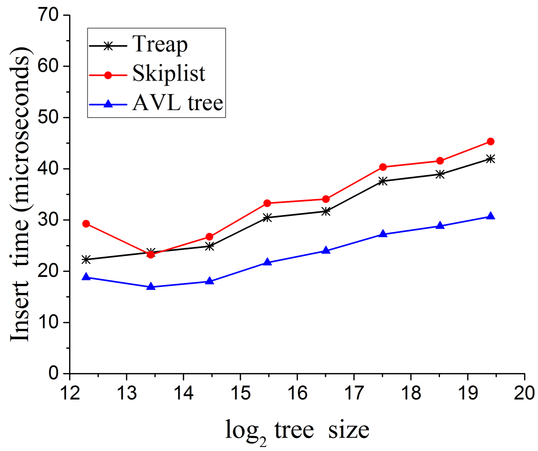
Figure 1: Left: proof size per modification for $B = 2000$, as a function of starting tree size n. Right: proof size per modification for a tree with $n = 1\,000\,000$ keys, as a function of batch size B. In both cases, half of the modifications were inserts of new (key, value) pairs and half were changes of values for existing keys.

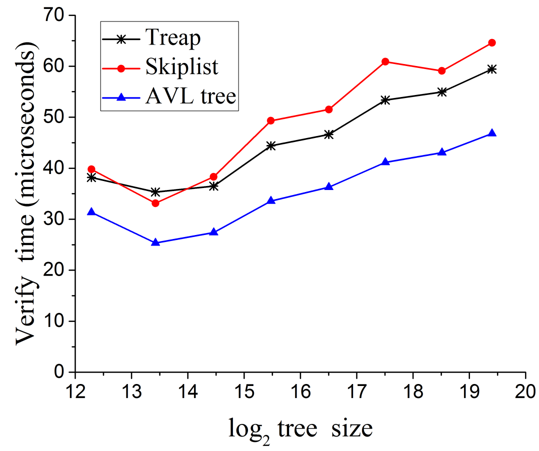
then applying gzip [GA], which, experimentally, never exceeded 150 bytes, regardless of the batch size. The improvements reported in this section are for uniformly random keys; biases in key distribution can only help our compression, because they result in more overlaps among tree paths used during the operations.

For example, for $n = 1\,000\,000$, the combined proof for 1000 lookups and 1000 inserts was only 358 bytes per operation. If a transaction in a block modifies two accounts, and there are 1 000 000 accounts and 1 000 transactions in the block (this number is realistic—see [tbp]), then we can obtain proofs of 716 bytes per transaction remaining at 128-bit security level. If some accounts are more active and participate in more than one transaction, then the per transaction space is even less, because our compression algorithm performs better when operations share paths.

We can compare our results with those reported in Miller et al. [MHKS14, Figure 13d], who report the results of batching together (using a "suspended disbelief" buffer to eliminate some labels and gzip to compress the stream) $B = 100\,000$ proofs for lookup operations on a red-black tree tree of size n ranging from 2^4 to 2^{21}. For these parameter ranges, our proofs are at least 2.4 times shorter, even though we use 1.6-times longer hashes, as well as longer keys and values. For example, for $n = 2^{21}$, our proofs take up 199 bytes per operation vs. 478 bytes of [MHKS14]. Proofs for insertions are even longer in [MHKS14], while in our work they are the same as for lookups. We emphasize, again, that the work of Miller et al. has the advantage of supporting general data structures.

Prover and Verifier Running times. The benchmarks below were run on an Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz Linux machine with 8GB of RAM running in 64-bit mode and using only one core. We used Java 8.0.51 and compiled our Scala code with scalac 2.11.8. The Java implementation of Blake2b hash function was obtained from the official Blake website `https://blake2.net/`. The average prover time for inserting a random key into our AVL+ tree with 1 000 000 random keys was 31 μs, while the average verifier time for the same operation was 47 μs.

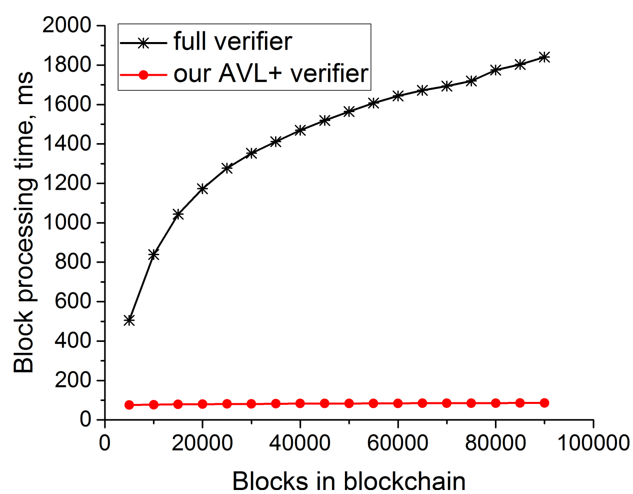




It is difficult to make comparisons of running times across implementations due the variations in hardware environments, programming language used, etc. Note, however, that regardless of those variables, the running times of the prover and verifier are closely correlated with path length k: the prover performs k key comparisons (to find the place to insert) and computes $k+1$ hash values (to obtain the label of two new nodes and $k-1$ existing nodes whose labels change), while the verifier performs two comparisons (with the keys of two neighboring leaves) and computes $2k+1$ hash values (k to verify the proof and $k+1$ to compute the new digest). Tree rotations do not change these numbers.

We therefore expect our AVL+ trees to perform about 1.4 times faster than skip lists, which is, indeed, what our benchmarks show.

When we batch multiple transactions together, prover and verifier times improve slightly as the batch size grows, in particular because labels of nodes need not be computed until the entire batch is processed, and thus labels of some nodes (the ones that are created and then replaced) are never computed.



Simulated Blockchain Proving and Verifying. We used a server (Intel(R) Core(TM) i7-5820K CPU @ 3.60GHz Linux machine with 64GB of RAM and SSD storage) to simulate two blockchain verifiers: a "full verifier" that simply maintains a full on-disk (SSD) data structure of (key, value) pairs and a "light verifier" who uses our data structure, and thus maintains only digests and verifies proofs, using very little RAM and no on-disk storage. The data structure was populated with 5 000 000 random 32-byte keys (with 8-byte values) at the start. Our simulated blocks contained 1500 modifications of values for randomly chosen existing keys and 500 insertions of new random keys. We ran the simulation for 90 000 blocks (thus ending with a data structure of 50 000 000 keys, similar to Bitcoin UTXO set size [Lop] at the time of writing).

Both the full and the verifier were limited to 1GB of RAM. Because the actual machine had 64GB of RAM, in order to prevent the OS from caching the entire on-disk data structure, we simulated a limited-RAM machine by invalidating the full verifier's OS-level disk cache every few 10 seconds. We measured only the data structure processing time, and excluded the time to read the block from the network or disk, to verify signatures on transactions, etc. The full verifier's running time grew rapidly, ending at about 1800 ms per block on average, while our light verifier stayed at about 85ms per block, giving our authenticated data structures a 20x speed advantage once the size gets large.

To make sure that generating proofs is feasible for a powerful machine, we also ran our prover, but permitted it to use up to 48GB of RAM. The prover stayed at about 70ms per block, which is a small fraction of a full node's total cost. For example, the cost to verify 1000 transaction signatures—just one of the many things a full node has to do in order to include transactions into a block—was 280ms on the same machine (using the Ed25519 [BDL+12] signature scheme).

5 Conclusion

We demonstrated the first significant performance improvement in two-party authenticated data structures since [PT07] and three-party authenticated data structures since [CW11]. We did so by showing that skip lists are simply a special case of the more general balanced binary search tree approach; finding a better binary search tree to use; and developing an algorithm for putting together proofs for multiple operations. We also demonstrated that our two-party authenticated data structures can be used to greatly improve blockchain verification by light nodes without adding much burden to full nodes—providing the first such demonstration in the context of cryptocurrencies.

6 Acknowledgements

We thank Andrew Miller for helpful and detailed explanations of his work [MHKS14], for running his code to get us comparison data, and for comments on our draft.

References

[AGT01] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In George I. Davida and Yair Frankel, editors, *Information Security, 4th International Conference, ISC 2001, Malaga, Spain, October 1-3, 2001, Proceedings*, volume 2200 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2001. Available at `http://aris.me/pubs/pad.pdf`.

[ANWOW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-OHearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[AVL62] Adel'son-Vel'skii and Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2), 1962. English translation in Soviet Math. Doklady 3, 1962, 1259–1263.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012. Avialable at `https://ed25519.cr.yp.to/`.

[BEG+91] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 90–99. IEEE Computer Society, 1991. Later appears as [BEG+94], which is available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991`.

[BEG+94] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991`.

[BGM16] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, volume 9604 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2016. Available at `http://arxiv.org/abs/1406.5694`.

[BGV11] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 111–131. Springer, 2011. Available at `http://eprint.iacr.org/2011/132`.

[BP07] Giuseppe Di Battista and Bernardo Palazzi. Authenticated relational tables and authenticated skip lists. In Steve Barker and Gail-Joon Ahn, editors, *Data and Applications Security XXI, 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Redondo Beach, CA, USA, July 8-11,* *2007, Proceedings*, volume 4602 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2007. Available at http://www.ece.umd.edu/~cpap/published/alex-ber-cpap-rt-08b.pdf.

[But16] Vitalik Buterin. Transaction spam attack: Next steps, 2016. https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On scaling decentralized blockchains. In *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[CF13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, volume 7778 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2013. Available at http://eprint.iacr.org/2011/495.

[CLH+15] Xiaofeng Chen, Jin Li, Xinyi Huang, Jianfeng Ma, and Wenjing Lou. New publicly verifiable databases with efficient updates. *IEEE Trans. Dependable Sec. Comput.*, 12(5):546–556, 2015.

[CLW+16] Xiaofeng Chen, Jin Li, Jian Weng, Jianfeng Ma, and Wenjing Lou. Verifiable computation over large database with incremental updates. *IEEE Trans. Computers*, 65(10):3184–3195, 2016.

[cod] Implementation of authenticated data structures within scorex. https://github.com/input-output-hk/scrypto/.

[CW11] Scott A. Crosby and Dan S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Trans. Inf. Syst. Secur.*, 14(2):17, 2011. Available at http://tamperevident.cs.rice.edu/Storage.html.

[DJ07] Brian C. Dean and Zachary H. Jones. Exploring the duality between skip lists and binary search trees. In David John and Sandria N. Kerr, editors, *Proceedings of the 45th Annual Southeast Regional Conference, 2007, Winston-Salem, North Carolina, USA, March 23-24, 2007*, pages 395–399. ACM, 2007. Available at https://people.cs.clemson.edu/~bcdean/skip_bst.pdf.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10. IEEE, 2013.

[EK13] Mohammad Etemad and Alptekin Küpçü. Database outsourcing with hierarchical authenticated data structures. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 381–399. Springer, 2013. Available at http://eprint.iacr.org/2015/351.

[GA] Jean-loup Gailly and Mark Adler. gzip. http://www.gzip.org/.

[GPT07] Michael T. Goodrich, Charalampos Papamanthou, and Roberto Tamassia. On the cost of persistence and authentication in skip lists. In Camil Demetrescu, editor, *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, volume 4525 of *Lecture Notes in Computer Science*, pages 94–107. Springer, 2007. Available at http://cs.brown.edu/cgc/stms/papers/pers-auth.pdf.

[GPTT08] Michael T. Goodrich, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Athos: Efficient authentication of outsourced file systems. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security, 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings*, volume 5222 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2008. Available at http://www.ece.umd.edu/~cpap/published/mtg-cpap-rt-nikos-08.pdf.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. Available from http://professor.ufabc.edu.br/~jesus.mena/courses/mc3305-2q-2015/AED2-13-redblack-paper.pdf.

[GSTW03] Michael T. Goodrich, Michael Shin, Roberto Tamassia, and William H. Winsborough. Authenticated dictionaries for fresh attribute credentials. In Paddy Nixon and Sotirios Terzis, editors, *Trust Management, First International Conference, iTrust 2003, Heraklion, Crete, Greece, May 28-30, 2002, Proceedings*, volume 2692 of *Lecture Notes in Computer Science*, pages 332–347. Springer, 2003. Available at http://cs.brown.edu/cgc/stms/papers/itrust2003.pdf.

[GT00] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical Report, Johns Hopkins Information Security Institute; available at http://cs.brown.edu/cgc/stms/papers/hashskip.pdf, 2000.

[GTS01] M.T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. Available at http://cs.brown.edu/cgc/stms/papers/discex2001.pdf; also presented in Proc. DARPA Information Survivability Conference & Exposition II (DISCEX II), 2001.

[HPPT08] Alexander Heitzmann, Bernardo Palazzi, Charalampos Papamanthou, and Roberto Tamassia. Efficient integrity checking of untrusted network storage. In Yongdae Kim and William Yurcik, editors, *Proceedings of the 2008 ACM Workshop On Storage Security And Survivability, StorageSS 2008, Alexandria, VA, USA, October 31, 2008*, pages 43–54. ACM, 2008. Available at http://www.ece.umd.edu/~cpap/published/alex-ber-cpap-rt-08b.pdf.

[KKR+16] Aggelos Kiayias, Ioannis Konstantinou, Alexander Russell, Bernardo David, and Roman Oliynykov. A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Report 2016/889, 2016. http://eprint.iacr.org/2016/889.

[Knu98] Donald Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[Lop] Jameson Lopp. Unspent transactions outputs in Bitcoin. `http://statoshi.info/dashboard/db/unspent-transaction-output-set`, accessed Nov 7, 2016.

[Mer89] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989. Available at `http://www.merkle.com/papers/Certified1979.pdf`.

[MHKS14] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 411–424. ACM, 2014. Project page and full version at `http://amiller.github.io/lambda-auth/paper.html`.

[Mil12] Andrew Miller. Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage), 2012. `https://bitcointalk.org/index.php?topic=101734.msg1117428`.

[Mil16] Andrew Miller. Private communication, 2016.

[MND+04] Charles U. Martel, Glen Nuckolls, Premkumar T. Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.75.3658`.

[MR98] Conrado Martínez and Salvador Roura. Randomized binary search trees. *J. ACM*, 45(2):288–323, 1998. Availabel at `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.243`.

[MWMS16] Meixia Miao, Jianfeng Wang, Jianfeng Ma, and Willy Susilo. Publicly verifiable databases with efficient insertion/deletion operations. *Journal of Computer and System Sciences*, 2016. Available on-line at `http://dx.doi.org/10.1016/j.jcss.2016.07.005`. To appearn in print.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[NN00] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7072`.

[nxt] The Nxt cryptocurrency. `https://nxt.org/`.

[Pap11] Charalampos Papamanthou. *Cryptography for Efficiency: New Directions in Authenticated Data Structures*. PhD thesis, Brown University, 2011. Available at `http://www.ece.umd.edu/~cpap/published/theses/cpap-phd.pdf`.

[Par15] Luke Parker. The decline in bitcoin full nodes, 2015. `http://bravenewcoin.com/news/the-decline-in-bitcoins-full-nodes/`.

[Pfa02] Ben Pfaff. GNU libavl 2.0.2, 2002.

[PT07] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *Information and Communications Security, 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceedings*, volume 4861 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007. Available at `http://www.ece.umd.edu/~cpap/published/cpap-rt-07.pdf`.

[PTT16] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990. Available from `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.9072`.

[Rog06] Phillip Rogaway. Formalizing human ignorance. In Phong Q. Nguyen, editor, *Progress in Cryptology - VIETCRYPT 2006, First International Conferenceon Cryptology in Vietnam, Hanoi, Vietnam, September 25-28, 2006, Revised Selected Papers*, volume 4341 of *Lecture Notes in Computer Science*, pages 211–228. Springer, 2006. Available at `https://eprint.iacr.org/2006/281.pdf`.

[SA96] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. Available at `https://faculty.washington.edu/aragon/pubs/rst96.pdf`.

[sca] The Scala programming language. `http://www.scala-lang.org/`.

[Sed08] Robert Sedgewick. Left-leaning red-black trees, 2008. Available at `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`.

[tbp] Transactions per block. `https://blockchain.info/charts/n-transactions-per-block`.

[Tea16] The Go Ethereum Team. Official golang implementation of the ethereum protocol, 2016. `http://geth.ethereum.org/`.

[Wei06] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java (Second Edition)*. Pearson, 2006.

[Wik13] Bitcoin Wiki. Cve-2013-2293: New DoS vulnerability by forcing continuous hard disk seek/read activity, 2013. `https://en.bitcoin.it/wiki/CVE-2013-2293`.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Available at `http://gavwood.com/Paper.pdf`, 2014.

A Definition of Security

Notation and Functionality. We assume that a data structure goes through a succession of states $S_0, S_1, \ldots$, where S_i is a result of applying some operation op_i to the state S_{i-1}. The state S_0 (e.g., a tree with a single $-\infty$ sentinel) is known to everyone. A data structure state is *valid* if it can be obtained from S_0 by a sequence of operations (e.g., an unbalanced AVL tree is not valid); we should never start with an invalid data structure, and our security and functionality definitions provide no guarantees in such a case. If an operation does not change the state, then $S_i = S_{i-1}$. In addition to possibly changing the state, the operation may also return a value ret_i. We assume that the change in the state and the return value are deterministic; randomized data structures are modeled by explicitly specifying the randomness used as part of op. There is an efficient deterministic function D that takes a state S and computes the digest $D(S)$; let $D_0 = D(S_0)$.

The honest prover has S_{i-1} (plus other information—e.g., labels of nodes—needed to make the data structure authenticated, which we do not include explicitly to avoid overburdening notation), and, in addition to performing the operation op_i, outputs a proof π_i, which goes to the verifier. Let the prover's algorithm be denoted by $\mathcal{P}(S_{\mathsf{old}}, \mathsf{op}) \to (\pi, S_{\mathsf{new}}, \mathsf{ret})$ and the verifier's algorithm be denoted by $\mathcal{V}(D_{\mathsf{old}}, \mathsf{op}, \pi) \to (\text{“accept”}/\text{“reject”}, D', \mathsf{ret}')$. Formally *completeness* (also known as correctness) requirement is that for any valid data structure state S_{old} and any operation op, if $D_{\mathsf{old}} = D(S_{\mathsf{old}})$, the algorithm $\mathcal{P}(S_{\mathsf{old}}, \mathsf{op})$ outputs $(\pi, S_{\mathsf{new}}, \mathsf{ret})$ such that $\mathcal{V}(D_{\mathsf{old}}, \mathsf{op}, \pi)$ accepts and outputs $D' = D(S_{\mathsf{new}})$ and $\mathsf{ret}' = \mathsf{ret}$.

Primary Security Goal: Soundness. The primary security requirement, called *soundness*, is that no computationally bounded adversary can make a proof π' that causes the verifier to accept and output incorrect D'_{new} or ret'.

Because we base our security on the security of collision-resistant hashing for a fixed hash function, we cannot talk of a “nonexistence” of such an adversary, because once the function is fixed, an adversary who knows a collision exists in a mathematical sense (even if no one knows how to build one). Instead, we talk about an efficient algorithm that transforms such an adversary into a hash collision, in the style of [Rog06].

Formally, the security requirement is formulated in terms of a collision-resistant hash function H, which is used in the algorithms of the prover and the verifier, and the algorithm D. Call a triple (S, op, π^*) *malicious* if S is a valid data structure state but $\mathcal{V}(D(S), \mathsf{op}, \pi^*)$ outputs (“accept”, D'_{new}, ret'), and either $D'_{\mathsf{new}} \neq D(S_{\mathsf{new}})$ or $\mathsf{ret}' \neq \mathsf{ret}$. The security requirement is that there exists an efficient algorithm R that, given any malicious triple as input, outputs $x \neq y$ such that $H(x) = H(y)$.

Secondary Security Goal: Verifier Efficiency. Our secondary security goal, to reduce the possibility of denial of service attacks on verifiers, is that a verifier is efficient regardless of what an honest or a malicious prover does. Specifically, we require that for a data structure with n elements, the verifier running time is guaranteed $O(\log n)$ regardless of the input. That is, if S is a valid data structure with n elements and $D(S)$ is its digest, then $\mathcal{V}(D(S), \mathsf{op}, \pi^*)$ completes its operation in $O(\log n)$ time for any op and π^* (in particular, this implies that $\mathcal{V}$ will not even read π^* that is too long, and that honest proofs π will always have length $O(\log n)$).

B Proof of Security

Our Algorithms. The exact structure of π and the exact verifier algorithm are not important for the security proof. Here are the salient features that we use to prove security.

$\mathcal{V}$ creates a partial tree T using the information in op and in π. This tree starts at the same root, but some branches terminate earlier than the corresponding branches in S. Every node in T can be either a *label-only* node, containing no children and no information other than the label (such a node terminates a branch), or a *content* node, containing all the same fields as the corresponding node in S (including a bit indicating whether it is a leaf or not), except omitting the key in the case of an internal node. Every content node in T is either a leaf or has both children (thus, every node in T has its sibling). The labels of label-only nodes are read from the proof, while the labels of content nodes are obtained by $\mathcal{V}$ via an application of H (and are never read from the proof). $\mathcal{V}$ checks that the label of the root of T matches the label of the root of S contained in $D(S)$.

$\mathcal{V}$ performs the same operation on T as $\mathcal{P}$ performs on S to compute ret and $D(S_{\mathsf{new}})$, with the following important difference: when search for a key key requires a comparison between key and $\mathsf{t.key}$ for some non-leaf node t in T in order to decide whether to go left or right, $\mathcal{V}$ does not perform the comparison, but reads the left-or-right direction from the proof. Then, when the search reaches a leaf f (and every such search must reach a leaf, or else $\mathcal{V}$ rejects), $\mathcal{V}$ checks that $\mathsf{f.key} \leq \mathsf{key} < \mathsf{f.nextKey}$ (in particular, if this inequality is strict, then $\mathcal{V}$ determines that the key is not found). $\mathcal{V}$ also rejects if T does not contain enough content nodes to compute the values ret and $D(S_{\mathsf{new}})$.

We do not address completeness here, as it is easy to see.

Soundness. We now need to show the collision-finding algorithm $R(S, \mathsf{op}, \pi^*)$. R will run the verifier $\mathcal{V}(D(S), \mathsf{op}, \pi^*)$ to get the partial verifier tree T^*. We will say that a node t^* in the verifier tree T^* does not match a node t in S if the nodes are in the same position (defined by the path from the root), but some information in t^* contradicts corresponding information in t. (Note that a verifier node may not have all of the information of the prover node—it may contain only a label, or not contain a key—but this alone is not reason to call the nodes "not matching." Information that is present in both nodes must be different in order for nodes to be considered not matching.)

If t^* exists in T^*, but a node t in the same position either does not exist in S or does not match t^*, then R can easily find a hash collision, as follows. First, find a highest t^* in T^* for which this is true (breaking ties arbitrarily). A node in the same position must exist in S for the following reason: if t^* is the root of T^*, then a node in the same position exists in S because S is never empty, because S has a sentinel. Else, the parent p^* of t^* is a higher node and a content node and a non-leaf, which means a node p matching p^* must exist in S and must also be a non-leaf (because the leaf-indicator bits of p and p^* must match), and every non-leaf in S has both children.

Now consider t^* and t. If the labels of these two nodes do not match, they are not roots (because $\mathcal{V}$ checks that the label of the root of T^* matches the label of the root of S contained in the digest $D(S)$). Therefore, the labels of their parents match (because t^* and t are a highest non-matching pair), and thus we find a hash collision in the computation of the parent labels (recall that the verifier tree T^* has every node's sibling, and thus the label of the sibling of t^* is available to R in order to create the input to H). If the labels of t^* and t match, then some other content does not; therefore, t^* is a content node, and all the contents needed to compute its label is in T^* and thus available to R. Thus R can output a hash collision on the contents of these two nodes.

It remains to consider the case when every node in T^* has a corresponding node in S and matches it. In this case, we will derive a contradiction to the statement that either $\mathsf{ret}' \neq \mathsf{ret}$ or $D' \neq S_{\mathsf{new}}$. Let key be the the key specified in op. In order for $\mathcal{V}$ to accept, the search path indicated in the proof must lead to a leaf f^* in T^* with $\mathsf{f.key} \leq \mathsf{key} < \mathsf{f.nextKey}$. Let f be the matching leaf in S. Because S is valid, there is only one leaf in S such that $\mathsf{f.key} \leq \mathsf{key} < \mathsf{f.nextKey}$, and therefore the honest prover's search path would lead to f. Thus, the search path of the $\mathcal{V}$ is the same as the search path of the honest prover $\mathcal{P}$ would be. All the other steps of $\mathcal{V}$ are also the same on T^* as the corresponding steps of the honest $\mathcal{P}$ on S, by design. Therefore, they perform the same steps on the same tree, and $\mathcal{V}$ will compute the same ret and $D(S_{\mathsf{new}})$ as $\mathcal{P}$.

Verifier Efficiency. Our AVL+ plus trees also satisfy our secondary security goal of verifier efficiency. We add the maximum tree depth $D(S)$ to the digest of S. In AVL+ trees, this depth is guaranteed to be at most $1.4405 \log_2(n+2)$ [Knu98, p. 460]. $\mathcal{V}$ will reject as soon as π leads $\mathcal{V}$ down a path that is deeper than this depth. Since honest π is proportional in length to this path (in fact, honest π contains only one path, except in case of deletions, when it contains up to two paths and some children/grandchildren of nodes on the paths), $\mathcal{V}$ can immediately reject any longer π. Thus, the longest possible π that $\mathcal{V}$ will read has length $O(\log n)$. The size of the partial tree T is bounded by the size of the proof π, and running time of $\mathcal{V}$ is linear in T.

Recall that when we compress proofs for multiple operations together, the verifier gets a partial tree for all the combined operations. In this case, the verifier continually checks the maximum depth condition when reconstructing the tree. The verifier also continually checks that the number of leaves in the reconstructed tree does not exceed twice the number of operations that the proof is for (because insertions, modifications, and lookups touch only one leaf, while deletions touch two). Thus, the size of the proof that the verifier will read is $O(B \log n)$, where B is the number of operations that the proof is for.