Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies

Leonid Reyzin[1,3](✉), Dmitry Meshkov[2], Alexander Chepurnoy[2], and Sasha Ivanov[3]

[1] Boston University, Boston, USA
reyzin@bu.edu
[2] IOHK Research, Sestroretsk, Russia
{dmitry.meshkov,alex.chepurnoy}@iohk.io
[3] Waves Platform, Moscow, Russian Federation
sasha@wavesplatform.com

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*.

A. Kiayias (Ed.): FC 2017, LNCS 10322, pp. 376–392, 2017.
https://doi.org/10.1007/978-3-319-70972-7_21

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 verification (e.g., so-called light nodes, such as Bitcoin's SPV nodes [Nak08, Sect. 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].

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 cut almost by half.

Our setting of authenticated data structures—in which verifiers are able to compute the new digest after modifications—is often called the "two-party" case (because there are only two kinds of parties: provers and verifiers). It should not be confused with the easier "three-party" case addressed in multiple works [Mer89,NN00,GT00,GTS01,AGT01,MND+04,GPT07,CW11], in which verifiers are simply given the new digest after modifications (e.g., by a trusted data owner). While we design primarily for the two-party case, our results can be used also in the three-party case, and can, for example, replace 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.

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, Sect. 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., [Tod16, 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 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 cost.

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

[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.

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 proofs (whether valid or not) that take the verifier more time to process than a prespecified upper bound.

Importantly, the model assumes that the verifiers and the provers agree on which data structure operations need to be performed (in our cryptocurrency application, the operations will come from the transactions, and the verifier will check whether the operations themselves are valid by, for example, checking the signature and account balance of the payer). A verifier is not protected if she performs an operation that is different from the prover's, because she may still compute a valid new digest; she will notice this difference only if she is able to see that her new digest is different from the prover's new digest. The model also assumes that the verifier initially has the correct digest (for example, by

[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.

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, update of a value for a given key, and deletion by key.

We provide formal security definitions in the full version [RMCI16].

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 (which we also improve), in which modifications are performed by a trusted author and only lookups are performed by the provers, who trust the digest. Some works also 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. The explanation here is terse for lack of space; a more detailed and accessible explanation is available in the full version of the paper [RMCI16].

Starting Point: Merkle Tree. We start with a Merkle tree [Mer89] (not necessarily perfectly balanced) with leaves storing (key, value) pairs, sorted by key. Each internal node stores the minimum of its right subtree to enable searching like in binary search trees (the same way as in [NN00,AGT01,MHKS14], but not in [CW11], where (key, value) pairs are stored also in internal nodes, which, as we demonstrate below in Sect. 4, results in longer proofs). To ensure every non-leaf has two children, and every insertion creates a new internal node with an existing left leaf and a new right leaf, we start with a $-\infty$ sentinel. To enabling proving nonmembership of a key (in particular, during insertion), each leaf stores the next key in addition to its own.

Updates and Simple Insertions. If the prover updates the value stored at a leaf (for example, subtracting from it money used for a transaction), the authenticating path for the leaf already contains all the information needed to compute the new digest. Thus, the proof for an update is the same as the proof for a lookup. Similarly for insertions: as long as an insertion doesn't require rebalancing the tree, the authenticating path to the leaf where insertion took place already contains the information necessary to compute the new digest.

3.1 Our Improvements

Observation 1: 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]), and treaps [SA96] (and their equivalent randomly-balanced binary search trees [MR98]). They all maintain some extra information in the nodes that enables the insertion and deletion algorithms to make a decision as to whether, and how, to perform tree rotations in order to maintain a reasonably balanced tree. We will add this information to the hash function input for computing the label of each node and to the authenticating path. For insertions, 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 authenticating path already has sufficient information for the verifier to perform the insertion and the tree-balancing operation. This is the case for AVL trees and treaps, but not for red-black trees.

However, of all the balanced tree 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. According to Miller [Mil16], proofs for insertions in the red-black trees of [MHKS14] are therefore approximately three times longer than proofs for lookups. Other balancing approaches enable us to keep insertion proofs short.

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 that skip lists [Pug90] are equivalent to binary search trees. This view enables us to test the performance of skip lists and treaps with essentially the same implementation, which no prior implementation has done. (More details are provided in [RMCI16]).

Observation 4: Deterministic is Better. Treaps and skip lists perform well in expectation 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, Sect. 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 Sect. 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, updates, 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.

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). Our approach for compressing a batch of operations is described in [RMCI16].

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 [RMCI16].

4 Implementation and Evaluation

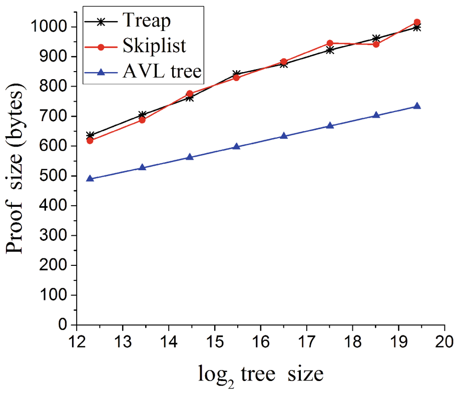
We implemented our AVL+ trees, as well as treaps and our tree-based skip lists, in the Scala [sca] programming language using the Blake2b [ANWOW13] hash function with 256-bit (32-byte) outputs. Our implementation is available at [cod][3]. For the AVL+ implementation, we used the textbook description [Wei06] with the same balance computation procedure as in [Pfa02, Chap. 5]. 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 updates 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

[3] Note that the implementation of AVL+ trees with proof compression for a batch of multiple operations is fully featured, while the other implementations (contained in subdirectory "legacy") are sufficient to perform the measurements reported in this paper, but are missing features, such as deletions, error handling, and compression of multiple proofs.

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.

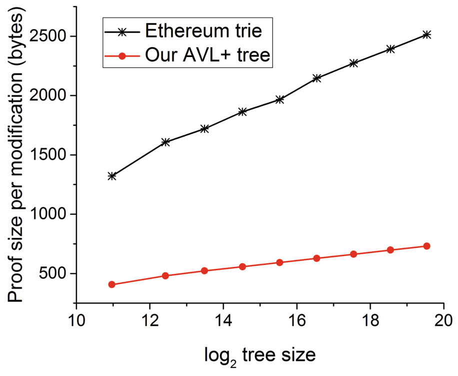
Proof length for deletions is more variable (because the deletion operation goes to two neighboring leaves and may also need off-path nodes for rotations), but is on average 50 bytes greater than for insertions, lookups, and updates.

Proof Length Comparison with Existing Work. Our numbers are consistent with those reported by Papamanthou and Tamassia [PT07, Sect. 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 (but not vice versa: three-party authenticated data structures do not work in our model, because they do not allow the verifier to compute the new digest, though some can be adapted to do so). 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, Fig. 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. Tendermint's implementation of Merkle AVL+ trees [Kwo16] has no provisions for proving absence of a key (nor for proving any modifications, because it is in the three-party model), but appears to have roughly the same proof length as ours when we adjust for hash and key lengths.

Proof Length for Multiple Operations. Compressing together proofs for a batch of B operations at once (using Observation 6 in Sect. 3) reduces the proof length per operation by approximately $36 \cdot \log_2 B$ bytes. This improvement is considerably greater than what we could achieve by concatenating individual proofs and then applying gzip [GA], which, experimentally, never exceeded 150 bytes, regardless of the batch size. The improvements reported in this section and in Fig. 1 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 updates and 1000 inserts was only 358 bytes per operation. If a transaction in a block modifies two

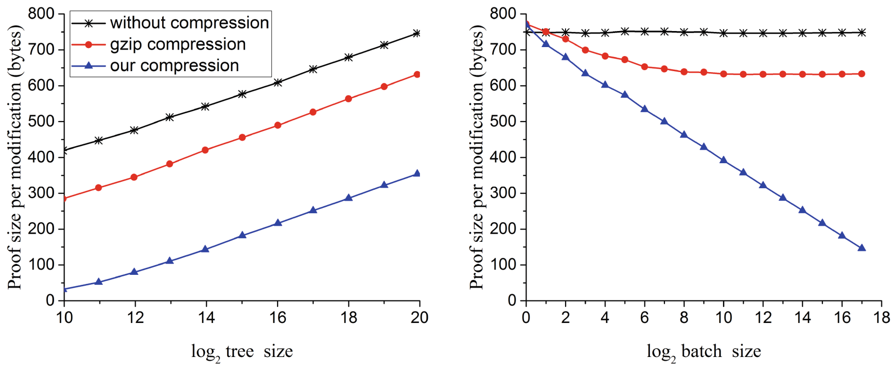
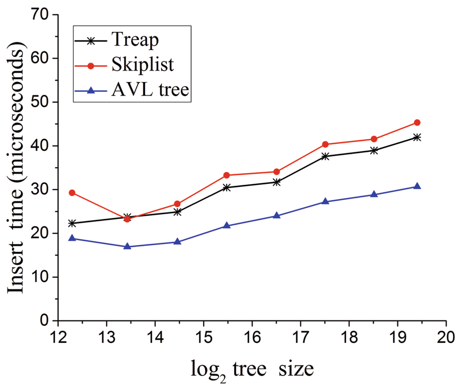


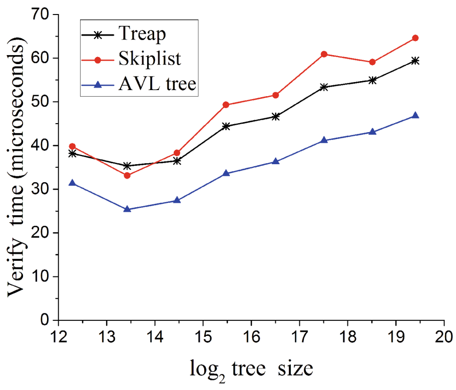
Fig. 1. Left: proof size per modification for $B = 2000$, as a function of starting tree size n. Right: proof size per modification for a tree with $n = 1\,000\,000$ keys, as a function of batch size B. In both cases, half of the modifications were inserts of new (key, value) pairs and half were changes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

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 repeated paths get compressed.

We can compare our results with those reported in Miller et al. [MHKS14, Fig. 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 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 @ 3.30 GHz Linux machine with 8 GB 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 (resp, verifier) time for inserting a random key into our AVL+ tree with 1 000 000 random keys was 31 μs (resp., 47 μs).

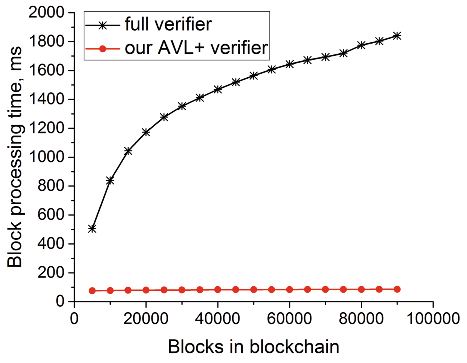




It is difficult to make comparisons of running times across implementations due the variations in hardware environments, programming language used, etc. Note, however, that regardless of those variables, the running times of the prover and verifier are closely correlated with path length k: the prover 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. Therefore, AVL+ trees perform about 1.4 times faster than skip lists.

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 @ 3.60 GHz Linux machine with 64 GB of RAM and SSD storage) to simulate two different methods of verifying account balances: simply maintaining a full on-disk (SSD) data structure of (key, value) pairs (similar to the process a traditional "full verifier" would perform) vs. maintaining only a digest of this data structure and verifying proofs for data structure operations, using very little RAM and no on-disk storage (similar to the process a "light verifier" would perform when provers use our AVL+ trees). 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 updates 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 light verifier were limited to 1 GB of RAM. Because the actual machine had 64 GB 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 s. 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 85 ms 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 48 GB of RAM. The prover stayed at about 70 ms 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 280 ms on the same machine (using the Ed25519 [BDL+12] signature scheme). The proofs size varied from 0.8 to 1.0 MB per block (i.e., 423–542 bytes per data structure operation).

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.

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. We thank Peter Todd and Pieter Wuille for fascinating discussions.

References

[AGT01] Anagnostopoulos, A., Goodrich, M.T., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: Davida, G.I., Frankel, Y. (eds.) ISC 2001. LNCS, vol. 2200, pp. 379–393. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45439-X_26

[ANWOW13] Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: simpler, smaller, fast as MD5. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 119–135. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38980-1_8

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

[BDL+12] Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.-Y.: High-speed high-security signatures. J. Cryptographic Eng. **2**(2), 77–89 (2012). https://ed25519.cr.yp.to/

[BEG+91] Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: 32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1–4 October 1991, pp. 90–99. IEEE Computer Society (1991). Later appears as [?], which is available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991

[BGM16] Bentov, I., Gabizon, A., Mizrahi, A.: Cryptocurrencies without proof of work. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 142–157. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_10

[BGV11] Benabbas, S., Gennaro, R., Vahlis, Y.: Verifiable delegation of computation over large datasets. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 111–131. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_7

[BP07] Di Battista, G., Palazzi, B.: Authenticated relational tables and authenticated skip lists. In: Barker, S., Ahn, G.-J. (eds.) DBSec 2007. LNCS, vol. 4602, pp. 31–46. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73538-0_3

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

[CDE+16] Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün, E.: On scaling decentralized blockchains. In: Proceedings of 3rd Workshop on Bitcoin and Blockchain Research (2016)

[CF13] Catalano, D., Fiore, D.: Vector commitments and their applications. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 55–72. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36362-7_5

[CLH+15] Chen, X., Li, J., Huang, X., Ma, J., Lou, W.: New publicly verifiable databases with efficient updates. IEEE Trans. Dependable Sec. Comput. **12**(5), 546–556 (2015)

[CLW+16] Chen, X., Li, J., Weng, J., Ma, J., Lou, W.: Verifiable computation over large database with incremental updates. IEEE Trans. Comput. **65**(10), 3184–3195 (2016)

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

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

[DJ07] Dean, B.C., Jones, Z.H.: Exploring the duality between skip lists and binary search trees. In: John, D., Kerr, S.N. (eds.) Proceedings of the 45th Annual Southeast Regional Conference, 2007, Winston-Salem, North Carolina, USA, 23–24 March 2007, pp. 395–399. ACM (2007). https://people.cs.clemson.edu/~bcdean/skip_bst.pdf

[DW13] Decker, C., Wattenhofer, R.: Information propagation in the bitcoin network. In: IEEE P2P 2013 Proceedings, pp. 1–10. IEEE (2013)

[EK13] Etemad, M., Küpçü, A.: Database outsourcing with hierarchical authenticated data structures. In: Lee, H.-S., Han, D.-G. (eds.) ICISC 2013. LNCS, vol. 8565, pp. 381–399. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12160-4_23

[GA] Gailly, J.-L., Adler, M.: gzip. http://www.gzip.org/

[GPT07] Goodrich, M.T., Papamanthou, C., Tamassia, R.: On the cost of persistence and authentication in skip lists. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 94–107. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72845-0_8

[GPTT08] Goodrich, M.T., Papamanthou, C., Tamassia, R., Triandopoulos, N.: Athos: efficient authentication of outsourced file systems. In: Wu, T.-C., Lei, C.-L., Rijmen, V., Lee, D.-T. (eds.) ISC 2008. LNCS, vol. 5222, pp. 80–96. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85886-7_6

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

[GSTW03] Goodrich, M.T., Shin, M., Tamassia, R., Winsborough, W.H.: Authenticated dictionaries for fresh attribute credentials. In: Nixon, P., Terzis, S. (eds.) iTrust 2003. LNCS, vol. 2692, pp. 332–347. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44875-6_24

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

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

[HPPT08] Heitzmann, A., Palazzi, B., Papamanthou, C., Tamassia, R.: Efficient integrity checking of untrusted network storage. In: Kim, Y., Yurcik, W. (eds.) Proceedings of the 2008 ACM Workshop On Storage Security and Survivability, StorageSS 2008, Alexandria, VA, USA, 31 October 2008, pp. 43–54. ACM (2008). http://www.ece.umd.edu/~cpap/published/alex-ber-cpap-rt-08b.pdf

[KKR+16] Kiayias, A., Konstantinou, I., Russell, A., David, B., Oliynykov, R.: A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Report 2016/889 (2016). http://eprint.iacr.org/2016/889

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

[Kwo16] Kwon, J.: Tendermint go-merkle (2016). https://github.com/tendermint/go-merkle

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

[Mer89] Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_21

[MHKS14] Miller, A., Hicks, M., Katz, J., Shi, E.: Authenticated data structures, generically. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 411–424. ACM (2014). http://amiller.github.io/lambda-auth/paper.html

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

[Mil16] Miller, A.: Private communication (2016)

[MND+04] Martel, C.U., Nuckolls, G., Devanbu, P.T., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica **39**(1), 21–41 (2004). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.75.3658

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

[MWMS16] Miao, M., Wang, J., Ma, J., Susilo, W.: Publicly verifiable databases with efficient insertion/deletion operations. J. Comput. Syst. Sci. (2016). http://dx.doi.org/10.1016/j.jcss.2016.07.005. To appear in print

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

[NN00] Naor, M., Nissim, K.: Certificate revocation and certificate update. IEEE J. Sel. Areas Commun. **18**(4), 561–570 (2000). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7072

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

[Pap11] Papamanthou, C.: Cryptography for efficiency: new directions in authenticated data structures. Ph.D. thesis, Brown University (2011). http://www.ece.umd.edu/cpap/published/theses/cpap-phd.pdf

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

[Pfa02] Pfaff, B.: GNU libavl 2.0.2 (2002). http://adtinfo.org/libavl.html/index.html

[PT07] Papamanthou, C., Tamassia, R.: Time and space efficient algorithms for two-party authenticated data structures. In: Qing, S., Imai, H., Wang, G. (eds.) ICICS 2007. LNCS, vol. 4861, pp. 1–15. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77048-0_1

[PTT16] Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables based on cryptographic accumulators. Algorithmica **74**(2), 664–712 (2016)

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

[RMCI16] Reyzin, L., Meshkov, D., Chepurnoy, A., Ivanov, S.: Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. Technical report 2016/994, IACR Cryptology ePrint Archive (2016). http://eprint.iacr.org/2016/994

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

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

[Sed08] Sedgewick, R.: Left-leaning red-black trees (2008). 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/

[Tod16] Todd, P.: Making UTXO set growth irrelevant with low-latency delayed TXO commitments (2016). https://petertodd.org/2016/delayed-txo-commitments

[Wei06] Weiss, M.A.: Data Structures and Algorithm Analysis in Java, 2nd edn. Prentice Hall, Pearson (2006)

[Whi15] White, B.: A theory for lightweight cryptocurrency ledgers (2015). http://qeditas.org/lightcrypto.pdf. (see also code at https://github.com/bitemyapp/ledgertheory)

[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] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2014). http://gavwood.com/Paper.pdf