Essay on Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies

Anders Martoja
University of Tartu
andersm9@ut.ee
November 24, 2017

Abstract

In cryptocurrencies a public ledger needs to be easily verifiable. However, maintaining a data structure of all the account balances can become quite problematic. As a first solution, keeping the data structure in secondary memory is slow due to its access times. From a different point of view, maintaining it in RAM upsets the whole idea of decentralized cryptocurrencies due to requiring powerful machines. The article by Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy and Sasha Ivanov proposes an improved design and implementation of two-party and three-party authenticated dynamic dictionaries which are applied to cryptocurrency ledgers. The performance of the proposed schemes are evaluated under realistic transaction loads. The aim of this essay is to highlight the key points of their article.

I. Introduction

The motivation behind [1] is to make transaction validation easier. It is hard to validate if a person A has the required amount of currency needed for successful transaction. Stateless part of the transaction: syntax check, signature of person A etc. is present in the transaction itself. While the stateful part of the validation which checks that A has the necessary funds requires knowledge of A’s previous transactions. A key-value-store is necessary for validation transactions. It contains information about public keys mapped to the values. The problem is that the data structure is large and growing. Currently such data structure for Bitcoin is roughly 1.5 GB. When the data structure is stored in secondary storage, access and validation of transactions become slow. When the dictionary is stored in RAM, then only powerful computers could store it, which would defeat the decentralization principle of cryptocurrency.

II. The model of Two-Party Authenticated Dictionaries

Authors of [1] use two-party model of authenticated dictionaries, because it permits insertion, lookup, update and delete. Each state of the authenticated dynamic dictionaries is accompanied by a unique digest. Only verifiers and provers are part of the model. Provers maintain a copy of the data structure and operate on it. Proofs of operations are sent to verifiers. The latter only has the digest of the current state of the data structure. Verifiers use the proof to obtain the result of the operation and update their digests when the data structure is modified. The goal is to prevent provers from tricking verifiers into accepting wrong results. It is also important to prevent the provers from generating proofs which would take verifiers more time to check than the prespecified upper bound. Two-party authentication assumes that the provers and verifiers are in unison about which data structure is worked with. And that the verifiers initially possesses the correct structure.

III. Construction

The authors of [1] started with a Merkle tree, where the digest is the label of the root. The proof that a key with a value is present consists of labels of sibling nodes on the path from root to the leaf. Information about going left or right at each step is also included. The proof also known as the authenticating path is checked by recomputing the alleged root label and comparing it to digest. The Merkle tree is modified into a variation of the Binary Search Tree (BST) to enable searches. Values are kept only at the leaves, so the internal nodes are used for searching. Further improvements were made by implementing balancing operations similar to those of AVL trees [2]. Extra information is stored in the nodes for the insertion and
deletion algorithms to maintain a reasonably balanced tree. Internal keys are not hashed, they are not in the proof. The verification of a leaf requires information about the hash chain that connects the root and to the leaf. AVL trees were chosen because they performed the best in lookups, updates, and insertions. Their cost is determined by the depth of the leaf. On average the distance from root to a random leaf for AVL trees after inserting $n$ random keys, close to the optimal $\log_2 n$, at the same time the worst-case is only 1.44 times the optimal. In expectations they outperformed other data structures that the authors considered. The authors also proposed several optimizations, e.g. when multiple operations are processed together their proofs can be compressed. The verifier simply reconstructs the relevant portion of the tree by using the proof and computes the label of the root of the reconstructed tree. The verifier runs essentially the same algorithm as the prover to execute the modifications. The difference is that the verifier uses the left-or-right directions from the proof on internal nodes and adds a check to make sure the sought after key is equal to the key in the leaf or between the key in the leaf and the key of the next leaf. The authors decided on using an AVL+ tree where the values are stored only in the leaves.

IV. Implementation and Evaluation

An evaluation was done by implementing AVL+ trees, treaps and tree-based skip-lists in Scala using Blake2b hash function with 256-bit outputs. Experiments measured the cost of 1000 random insertions, into the data structures that already had the keys in them. The authors also predicted correctly, that the AVL+ tree performed only 2-3 percent worse than the predicted optimal $\log_2 n$. The average length of the proof for inserting a new key into a 1000000-node tree was 753 bytes. The proof length is almost directly proportional to the path length. AVL-tree-based proofs are roughly 1.4 times shorter than skip-list-based ones. Deletion proof is roughly 50 bytes longer than for other operations. Compressing together proofs for a batch of operations at once, reduces the proof length per operation of approximately 36 times $\log_2 B$ bytes. Regardless of what hardware and programming languages are used (it is difficult to make comparisons), the running times of the prover and verifier are closely correlated with the path length $k$: the prover performs $k$ key comparisons and computes $k + 1$ hash values, while the verifier performs two comparisons and computes $2k + 1$ hash values. Rotations do not change these numbers. Two different account balance verifications are simulated. Firstly, the data structure was put onto a SSD. In the second case the data structure was kept in RAM. They measured only the data structure processing times. The full verifier running time grew rapidly, ending at about 1800 ms per block on average, while the light verifier stayed at about 85 ms per block. The proposed data structure had 20x speed advantage with large blocks, it is illustrated in figure 1.

![Figure 1: Full verifier vs proposed AVL+ verifier, taken from [1]](image)

V. Conclusion

The authors demonstrated significant performance improvements for two- and three-party authenticated data structures. An algorithm was developed for compressing operations. And the data structure can be used to improve blockchain verification in cryptocurrencies by light nodes without affecting full nodes [1].

References
