Providing Non-inclusion Proofs in Bitcoin append to right UTXOS Merkles

Abstract. Stateless nodes in Blockchains store a constant number of cryptographic hash values that make them able to verify transactions correctly via proofs without storing the complete system state. The cryptographic hash values are usually the root(or roots) of some Merkle strucutre stored in a bridge server holding the complete system state, which is the current set of all Unspent Transaction Outputs UTXOS in Bitcoin. While earlier research keyed the Merkle strucutre on the UTXO hash value, newer research achieved more locality of reference and thus shorter batch proofs by simply appending new UTXOS to the right. However, this comes with the drawback of losing the ability to provide non-inclusion or fraud proofs. In this paper, we propose a design that can provide fraud proofs and reserve locality of reference for batch proofs at the same time with minimized overhead. The idea is that any append to the right design must keep on its server some kind of mapping from the original indexing, used by client nodes, of the UTXOS on hash value to the UTXOS as leaves arranged on their creation order. *We show how this mapping data strucutre could be carefully designed to provide fraud proofs* when needed by building a complete Merkle tree on top of it; compared to existing built-in language maps we can arrange a special purpose design to achieve an acceptable performance. Due to well studied bit-randomness of cryptographic hashes used in blockchains, we can assume UTXOS to fall in buckets based on any subset of hash bits according to a uniform distribution. This, still longer, fraud proof will only be send once per block in the worst case since an invalid UTXO invalidates the whole block in hand, and a per-block root calculation will allow discovering the details of double spending attempts within the same block.

Keywords: UTXO, Bitcoin, Merkle Tree, Fraud Proofs, Stateless Nodes, maps, hash tables.

1 Introduction

Blockchains could be viewed as an append only data structure, which size keeps increasing endlessly with more blocks being added. Full nodes, that form the consensus, have to keep the whole chain in its secondary storage and the whole UTXOS set which is about 4G in its local memory in order to correctly verify transactions in each block. SPV clients or lightning nodes on the otherhand keep only block headers and depend on full nodes to verify transactions, however this makes them more vulnerable to attacks. The concept of stateless nodes was a reasonable compromise where nodes ought to keep only an accumulated hash value in its local memory and fetch what is called a proof or witness needed to verify UTXOS in the current block. UTXOS Merkle Tree works the same way as the transactions Merkle Tree which it's root is stored in the block header [1]; Fig.1 summarizes the steps and the basic

idea: hashes of target items to be verified are inserted as leaves and a tree is built on them, the target hash is calculated by extracting hashes of sibling nodes (called a proof) along the path from the already stored accumulated root value and compared with the given UTXO hash by the transaction[2]. The fact that a data item does not exist in a Merkle Tree can be verified too, although not usually needed in the transactions Merkle Tree case. A Fraud *Proofs,* or sometimes called *Non-Inclusion* or *Non-Membership* proof, that a data item, say **X**, does not exist as a leaf in the Merkle Tree can be done by fetching the proofs of the largest data items **i**, and the smallest data item ***j*** that satisfy **i < X < j.** Then the receiver can verify from their siblings that they are adjacent in the Merkle Tree. For example in Fig.1, if the data item in question is hashed to a value X where **Hash1 < X < Hash2**; we reply with the proofs of Hash1 and Hash2. When the sender verify Hash1 & Hash2 and also verify they are adjacent in the Merkle, now it is proved that there are no items such as X exists between them.

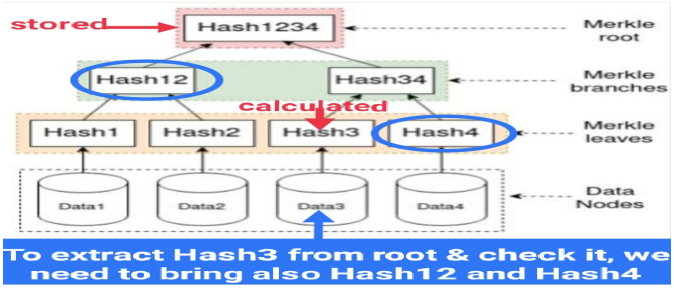


Fig.1 a simple Merkle Tree, annotated version of fig in [2]

Although stateless nodes concept is common to all types of cryptocurrency blockchains, this paper is concerned with Bitcoin which is an ***Unspent Transaction Output based Blockchain***, where coins or currency is the main component the system has to keep track of[3]; ie unlike the normal banking and programming model a transaction output UTXO is only used once as an input to a new transaction then its lifespan ends (becomes spent). The concept aims to prevent double spending attempts, where any money transfer on the ledger is viewed as the end of existence of the transaction inputs (that was before an Unspent Output of a previous transaction) and hence the creation of new Unspent outputs tied to this transaction. The state of the blockchain at any instance(block) is defined as the current set of Unspent Transaction Outputs UTXOS.

In general there is another type of cryptocurrency blockchains, ***account-based blockchains,*** Ethereum is the most widely used [4]. Cardano [5,6] is an example of a two-mode cryptocurrencyis, where the UTXO-based mode divide the UTXOS into 4 types and restricts the use of UTXOS to 1 per transaction. An AVL^{+} tree is used as the underlying Merkle for the UTXOS, and hence we think fraud proofs are available in Cardano main design.

In Bitcoin, the problem of the growing size of the UTXOS set have been addressed in [7,8] even before the concept of stateless nodes, research efforts aimed to reduce proof sizes by choosing an appropriate underlying Merkle structure that reduce the leaf-to-root path length (depth of the tree) using the

UTXO hash as the search key for that structure. Newer research [9,10] relied more on a main heuristic that is similar to the multi-level cache heuristic "outputs that were spent recently are more likely to be re-spent (the new UTXOS resulting from them) than outputs that have not been spent in a long time" [6]; this locality of reference heuristic was best achieved by appending new UTXOS to the right and dropping the use of a hash as a search key. However, since leaves are arranged now according to their creation not to their hashes, such arrangements made it not possible to provide non-inclusion or fraud proofs from the resulting Merkle strucutre. Hash-keyed designs prove the non-existence of a certain UTXO hash *by sending the proofs of the nearest existing hashes preceding and succeeding the hash in query and showing they are adjacent leaves in the Merkle* [11], while in append to right designs those preceding and succeeding UTXOS are not adjacent in the Merkle Tree.

In this paper we present a suggestion to have both advantages, locality of reference and non-inclusion proofs, by slightly changing the "needed anyway" map/dictionary data structure in such designs, to map the UTXOS given their hash value to their positions as Merkle leaves ordered on creation time (block height and transaction ID). The "not found" (not OK in Go lang maps) already provided by any built-in map can not be trusted by client nodes as a non-inclusion proof without possessing a previously calculated value like in case of the Merkle root; we show how we can achieve that by building a shadow Merkle tree on top of it.

The rest of the paper is organized as follows section2 gives a brief summary of previous research on hash-keyed bdata structures suggested for storing UTXOs hashes, Section3 explains the main concepts of locality of reference and fraud proofs, then section4 get us back to the Merkle strucutres survey on append to the right designs this time ending with the Utreexo project forest. Section5 introduces our proposal; the idea, the proposed map design, and how it can provide non-inclusion proofs and its impact on performance. Finally section6 concludes the paper with some possible future work.

2 A brief History of Bitcoin UTXO Merkles

As early as 2011, both Greg Maxwell [12] and Dithi [7] suggested using a Merkle tree in order to query the UTXO set more efficiently. This was followed by recommending a TX-id indexed Merkle prefix Trie in 2012[13], Red-Black trees [14] to make incremental modification to the tree more efficient. Ever since then one can find a considerable number of projects trying to suggest different kinds of trees or data structures to store the hashes; the accumulator batching techniques introduced in [15] could be viewed as the end of those using the hash as the BST key of the Merkle strucutre. The designers emphasized in the abstract that they support batch non-membership proofs; to prove a UTXO doesn't exist, two proofs can be submitted that the nearest UTXO hashes before and after it are adjacent leaves in the Merkle strucutre. The accumulator, which was not specifically designed for the UTXOS case, has superior properties in terms of proof size and batch reductions. However, sometimes batching all required UTXOS proofs in a block is not valid; as will be explained in the following section.

However, we end this brief section by summarizing, in Fig.2 from[10], the main common model of the UTXOS Merkle design with a bridge server holding the Merkle structure answering proof queries by stateless nodes. We did not discuss an earlier idea was to link the proof(or the creation path) to the UTXO itself inside the transaction as it was found costly and appears now in Testnet only for demonstration purposes.

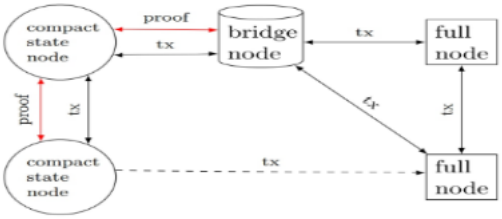


Fig. 2: The bridge server concept behind Stateless Node

3 Related Concepts

After the more basic idea of UTXOS, stateless nodes, Merkle trees, and proofs were introduced, in this section we explain and demonstrate the main concepts related; namely locality of reference and reductions in proof batching, and fraud proofs.

3.1 Locality of Reference

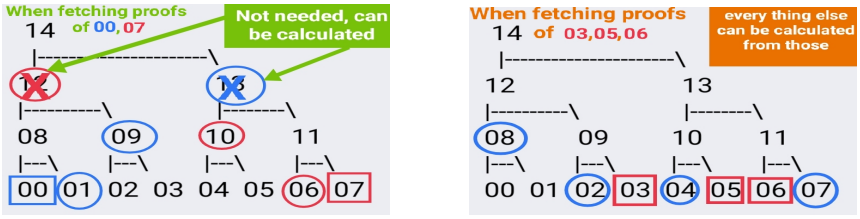
Many describe this heuristic governing the UTXOS set as a varient of the multi-level cache heuristic, with the main idea is that newer UTXOS are more likely to spend than elder ones. Since the main research target was to achieve smaller proof sizes to minimize time and memory requirements in sending them, researchers worked on achieving some inner nodes hash calculation reductions, and also having a large number of common nodes when sending batch proofs for many UTXOS say in a certain block.

Examples of those reductions and how the leaves adjacency leads to even more are illustrated in Fig.3 a&b. In Fig. (3-a) the required proofs are for two apareted nodes, 00&07, so we find that the proofs do not contain any common nodes; the only saving we can do is not sending nodes 12&13 as they can be calculated from already sent nodes...

12= concatenation of((00,01),09)
13=concatenation of(10,(06,07))

(Still not achievable if 00,07 are not batched together)
While on the otherhand, in Fig.(3-b) when proofing nodes 03,05,06 we have more common nodes in the fetched proofs that we can send only 4 nodes for proving 3 nodes (not 6 nodes if we compared to Fig.2a). Finally, if we are to prove a batch of nodes 00-03, or 04-07, we will only need 1 node as prove; 13 or 12 respectively. Note however, as mentioned previously, that batch proves does not always work correctly for cryptocurrency blockchains; sometimes

you must recalculate the accumulator root value and some modified Internal nodes before fetching any more proves to prevent double spending attacks. For example , if you have to recalculate the root between fetching node 00& node 07 in (3-a), or nodes 03,05,06 in (3-b) this will change the internal nodes values and make the reduction invalid. Some exiting projects like [10] use *Copy On Write (COW)* form of their Merkle data structure[16].



a) 4nodes needed for 2 apart nodes. **b)** still 4nodes for 3 near nodes

Fig.3 examples of batch proof reductions when some siblings can be calculated

Append to the right UTXOS Merkle designs [9,10] claim they achieve more locality of reference than hash-keyed designs [12,13,14] because recent UTXOS get stored together at the right, and also get spent together before older UTXOS who remain at the left, which results in more node reductions in batch proofs. While in hash keyed Merkle trees, UTXOS are stored according to their hashes whatever their creation time or even block; two input UTXOS in the same transaction could end up stored scattered even to the most right and most left positions..Although an optimal path length (tree depth) is O(log N), the Utreexo project in [10] claim they fetch only O(10 K) nodes batch proof for O(2K) UTXOS Proof request per block.

3.2 Fraud Proofs

When a stateless client requests the proof of a certain UTXO appearing in some transaction, and the hash of this UTXO is not found as a leaf in the Merkle strucutre, a simple *"not found"* answer can not be trusted by client nodes as a cryptographic proof of non-inclusion. Like inclusion proofs, the client node must possess a previously calculated accumulator hash value to check against.

Fraud Proofs in Bitcoin hash-keyed Merkles

In all hash-keyed UTXOS Merkle structures suggested in Bitcoin, non-membership proofs were achieved using the same UTXO Merkle root by sending the proofs of the nearest existing hashes preceding and succeeding the hash in query and showing they are adjacent leaves in the Merkle [11]. Fig.4 shows two examples of fraud proofs, considering the same Merkle Tree in Fig.3 after node 03 is deleted. When a proof is requested for node 03 after it was deleted (a possible double spending attempt), the server replies by sending two proofs for nodes 02&04; the client node then checks the correctness of their proofs against the stored root hash value and that they are adjacent in the tree. Note that adjacency could be discovered in this case

by knowing that 02 is the rightmost leaf branching from node 12, and 04 is the leftmost leaf branching from node 13, and that nodes 12,13 are siblings. The second example is of a proof request for a non existing hypothetical address value 005 that was never a leaf in the tree (a fake UTXO), the server will reply with proofs of nodes 00&01 and the client will find out simpler this time from the first sibling values that they are adjacent.

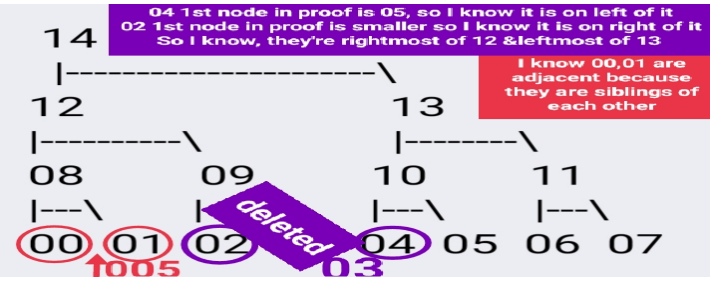


Fig.4: Examples of Fraud Proofs in hash-keyed UTXO Merkle designs in Bitcoin, here proofs are asked for a deleted value "03", and a never existed value "005"

Fraud Proofs in IoTs and other kinds of Blockchains

There are other implementations that use default NULL hash value, say the hash of all zeros for example, in place of non existing data items [17,18]. This is usually done in *permission Blockchains* like IoTs [18] where the number of data items is limited which make it more easier to keep a complete tree of all possible values. In this case, a non-inclusion proof is fetched systemically like an inclusion proof leading to the default NULL hash value; Fig.5. In our design we will use this kind of simpler fraud proofs for empty buckets, since empty locations in the hash table pointer vector are reserved anyway. In fact Fig.2 is inspired by comparison diagrams between (B+ and B-) Merkle trees in [16].

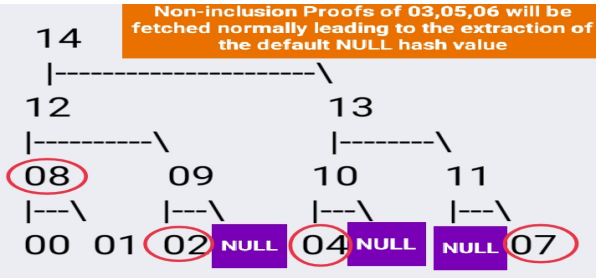


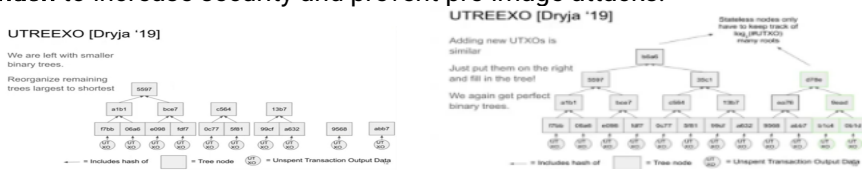
Fig.5: Fraud Proofs using a NULL default hash value

4 Append to The Right UTXO Merkles

In this section we explore the newer research series that chose to sacrifice the proof of non-inclusion feature and just append new UTXOS to the right in order to promote locality of reference among fetched proofs, with more emphasize on the Utreexo project.

Merkle Mountain Range MMR [19] was the first to our knowledge that dropped the hash keyed tradition and appended new UTXOS to the right. We could say that they also originated back in 2016[8] the intuition of using a number of *complete binary trees*, ***mountains***, that corresponds to the *1s* in the binary representation of *N* the number of leaves(UTXOS) with depth equal to their position. As an example a set of say 11 UTXOS =$(1011)_2$ can be stored in three complete Merkle binary trees of depth 3,1,0; *another hidden advantage of complete binary trees is that they are stored compactly as vectors without the need to use left & right pointers*. MMR is available as API and are now used in *Tari* cryptocurrency[20].

On the same track followed the widely known MIT-lab project **Utreexo**[10,21], which suggested ***a forest*** of complete binary trees stored in a bridge server, Fig.3, and new UTXOS get appended to the right. Specifically, new leaves start building smaller subtrees in powers of 2 that only joins the next larger subtree (next power of 2) when it becomes full. This guarantees a *bounded tree depth* of ***log(no of UTXOS)*** , and hence number of siblings in fetched proofs, with adding only ***log(no of UTXOS)*** *accumulated roots;* Fig.6 (taken from [22]). The designers say [23] an average block (about 2K transactions) with a few thousands of insertions & deletions only causes something on the order of 10K hashes, which is *much less than logN* per element (26-27 for the current UTXO set size of 76m) due to the *locality of reference* coming from the ***append to the right*** rule; the only missing feature is fraud proofs. Utreexo also studied the cache deployment for the fetched proofs, IBD Initial Block Download; they ***added block number to the UTXO hash*** to increase security and prevent pre-image attacks.



a)successive insertions **b)**complete subtrees join to form the next power of2 subtree

Fig.6 the Utreexo Forest

5 The proposed multi-functional map design

The specific design details of the Go language map used in Utreexo project for mapping UTXO hashes can be found in [23,24,25] where it is stated that it's average load factor is ~ 6.5 items per bucket; Fig5.

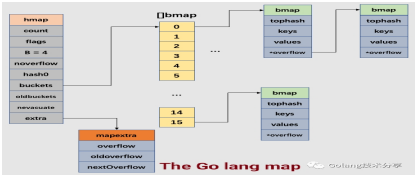


Fig.7: the Go lang map

Our proposed map design for this specific requirement of providing non-inclusion proofs, Fig. 8, can be summarized in the following steps:

1. A 2^{26} (could be tuned) pointer vector is allocated to put each UTXO in a bucket according to its hash.
2. Due to the Cryptographic hash robustness & bit randomness, SHA256 in Bitcoin [26,27], UTXO hashes should fall uniformly as all buckets are equiprobable and each bucket is expected to contain 1-2 UTXOS (less than Go lang map load factor with less book keeping fields, 2^{25} entry may lead to the same)
3. Insertion is adding a node (in order)to the bucket linked list, deletion is deleting a node from the bucket list; the vector bucket pointer maybe adjusted in the process from or to NULL value.
4. To save computation time, we will calculate and send the accumulated root value of this secondary tree only once per block; either at the end if valid or when encountering an invalid UTXO to send the non-inclusion proof and abort the whole block. According to the time versus space constraints we may choose to store the complete tree on top of the hash table, or recalculate it from scratch with each block like the block Transactions Merkle Tree.
5. The non-inclusion proof will be send according to the previous block accumulated root, so during batch preparation **a newly inserted UTXO will be flagged 1, a deleted UTXO is flagged -1, and 0 means value the same as previous block.**
6. If the block is valid, update and reset flags to 0 while computing the new root. A hash of a bucket is the concatenation of all its nodes (expected to be 2), and empty buckets are substituted by a default NULL hash value like the Merkle strucutres in [16,17]
7. If we had an invalid UTXO, we have two cases:
   a) The hash doesn't exist in any case (not even with a deleted flag), in this case we send the usual non-inclusion proof considering old status before any UTXO from this block. Note that if the UTXO bucket was completely empty, the proof of the default NULL hash[1] will be sent as in [16,17].
   b) The UTXO hash is there, but has a deleted flag (it was spent before during this block); ie, we are facing a double spending within the same block. In this case, we can provide the details of the original spending transaction; for example could send something like "TX ID .... tried to spend it with proof ....". Note that the original transaction will not be executed since the whole block will be aborted and the old state will be resumed, that's why the 2 bit flag is needed.

Each entry in the pointer vector is a pointer to its bucket list, ***a concatenated hash of its bucket elements***[2]***;*** could possibly contain the number of elements in its bucket and a bit flag indicating if one of its elements has changed this round to enhance the performance.

---

[1] Note that any NULL bucket must be substituted by the default NULL hash, and can not be just omitted, in tree calculations; because the tree must differ according to the place and number of NULL values to prevent second pre-image attacks. See EQ.1, p.10 in [18] for more details

[2] This field will not be needed if the whole Merkle tree is recalculated from scratch each block.

The strucutre of each item inside the bucket list:
-A pointer to the UTXO leaf in the main Merkle
-The value of the remaining hash bits, or could be omitted with its value stored with the leaf node.
-Two bit flag to facilitate the *Copy On Write* process:
0=no change
1= inserted this block
-1=deleted this block

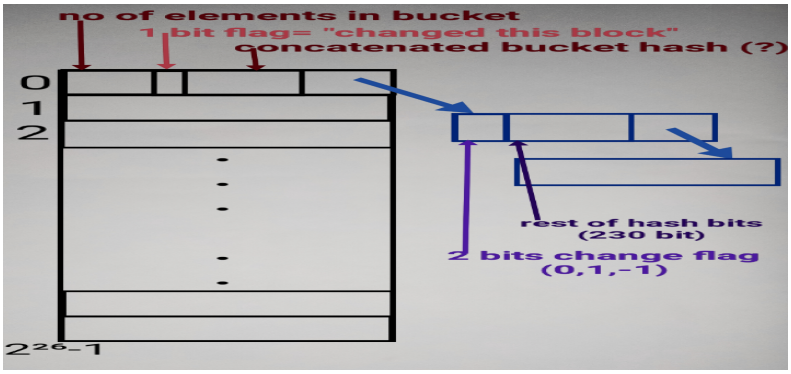


Fig.8: The proposed map design

Performance Analysis

If the shadow Merkle is recalculated each block no extra storage is needed, in fact the total map size is expected to be less than that of the Go language map. If the complete Merkle is stored on the other hand, or a halfway decesion of storing pointer vector hashes only, an *O(N)* storage will be added (no left and right pointers will be needed with the use of the vector representation to store the complete shadow Merkle) but with a better average time performance. In general, we do not expect a time performance degradation since the tree will be recalculated per block instead of per UTXO; we prefer to defer the firm statement till the experimental results. Finally, from security perspective in addition to fraud proofs, we think that ***being able to identify the two candidate transactions involved in a UTXO double spending*** is an added useful piece of information.

6 Conclusions & Future Work

In Blockchains stateless nodes stores only an accumulated Merkle root hash value of the system state, the UTXOS set in Bitcoin, and fetch proofs needed to verify the current block. Literature results show that append to the right Merkle designs like [8,9] achieve shorter barch proofs due to locality of reference between quiered UTXOS; however, these designs, as opposed to hash-keyed ones, lack the ability to provide fraud or non-inclusion proofs for non existing UTXOS. In this paper we proposed a design that can offer such

proofs while keeping the locality of reference. We suggest to make use of the already existing map data structure by replacing it with a special purpose designed one and building another shadow Merkle on top of it. The extra Merkle Tree will be calculated once per block to keep the overall system performance, and will send only fraud proofs once per block in the worstcase when an invalid UTXO is encountered. If the requested to spend UTXO was a subject of a double spend within the same block we can identify the two attempting to spend transactions; we believe this is an added security value.

The proposed map design need to be experimented to tune the number of buckets, number of UTXO hash bits used in the pointer vector, for the current size of the UTXO set ($2^{26} \leq 76m \leq 2^{27}$) we suggest the use of 26 bits leading to 2^{26} buckets as a starting point. Another design decision is whether to store this extra Merkle Tree or rebuild it each round (each block); as a starting point we are in favor of recalculating it since this is what is done for the block transactions Merkle Tree, a halfway solution could be to store each bucket concatenated hashes and rebuild the rest of the tree.

Acknowledgment

I thank Bolton Bailey, the author of [9], for patiently explaining how non-inclusion proofs are calculated through email; it originally explained in MIT lecture 23 about research directions [11]that I delayed studying for a while. Then I should also thank Mark Friefenbach, the author of [13], for originally pointing out the missing of such feature, while arguing with the Utreexo project team.

References

1. Russell O'Connor, "A Method for Computing Merkle Roots of Annotated Binary Trees", May 2017, https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2017-May/014362.html
2. Teemu Kanstrén, "Merkle Trees: Concepts and Use Cases", https://medium.com/coinmonks/merkle-trees-concepts-and-use-cases-5da873702318, 16 Feb 2021
3. C. Pérez-Solà, S. Delgado-Segura, G. Navarro-Arribas and J. Herrera-Joancomartí, "Another coin bites the dust: an analysis of dust in UTXO-based cryptocurrencies", January 2019, Royal Society Open Science 6(1):180817 DOI:10.1098/rsos.180817
4. https://blog.ethereum.org/2020/11/30/the-1x-files-code-merkleization/; accessed 6/2021
5. "Understanding UTXOS in Cardano", Design specification paper, https://emurgo.io/blog/understanding-unspent-transaction-outputs-in-cardano, accessed 24/5/2021.
6. https://hydra.iohk.io/build/790053/download/1/delegation_design_spec.pdf
7. https://en.bitcoin.it/wiki/User:DiThi/MTUT#PROPOSAL:_Merkle_tree_of_unspent_transactions_.28MTUT.29.2C_for_serverless_thin_clients_and_self-verifiable_prunned_blockchain
8. Todd P. 2016, "Making UTXO set growth irrelevant with low-latency delayed TXO commitments". bitcoin-dev mailing

list.https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-May/012715.html
9. Bolton Bailey and Suryanarayana Sankagiri, "Merkle Trees Optimized for Stateless Clients in Bitcoin", ifca, 5th Workshop on Trusted Smart Contracts, WTSC21
10. Dryja T.,"Utreexo: A dynamic hash-based accumulator optimized for the bitcoinutxo set.", IACR Cryptol, ePrint https://eprint.iacr.org/2019/611.
11. MAS.S62, Cryptocurrency Engineering and Design, lec23: "New Directions in Crypto", https://youtu.be/74_BKWR3n0k, Spring 2018.
12. https://bitcointalk.org/index.php?topic=21995.0
13. https://bitcointalk.org/index.php?topic=88208.0
14. Miller, A., Hicks, M., Katz, J., Shi, E.: "Authenticated data structures, generically", ACM SIGPLAN Notices 49(1), 411423 (2014)
15. Dan Boneh, Benedikt Bünz, and Ben Fisch, "Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains", Cryptology ePrint Archive: Report 2018/1188. https://eprint.iacr.org/2018/1188, last revised 20 May 2021.
16. Kasampalis, Sakis (2010). "Copy On Write Based File Systems Performance Analysis And Implementation" (PDF). p. 19.
17. Rasmus Dahlberg, Tobias Pulls, and Roel Peeters2, 'Efficient Sparse Merkle Trees Caching Strategies and Secure Non-Membership Proofs", NordSec 2016 Computer Science, DOI:10.1007/978-3-319-47560-8_13, Corpus ID: 3812588, https://eprint.iacr.org/2016/683.pdf
18. Alex Shafarenko, "Indexing structures for the PLS blockchain", arXiv:2107.08970v2 [cs.CR] 3 Aug 2021
19. https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md, last visited 14/10/2021
20. https://crates.io/crates/merklemountainrange, last visited 14/10/2021
21. Colin Harper, https://www.coindesk.com/human-rights-foundation-gives-new-bitcoin-development-grants, 31May2021
22. https://youtu.be/HEKtDILPeaI, Bolton Bailey, "Merkle Trees Optimized for Stateless Clients in Bitcoin", Mar,2021
23. https://dci.mit.edu/utreexo, https://github.com/mit-dci/utreexo
24. https://www.fatalerrors.org/a/a-comprehensive-analysis-of-golang-s-map-design.html
25. https://dave.cheney.net/2018/05/29/how-the-go-runtime-implements-maps-efficiently-without-generics
26. https://cs.stackexchange.com/questions/144081/how-does-the-go-language-implements-maps/144082?noredirect=1#comment304249_144082
27. Dan Michael A. Cortez, Ariel M. Sison ,Ruji P. Medina, "Cryptanalysis of the Modified SHA256", HPCCT&BDAI 2020, July 2020, pages: 279-183, https://doi.org/10.1145/3409501.3409513