TOWARDS CREATING A THIN CLIENT FOR MONERO

BY

KEVIN LEE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Electrical and Computer Engineering
in the College of Engineering of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Andrew Miller

ABSTRACT

As an increasing number of users begin to use cryptocurrencies, the number of transactions per day has also been steadily increasing. Users traditionally run full nodes which involve downloading a copy of the blockchain associated with that cryptocurrency and updating it through the network. This method guarantees full security at the cost of greater power consumption and data storage. The recent ubiquity of data connections and low-capacity hardware such as smartphones has only furthered the demand to delegate cryptocurrency-related computations to capable servers. This raises the need to verify results that are returned to a light client by a server, and also to perform conflict resolutions whenever servers disagree upon a query. In this study, we implement the storage of the Monero blockchain as authenticated data structures on powerful servers, and perform computations on the data structures as queries are made or as the blockchain is updated. Experimental results have shown that authenticated data structures can be used to create a secure thin client for Monero, and we conclude with a discussion on the possibility of integrating that thin client into the network.

Keywords: Cloud Computing; Verifiable Computation; Security; Cryptocurrency

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 4
2.1 Cryptocurrencies 4
2.2 Delegation of Computation 6

CHAPTER 3 SYSTEM DESIGN 8
3.1 Security Goals 8
3.2 Overview of Design 8
3.3 Data Structure Details 17

CHAPTER 4 SECURITY ANALYSIS 19
4.1 Security Properties 19
4.2 Incentives 19
4.3 Errors 20

CHAPTER 5 IMPLEMENTATION AND BENCHMARKS 22
5.1 Benchmarks 22

CHAPTER 6 CONCLUSION AND FUTURE WORK 25

REFERENCES 26

CHAPTER 1

INTRODUCTION

In the past decade, cryptocurrencies have gained worldwide attention for their use of cryptography to make secure transactions and verify transfers of assets between users. With the start of Bitcoin [1] in 2008, these systems provide public logs that are available to everyone. These logs, or *blockchains*, are immutable and large (the Bitcoin blockchain is 160 GB as of November 2017), and blocks containing new transactions are constantly being added. With the simultaneous rise of Cloud Computing, more users are foregoing the costs of running a *full node*, a program that downloads all block and verifies them, instead delegating the storage and all relevant computations associated with the blockchains to powerful servers.

However, this strategy of outsourcing computations comes at the cost of full security. There are many reasons for a cloud to answer a request incorrectly. For instance, a cloud might benefit from particular outputs of a computation, and thus would be incentivized to maximize those results [2]. A client with just the result would usually not be able to differentiate between correct and incorrect information. To mitigate this, *thin clients* can employ *authenticated data structures* [3], [4] at the server side in order for the client to verify that a query was correctly returned. Whenever a server fetches data from the blockchain, it will additionally return a compact proof that can be efficiently performed by the less powerful client to check whether the data returned was valid, and furthermore whether the server is honest.

Merkle hash trees [5] are the earliest known examples of authenticated data structures (ADS). By associating each value in an array of data with its hash, the cryptographically hashed data, or *digests*, serve as leaf nodes in a binary tree that is formed by recursively hashing the concatenation of digests of its siblings until there is a single node left. This top node, the *Merkle root*,

is stored at the client and acts as a value to be checked against whenever a proof is returned by a server. When a value is looked up in the tree on the server side, the digests involved in the path taken to reach the value are packed into a compact proof that is sent to the client. The client can then reconstruct the path taken by the server in the Merkle tree, and accept the data returned if the proof ends up with the same value of the Merkle root it has stored. For N values, the server would only need to include $\log N$ digests in its proof, and the client would only need to take $\log N$ steps to verify a response. A visualization of our method is given in 1.1.

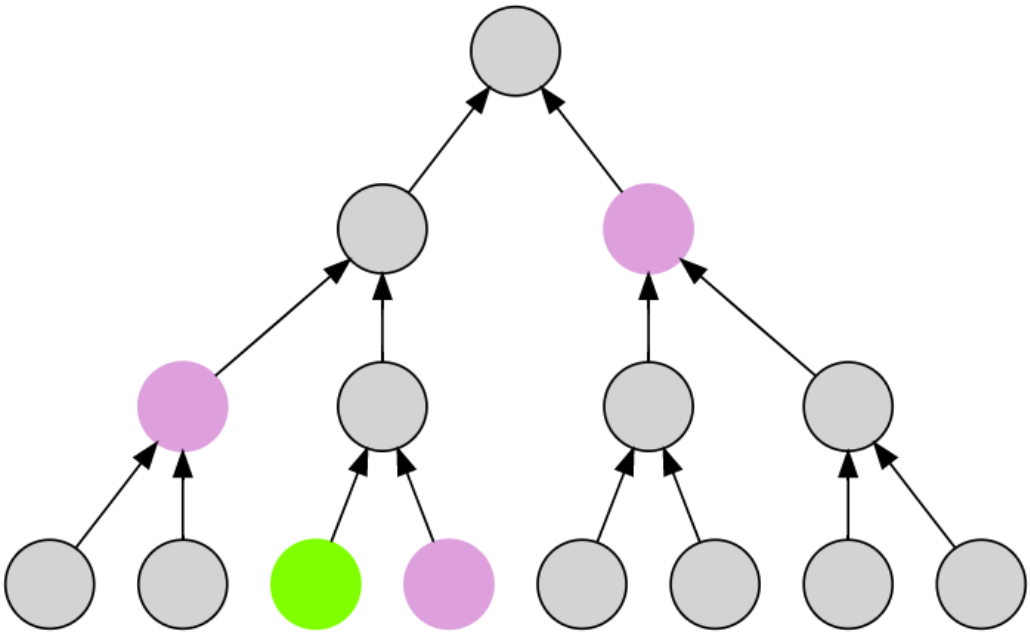


Figure 1.1: To prove the node in green was correctly fetched, a server would provide the hashes of the nodes in purple in ascending order as a proof.

There are currently several thin clients that Bitcoin users can use to retrieve account balances and send transactions without downloading the whole blockchain. One such client is Electrum [6], that maintains its own servers that run Bitcoin nodes. Electrum fetches information from the servers and lets clients get their balances and check transactions by using *Simple Payment Verification*, which involves downloading only the block headers and filtering for relevant transactions. Electrum servers use SSL authentication to protect users from *man-in-the-middle attacks*, but has no robust mechanism to protect users from dishonest servers.

Presently, there is no well-known thin client implementation for any cryptocurrency that uses authenticated data structures to provide security to its users. There have been many previous studies [4], [7] on the feasibility of using ADS in a thin client for Bitcoin, but none of them have been widely implemented yet.

This study presents a method for computing an authenticated data structure over the current Monero [8] blockchain. Monero, a CryptoNote-based digital currency that aims at providing privacy in its transactions by using chaff coins called *mixins*, along with the actual coins that are spent, requires constant sampling of these mixins from the blockchain. A client would need to make multiple calls to a server running a Monero full node when it asks for the chaff coins to use in its transactions. This introduces many opportunities for a server to answer incorrectly to a query, and a single invalid mixin would invalidate the transaction. Due to this, it is imperative for a client to be guaranteed correctness of a successful query as well as the ability to find honest servers while eliminating dishonest ones.

CHAPTER 2

BACKGROUND

2.1 Cryptocurrencies

A cryptocurrency is a decentralized peer-to-peer network that uses a public distributed ledger, called a blockchain, to keep track of user account balances. To spend cryptocurrencies, users broadcast digitally signed messages to the network, which validate the messages and append them to the ledger as transactions. Each cryptocurrency transaction contains inputs and outputs, where the inputs are funds that the sender owns and the outputs are amounts transferred to another user in the network or back to the wallet of the sender. In essence, each outgoing transaction is a reference to an earlier incoming transaction to the senders account. Because the blockchain is an append-only data structure with each new block including the hash of the previous block, transactions cannot be modified once they have been confirmed.

2.1.1 Monero

Because a blockchain is public, users can potentially be at risk of privacy attacks long after they have committed transactions. Transactions can be easily linked by data miners and may even expose identifying information about the users who were involved. Monero, a CryptoNote-based cryptocurrency, aims at obscuring transactions in order to remedy this.

When a user creates a transaction in Monero, he specifies the number of mixins, or chaff coins, to be included in the transactions. These mixins are actually valid outputs that were generated in previous spends by other users in the network, and added to the anonymity set. The wallet protocol will sample outputs that originate from earlier transactions to be included as fake

spends, as well as the user's own coin to be spent. The mixins are sampled based on their age and then sorted in increasing order along with the real coin. This makes it impossible for anyone to tell which coin is the real spend, and renders transactions effectively unlinkable.

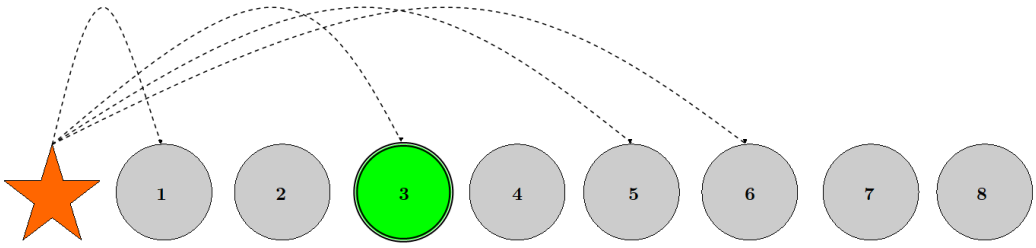


Figure 2.1: A transaction with 3 mixins. The output with index 3 is the real spend, while outputs 1, 5, and 6 are selected as chaff

Transactions also generate new outputs that are added to the blockchain. These outputs are assigned a unique 64-character public key identifier and a global index integer based on their denomination. For instance, if an output of the denomination 10 XMR (10 units of Monero) is the fifth output of that denomination in the blockchain, then it would have a global index of 5.

With the introduction of RingCT in January 10, 2017, denominations no longer appear in the Monero blockchain. The new protocol masks amounts to all those not involved in the transaction; peers would be able to see that a transaction occurred, but would not be able to know the amount that was sent. On popular block explorers, amounts usually show up as 0 XMR. Even though RingCT outputs are required to be mixed with other outputs coming from RingCT transactions, the anonymity set is now much larger because outputs are no longer split up by denomination.

Though it may seem that one would not be able to tell which mixin is the actual one being spent in a transaction, Monero's sampling protocol has a few weaknesses. We have previously done an empirical analysis [9] of traceability in the Monero blockchain based on the different mixin-selection schemes the protocol has employed, and were able to find a substantial amount of transactions in the blockchain to be vulnerable. Our programs were able to expose the real spent coins in transactions due to weaknesses in both sam-

pling strategies and anonymity set. This issue is not the focus of this study, however.

2.2 Delegation of Computation

There have been several previous works done in the subject of verifying computations. Quin, a protocol developed for clients to verify the correctness of a clouds computation, is a *refereed delegation of computation*, or RDoC, system [2]. By splitting computations of servers in conflict into smaller parts, the protocol can find dishonest servers in logarithmically bound rounds. The conflict resolution protocol involves using a binary search to find the initial point of disagreement between servers. VERSUM [10], a system that is based on Quin, achieves low overhead in incremental computation and conflict resolution. It makes use of SEQHASH, an ADS that allows servers to construct digests of computation histories using incremental updates. While our system supports incremental updates as new blocks are created in Monero, we do not need to use an ADS like SEQHASH to keep track of computation histories, as we are only concerned about the correctness of the leaves of the Merkle tree, which are transaction outputs.

2.2.1 TrueBit

A dispute resolution layer similar to that of Quin is implemented in Truebit [11], which is built over the Ethereum [12] blockchain. TrueBit is a system that efficiently processes and verifies transactions in Ethereum by reducing the number of redundant network computations used in traditional smart contracts. In TrueBit, the objective of the verification game is to resolve a dispute between a server and a client. The *judges* in the network are all the Ethereum miners in the network who reach verdicts through *Nakamoto consensus* [1]. The client will challenge the server to provide its computation over a requested time interval, and repeatedly challenge over a smaller subsets with the previous subset until the judges can easily decide on whether the challenge is justified. At the end of the game, either the server is found to be cheating and penalized, or the client is penalized for the false alarm. Our conflict resolution protocol also repeatedly makes calls over a subset of

digests until it is trivial for the client to catch any dishonesty.

CHAPTER 3

SYSTEM DESIGN

3.1 Security Goals

The goal of our design is to provide a way for a thin client connected to a server to verify that queries returned are correct. Without its own copy of the blockchain, a client would need a *proof-of-correctness* to make sure that the data received is indeed the one it requested. Merkle hash trees allow for the verification of large data structures, notably blockchains.

When servers are in disagreement, the client also needs to be able to distinguish the honest server from the dishonest one. Querying each server for every single output in order can take a long time, especially with network delays, as well as space. An efficient conflict resolution protocol is needed for this system.

3.2 Overview of Design

The design of our authenticated data structure over the Monero blockchain involves utilizing nested Merkle hash trees. A tree is built over the outputs, and then used as a leaf in the construction of the tree over the transactions, and so on. The reason for doing this rather than building a single tree over the outputs is mainly for the conflict resolution protocol, which is discussed later. In the protocol, servers are asked at several stages to retrieve the number of leaves in their tree structures, which could correspond to the number of blocks, transactions, or outputs present. Having a nested structure makes it quicker for a server to respond to that query and avoid being timed out.

The calculation is threefold: First, the outputs of a transaction are used to compute a Merkle tree. The root of the resulting tree is used in the computation of another Merkle tree over the transactions that belong to the same block. Lastly, a top Merkle tree is calculated over the current blocks in the Monero blockchain. Because Monero transactions are constantly being made and new blocks are added to the ledger, our implementation supports efficient incorporation of the new information into the top Merkle tree structure without entire recomputation. The authenticated data structures generated are stored alongside the unmodified Monero blockchain on multiple servers. Using standard queries, the client, in reasonable time, can get a result as well as a proof to verify the result.

3.2.1 Retrieving Queries

From storing the top Merkle root of the server, the client now knows of the number of outputs stored at the server, which is trusted to be running a full node of the Monero blockchain and periodically updating. When the client wants to send a transaction, it will query the server for the output at the requested global index, sample the indices of the mixins to be included in the transaction, and request the outputs from the server in the same fashion. The query for the real spend should be randomly made between the queries for the mixins so as to ward off any guessing attacks by listeners on the connection. Now with all the outputs requested, the client can generate the transaction using its wallet and commit the transaction to the Monero network.

On the server side, when the request for an output from a client is received, the server will first determine the validity of the query. If the requested index is negative, greater than its maximum index, or cannot be parsed from the request, then the server will return a failure message to the client. Otherwise, the server will retrieve the output at the requested index. It does so by doing a modified bisection search over the leaves of the top Merkle tree. Because we design nodes to include the greatest global index they contain, our goal is to find the node with the smallest index that is greater than or equal to the requested index. At the end of the bisection search we are guaranteed to

have the block in which the requested output resides.

Upon retrieving the correct leaf of the top Merkle tree, the fetch process continues by running the modified bisection search again on the block Merkle tree. Like the nodes of the top Merkle tree, the nodes in the block Merkle tree also include the maximum global index they contain. Upon completion of the second bisection search, we will have the transaction in which the requested output resides. Performing the modified bisection algorithm one more time on the transaction Merkle tree will yield the requested output, given the server has built the authenticated data structure correctly in the build phase. The server returns the output alongside the proof of correctness in a JSON-encoded response to the client, which can verify the path taken by the server against the top root it has stored.

Having a Merkle hash tree structure over the outputs and a modified bisection search means that the server can retrieve the output in $\mathcal{O}(\log N)$ time. Having multiple levels of Merkle trees in the server entails fast adjust times in the top tree when blocks come in.

3.2.2 Generating Proofs at the Server

Whenever the server returns a requested output, it is also required to return the steps it took in order to reach the output. After traversing down to the found output, we call a function, which is shown in Figure 3.1, to traverse back up to the root of the Merkle tree, collecting the digests of the node it visits.

This process takes $\mathcal{O}(\log N)$ time, as the server does a backwards traversal from the output back up to the root, appending the hash of the sibling node at each level. In our design, each node is designated as either the left sibling or right sibling, which we denote as LEFT or RIGHT, respectively. This is essential to the client in verifying the proof. In our design, the function traverses back up to the root of the transaction Merkle tree, then traverses back up to the root of the block Merkle tree, and lastly it traverses back up to the root of the top Merkle tree. At the end of each phase the proof for each individual tree is stored in a vector, and given to the client as a 3-part

```
GetProof(index):
    node ← leaves[index]
    proof ← []
    proof.append((node, "SELF"))
    while node.parent ≠ NULL
        sibling ← node.sibling
        proof.append((sibling, sibling.side))
        node ← node.parent
    proof.append((node, "ROOT"))
    return proof
```

Figure 3.1: The algorithm traverses from the output back up to the root, adding the hash of the sibling of the current node to the proof.

proof.

3.2.3 Verifying Proofs

When a client receives a valid response to its query, it must first make sure the server correctly fetched the output from its running node. The client does this by checking the proof returned along with the response against the Merkle root it has stored. The algorithm is shown in Figure 3.2.

```
CheckProof(proof, root):
    link ← proof[0]
    for i ← 1 to len(proof)
        if proof[i].side = "LEFT"
            link ← hash(proof[i] + link)
        else if proof[i].side = "RIGHT"
            link ← hash(link + proof[i])
        else
            return "Proof is invalid."
    if link ≠ root
        return "Proof is invalid."
    else
        return "Proof is valid."
```

Figure 3.2: The client completes this short proof by hashing the proof elements together, and then checking against the stored root.

The client-side verifier goes through the three-step proof in $\mathcal{O}(\log N)$ time, invoking the verifier for checking a single Merkle tree each time. The single Merkle tree verifier takes the initial value of the proof, and hashes the rest of the values in the proof according to the side. Specifically, if the next value

in the proof corresponds to the left sibling of the node in the Merkle tree, it will concatenate the next value to the left of the current value, and hash the concatenation. Otherwise, the verifier will concatenate the next value to the right of the current value, and hash the concatenation. The single tree verifier ends when the list is exhausted. The three individual proofs are linked such that the result of the proof over the transaction Merkle tree is the starting element of the proof over the block tree, and that the result of the proof over the block tree is the starting element of the proof over the top Merkle tree. This makes it difficult for a dishonest server to compromise the overall proof as it would involve recomputing hashes over the whole blockchain. During the verification process, if any of the steps fail, the client will mark the proof as invalid, and consequently the server would be considered dishonest. At the end of the proof, the client will perform a final check by determining if the proof is equal to the top Merkle root it has stored. If the check passes, then the client knows that the requested output was fetched correctly.

3.2.4 Updating the Merkle Tree

The current Monero block rate is about 2 minutes, which means that a new block of transactions is appended to the ledger in that time. When a new block is added, the server must be able to quickly parse information from it and extend the ADS over the block. New transaction Merkle trees and the block tree can quickly be built; however, the top Merkle tree would need to extended. It will take too long to rebuild the top Merkle tree from scratch, as the downtime will lead to clients being unable to send transactions. Instead, we must have a method of adding to the top Merkle tree while adjusting only the nodes affected.

The design of our Merkle tree[1] uses a method for efficiently adding to an existing tree structure. The algorithm, which is visualized in Figure 3.3 and shown in Figure 3.4, is as follows:

1. In a given Merkle tree, return a list of nodes in the tree that have complete subtrees beneath them, that is, the internal nodes that are

[1]Based on Jamie Steiner's Merkle tree implementation (MIT license): `https://github.com/jvsteiner/merkletree`

roots of complete subtrees. This nodes in the list are ordered from left to right.

2. In a reverse iteration of the list, compute a new parent node from the list element and the node to be added. The node added will receive a reference of RIGHT from the new parent node, and the list element will receive a reference of LEFT. The new parent node is now designated as the new node to be added, and we repeat this as we iterate through the list.

3. At the end of the iteration, the new parent node is effectively the new Merkle root of the data structure. We return the node as such.

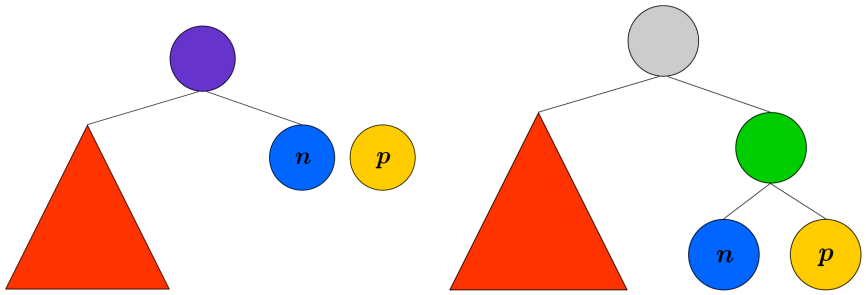


Figure 3.3: In (a), the red section is a complete subtree and node n also has a complete subtree underneath (empty tree). We wish to add node p. In (b), p has been added, and the green and top nodes recomputed according to our design.

```
AddAdjust(complete-subtrees, new-node):
    right-child ← new-node
    for left-child in reversed(complete-subtrees)
        parent ← NewTree(left-child, right-child)
        right-child ← parent
    return parent
```

Figure 3.4: The function NEWTREE is the same as building a Merkle hash tree over the two nodes passed in; it will return the Merkle root

When a the server receives the new block, it will first compute the transaction Merkle trees in the block, and then build the block Merkle tree. Next, the server will add the new block as an additional leaf in the top Merkle tree by using the `add_adjust` function. This is much faster than rebuilding the whole tree from scratch.

3.2.5 Conflict Resolution

The thin client would be communicating with multiple servers running a full Monero node with our design on top of the blockchain. The client would store the top Merkle roots of the servers in a list or similar data structure, and update the roots when the server adds new blocks to the network. If there is at least one honest server in the group of servers the thin client is communicating with, the client can eventually distinguish that server from the dishonest ones using the resolution protocol.

The conflict resolution protocol consists of the following three parties:

- the two servers that are in disagreement on their computation over the Monero blockchain, and
- the client who discovers that the two servers are in disagreement based on the Merkle roots it has stored.

The two servers do not have to be aware of each other in the network for the conflict resolution protocol to be successful, nor do we need to assume that they are not colluding. However, the client needs to be communicating with at least one honest server in order to realize there is conflict. It keeps track of the computation histories of each server by storing its top Merkle root, which is periodically updated as new blocks come in. Servers that are in agreement with each other would produce identical top Merkle roots, whereas those in disagreement would produce vastly different roots due to uniformity.

When the client initiates the conflict resolution protocol, the servers need not be notified that the protocol has started. The client can carry out the protocol by making requests to each server. The algorithm is as follows:

1. The client queries each server for the number of leaves present at its top Merkle structure. If the numbers match, the protocol proceeds. Otherwise, the client can query several block explorers, which are running Monero nodes, to check the current height. The server that is not at the current height of the blockchain is determined to be lagging behind the network and automatically marked as dishonest.

2. The client sends a challenge to the servers. Specifically, each server is asked to fetch the left and right child hashes of its top Merkle root. If any of the children nodes are leaf nodes, then we also return the pre-hashed data. Any server that fails to respond in bounded time is marked as dishonest.

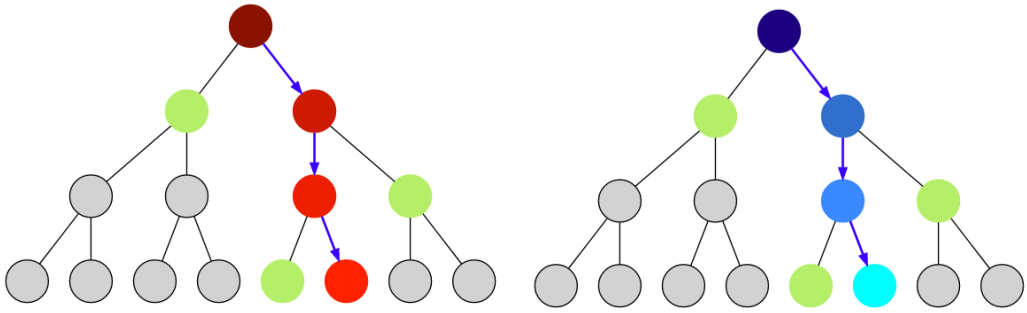
3. Upon receiving the requested left and right children, the client checks the left child of the first server against that of the second. If they match, then the client will add an additional command RIGHT to the `/getchildren` requests. Otherwise, it will add an additional command LEFT to the requests.

4. At the server side, additional commands will result in the server first traversing downward from the Merkle root in the corresponding direction, and then fetching the children at the resulting internal node. If the commands make the traversal go out of bounds, then the response will include a failure message.

5. Steps 2-4 are repeated until the client receives the leaf node at which the servers initially disagree. This leaf node corresponds to the block in which in conflict arises, and the protocol starts from step 1 again to find the transaction in which the conflict arises.

6. Upon reaching the transaction in which the two servers disagree, the client checks for the number of leaves, which correspond to outputs, in the transaction. If the number of leaves is the same, then the client will ask each server to respond with all of the outputs in that transaction. The client can verify the outputs of a transaction by querying several block explorers for the outputs indexed at a given transaction hash. Because the number of outputs returned is small compared to the set of all outputs, the client can quickly check the result against the servers' responses. The server that contains the modified output is marked as dishonest.

7. If the number of leaves is not the same, then all the client has to do is call upon the block explorers to return the number of outputs indexed at that given transaction. The server that produced the incorrect number of leaves is marked as dishonest.



(a) Server 1's tree (b) Server 2's tree

Figure 3.5: The conflict resolution protocol starts from the root of each server's top Merkle tree, and reaches the origin of conflict when the client finds the leaf in which both servers disagree.

```
RESOLVECONFLICT(server1, server2):
  if NUMLEAVES(server1) = NUMLEAVES(server2)
      path ← []
      while(TRUE)
          left1, right1 ← GETCHILDREN(server1, path)
          left2, right2 ← GETCHILDREN(server2, path)
          if left1 = left2
              path.append("RIGHT")
          else
              path.append("LEFT")
          if left1.isleaf or right1.isleaf
              break
      if left1 = left2
          return right1, right2
      else
          return left1, left2
```

Figure 3.6: The client runs the conflict resolution protocol by making calls to the servers in conflict.

In this design, which is visualized in Figure 3.5 and shown in Figure 3.6, we want to find the earliest point in the Merkle tree that caused the two servers to disagree. Because the servers have the same of number of leaves in their top Merkle tree, they must have computed over the same number of Monero blocks. This also implies that their trees were built in the same manner and so we can traverse them with the same path.

3.3 Data Structure Details

Our design for the authenticated data structure over the Monero blockchain is as follows:

1. For a transaction with N outputs, we will build a Merkle tree over the outputs $o_1, ..., o_N$. The Merkle root returned will be a digest of all the outputs, as well as the maximum index of the outputs. Let us denote the Merkle root returned as $(tr, \max(idx(o_1, ..., o_N)))$, where tr is the digest of all the outputs in the transaction. Since outputs are ordered, we can be sure that output o_N will have the maximum global index.

2. From the first step, each block will have a set of Merkle roots corresponding to the set of transactions in that block. Let us denote the ordered list of Merkle roots as $[(tr_1, idx(tr_1)),...,(tr_N, idx(tr_N))]$. We will build a Merkle tree over this list in the same way as the first step. The Merkle root returned will be a digest of all the transactions in the block, as well as the maximum index of the outputs. We denote this as $(br, \max(idx(tr_1, ..., tr_N)))$. Since transactions are ordered, we can be sure that tr_N will have the maximum global index.

3. From the second step, there will now be a set of Merkle roots corresponding to the blocks in the network. More specifically, each block in the Monero blockchain will have a Merkle root associated with it. We will build a Merkle tree over the roots of the blocks, and end up with a top Merkle root. We denote this root as $(top, \max(idx(b_1, ..., b_N))$, where b_i is the i^{th} block in the blockchain. Because the blocks are in temporal order, we can be sure that b_N will have the maximum global index over all the outputs in the blockchain. The building phase concludes after this step.

A detailed diagram of the three-layer Merkle tree can found found in Figure 3.7. In this design, the top root contains the digest of all the outputs in the Monero blockchain and also the maximum global index of the outputs. The top root can be stored at a client, which now does not need to download the blockchain. It is evident that the top root would need to be updated and pushed out to the client as new blocks come in. The design of our Merkle tree supports adding to the tree without rebuilding the whole top structure.

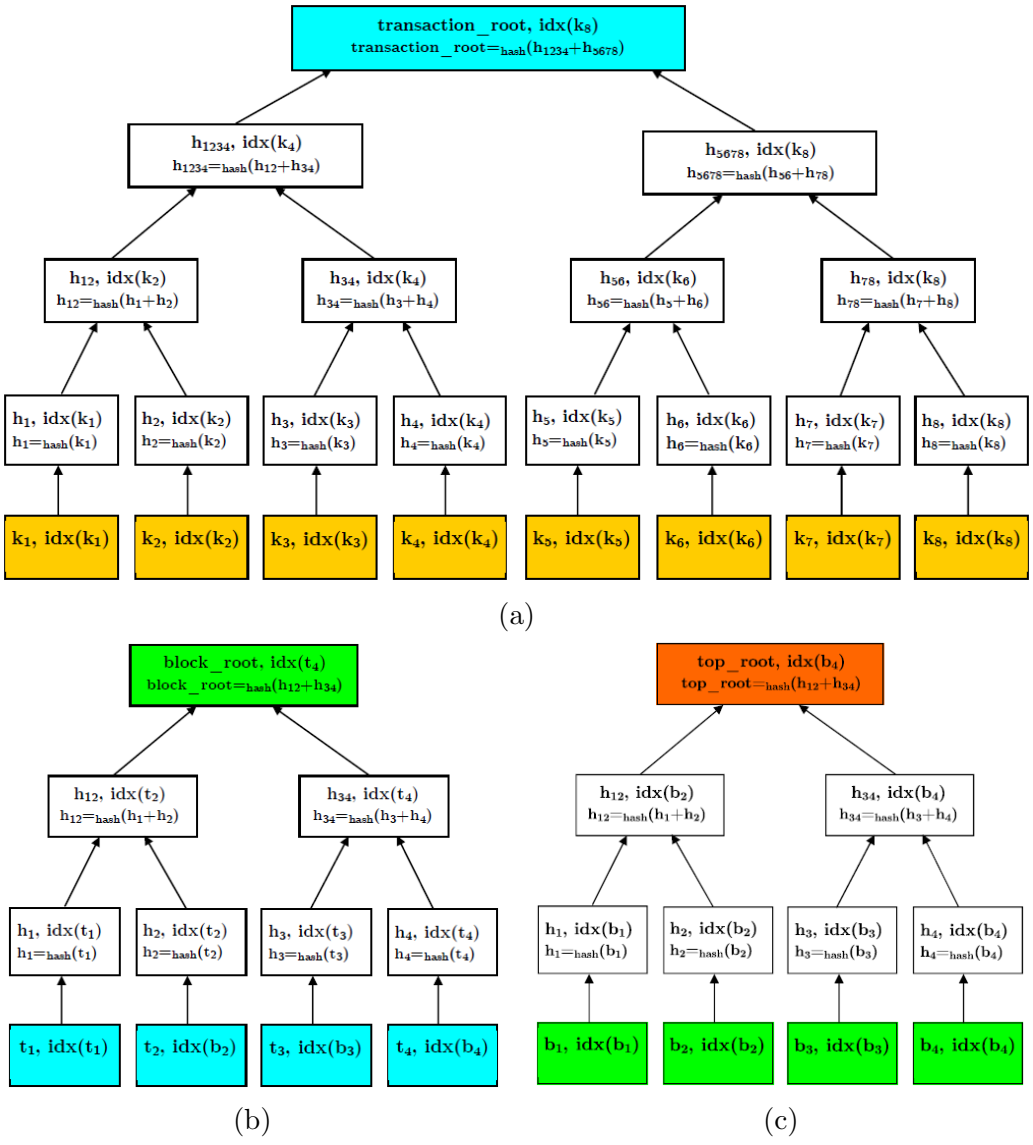


Figure 3.7: The tree over the outputs (a) is constructed first. The root of (a) is used as the leaf in (b), whose root is subsequently used as the leaf for (c).

CHAPTER 4

SECURITY ANALYSIS

4.1 Security Properties

Because hash functions are thought to be one-way functions, it is computationally impossible for a server to deceive a client by providing an incorrect output while giving its proof that leads up to the correct Merkle root. The request would time out, and the thin client would mark the server as dishonest.

By using a Merkle tree over the blockchain, the client can also narrow down the source of conflict between servers using a binary search on the hashes of the internal nodes. This greatly reduces the time a client would have to take to find errors.

4.2 Incentives

Unlike Ethereum, which supports *smart contracts* [12], Monero currently cannot provide incentives that preserve its privacy goals. If we employed a "verification game" dispute resolution layer identical to that in TrueBit, the role of judges could be played by the miners in the network. In order to get judges to referee the verification components of the conflict resolution protocol correctly, rewards can be sent to them if the protocol successfully terminates. At the start of a resolution, the client and two servers will be mandated to deposit funds into escrow. At the end of the process, the funds are returned to the honest server in full, whereas the dishonest server is charged a fraction as penalty and the client is charged a service fee. If both servers are found to be dishonest, then both are penalized and the service fee on the client is discounted. A bad client call on two servers that are not

in conflict will result in a higher service fee on the client and no deduction on the servers. The refusal of the client or any server to participate in the protocol will result in termination of the protocol and the designation as dishonest in the case of the server. Having a reward gives an incentive to the judges to lend their computing power and cooperation to the client. We assume that the judges in the protocol referee each step fairly.

However, having a client pay miners in the network to referee would violate the untraceability aspect of Monero's privacy, as the identity of the client would be exposed in the transaction. Also, because there are no smart contracts in Monero, it is currently impossible for financial penalties or rewards to be handed out.

4.3 Errors

As seen in the explanation of our protocol, there can be two causes for an erroneous transaction Merkle root. First, the server could have included extra outputs from other transactions or excluded transaction outputs during the building phase. Having the incorrect number of outputs guarantees a different Merkle root from one built over the correct number of outputs. Second, the data of the outputs could have been modified by the server. Even if the server correctly builds the Merkle tree over the correct number of outputs, the Merkle root would be different. A combination of these two errors can also occur and result in an erroneous root.

A server can have reasons other than unintentional mistakes to deceive the client. By receiving false information about the Monero blockchain, the client can potentially create invalid transactions. Although those broadcasted transactions would never receive any confirmation, the client would be rendered unable to send funds in Monero. This would usually be done by a disgruntled server.

4.3.1 Handling Uncooperative Participants

In order to resolve conflicts in a timely manner, the conflict resolution protocol must be able to determine when a participant has become uncooperative and ignore it. When the client asks the two conflicting servers to provide their left and right children hashes of a requested Merkle root, the responses must not only be valid, but also returned in a bound time. Whenever a server fails to provide a valid response to a query for the children of a root in that frame, the client will denote the server as being uncooperative and consequently dishonest. Further penalties can be imposed on the dishonest server if needed, including expulsion from the Monero peer-to-peer network.

In another scenario, the client can be the malicious participant in the protocol. By repeatedly sending requests, the client can overload the server and prevent other clients from sending transactions. Because finding the conflicting leaf in a balanced binary tree should take no more than $\log N$ steps, the number of requests for a tree's children at a server can be limited within a timeframe. If a client exceeds that limit for `/getchildren` requests, the server can prevent the client from making further requests until a period of time elapses. Because the conflict resolution process utilizes the computing power of the servers, the client should also be punished if it calls the protocol on two servers that do not have conflicting Merkle roots. A check is already implemented on the client side so as to prevent accidental calls from the script, however, should the client circumvent this check, the servers can terminate the protocol and impose penalties on the client, including denial of service. This would force the client to utilize another server to fulfill its requests. If all of the servers available to the client deny service, then the malicious client would be starved.

CHAPTER 5

IMPLEMENTATION AND BENCHMARKS

Our implementation of this design is done in Python. All blockchain data was scraped using a C++ scraper we developed over the Monero blockchain, collecting the following information for each RingCT output:

(block hash, transaction hash, output public key, global index)

All the raw data from the blockchain is written to a database, which we then read out of using the `sqlite3` module for Python. The database contains 2985559 RingCT outputs, which range from the launch of RingCT to November 5, 2017. We make an arbitrary split on October 23, 2017, and use the data from January 10, 2017, to the split date to build our initial authenticate data structure. The data from October 23, 2017, to November 5, 2017, is used to test efficient adding to the already-present ADS.

All of the Merkle trees created in our design are too large to fit into main memory, and so we utilize a disk-backed persistent dictionary-like object to store the data structures. The reason for using this over "dbm" databases is that the values shelved can be arbitrary Python classes, which is not allowed in conventional databases.

5.1 Benchmarks

Our design is deployed on three separate LinuxONE medium virtual servers running a SLES12 SP2 image. There are two virtual CPUs and 4096MB RAM on each server. Two servers are deployed as full node servers, which contain the Monero blockchain, and the other is deployed as a client.

In order to profile the performance of our design, a script is set up at each server to automatically carry out and measure the actions pertinent to each test. For the build test, each full node server built the three-layer Merkle tree from the scraped blockchain data, tore down the design, and repeated the build phase. In the query test, the client generated a random global index and queried each server for the output located at that index. Evaluation of the query test also includes the time to generate the proof at the server side, as well as network delays. The proof verification test was simultaneously carried out as the client checked the correctness of the query. In the add and adjust test, each server added a new block to the ADS and adjusted its top Merkle tree. The updated top root of each server was also checked for correctness. In the conflict resolution test, one server modified a random transaction before building over the blockchain. The client would then initiate the conflict resolution protocol and find the tampered transaction. The results of the tests are shown in Table 5.1 and Figure 5.1.

Table 5.1: Combined performance results

	Average Performance	Number of Trials
Build	4886.182153 seconds	100
Query response	0.714903 seconds	1000
Proof verification	0.000132 seconds	1000
Update top tree	13.320195 seconds	100
Conflict resolution	236.235168 seconds	100

5.1.1 Disk Latency

A detailed profile on the conflict resolution protocol reveals that a majority of the time was spent by the server retrieving the requested Merkle tree from its disk-backed dictionary-like object and traversing it based on the client's request. A way to mitigate this would be to use a server with more available RAM so that all of the trees would fit in main memory, removing the need for disk seeks. Another method for greatly improving the speed of the protocol would be to memoize the `/getchildren` calls. Because the client sends traversal commands that are extended from the call before, the server should not have to re-traverse the path from the root, instead picking

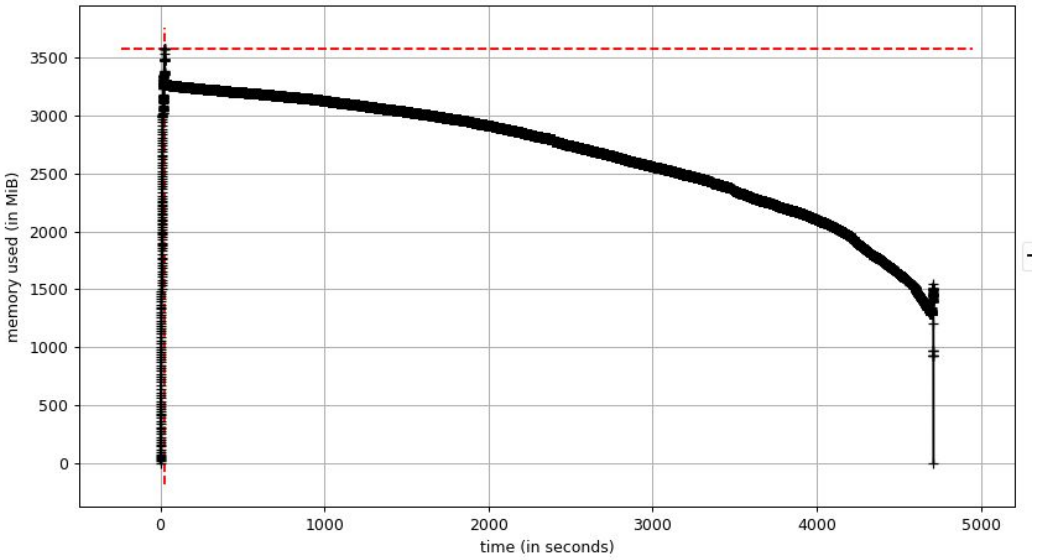


Figure 5.1: Memory usage at the server during the build phase.

up from where it last ended.

5.1.2 Storage

Scraping all of the necessary data for our design from the Monero blockchain into a database takes 1.57 GB of disk space. The storage of all the Merkle trees in our design takes 2.17 GB. For comparison, the current size of the Monero blockchain is 33.52 GB (as of November 29, 2017).

CHAPTER 6

CONCLUSION AND FUTURE WORK

This study serves as a proof-of-concept for a thin client to be built over the Monero blockchain. By building a three-layer Merkle tree over all the blocks, a server can efficiently fulfill queries on the blockchain from a client while proving it has provided the correct response. A thin client can securely get outputs to be used and generate transactions without having to store and run a full copy of the ledger. In the event of a conflict, the client can quickly perform a resolution by sending challenges to each server and narrowing down to the source of conflict. The server also has the capability to ensure that the conflict resolution does not drain its resources by blocking out extraneous calls.

Further study will require extensive testing and more data from the Monero blockchain to be built over. The Merkle tree used now should also be updated to orphan blocks An implementation will only be useful if the changes are made to the Monero protocol, including adding a Merkle root to each block and adding privacy preserving incentives. As more people turn to cryptocurrencies, we expect the innovations stemming from building authenticated data structures over blockchains to grow.

The repository used in this study is publicly available at: `https://github.com/kvnl33/merkletree`

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf."

[2] R. Canetti, B. Riva, and G. N. Rothblum, "Practical delegation of computation using multiple servers," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046759 pp. 445–454.

[3] R. Tamassia, *Authenticated Data Structures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–5. [Online]. Available: https://doi.org/10.1007/978-3-540-39658-1_2

[4] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically," *SIGPLAN Not.*, vol. 49, no. 1, pp. 411–423, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535851

[5] R. C. Merkle, "Secure communications over insecure channels," *Commun. ACM*, vol. 21, no. 4, pp. 294–299, Apr. 1978. [Online]. Available: http://doi.acm.org/10.1145/359460.359473

[6] "Electrum bitcoin wallet," 2014. [Online]. Available: https://electrum.org

[7] L. Reyzin, D. Meshkov, A. Chepurnoy, and S. Ivanov, "Improving authenticated dynamic dictionaries, with applications to cryptocurrencies," Cryptology ePrint Archive, Report 2016/994, 2016, https://eprint.iacr.org/2016/994.

[8] N. van Saberhagen, "CryptoNote v 2.0," Oct. 2013. [Online]. Available: https://cryptonote.org/whitepaper.pdf

[9] A. Miller, M. Möser, K. Lee, and A. Narayanan, "An Empirical Analysis of Linkability in the Monero Blockchain," 2017. [Online]. Available: http://monerolink.com/monerolink.pdf

[10] J. van den Hooff, M. Frans Kaashoek, and N. Zeldovich, "Versum: Verifiable computations over large public logs," 11 2014.

[11] J. Teutsch and C. Reitwiener, "A scalable verification solution for blockchains," Nov. 2017. [Online]. Available: http://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf

[12] "Ethereum," 2015. [Online]. Available: https://www.ethereum.org/