2018 IEEE European Symposium on Security and Privacy Workshops

Authenticated Data Structures for Privacy-Preserving Monero Light Clients

Kevin Lee
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
klee160@illinois.edu

Andrew Miller[†]
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
soc1024@illinois.edu

***Abstract*—Monero, a leading privacy-oriented cryptocurrency, supports a client/server operating mode that allows lightweight clients to avoid storing the entire blockchain, instead relying on a remote node to provide necessary information about the blockchain. However, a weakness of Monero's current blockchain data structure is that lightweight clients cannot authenticate the responses returned from a remote node. In this paper, we show that malicious responses from a remote node can lead to reduced privacy for the client. We discuss several lightweight mitigations that reduce the attack's effectiveness. To fully eliminate this class of attack, we also show how to augment Monero's blockchain data structure with an additional index that clients can use to authenticate responses from remote nodes. Our proposed solution could be implemented as a hard fork, or alternatively through a "Refereed Delegation" approach without needing any fork. We developed a prototype implementation to demonstrate the feasibility of our proposal.**

***Index Terms*—Cryptocurrencies, Privacy, Authenticated Data Structures**

I. Introduction

Monero is a popular privacy-preserving cryptocurrency. At the time of writing, it has a market cap of over $5 billion USD, making it a top-15 cryptocurrency[1]. Like other blockchain-based cryptocurrencies, such as Bitcoin and Ethereum, the Monero blockchain is large and growing at a steady pace (nearly 40 gigabytes at the time of writing). However, also like other cryptocurrencies, Monero client software can be run in a lightweight configuration, which makes use of a "remote node" to provide access to public blockchain data. The lightweight client only stores a minimal amount of data, such as the user's private keys.

Although most documentation recommends that users store the entire blockchain locally if possible, we anticipate increased usage of lightweight clients in the future. The `getmonero.org` user guides include instructions for connecting to remote nodes, and the `moneroworld.com` website provides listings of publicly reachable remote nodes, to which new volunteers can add themselves.

The Monero client is designed with countermeasures to enhance privacy even if the remote node is corrupted. In particular, when fetching blockchain information from the remote node, the client asks for redundant information to avoid revealing which data it already knows about (and hence which transactions belong to it). To our knowledge, all publicly available discussions suggest that clients are guaranteed "untraceability" even in the case of a malicious node:

> Use of a remote node doesn't reveal much about you to the node operator; your secret keys, *which output key is yours,* how much fund you have, etc ... all this kind of information stays private to yourself.
> (`stoffu`[2], MRL Researcher)

In this paper we demonstrate new attacks on the remote node setting of Monero, which allow a corrupt remote node (or an on-path network adversary, such as an ISP) to trace a client's transactions. These attacks involve the remote node inserting corrupted responses about blockchain data, which the client cannot authenticate. The remote node can deduce from the client's behavior information about which output key in the transaction is the real one, thereby violating the untraceability guarantee. We discuss several lightweight countermeasures that make such attacks less effective, which we have suggested to Monero core developers as part of a responsible disclosure.

To systematically resolve the vulnerability underlying our attacks, we also discuss how to extend the Monero blockchain data structure to enable lightweight clients to authenticate remote node responses. The blockchain data structure is already a hash-based authenticated data structure [1]–[3], and so in principle the client could traverse it to authenticate responses. However, this would be inefficient, requiring a linear scan. In this paper we show how to add an additional layer of Merkle trees to the Monero blockchain data structure, which clients can use as a secure index to authenticate responses.

Our proposed data structure could be deployed in Monero as a hardfork upgrade, which is plausible since Monero already conducts regularly-scheduled hardforks to adopt improvements. However, we also present an alternative deployment approach based on "Refereed Delegation" [4] that avoids the need to fork at all. In this model, which is shown in Fig. 1, the client connects to multiple peers, requests data from each of them, and can efficiently identify the correct response if *any* of the peers are functioning correctly. We show

[†]Andrew Miller is also a board member of Zcash Foundation
[1]according to `coinmarketcap.com`

[2]https://www.reddit.com/r/Monero/comments/7no0nh/running_your_own_node_vs_using_a_remote_node/

DOI 10.1109/EuroSPW.2018.00010

through an implementation and experiment with the actual Monero blockchain that the overhead would be reasonable in computing power and storage to generally all nodes running the current protocol.

II. Background and Related Work

A. Monero and 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 sender's 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.

Financial transactions are often sensitive in nature, and so privacy is an important concern for cryptocurrencies. Since blockchain transaction logs are inherently public and broadly distributed, these data may be analyzed long after the transactions occur. Several prior works have demonstrated the feasibility of transaction graph analysis [5]–[7] in Bitcoin and other cryptocurrencies.

Monero is a privacy-oriented cryptocurrency that employs additional cryptographic techniques to obscure the transaction graph. Monero is designed to provide three main privacy guarantees:

1) *Untraceability:* a transaction spending a coin cannot be linked to the transaction where that coin was received.
2) *Unlinkability:* this is achieved through the use of one-time "stealth addresses", whereby the client's long term address cannot be used to infer spends. A "view key" associated with an address must be used to detect incoming transactions. This can be outsourced to a hosted Monero account (as is this case with hosted wallets), but reduces privacy.
3) *Confidential Values:* the amount of money transferred is not visible (This is associated with RingCT).

Our security focus in this paper is untraceability; confidential values and unlinkability are not affected by our attacks or countermeasures.

To be untraceable, a Monero transaction includes a number of "mixins", or chaff coins, along with the real coin being spent. These mixins are randomly chosen by the client among all outkeys that were generated in previous transactions by anyone in the network. A visualization of the protocol is shown in Fig. 2. The transaction includes a ring signature to prove the user is authorized to spend one of the coins, without revealing which one.

Each transaction also generates new outputs that are added to the blockchain, and made available for future transactions to use as mixins. Each output comprises an outkey structure, which contains a unique 32-byte public key identifier and a global index integer, and are included in the mixin pool for future transactions.[3]

Previous work has analyzed privacy hazards in Monero resulting from poor sampling of mixins, though recent versions of the software employ more robust sampling strategies to mitigate this [8], [9]. Our focus in this paper is on the interaction that occurs *after* the client has sampled the mixin indices, namely that the client must then query the blockchain to retrieve the public keys corresponding to the mixins. We show that if the client receives erroneous data here, it can render the transaction traceable.

B. Light clients

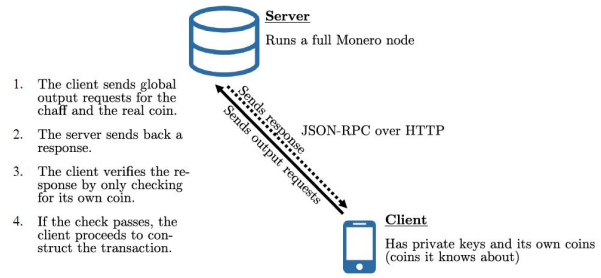
Lightweight clients for cryptocurrencies are based on the Simplified Payment Verification (SPV) mode [10], in which clients to verify transactions by making use of mostly-untrusted network peers. A light client connects to several such peers, at least one of which must be assumed to function correctly. The client requests the proofs-of-work in the current blockchain, but otherwise skips all transaction data. By identifying the blockchain with the most proofs-of-work, the client obtains a root hash of a block that is finalized with high probability. The client then requests any relevant transactions by providing a bloom filter containing the client's addresses. Finally, the client uses an authenticated data structure proof to validate that returned transactions are actually included.

Remote Nodes in Monero. Unlike in Bitcoin, whose wallet manages both transactions and the blockchain, Monero makes use of a daemon to handle the Monero network. The daemon downloads its own copy of the blockchain and synchronizes with the network to send and scan for incoming transactions. The wallet, communicating with the daemon, detects the transactions that have been sent to it. Despite the security risks of doing so, several users have elected to forgo running their own daemons, instead connecting their wallets to open daemon services like MoneroWorld. These remote node services are networks of users who have offered their nodes up in order for others to use Monero. While convenient, using a remote node is insecure and does come with risks. For instance, a remote node is able to get the IP address of a connecting wallet and potentially identify open ports at that address.

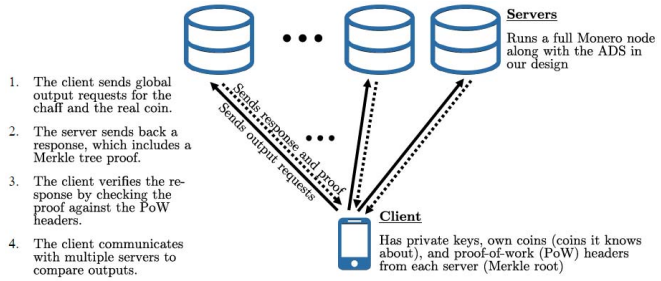
C. Merkle Trees and Authenticated Data Structures

Hash-based authenticated data structures enable a client with limited storage capacity to verify queries made against a remote data structure. The key idea is that the client stores a short hash digest representing the whole data structure, and the

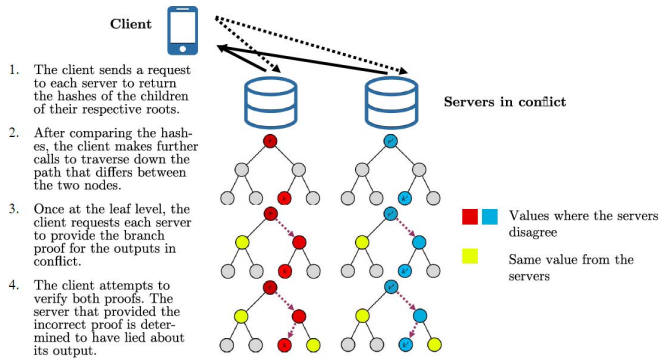
[3]With the introduction of RingCT in January 10, 2017, denominations no longer appear in the Monero blockchain. The new protocol masks amounts to all those not involved in the transaction; peers would be able to see that a transaction occurred, but would not be able to know the amount that was sent. 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.



(a) An overview of current remote node operations at a high level.



(b) An overview of our proposed system at a high level. Communication between the client and server can use SSL.



(c) A visualization of the conflict resolution in our proposed system. The protocol uses the Refereed Delegation of Computation model [4] to find the earliest conflicting outputs, and Merkle proofs to determine the correct output. Monero transactions already generate these Merkle trees in every new transaction.

Fig. 1. High level overviews of the operations analyzed in this paper. Current Monero transaction procedures are shown in (a), whereas (b) outlines transaction procedures in our proposed system. Our proposed system also handles conflict resolution between disagreeing remote nodes; the protocol is shown in (c).

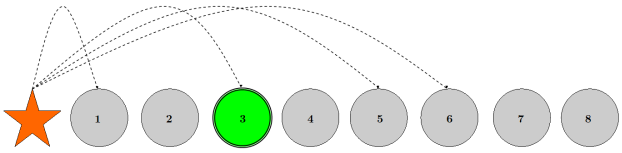


Fig. 2. A transaction with 3 mixins. The outkey with index 3 is the real spend, while outputs 1, 5, and 6 are selected as chaff.

remote server provide authenticating information that the client can use to check consistency. The most well-known example of an authenticated data structure is the Merkle tree [11], which is a balanced binary tree representing an array. The leaves of the tree are the hashes of elements of the array, and the overall digest (the *Merkle root*) is computed by recursively hashing the concatenation of the digests of its children. When the client queries an element of the array, the remote server returns the response along with all the hashes of sibling along the path up to the tree. When verifying the query the client reconstructs the path taken by the server in the Merkle tree, and accepts the data returned if the proof ends up with the same value for the Merkle root. 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.

D. Refereed Delegation of Computation

The refereed delegation model was introduced by Canetti et al. [4]. Their protocol, Quin, allows clients to verify computations by querying multiple servers and applying a dispute resolution procedure to identify incorrect responses. The approach involves splitting the 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 [12] is a system that extends Quin with a new data structure, SEQHASH, which enables servers to efficiently update refereed delegation for arbitrary incremental computations. While our setting also involves incremental computations (as each new committed changes adds a small number of additional transaction outputs that may be queried), the generality of SEQHASH is not necessary, and ordinary Merkle trees are sufficient.

1) TrueBit: A dispute resolution layer similar to Quin is implemented in TrueBit [13], which is built over the Ethereum [14] 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* [10]. The client will challenge the server to provide its computation over a requested time interval, and repeatedly challenge over smaller subsets of the previous computation 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. This approach would be interesting to consider for future work, although Monero does not support the necessary smart contract programming features.

III. Tracing attacks from remote nodes

In this section we show how corrupted data sent from a Monero remote node can be used to expose linkages between transactions made by a lightweight client.

A. Remote node operation in Monero

Before explaining the attacks, we first describe in more detail how Monero light weight clients operate today. When the Monero client initiates a transaction, it first queries the remote node for information about the blockchain. Among the information received is the maximum global index, which the wallet will use to determine which mixins to include in the transaction. The wallet will go through the mixin-selection protocol, which samples candidate outputs (by global index) to be used as mixins (e.g., 58 outputs are requested for a default 4-mixin transaction). A subset of these outputs are selected over a recent period and the rest are selected over a triangular distribution based on their age (the older an output is, the smaller the likelihood of it getting selected). To avoid revealing which is the real input, the client also includes the global index of the actual outkey it is spending along with the request, even though this data is already stored by the wallet and therefore redundant to ask for. The request for outputs is sent to the remote node via the `get_outs.bin` API endpoint, which will return each requested outkey. When receiving a response from `get_outs.bin`, the Monero client currently performs a partial validation; if the client's actual outkey (which is already stored in the wallet) is not among the returned responses, then the transaction is aborted and an error is displayed to the user. After the wallet receives all of the outkeys, it selects uniformly from among these the final number of mixins, and includes these alongside the real outkey to form the transaction. After signing the transaction, the client marks those outputs as spent and transmits the transaction to the remote node.

B. The Retry-and-Intersect Attack

We first make an observation about the client behavior described above: if the remote node returns an invalid response to `get_outs.bin` and triggers an exception at the client, then the client aborts the transaction and displays a message to the user. If the user tries to initiate the same transaction again, the client software begins the process anew, including sampling all new outputs to use as mixins. The end result is that two queries are sent to the remote node, which with high likelihood intersect in a single index which is the correct output.

As a proof of concept, we carried out the above scenario 10 times on a private network with an unmodified instance of `monero-cli`, connected to an instance of `monerod` which we modified to generate invalid responses to `get_outs.bin`. We found that a unique intersection was available in each of our trials. In a larger sample size, we would expect that in some cases the two requests to `get_outs.bin` would intersect in more than one index, but this appears to happen with low frequency.

We note that this is an active attack, and involves showing an error message to the client. This would therefore likely raise suspicion if it occurred many times in a row. However, since the transaction goes through without error on the second request, used sparingly the attack may not raise much suspicion.

a) Mitigation 1 - Caching mixin choices: The main cause of this variation of the attack is that the client rerandomizes its choice of mixins after a `get_outs.bin` request if the transaction is aborted. Hence an effective mitigation would be to persist the choice of mixins to disk, such that if the transaction must be retried then an identical request to `get_outs.bin` would be made, such that no more information is revealed in the second request than in the first.

b) Mitigation 2 - Using TLS for remote node communications: This attack is exacerbated by the fact that peer-to-peer communications between a Monero client and remote node are not encrypted currently (i.e., they use JSON-RPC over plain HTTP). Hence, this active tracing could be performed by an on-path network adversary (e.g., a malicious router or ISP). Hence a mitigation would be to upgrade to an authenticated channel such as SSL/TLS, meaning the attack would require a compromise of the remote node itself.

C. The Guess-and-Check Attack

Even if mitigations 1 and 2 above are adopted, a remote node can still mount an attack that reveals partial information about the client's actual outkey, and with some probability tells the attacker exactly which one is real. In this variation, instead of sending a completely invalid response, the attacker corrupts just one of the requested outkeys. There are then two different client behaviors:

- *Case 1: the corrupt outkey is not the client's real one.* Since the client cannot validate the response, the client constructs a signed transaction and transmits it to the remote node. The transaction is invalid and does not show up in the blockchain. However, the client stores the transaction in the wallet as *pending*, such that its inputs will not be reused for 24 hours.
- *Case 2: the corrupt outkey is the client's real one.* The client identifies the response as invalid, and does not transmit any signed transaction. The attacker learns partial information about the client's real outkey. This

case occurs with probability $\frac{1}{58}$ given the default settings of the client.

This attack is less appealing for the attacker since it identifies the client's real outkey with a fairly low probability. However, this attack appears difficult to eliminate entirely without authenticating the responses from remote nodes, an approach we explain shortly. If the client cannot tell that the response is invalid, it must behave the same as if the response is correct, i.e. by transmitting a transaction with a valid ring signature. However, if the client's actual outkey is the one corrupted, then there is no way for the client to construct the necessary ring signature.

IV. Authenticated Data for Monero Light Clients

A. Security Goals

The goal of our design is to provide a way for a light client connected to a remote node to construct a Monero transaction without revealing which is the real coin it's spending. Our attack model assumes that the node may be compromised and therefore provide maliciously corrupted responses. Our approach is to use hash-based authenticated data structures to enable the client to validate the responses it receives. Our scheme therefore relies on a collision-resistant hash function. Our main performance constraint is that the client should not have to store more than a constant amount of blockchain data, although we assume that the client is able to obtain the root hash of a valid blockchain (i.e., by downloading the block headers and checking proofs-of-work just like a Bitcoin SPV client [10]).

B. Overview of Data structure

The design of our authenticated data structure over the Monero blockchain involves utilizing nested Merkle hash trees. A tree is built over the transaction outputs in each transaction, the roots of which are used as a leaves in a Merkle tree over all the transactions in a block, the roots of which become leaves in a Merkle tree over all blocks. The maximum output index among the transactions are accumulated in each layer. Because new transactions are constantly committed as blocks are added to the ledger, we design our data structure to support efficient updates without needing to recompute the entire tree. The authenticated data structures generated are stored alongside the unmodified Monero blockchain on every full node. Using standard queries, the client, in reasonable time, can get a result as well as a proof to verify the result.

We choose to have a nested three-layer Merkle tree structure instead of simply building a Merkle tree directly over the entire set of transaction outputs for the following performance considerations:

- Many transactions contain a large number of outputs. Having a finer grained Merkle tree lets the client avoid downloading an entire transaction to fetch a single output.
- As there are a large number of blocks, a higher layer of Merkle trees avoids the client needing to download all the block headers starting from genesis.

1) Retrieving Queries: From storing the top Merkle root of the server, the client 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. Including the maximum index allows the client to sample an output based on its index. 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. 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 using a standard bisection search on the indices of the tree. 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.

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, the server traverses back up to the root of the Merkle tree, 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. At the end of each layer the proof for each individual tree is stored in a vector, and given to the client as a three-part proof.

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 proofs returned along with the response against the top-level Merkle root it has stored.

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 standard single Merkle tree verifier, shown in Fig. 3, takes the initial value of the proof and hashes the rest of the values in the proof according to the side. If the next value in the proof corresponds to the left sibling of the node in the Merkle tree, the verifier will concatenate the next value to the left of the current value, and proceed. Otherwise, the verifier will concatenate the next value to the right of the current value, and continue. 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

```
CHECKMERKLEPROOF(proof, root, output):
  link ← proof[0]
  if proof[0].index ≠ output.index
      return "Proof is invalid."
  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."
```

Fig. 3. The client validates this short Merkle proof by hashing the proof elements together, and then checking against the proof-of-work header. The global index check before the proof starts asserts that we are checking the correct output.

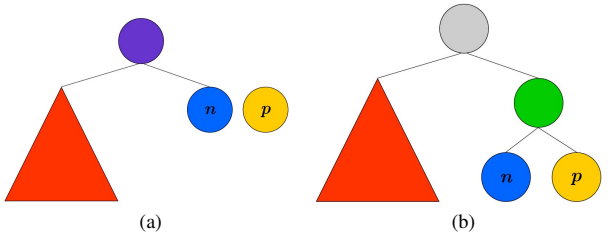


Fig. 4. *Appending to a Merkle Tree.* In (a), the red section is a complete subtree and node n also has a complete subtree underneath (empty tree). We wish to add node p. In (b), p has been added, and the green and top nodes recomputed. Appending the N^{th} element to a Merkle tree requires $O(\log N)$ steps.

requested output was fetched correctly.

4) Updating the Merkle Tree: The current Monero block rate is about 2 minutes. 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 be extended. The append operation for a Merkle tree is visualized in Fig. 4 and shown in Fig. 5. The steps are as follows:[4]

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 roots of complete subtrees. These 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.

[4]Our presentation of Merkle trees is based on Jamie Steiner's Merkle tree implementation (MIT license): `https://github.com/jvsteiner/merkletree`

```
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
```

Fig. 5. The ADDADJUST subroutine adds a new element to the Merkle tree without rebuilding the existing structure. The subroutine NEWTREE builds a Merkle tree over the two nodes passed in and returns the Merkle root.

When a 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.

C. 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 Fig. 6. 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 root would need to be updated and pushed out to the client as new blocks come in.

Fig. 6. Proposed 3-layer data structure to support authenticated Monero light clients. The leaves of the tree (yellow) are outkeys; throughout the entire tree, every node is augmented with the greatest global index of any outkey descendent from it. The bottom layer of the tree (a) contains one leaf for each outkey (yellow), and is rooted at an individual Monero transaction (blue). The hash of each transaction is used as a leaf in the next layer (a), which is rooted at an individual block (green). This layer has the same structure as the Merkle tree used in the existing Monero block chain. Finally, we build a Merkle tree over each of the blocks (c), such that the final root represents the entire blockchain.

V. Refereed Delegation

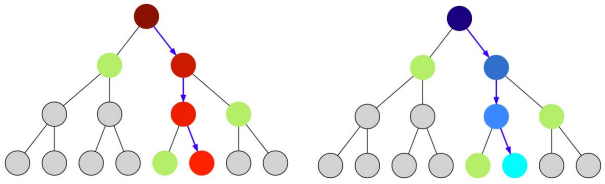
As an alternative to upgrading Monero with a hardfork to incorporate our data structure, we can instead use the referred delegation approach [4]. The main idea is that the clients query multiple different servers, and if the responses are inconsistent, the client runs an efficient conflict resolution protocol to reject the incorrect ones. If there is at least one honest server among those communicating with the client, the client can distinguish the correct one.

The conflict resolution protocol consists of the following three parties:

- 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.

When the client needs to fetch information from the blockchain to construct a transaction, it makes replicated queries to both of the servers. Along with the response, each server produces the top Merkle tree root built over all the outkeys. If both responses are identical, then the client accepts the answer; otherwise, the conflict resolution algorithm then proceeds as follows:

1) 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.
2) 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 *path* given in the next request. Otherwise, it will add an additional command LEFT to the request path.
3) 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.
4) 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.



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

Fig. 7. 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):
  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
      s1 ← GetBranchProof(server1, right1)
      s2 ← GetBranchProof(server2, right2)
  else
      s1 ← GetBranchProof(server1, left1)
      s2 ← GetBranchProof(server2, left2)
  if CheckProof(s1)
      return server1
  else if CheckProof(s2)
      return server2
  else
      return None
```

Fig. 8. The client runs the conflict resolution protocol by making calls to the servers in conflict. The GetChildren subroutine allows the client to traverse through a server's ADS from the root to the leaf. Afterwards, the client retrieves the Merkle branch proof for that leaf using the GetBranchProof subroutine. Using limited computing power, the client can verify proofs and return the honest remote node. If the proofs of both remote nodes are invalid, then the conflict resolution will not return any honest nodes.

5) Upon reaching the transaction in which the two servers disagree, Steps 2-4 are repeated until the client receives the leaf node at which the servers initially disagree. This leaf node corresponds to the output in which in conflict arises, and marks the overall initial point at which the two servers first disagree.
6) The client asks each server to provide the branch proof for its version of the leaf node found in Step 6. The branch proofs in here are the ones that are already present in the Monero blockchain, that were created to support Simple Payment Verification.
7) The client verifies the two proofs returned, and finds that one of the proofs is incorrect. Thus, the server that provided the incorrect proof is the one that has the modified output, and is marked as dishonest.

In this design, which is visualized in Fig. 7 and shown in Fig. 8, 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.

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.

VI. Implementation and Benchmarks

To demonstrate the feasibility of our approach we developed a prototype implementation, including both the authenticated data structure and the refereed delegation conflict resolution protocol. The implementation of the prototype is publicly available at: `https://github.com/kvnl33/monero-ads`

A. Storage Requirements and Disk Latency

For our experiments, we made use of the actual Monero blockchain as of November 2017. Extracting the relevant data from the Monero blockchain (the public keys associated with each RingCT outkey) results in 1.57 GB of data in total. Computing all of the Merkle tree hashes on top of this takes a total of 2.17 GB. For comparison, the current size of the Monero blockchain is 33.52 GB.

B. Benchmarks

We deployed a test network across three Lenovo System x3650 M5 servers, each running an Ubuntu 14.04 image. The network ran as a single-threaded process on an Intel E5-2650 2.2 GHz processor, with 1 TB HDD as permanent storage and 128 GB of RAM. Two instances are deployed as remote node servers, which contain the Monero blockchain, while the other acts as a client. We characterize the performance of our proposal by measuring the time it takes to carry out each separate task. Our results are summarized in Table 1.

The Build task involves constructing the Merkle tree index on the Monero blockchain (for the the RingCT outputs only),

TABLE I
COMBINED PERFORMANCE RESULTS, 1000 TRIALS

	Average Performance
Build	1785.935888 seconds
Query response	0.007219 seconds
Proof verification	0.000132 seconds
Update top tree	0.003195 seconds
Conflict resolution	0.171619 seconds

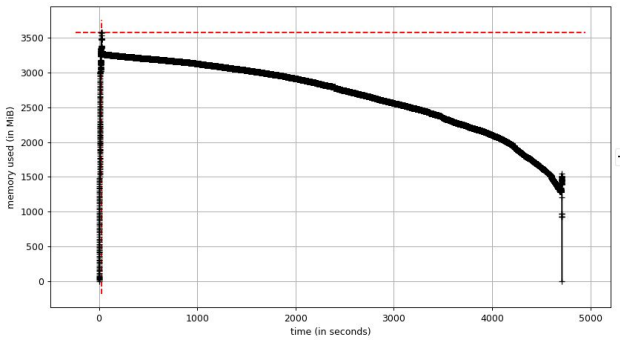


Fig. 9. Memory usage at the server during the build phase.

taking around 30 minutes (the memory requirements for this process are shown in Fig. 9). This represents the one-time setup cost that would need to occur when upgrading full nodes prior to a hardfork to adopt our proposed change. To respond to client queries, the server takes 0.007 seconds to fetch and produce a Merkle tree proof for a single outkey (and hence less than a minute to respond to all the requests made for constructing one transaction). Verifying the proof however takes less than a millisecond. In the current Monero protocol, it takes 0.002 seconds on average for a remote node to return all the outkeys needed for a default transacton (58 outkeys). In our proposed protocol with authentication, it takes 0.093329 seconds for the same number of outkeys. We suspect that significant optimization is possible and that the performance is largely due to our choice of serialization library. The updated top root of each server was also checked for correctness.

In the Conflict resolution test, we simulated the worst-case scenario where one server modifies a random transaction before building the Merkle tree over the blockchain. The client must then carry out the entire conflict resolution protocol to identify the tampered transaction. The running time reflects the entire process occurring sequentially, including network delays and a single-threaded client making parallelized calls to the remote nodes. A future optimization 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 up from where it last ended.

VII. CONCLUSION AND FUTURE WORK

In this paper we have identified a new privacy hazard in Monero light clients, where a malicious remote node sends the client bogus blockchain data and can thereby trace the client's transactions. We have disclosed this to Monero core developers, along with short term countermeasures that can make these attacks less effective. To systematically eliminate the underlying hazard, we propose to augment the Monero blockchain structure to enable clients to validate responses from remote nodes.

Our prototype implementation shows feasibility along with initial performance criteria. There are several improvements in design choice that could be made, as mentioned throughout this paper, that would further optimize runtime and is the focus of ongoing work. Our focus has been on Monero, however we conjecture that other privacy-preserving cryptocurrencies (e.g., Dash, Zcash) would need to overcome analogous limitations, for which a similar approach based on authenticated data structures may prove useful.

REFERENCES

[1] 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

[2] 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

[3] L. Reyzin, D. Meshkov, A. Chepurnoy, and S. Ivanov, "Improving authenticated dynamic dictionaries, with applications to cryptocurrencies," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 376–392.

[4] 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, pp. 445–454. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046759

[5] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, "A fistful of bitcoins: characterizing payments among men with no names," in *Proceedings of the 2013 conference on Internet measurement conference*. ACM, 2013, pp. 127–140.

[6] D. Ron and A. Shamir, "Quantitative analysis of the full bitcoin transaction graph," in *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 6–24.

[7] F. Reid and M. Harrigan, "An analysis of anonymity in the bitcoin system," in *Security and privacy in social networks*. Springer, 2013, pp. 197–223.

[8] 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

[9] A. Kumar, C. Fischer, S. Tople, and P. Saxena, "A traceability analysis of monero's blockchain." *IACR Cryptology ePrint Archive*, vol. 2017, p. 338, 2017.

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

[11] 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

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

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

[14] (2015) Ethereum. [Online]. Available: https://www.ethereum.org/