TICK: Tiny Client for Blockchains

Wei Zhang*, Jiangshan Yu†, Qingqiang He*, Nan Zhang* and Nan Guan*
*Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China
†Department of Software Systems & Cybersecurity, Monash University, Australia

***Abstract*—In Bitcoin-like systems, when a payee chooses to accept zero-confirmation transactions, it needs to verify the validity of the transaction. In particular, one of the steps is to verify that each referred output of the transaction is not previously spent. The conventional lightweight client design can only support such operation in the complexity of O(N_T), where N_T is the total number of transactions in the system. This is impractical for lightweight clients.**

The latest proposals suggest to summarize all the unspent outputs in an ordered Merkle tree. Therefore, a light client can request proof of presence and/or absence of an element in it to prove whether a referred output is previously spent or not, in the complexity of O(log(N_U)), where N_U is the total number of unspent output in the system. However, updating such ordered Merkle tree is slow, thus making the system impractical — by our evaluation, when a new block is generated in Bitcoin, it takes more than one minute to update the ordered Merkle tree.

We propose a practical client, TICK, to solve this problem. TICK uses the AVL hash tree to store all the unspent outputs. The AVL hash tree can be update in the time of O(M*log(N_U)), where M is the number of elements that need to be inserted or removed from the AVL hash tree. By evaluation, when a new block is generated, the AVL hash tree can be updated within 1 second. Similarly, the proof can also be generated in the time of O(log(N_U)). Therefore, TICK brings negligible run-time overhead, and thus it is practical. Benefited by the AVL hash tree, a storage-limited device can efficiently and cryptographically verify transactions. In addition, rather than requiring new miners to download the entire blockchain before mining, TICK allows new miners to download only a small portion of data to start mining.

We implement TICK for Bitcoin and provide an experimental evaluation on its performance by using the current Bitcoin blockchain data. Our result shows that the proof for verifying whether an output of a transaction is spent or not is only several KB. The verification is very fast – generating a proof generally takes less than 1 millisecond, and verifying a proof even takes much less time. In addition, to start mining, new miners only need to download several GB data, rather than downloading over 230 GB data.

***Index Terms*—Block chain, Light client, Transaction verification.**

I. Introduction

Since the introduction of Bitcoin [1], blockchain has emerged as a very attractive technology and promises the tremendous potential for creating new applications and business models. One of the most exciting aspects of blockchain technology is that it is entirely distributed, rather than controlled by one central point [2]. The lack of a single authority makes the system fairer and considerably securer. To this end, all the nodes in the system must maintain the whole transaction history locally and agree on a unique order in these transactions, therefore imposing a huge requirement on the storage capacity. For instance, currently, the sizes of the transaction history of Bitcoin and Ethereum are more than 230 GB [3] and 3 TB [4], respectively, which are too large to be implemented on storage-limited devices (e.g., mobile devices).

To lift the restriction caused by the heavy storage requirement, instead of downloading the whole transaction history, most blockchains support the lightweight client, also known as the light client, which only needs to download the block headers and use the simplified payment verification (SPV) to verify transactions. The light client, to some degree, is more applicable to storage-limited mobile devices, but still has several limitations in practice. First, its data size is linearly increased as the number of block headers is ever-growing. As storage limited devices generally have a constant and small storage capacity, the size of the light client had better be constant and small. Second, it is nearly impossible to verify transactions. Bitcoin uses the output to denote the asset. During a transaction, a payer refers to its outputs to denote the asset paied by the payer, and a new output belonged to the payee is generated to denote the asset received by the payee. Therefore, in a transaction, the referred output can not be referred to previously. However, using the SPV protocol, a light client can not verify whether an output is referred by other transactions unless querying all its following transactions which is far from practical.

Generally, a transaction is valid if it is permanently recorded on the blockchain (for example, bitcoin recommends to have 6 confirmations, i.e., the block containing the transaction is followed by at least 5 blocks in the chain). However, in many cases, a payee cannot wait for an hour to confirm a transaction, and therefore the zero-confirmation transaction [5] is proposed and considered by many Bitcoin merchants. That is, merchants may consider accepting micro-transactions with no confirmation in the blockchain, as this provides a faster way to manage small-value transactions. However, since no miner has verified this transaction yet, it is vital for the merchants to at least verify the validity of the transaction. Unfortunately, a light client is hard to perform such fast payment as it can not validate newly issued transactions.

Miners play an important role in the proof-of-work (POW) based blockchains, they compete for the right to generate the new block and then get a reward. During the competition, miners validate transactions and audit the whole blockchain, thus guaranteeing the system's security. In principle, if an attacker controls more than 51% computing power, it can control the system and perform the 51% attack [6]. However, in existing blockchains, becoming a miner is time-consuming and storage-consuming as a new miner has to download the whole transaction history before mining. Consequently, the system may lose some potential miners, which is a big loss for the development of the system.

To solve the aforementioned problems, some proposals [7], [8] store all the unspent outputs, also called as UTXO, in an ordered Merkle tree. Hashes of all the UTXOs are lexicographically stored in the leaves of the Merkle tree, and the root hash of the Merkle tree is stored in the block header. The ordered Merkle tree is maintained by the full nodes. Therefore, a light client can verify whether the output is a UTXO by the following steps.

1) The light client sends the objective output to a full node.
2) The full node generates the proof of presence if the output is in the ordered Merkle tree, otherwise generating the proof of absence.
3) The light client computes the root hash according to the received proof. If the computed root hash equals the root hash stored in the block header, the received proof is valid, otherwise, the received proof is invalid.

Since UTXOs are lexicographically stored, the objective output can be efficiently located and the proof can be efficiently generated. However, users that maintain the tree is responsible to update the tree when a new block is generated. In the worst case, when inserting/deleting an element to/from an ordered Merkle tree, the hash of all the elements need to be re-computed, leading a long delay and making the system impractical. The time complexity of proof generation and self-update of the ordered Merkle tree are shown in Table I.

In this paper, we propose TICK, a Tiny Client for blockchains. In TICK, all the UTXOs, are stored in an AVL hash tree [11], [12] which combines the hash tree with the AVL tree. Since the hash of each element is stored in its parent node, the AVL hash tree supports efficient verifiable cryptographic proofs on the presence and/or absence of data in it, in the size of O(log N_U). Therefore, a light client can verify transactions through requesting cryptographically verifiable proofs from a full node, rather than having to blindly trust it. More importantly, since it keeps the self-balancing nature of the AVL tree, the tree can be updated in the time of O(M(log N_U)) when a block is generated, which is significantly less than that of the ordered Merkle tree O(N_U), where M denotes the total number of inputs and outputs in a block. Therefore, it introduces a negligible run-time overhead, making the system practical. We have evaluated the performance of the AVL hash tree and the ordered Merkle tree. On the same computer, using the actual Bitcoin data, updating the AVL hash tree and the ordered Merkle tree cost 0.3s and 80s respectively. Since the average block time of Bitcoin is around 10 minutes, using the ordered Merkle tree significantly delays the mining process, thus making the system impractical. In contrast, $tick$ is practical. We store the root hash of the AVL hash tree in the coinbase of the corresponding coinbase transaction which is the first transaction in a block and provides up to 100 bytes for arbitrary data. Therefore, implementing TICK does not need to change the current Bitcoin structure.

Benefited by the UTXO tree, a light client can cryptographically verify transactions by only downloading the latest block header, therefore the size of the light client is small and constant. Similarly, a miner can start to mine by downloading the finalized UTXO tree and the latest block rather than the whole transaction historys. Our contribution can be summarized as follows:

- We propose TICK, the first piratical blockchain client that is able to efficiently verify a transaction without having to download all the blockchain data or trusting a full node. Moreover, TICK does not require the change of the current blockchain structure.
- In TICK, the size of the light client is small and constant and therefore is more applicable to storage-limited devices. In conventional blockchains, a light client should download all the block headers, which data size is large and ever-growing. But in TICK, a light client can only download a preferred constant small number of block headers.
- TICK reduces the data size that miners need to download before mining. In conventional blockchains, a newly joint miner needs to download the whole transaction history which may take several days and consume hundreds of GB memory; whereas in TICK, to start mining such miner only needs to download several GB of data.

Our work is applicable to other UTXO-based blockchains. However, for simplicity, we demonstrate how it works by using Bitcoin as an example. We collect the Bitcoin data from the genesis block to the 611707-th block[1]. Experimental results show that in TICK a light client only needs to download 80 bytes data (a block header) which is much smaller than 43 MB the size of the conventional light client, and the data size is constant over time rather than that of Bitcoin which is linearly increased. Moreover, the light client in TICK can verify transactions through requesting proof from an untrusted full node. Both the proof of presence and the proof of absence are less than 4 KB and therefore the communication latency is generally slight. In TICK, a miner can start to mine while only downloads around 5.9 GB data, which is much less than 230 GB data of Bitcoin. Hence, a large amount of time and storage space are saved, and furthermore becoming a miner is easier.

II. Related Work

To reduce the storage requirement, block summarization is proposed in several works [13], [14]. The main idea behind the block summary is that instead of storing all the blocks nodes can store the change in a sequence of blocks(called block summary). This summary can store all the input resources for the given blocks and the total change that was introduced by these blocks. But the compression ratio of the block

[1]This block is generated at 2020.01.07

Client	Data structure	Time complexity of proof generation	Time complexity of update	Transaction verification
Proposals [7], [8]	Ordered Merkle Tree	O(log N_U)	O(N_U)	Yes
Flyclient[9]	Merkle Mountain Rnage	O(log N_T)	O(log N_T)	No
UTREEXO[10]	Forest of Merkle Tree	O(N_U)	O(log N_U)	No
TICK	AVL Hash Tree	O(log N_U)	O(M *log N_U))	Yes

TABLE I: A comparison among related work and TICK, where N_U denotes the number of UTXO, M denotes the sum of the count of outputs and the count of inputs in a block, N_T denotes the number of transactions. TICK is the only one that supports transaction verification, efficient data items update, and efficient proof generation.

summarization method is not high enough for mobile devices. For example, in Bitcoin, they achieve a compression ratio of 0.54, as a result, a node still needs to download more than 100 GB data.

Flyclient [9] is a concurrent work proposing a light client for blockchain. It uses a Merkle Mountain Range (MMR) to store the hash of all the previous block headers. The MMR is maintained by a full node and the root hash of the MMR is stored in the block header. Therefore, a light client can verify all the previous block headers through requesting proofs from a full node, if it only stores the latest block header. Once a block header is verified, a light client can use the SPV protocol to verify whether a transaction is in the block. However, similar to the conventional light client, it still can not verify whether an output is previously spent unless requesting all its following transactions which is too complex to be practical.

Instead of summarizing all the block headers, Utreexo [10] introduces a hash based accumulator, a forest of perfect Merkle trees, to store all the UTXOs. The accumulator can provide cryptographically verifiable proofs of presence of a UTXO. If an output is in the forest, it is a UTXO. Therefore, a light client can verify whether an output is not preciously spent or not through requesting proofs of presence. However, since the accumulator is an unordered data structure, locating the objective output is inefficient which needs to search all the items in the worst case. Therefore, the proof generation is slow, and thus the system has low scalability. Moreover, it can not provide the cryptographically verifiable proofs of absence. If the output spent by the newly issued transaction is not in the accumulator (it may be spent before or is not exist), it can not provide a cryptographical proof to prove the invalidity of the transaction, unless providing the proof of presence of all the items in the accumulator which is impractical.

As discussed in section I, some works [7], [8] on the forum proposed to use the ordered Merkle Tree storing all the UTXOs. But they are impractical as the ordered Merkle tree is slow to update. We have evaluated that, when a new block is generated, more than 1 minute is costed to updated the ordered Merkle tree. Therefore the mining process is significantly delayed.

To sum up, to reduce the data size for transaction verification, the data for transaction verification can be summarized in a cryptographic data structure. Then, a node can verify transactions remotely while it only holds the digest of the data structure. However, to be practical and scalable, there are two challenges.

- **C1-Efficient in Proof Generation.** The proof to prove that whether an output is a UTXO should be efficiently produced. Otherwise, it can not handle a large number of proof requests, and thus has low scalability.
- **C2-Efficient in Self-update.** Updating the data items should be efficient. Otherwise, it delays the mining process, making the system impractical.

All the aforementioned works are summarized in Table I. Compared with all the related works, our work, TICK, is the only one that overcomes all these two challenges.

Besides, there are some works that protect the privacy of the light client which are orthogonal to our method. In order to verify transactions, a light client needs to request data from a full node through a peer to peer network. However, such a payment verification may leak considerable information about the clients, thus defeating user privacy that is considered one of the main goals of decentralized cryptocurrencies. In [15], [16], they use available trusted execution capabilities, SGX enclaves to protect the user privacy.

III. Preliminaries

Using Bitcoin as an example, we briefly introduce the UTXO-based blockchains in Section III-A. Then we show its limitations in Section III-B and discuss the causes of these limitations in Section III-C.

A. Bitcoin

Bitcoin [1] is the first and still the most popular cryptocurrency blockchain system. Using Bitcoin, users can issue transactions to transfer BTC, a token, with each other. Bitcoin includes two types of transactions: the coinbase transaction and the standard transaction. Coinbase transactions, the first transaction of each block, create new BTC from nothing by the miner. It records the block rewards and contains the coinbase which provides 100 bytes for arbitrary data. Standard transactions, the more common one, record the BTC transfer between the payer and the payee. A standard transaction contains a Txid, at least one input, and at least one output. During a transaction, a payee may receive some BTC, which is denoted by the output. Also, a payer may spend some BTC, which is denoted by the input. Therefore, the spent BTC should be owned by the payer and not be previously spent. To this end, the input should refer to an output owned by the payer, and the referred output can not be referred by other input. For example, if an input, $(\mathsf{Tx}20, 3, \mathsf{signature})$, is recorded on the blockchain, the BTC received in output3 of Tx20 is spent, and it can not be spent again. Specifically, an input denotes the spent BTC in this transaction, and it must refer to an output that is not spent yet to indicate the capital source. Once an output is referred (i.e. it is consumed), it can not be referred by others again.

Transactions are included in blocks by miners and then hashed as parts of a Merkle Tree. Before grouping transactions into a block, transactions should be verified by miners to be valid. This process is also known as the famous mining process. In Bitcoin, the transaction verification is generally divided into the following four steps.

1) The transaction is issued by the payer.
2) In each transaction, the sum of the value of all the outputs is no larger than the sum of the value of all the referred outputs.
3) The referred output is existing.
4) The referred output has not been referred by any other inputs.

Among these four steps, the first one can be easily performed by checking the signature of each input. The combination of the 3rd step and the 4th step is to make sure the referred output is not previously referred. The 2nd step needs to get the value of all the referred outputs. The output that not referred by any other input is also called Unspent Transaction Output (UTXO). Therefore, when a new transaction is issued, the major job of transaction verification is to make sure the referred outputs are UTXO. In Bitcoin, a full node stores the whole transaction history, and it therefore can easily verify a newly issued transaction locally.

The mining process is not coordinated by any central party, different miners may generate different blocks at the same time, and therefore the blockchain may fork into multiple chains. To agree on the same chain consistently with other nodes, each node downloads all blocks in every chain and picks the one with the highest total difficulty to follow. Using this strategy, it is shown that, in the long run, the network will agree on a single chain [17], [18], [19], known as the main (valid) chain. Therefore, a transaction is not guaranteed on the blockchian if there are not enough blocks behind it. In Bitcoin, generally, a block is considered on the chain permanently if there are 5 blocks behind it. In other words, when issuing a transaction, if a user can not verify its validity, he needs to wait at least 6-confirmations (i.e., around 60 minutes) to make sure the transaction is recorded on the main chain.

In a block, transactions are stored in a Merkle tree. We show an example of a Merkle tree in Figure 1. In order to differ from another Merkle tree defined in the following, we call the Merkle tree that stores transactions as a **transaction tree**. In the transaction tree, the leaf nodes are transactions and the non-leaf nodes are the hash of its children nodes. The hash of the tree, root hash, is stored in the block header. Then the block header is hashed as Prev Hash, which is stored in the next block header. Therefore, any modifications to the transaction history will lead to a different root hash value and different Prev Hash in the next block, and it can be detected by comparing the computed root hash with the root hash stored in the block header.

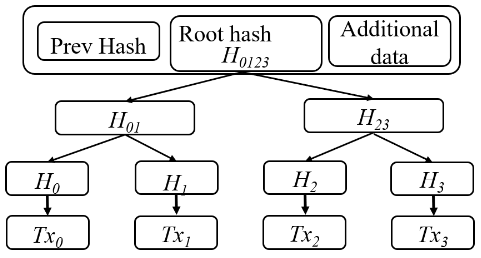


Fig. 1: A Merkel tree

In Bitcoin, a full node stores all the blocks and thus it can easily verify transactions. However, storing all the blocks needs a huge storage space which is impossible for storage-limited devices. In order to reduce the storage requirement, Bitcoin also supports the light client which only stores the headers of each block. However, the light client can not verify transactions, while it can only use the Simplified Payment Verification (SPV) protocol to perform the first 3 steps of transaction verifications. For the example shown in Fig. 1, if a light client wants to know whether Tx0 is in the block, it requests a proof from a full node. The full node, in turn, replies hash values of H_1 and H_{23}. These returned hash values, called critical hashes of Tx0, are sufficient for the light client to reconstruct the root hash of the Merkel tree. If the computed root hash equals the root hash stored in the block header, the light client is convinced that Tx0 is in the block. Therefore, it can verify whether an output is in Tx0 (step 3) and get the value of it (step 2). However, it can not verify whether the output is referred by other inputs of following transactions or not (step 4) unless requesting proofs of all the following transactions which is too cumbersome to be feasible. Therefore, a light client can not verify a newly issued transaction, and thus has to wait for 6-confirmations.

B. Limitations

As discussed above, the light client which only stores the block headers can significantly reduce the storage requirement, but it can not verify transactions. Therefore, when issuing a transaction, in order to make sure it is valid, a light client needs to wait about one hour (the block is recorded on the chain and five blocks behind it) to validate the correctness of the transaction. But for many cases, waiting for such a long time is infeasible. Now many merchants (such as Bitcoin ATMs) consider accepting micro-transactions with no confirmation in the blockchain, as this provides a faster way to manage small-value transactions. However, since no miner has verified this transaction yet, it is vital for the merchants to at least verify the validity of the transaction. But unfortunately in existing blockchains, the light client can not verify transactions. Therefore, to perform the fast payment, the light client has to either blindly accept an unconfirmed transaction or blindly trust a full node, which are both insecure.

On the other hand, although the light client needs less storage space compared with the full node, its storage space requirement still increases linearly over time. For example, the

Ethereum blockchain has more than 7.4 million blocks [4] and each block header is 528 bytes. Consequently, a light client in Ethereum should store more than 4.2 GB of data and the size increases 528 bytes every 13 seconds. Therefore, with the growth of the block height, storage space, the limited resources of mobile devices, will be consumed continuously and unlimitedly. In summary, when applying the light client on mobile devices, it causes the following limitation.

Limitation 1: The light client can not validate newly issued transactions, so that light client has to perform fast micropayment with higher risk when accepting 0-confirmation transactions. In addition, its data size increases linearly and unlimitedly with the number of blocks, which is infeasible for the storage-limited devices.

Another limitation of the blockchain system is that participating in the mining process is hard. It only allows the full node to participate in the mining process. In other words, before mining, a node needs to download the full transaction history, consuming a large amount of storage space and time. For example, the miner in Bitcoin and Ethereum should have at least 230 GB [3] and 3 TB [4] storage space respectively. Even a user with enough storage capacity also needs several days to synchronize its local blockchain, causing a long delay. Note that, mining is an important part of the blockchain system which ensures fairness and keeps the blockchain network stable and secure. More miners in the blockchain system, securer, fairer and more stable the system will be. However, such a long delay and a high storage requirement are very unfriendly to users, thus may causing the loss of some potential miners and the emergence of the authority node. Therefore, the system's fairness, stability, and security are harmed. To summarize, such a huge amount of data that a miner needs to store may bring the following limitation.

Limitation 2: A miner must download the whole transaction history before it can start mining, which is time-consuming and expensive in storage. Therefore, for a new joint member, becoming a miner is extremely unfriendly, thus bringing a negative impact on the stability and security of the blockchain system.

C. Discussion

The root cause of **Limitation** 1 and **Limitation** 2 is that the size of the cryptographic data (the full transaction history in Bitcoin) for the transaction verification is too large. A user can verify transactions if he only holds the UTXO set, but it is vulnerable as the validity of the UTXO set can not be verified without the whole transaction history. The miner needs to verify transactions so that it must store all the transaction history. The light client needs to check whether a transaction is in the block or not so that it must store all the block header, and even so it still can not validate transactions. Therefore, how to reduce the cryptographic data size for transaction verification is crucial.

IV. Methodology

This section details the proposed method, TICK. TICK adopts an efficient data structure, AVL hash tree, to store all the UTXOs. As a result, the storage requirement of the light client and the miner are significantly reduced. Meanwhile, it introduces the negligible overhead, thus making the proposed system practical and scalable. Section IV-A details the AVL hash tree. The structure of the full node, miner and light client are presented in section IV-B. Then, the transaction verification of light client is detailed in section IV-C.

A. AVL hash tree

The major job of transaction verification is to make sure that all the referred outputs of the transaction are UTXO. Currently, in Bitcoin, all the UTXOs are already extracted and stored in a UTXO set. Since a full node stores the whole transaction history, it can audit the UTXO set and therefore can use the UTXO set to verify transactions. However, the UTXO set can not provide cryptographic proofs to prove the inclusion or exclusion of an element in it. Therefore, a full node can not help a light client to verify newly issued transactions by providing the proofs of absence and/or presence of a UTXO. In conventional blockchain system, only a node that stores the whole transaction history can verify transactions. To assign the ability of transaction verification to light clients, we use a cryptographic data structure to store all the UTXOs. Once a user holds the digest of the data structure, it can cryptographically verify a UTXO through requesting proofs of absence and/or presence of an element in the data structure, and then verify transactions.

In the past, lots of cryptographical data structures have been proposed (i.e. Merkle tree [7], [8], Merkle tree forest[10]), but they are either impractical or not scalable. To be practical and scalable, the adopted data structure should have the following two properties:

1) Inserting/deleting items into/from it should be efficient. The node that maintains the data structure is responsible to update it when a new block is generated. Therefore, updating the data items of the data structure must be in a negligible time. Otherwise, the mining process is delayed, and the system is impractical.
2) The proof of absence and presence should be quickly produced. Otherwise, it can not simultaneously handle a large number of requests from different light clients, and thus has low scalability.

However, sometimes, these two properties are contradictory. That is, an ordered data structure can generate proofs quickly but slow to update, as the updated structure should keep the order relation among elements in it. Simply appending or removing a node from the data structure may be quick to update, but it is slow or even hard to generate proofs of absence and/or presence as it is not an ordered data structure.

In TICK, we adopt the AVL hash tree which is an ordered data structure and quick to update. The data structure of an element in the AVL hash tree is defined as follows:

Definition 1 (Element of AVL hash tree): For a UTXO in an AVL hash tree, u_i, the label of u_i is defined as a tuple containing two elements, one is the hash of itself $\textit{Hash}(u_i)$, and the other one is the hash of the combination of the following elements:

- $\textit{Hash}(u_i)$: the hash of the UTXO.
- $\textit{Height}(u_i)$: the height of the subtree rooted by u_i.
- $\textit{Diff}_H$: the height of the left subtree of u_i minus the height of the right subtree of u_i.
- h_l: the hash of the label of the left subtree of u_i (*null* if it is not exist).
- h_r: the hash of the label of the right subtree of u_i (*null* if it is not exist).

For a set of UTXOs, building an AVL hash tree includes the following two steps:

1) According to the hash value of each UTXO, building a classic AVL tree. In this stage, the label of each element is its hash value.
2) From the last level to the first level (i.e. the root node), according to Definition 1, computing *Height*, $\textit{Diff}_H$, h_l and h_r of each element, and modifying its label accordingly.

For example, we have 7 UTXOs, u_1, u_2, ..., u_7. Among them, $\textit{Hash}(u_i) < \textit{Hash}(u_{i+1})$. Then, we follow the aforementioned instructions to build an AVL hash tree. First, we build a classic AVL tree as follows:

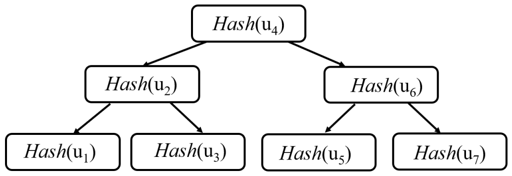


Fig. 2: A classic AVL tree

Then we compute the *Height*, $\textit{Diff}_H$, h_l and h_r for each element, and change its label accordingly. For simplicity, we define:

$$H_{u_i} = \textit{Hash}(\textit{Hash}(u_i), 0, 0, \textit{null}, \textit{null})$$

Therefore, the label of u_1, u_3, u_5, u_7 can be denoted by $(\textit{Hash}(u_1), H_{u_1})$, $(\textit{Hash}(u_3), H_{u_3})$, $(\textit{Hash}(u_5), H_{u_5})$, $(\textit{Hash}(u_7), H_{u_7})$ respectively, and finally the AVL hash tree can be obtained as follows:

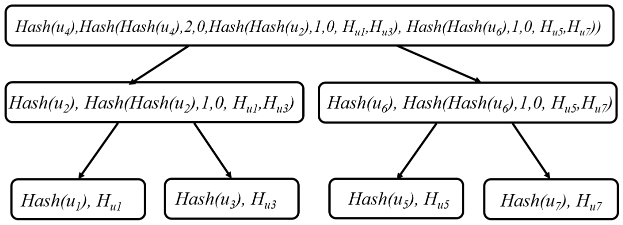

Hash(u4),Hash(Hash(u4),2,0,Hash(Hash(u2),1,0, Hu1,Hu3), Hash(Hash(u6),1,0, Hu5,Hu7))
Hash(u2), Hash(Hash(u2),1,0, Hu1,Hu3)
Hash(u6), Hash(Hash(u6),1,0, Hu5,Hu7)
Hash(u1), Hu1
Hash(u3), Hu3
Hash(u5), Hu5
Hash(u7), Hu7

Fig. 3: An AVL hash tree

In order to differ with the **transaction tree**, the AVL hash tree that stores all the UTXOs is denoted as **UTXO tree** in the following. The AVL hash tree combines the hash tree with the AVL tree. On one hand, it keeps the nice natures of a self-balancing binary search tree by ordering all the elements. Therefore, it is efficient when an element is inserted or deleted. Specifically, the computational complexity of inserting/removing an element is O(log N_U). Moreover, as it is an ordered data structure, users can efficiently locate an element in it and generate the proof. On the other hand, the AVL hash tree keeps the idea of a hash tree by storing the hash of the labels of a node in its parent node. Therefore, it can provide the cryptographic proofs of presence and absence of an element in it. Therefore, users who maintain the UTXO tree can efficiently provide proofs to prove the presence/absence of elements. The method of how to generate the proof of presence/absence is detailed in section IV-C.

Note, all the UTXOs can be straightforwardly obtained by traversing the transaction history. Furthermore, in some existing blockchain clients (e.g. Bitcoin client), UTXOs are already extracted. Once the UTXO tree is built, users only need to update it by inserting/deleting elements into/from it.

B. Node structure

In this section, we present the structure of the full node, miner and light client in TICK.

Full node. In TICK we introduce a new cryptographic data structure, UTXO tree, to store all the UTXOs. The UTXO tree is maintained by the full node locally. The root hash of the UTXO tree is stored in the coinbase. Note, coinbase has up to 100 bytes for arbitrary data to store. Therefore, TICK can be adapted without changes to the blockchain itself, and adopting TICK in Bitcoin does not lead to a hard fork. But the client implementation needs to be modified to support the proof generation and the light client transaction verification which are detailed in section IV-C. In addition, any malicious modifications to the UTXO tree will lead to a different root hash of the **UTXO tree** and a different root hash of the **transaction tree**, and therefore it will be detected as it will lead to a different Prev Hash in the next block.

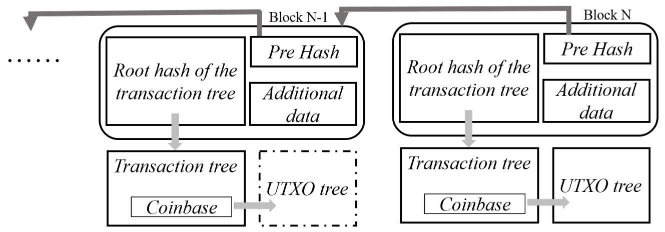


Fig. 4: Structure of full node

The structure of a full node is shown in Figure 4. We do not maintain the UTXO tree for each block, on the contrary, at a point, there is only one UTXO tree stored in the full node. The UTXO tree records all the UTXOs when the latest block is generated. When a new block is generated, the UTXO tree can be updated through function Update. Anyone maintaining the UTXO tree is responsible for calculating the UTXO tree for the newly generated block. Since the latest block may not be on the main chain in the future, in order to track the main chain, we also provide the function Rollback to get the UTXO tree of past blocks. As the UTXO tree is a self-balancing tree and the computational complexity of inserting/removing an element into/from it is O(log N), Update operation and Rollback both introduce a negligible runtime overhead.

To update the $utxo$ tree when a new block is generated, we define the function of Update as follows. Update is performed with each transaction Tx in the new block. For each Tx, Update function first deletes all the outputs referred by it from the UTXO tree, and then adds all the new introduced outputs to the UTXO tree.

ALGORITHM 1: Update($block$)

```
1: for every Tx in $block$ do
2:     for every input in Tx do
3:         delete the UTXO referred by the input
4:     end for
5:     for every output in Tx do
6:         insert the output
7:     end for
8: end for
```

Since there may be some splits in the chain and a new block may be not on the main chain eventually, in order to track the main chain and get the UTXO tree on other splits, we define the function of Rollback in the follows to get the UTXO tree of previous blocks. Similar to Update function, Rollback function just does the opposite.

ALGORITHM 2: Rollback($block$)

```
1: for every Tx in the $block$ do
2:     for every output in Tx do
3:         delete the UTXO
4:     end for
5:     for every input in the Tx do
6:         insert the output referred by the input
7:     end for
8: end for
```

Update and Rollback both need to delete and insert elements from the UTXO tree (i.e. line 3 and line 6 of Update and Rollback). The insertion and deletion operation are similar to the insertion and deletion operation of the AVL tree. The only additional operation is changing the label of elements when the label of its child node is changed. For simplicity, we do not detail them again in this paper. Similar to the AVL tree, the time complexity of the AVL hash tree insertion and deletion are both O(log N_U). Assuming that the total number of outputs and inputs of a block is M, the time complexity of updating a tree is O(M *log N_U).

Compared with the conventional blockchain system, the storage overhead is only the one hash value (i.e., the root hash of the UTXO tree) which is 32bytes for each block and the UTXO tree. For instance, the size of the UTXO tree in Bitcoin is around 5.9 GB[2] which is significantly less than 230 GB, the size of the whole transaction history [3]. Besides, the total size of all the root hashes of past UTXO trees are $611707 *$ 32bytes (e.g. 18 MB). Therefore, in TICK a full node needs to additionally download around 5.9 GB data. Note, a full node needs to store more than 230 GB transaction history originally, and therefore the introduced storage overhead is relatively slight.

Miner. In Bitcoin, a node can start to mine only if it owns the whole transaction history, However, downloading such a huge size of data (i.e. more than 200 GB) is time consuming and storage-space consuming. In TICK, as all the UTXOs are cryptographically stored in the UTXO tree, users can verify whether an output is a UTXO or not while only downloading the UTXO tree and the latest block header. This is because, users can verify the UTXO tree by comparing the root hash of the UTXO tree with the root hash stored in the coinbase. Therefore, a user can verify newly issued transactions locally by only downloading the latest block header and the UTXO tree. By now, the data size that a miner needs to download is around 5.9 GB which is significantly less than the whole transaction history. Furthermore, this gap increases over time.

Due to the introduction of UTXO tree, the mining process is slightly changed. In TICK, the mining process is:

a) solve a puzzle;
b) generate the hash value of the previous block header;
c) verify all the transactions that they want to group in the new block;
d) build the UTXO tree for the new block.

When a miner wants to group a set of Txs into a block, it uses Update to get the new UTXO tree, fill the root hash of the UTXO tree into the Tx tree, and then get the root hash of the Tx tree. Compared with the conventional blockchains, the only additional operation is d), building the UTXO tree for the new block according to the aforementioned Update function. According to our evaluation, step d) generally consumes less than 1 second, which is significantly less than the average block time of Bitcoin (i.e. 10 minutes). Therefore, the mining process is not delayed. Consequently, in TICK, a miner can only download the UTXO tree and the latest block rather than the whole transaction history. Hence, becoming a miner consumes less storage space and less time.

Light Client. In TICK, instead of downloading all the block headers, similar to the proposed miner, the light client can also only download a preferred small number of block headers. Since a light client does not need to track the main chain, in principle, a light client can always download the latest block header from the main chain. Therefore, compared with the conventional light client, it downloads fewer and constant data. As a result, no more storage spaces will be consumed with the growth of the block height.

Since all the UTXOs are stored in the UTXO tree, a full node can provide the proof to verify whether a UTXO is

[2]We collect the data of Bitcoin from the 1st block to the 611707 block.

in it or not. Therefore, a light client can verify whether an output is a UTXO or not, through requesting proofs from a full node. With the received proofs, the light client can re-construct the root hash of the UTXO tree. If the computed root hash equals the root hash stored in the coinbase, the proof is valid. Note, the coinbase is in the transaction tree and a light client can verify it by the SPV proctcol. Therefore, a light client can cryptographically verify the validity of a newly issued transaction. The light client transaction verification is detailed in section IV-C.

In contrast with the conventional light client which data size linearly increases over time, the data size of a light client in TICK is significantly small and constant, which is more applicable to storage-limited mobile devices. Moreover, the proposed light client can verify transactions through requesting proofs from an untrusted full node, and thus the light client is given the ability to perform the fast payment.

C. Light client transaction verification

In conventional blockchains, the light client can not verify newly issued transactions, as it can not verify whether an output is a UTXO or not. In TICK, all the UTXOs are stored in the UTXO tree, and its digest is stored in the transaction tree. Therefore, users can verify transactions, if they know the root hash of the transaction tree (which is stored in the block header). Therefore, in TICK, even though a light client may only store the latest block header, it can cryptographically verify newly issued transactions. To prove an output is a UTXO, a light client needs to send the output to a full node which maintains the UTXO tree to get the proof of presence or absence.

Proof of presence. Since the UTXO tree is a hash tree, where each node's label is hashed and stored as a part of the label of its parent node, it can provide the cryptographic proof to prove a node is in the tree. Similar to the SPV proof discussed in section III-A, in the AVL hash tree, the proof of the presence of an element also includes its critical hash values. We denote the critical hash value of a UTXO, u_i, as $\textit{CriHash}(u_i)$. For the example in figure 3, the critical hashes of u_3 are:

$$(H_{u_1}, \textit{Hash}(\textit{Hash}(u_6), 1, 0, H_{u_5}, H_{u_7}))$$

Except for these critical hashes, the proof of presence also includes $\textit{Hash}(u)$, $\textit{Height}(u)$, $\textit{Diff}_H$ of all the elements that on the path from the root (i.e. u_4) to u_3 and h_l, h_r of u_3 (itself). With these data, users can re-construct the root hash of the UTXO tree. The Proof of Presence can be defined as follows.

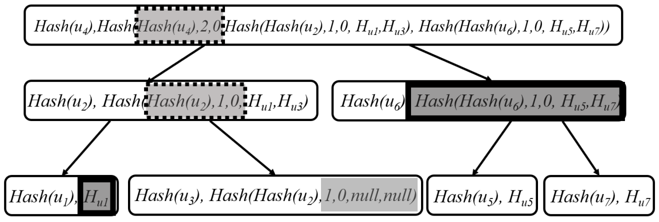
Definition 2 (Proof of Presence): For a UTXO, u_i, its proof of presence, denoted by $\mathsf{PoP}(u_i)$, is the combination of the following data:

- $\textit{CriHash}(u_i)$
- $\textit{Hash}(u)$, $\textit{Height}(u)$, $\textit{Diff}_H$ of all the elements that on the path from the root to u_i (including the root, excluding u_i).
- $\textit{Height}(u)$, $\textit{Diff}_H$, h_l and h_r of u_i.

With this proof, the light client can re-construct the roof hash of the AVL hash tree, if it only holds u_i. For the example in figure 3, as represented figure 5, proof of presence of u_3 is:

$$\begin{aligned}\mathsf{PoP}(u_3) = (&H_{u_1}, \textit{Hash}(\textit{Hash}(u_6), 1, 0, H_{u_5}, H_{u_7}),\\ &\textit{Hash}(u_4), 2, 0, \textit{Hash}(u_2), 1, 0,\\ &0, 0, \textit{null}, \textit{null})\end{aligned}$$

In figure 5, we expand H_{u_3} and use the shadow boxes to represent the data of $\mathsf{PoP}(u_3)$. Among the data covered by these shadow boxes: the data surrounded by the solid lines are critical hashes of u_3; the data surrounded by the dotted lines are the required data of the elements on the path from the root (u_4) to u_3; the rest are the required data of u_3.


Hash(u4),Hash(Hash(u4),2,0,Hash(Hash(u2),1,0, Hu1,Hu3), Hash(Hash(u6),1,0, Hu5,Hu7))
Hash(u2), Hash(Hash(u2),1,0, Hu1,Hu3)
Hash(u6), Hash(Hash(u6),1,0, Hu5,Hu7)
Hash(u1), Hu1
Hash(u3), Hash(Hash(u2),1,0,null,null)
Hash(u5), Hu5
Hash(u7), Hu7
Fig. 5: proof of presence of u_3 in figure 3

Proof of absence. To prove that a particular output is not a UTXO, the full node needs to prove that the output is absent from the UTXO tree by performing the following steps:

- Locate node u_i such that its hash value is lexicographically the largest one smaller than the output.
- Locate node u_j such that its hash value is lexicographically the smallest one greater than the output.
- Prove that u_i and u_j are present in the AVL hash tree, and no elements in the AVL hash tree has the hash value that is larger than $\textit{Hash}(u_i)$ and smaller than $\textit{Hash}(u_j)$. The former is proved by using the proof of presence of u_i and u_j, and the latter one can be verified by checking the path between u_i and u_j which is denoted as $\textit{Path}(u_i, u_j)$. Specifically, if $\textit{Height}(u_i) > \textit{Height}(u_j)$, u_j must in the right subtree of u_i as $u_j > u_i$. Then we can provide the path from the root of the right subtree of u_i to u_j to prove u_j is the largest one in the right subtree of u_i. If from the root of the right subtree of u_i to u_j, the path only goes to the right child and u_j has no right child, u_j is the largest one in the right subtree of u_i. Therefore, no elements in the AVL hash tree has the hash value that is larger than $\textit{Hash}(u_i)$ and smaller than $\textit{Hash}(u_j)$. Similarly, if $\textit{Height}(u_i) < \textit{Height}(u_j)$, we can find a path from the root of the left subtree of u_j to u_i to prove the absence. Therefore:

$$\begin{aligned}&\mathsf{PoA}(u) =\\ &\qquad (\mathsf{PoP}(u_i), \mathsf{PoP}(u_j), \textit{Path}(u_i, u_j))\end{aligned}$$

In the following, we detail how the light client in TICK verify whether an output is a UTXO. Firstly, the light client sends each referred output of a newly issued transaction to a full node individually and then gets the proof of PoP or PoA.

After receiving the proof, the light client re-construct the root hash of the UTXO tree, and use the SPV protocol to verify whether the computed root hash equals the data stored in the coinbase. If the computed root hash equals the root hash stored in the coinbase, the computed root hash of the AVL hash tree tree is valid and the proof is valid. In more detail, as presented in Figure 6:

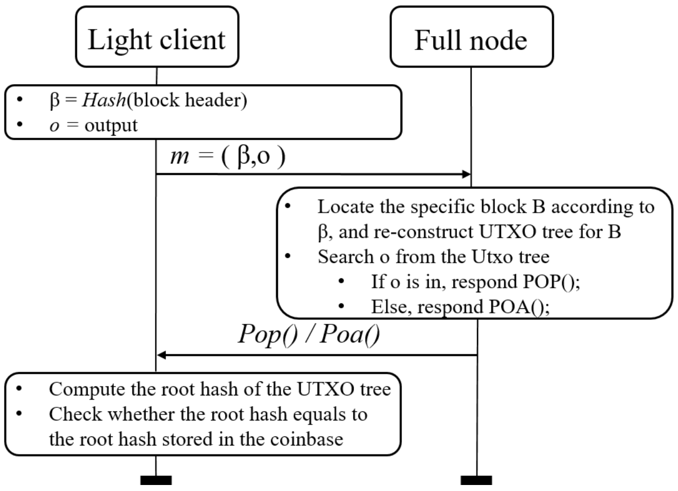

Light client
Full node
β = Hash(block header)
o = output
m = (β,o)
Locate the specific block B according to β, and re-construct UTXO tree for B
Search o from the Utxo tree
If o is in, respond POP();
Else, respond POA();
Pop() / Poa()
Compute the root hash of the UTXO tree
Check whether the root hash equals to the root hash stored in the coinbase

Fig. 6: light client transaction verification

- The light client gets the hash value β of the latest block header, and groups it with the referred output o as a message *m*. Then the light client sends *m* to a full node.
- When the full node receiving the message, it first locates the objective block according to the received β, and constructs the UTXO tree T_{UTXO} of the block. Since the UTXO tree is an ordered data structure, the full node can efficiently find whether the output is in the UTXO tree. If the output is in the UTXO tree, the full node generates the proof of presence PoP(output) and replies the proof to the light client. If not, the full node generates the proof of absence PoA(output) and replies the proof to the light client. In addition, the full node also replies the SPV proof of the coinbase transaction, which is denoted as $\text{SPV}_{\text{coinbase}}$.
- When the light client receiving the proof, it first computes the root hash of the UTXO tree. Then, with the computed root hash of the UTXO tree and the received $\text{SPV}_{\text{coinbase}}$, the light client computes the root hash of the transaction tree. If the computed root hash of the transaction tree equals the root hash stored in the block header, the received proof is considered as valid. Otherwise, the received proof is considered as invalid. In case the received proof is valid, the output is a UTXO if the received proof is PoP, otherwise, it is not a UTXO.

If all the outputs of a transaction are UTXO, the light client then computes the sum of the value of all the referred outputs. If it is no less than the sum of the value of all the outputs, the transaction is valid, otherwise, it is invalid.

During the transaction verification, the light client needs to communicate with the full node to request proofs, causing network latency. When the light client sending a message to the full node, the consumed bandwidth is the size of *m* which is $2 * 32$ bytes, i.e., the size of two hash values. When the light client receiving messages from the light client, the consumed bandwidth is the size of the PoA or PoP and the size of SPV(coinbase). Assuming the number of UTXOs in the UTXO tree is N_U, PoP is in the size of $O(\log(N_U))$, and the size of PoA is approximately twice of PoP. By our evaluation, the size of PoP is less than 2 kB and the size of PoA is less than 4 kB, which are both small. Therefore, transmitting PoP and PoA also consumes less bandwidth, causing negligible latency. Moreover, the size of PoA and PoP increases slowly with time. Specifically, the size of PoP increases $32+34$ bytes when the number of UTXOs doubled. That is because, when the number of UTXOs doubled the height of the UTXO tree increases 1. Therefore PoP may include one more critical hash value (32 byte) and there is one more node (the sum of size of *Hash*(u), *Height*(u) and *Diff*$_H$ is 34) on the path from the root to the leave.

V. Evaluation

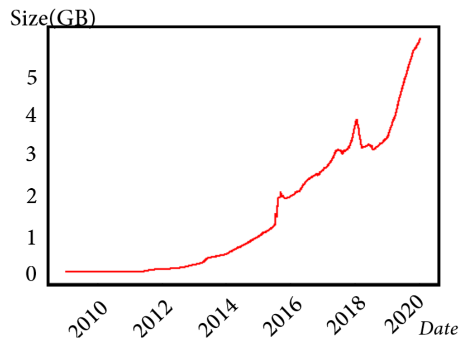
This section evaluates the performance of TICK in two aspects. On one hand, we analyze the data size of the full node, miners and light client in TICK, and compare them with the conventional Bitcoin to demonstrate that:

- the light client in TICK is more applicable to storage-limited devices;
- becoming a miner is easier;
- the introduced storage overhead to the full node is slight.

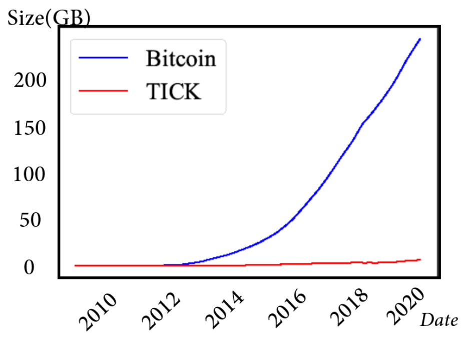
On the other hand, we also analyze the introduced runtime overhead and compared it with related works [7], [8] to demonstrate the scalability and the practicality. The evaluation is based on the data of the actual Bitcoin blockchain collected from the genesis block to the 611707-th block[3]. All the experiments are performed on a PC equipped with an Intel i9 processor and 64 Gb RAM.

Among all the collected blocks, there are 99431704 UTXOs. Therefore, the UTXO tree size is 5.9 Gb(99431704 * 32 *2 bytes). In TICK, although a full node needs to additionally store the UTXO tree, the size of the UTXO tree is two orders of magnitude smaller than its original size (the size of the whole transaction history). Moreover, a full node generally has a high storage capacity. **Therefore, the introduced storage overhead to the full node is slight.** We also present the dynamic change of the UTXO size with the growth of the blockheight in Fig. 7(a). According to the changing trend, the UTXO size grows slower than the data size of a full node in Bitcoin. Specifically, in the past year, the size of UTXOs only increases by 5% which is significantly less than the 29% increase of the data size of a full node in Bitcoin. Therefore, in the near future, the introduced storage overhead is relatively slight.

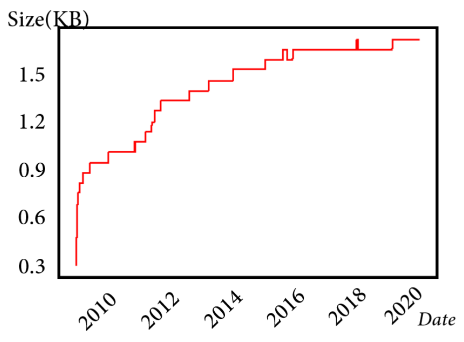
[3]This block is generated at 2020.01.07



(a) Size of the UTXO tree



(b) Minimum size of required data by miner



(c) Size of PoP

Fig. 7: Performance evaluation of TICK

In TICK, miners also need to store the UTXO tree, consuming storage space. However, users can start to mine while only download the latest block and the UTXO tree rather than the whole transaction history. Therefore, in TICK, users can start to mine if he only downloads around 5.9 GB data rather than more than 230Gb data. Compared with the conventional Bitcoin, the size of data that a miner needs to download is shown in Fig. 7(b). According to the changing trend, the expected gap between these two data sizes will increase over time. **Therefore, becoming a miner is easier which is good for the development of the blockchain.** Similar to the miner, a light client in TICK can only download the latest block header rather than all the block headers. Since the comparison between the data size of the light client in TICK and Bitcoin is straightforward (1 block header against 611707 block headers), we do not show it. Moreover, in contrast with the light client in Bitcoin which data size increases linearly, the data size of the light client in TICK is constant. **Therefore, it is more applicable to storage-limited devices.**

Benefited by the UTXO tree, in TICK, a light client can verify transactions through requesting proofs from a full node. Meanwhile, the runtime overhead is introduced. During the mining process, the miner and the full node need to update the UTXO tree. Therefore, the UTXO tree must be updated in a negligible time, otherwise, it may delay the mining process and thus making the client impractical. Other similar works[7], [8] propose to use the ordered Merkle tree to store all the UTXOs and generate proofs for the light client transaction verification. However, the ordered Merkle tree is slow to update. Compared with the ordered Merkle tree, we update the latest 100 blocks and get the average update time of the AVL hash tree and the ordered Merkle tree separately. Experimental results show that, the average update time of the AVL hash tree is around 0.3s, while the average update time of the ordered Merkle tree is around 84s. Therefore, the system that adopts the ordered Merkle tree delays the mining process and therefore is impractical. In contrast, when a new block is generated, updating an AVL hash tree only costs less than 1 second, bringing a negligible negative effect.

Except for the overhead caused by the UTXO tree maintenance, the light client transaction verification also introduces overhead. During the light client transaction verification, a light client needs to send the hash of the output and the hash of the latest block header to the full node. During this process, the light client only needs to send two hash values, i.e., 64 bytes, thus causing slight network latency.

After receiving the message from a light client, the full node should generate the proof accordingly. Since the AVL hash tree is an ordered data structure, the objective UTXO can be efficiently located. In the experiment, we randomly select 1000 UTXOs from all the 99431704 UTXOs, getting their proofs respectively and accounting the time consumption individually. Experimental results show that more than 99% of proofs can be generated within 1 millisecond, which is negligible. When the proof is generated, the full node should send the proof to the light client. With the growth of the number of UTXOs, the maximum size of PoP is shown in Fig. 7(c). Since the size of PoA is about twice the size of PoP, we do not show it again. When the block height is 611707, the maximum size of PoP is around 1.6KB, and the maximum size of PoA is around 3.2KB. Therefore, sending both PoP and PoA both cause slight latency. Moreover, the size of these proofs is logarithmic in the number of UTXOs, so it only increases dozens of bytes when the number of UTXOs is doubled. According to the changing trend of the UTXOs shown in Fig. 7(a), the size of UTXOs only increases by 5% in the past year. With this speed, the size of PoP and PoA are both very small in a long period. Last, the light client should compute the root hash of the UTXO tree to verify the proof. This process only needs to compute dozens of hash values, which can be completed in negligible time.

The runtime overhead brought by TICK can be summarized as follow:

- The time used to update the UTXO tree is less than 1 second.
- The time used to generate the proof for light client transaction verification is generally less than 1 millisecond.
- The network latency caused by transmitting the data (which size are both less than 10 KB) for light client transaction verification is negligible.

Therefore, the introduced runtime overhead is negligible, and thus TICK is practical and scalable.

VI. Conclusion

This paper proposes TICK, the first practical client that allows the light client to verify transactions without modifying the current blockchain structure. Therefore, the light client can perform fast payment for small value transactions, increasing the system's throughput. Moreover, the size of the proposed light client is smaller and constant, which is more applicable to storage-limited devices. Similarly, in TICK, the miner can download two orders of magnitude less data than the original, lowering the barrier for new miners. Last, TICK is scalable, as the run-time overhead caused by the light client transaction verification is negligible.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] C. Natoli, J. Yu, V. Gramoli, and P. J. E. Veríssimo, "Deconstructing blockchains: A comprehensive survey on consensus, membership and structure," *CoRR*, vol. abs/1908.08316, 2019.

[3] "Blockchain charts: Bitcoin's blockchain size," https://blockchain.info/charts/blocks-size/, (Accessed on 09/19/2019).

[4] "Ethereum blocks." https://etherscan.io/blocks/, (Accessed on 09/19/2019).

[5] C. Chen, "The mathematically secure way to accept zero confirmation transactions," *Cryptocoin news*, vol. 13, 2014.

[6] J. Bonneau, E. W. Felten, S. Goldfeder, J. A. Kroll, and A. Narayanan, "Why buy when you can rent? bribery attacks on bitcoin consensus," 2016.

[7] "Ultimate blockchain compression w/ trust-free lite nodes," https://bitcointalk.org/index.php?topic=88208.0.

[8] "Proposal: Merkle tree of unspent transactions (mtut)," https://en.bitcoin.it/wiki/User:DiThi/MTUT.

[9] L. Luu, B. Buenz, and M. Zamani, "Flyclient super light client for cryptocurrencies," accessed 2018-04-17.[Online]. Available: https://stanford2017 . . . , Tech. Rep.

[10] T. Dryja, "Utreexo: A dynamic hash-based accumulator optimized for the bitcoin utxo set."

[11] J. Yu, V. Cheval, and M. Ryan, "DTKI: A new formalized PKI with verifiable trusted parties," *Comput. J.*, vol. 59, no. 11, pp. 1695–1713, 2016.

[12] J. Yu, M. Ryan, and C. Cremers, "DECIM: detecting endpoint compromise in messaging," *IEEE Trans. Information Forensics and Security*, vol. 13, no. 1, pp. 106–118, 2018.

[13] A. Palai, M. Vora, and A. Shah, "Empowering light nodes in blockchains with block summarization," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[14] U. Nadiya, K. Mutijarsa, and C. Y. Rizqi, "Block summarization and compression in bitcoin blockchain," in *2018 International Symposium on Electronics and Smart Devices (ISESD)*. IEEE, 2018, pp. 1–4.

[15] S. Matetic, K. Wüst, M. Schneider, K. Kostiainen, G. Karame, and S. Capkun, "Bite: Bitcoin lightweight client privacy using trusted execution," IACR Cryptology ePrint Archive 2018, XXXX, Tech. Rep., 2018.

[16] K. Wüst, S. Matetic, M. Schneider, I. Miers, K. Kostiainen, and S. Capkun, "Zlite: Lightweight clients for shielded zcash transactions using trusted execution," in *International Conference on Financial Cryptography and Data Security. Springer*, 2019.

[17] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 281–310.

[18] L. Kiffer, R. Rajaraman *et al.*, "A better method to analyze blockchain consistency," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 729–744.

[19] R. Pass, L. Seeman, and A. Shelat, "Analysis of the blockchain protocol in asynchronous networks," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 643–673.