LVQ: A Lightweight Verifiable Query Approach for Transaction History in Bitcoin

Xiaohai Dai, Jiang Xiao, Wenhui Yang, Chaofan Wang, Jian Chang, Rui Han, and Hai Jin
National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
Email: jiangxiao@hust.edu.cn

***Abstract*—In the Bitcoin system, transaction history of an address can be useful in many scenarios, such as the balance calculation and behavior analysis. However, it is non-trivial for a common user who runs a light node to fetch historical transactions, since it only stores headers without any transaction details. It usually has to request a full node who stores the complete data. Validation of these query results is critical and mainly involves two aspects: correctness and completeness. The former can be implemented via Merkle branch easily, while the latter is quite difficult in the Bitcoin protocol. To enable the completeness validation, a strawman design is proposed, which simply includes the BF (*Bloom filter*) in the headers. However, since the size of BF is about KB, light nodes in the strawman will suffer from the incremental storage burden. What's worse, an integrated block must be transmitted when BF cannot work, resulting in large network overhead. In this paper, we propose LVQ, the first lightweight verifiable query approach that reduces the storage requirement and network overhead at the same time. To be specific, by only storing the hash of BF in headers, LVQ keeps data stored by light nodes being little. Besides, LVQ introduces a novel BMT (*BF integrated Merkle Tree*) structure for lightweight query, which can eliminate the communication costs of query results by merging the multiple successive BFs. Furthermore, when BF cannot work, a lightweight proof by SMT (*Sorted Merkle Tree*) is exploited to further reduce the network overhead. The security analysis confirms LVQ's ability to enable both correctness and completeness validation. In addition, the experimental results demonstrate its lightweight.**

***Index Terms*—Bitcoin, verifiable query, transaction history, Merkle tree, Bloom filter**

I. Introduction

Due to its merits of decentralization, traceability, and anonymization, Bitcoin has aroused massive attention since its birth in 2008 [1]. Essentially, Bitcoin can be considered as a distributed ledger, which is able to process and record transactions reliably, without relying any third-party entities [2]. In terms of the data integrity, Bitcoin nodes can be divided into two types: full node and light node [3]. The former stores the complete ledger data, including both block headers and bodies, which takes up more than 250GB storage space by December 2019. By contrast, the latter only stores the block headers, which reduces the storage requirement largely.

Without storing the block bodies, a light node usually needs to make requests to a full node for detailed ledger data. Particularly, a light node may request the balance or transaction history of a certain address. Let us take a real-life scenario as an example. A trendy coffee shop accepts Bitcoin as the payment method and the shop owner runs a light node on his/her smart phone. When a suspicious-looking man comes to pay by issuing a new transaction from an account, the owner wants to check if the man's account has enough balance immediately. In this case, the shop owner needs to make a request to a full node for balance calculation.

However, what if the full nodes accessed by a light node are malicious? In other words, a full node may return the wrong balance value to the light node, just to cheat the shop owner. As a result, a natural problem is that how to return a verifiable balance from a full node to a light node. Since there is essentially no 'balance' concept in UTXO-based blockchain system (i.e., Bitcoin) [4], the problem can be extended to 'how to return verifiable transaction history of a particular address'. With the transaction history, the balance of an address can be easily calculated out.

The problem of verifiable data query in Bitcoin can be divided into two parts: *correctness* and *completeness* [5]. *Correctness* requires that no extra data is returned, neither falsified nor incorrect. On the contrary, *completeness* demands that all the satisfactory data is returned without omitting. With regard to the query of verifiable transaction history, the full node is asked to return exact number of transactions relevant to a certain address.

There have been some attempts to implement the verifiable query [6] [7], the most fundamental one of which is named as the 'strawman' design. By simply including a Bloom filter (BF) in each header [8], the light node is able to check if an address appears in a block. To reduce the probability of *false positive match* (FPM), the size of BF is usually set much larger than the original header in Bitcoin (i.e., 80KB). In other words, it requires the light node to store a mass of data, which deviates the original design merits of light nodes.

In this paper, we propose LVQ, a lightweight verifiable query approach for transaction history in Bitcoin. By only storing the hash of BF in the header, LVQ is able to make the size of header small. Without storing the BFs in the headers, a light node relies on the full node to transmit the BF values along with the query result. To deal with the massive network overhead resulted from BF transmission, LVQ makes use of BMT (Bloom filter integrated Merkle Tree) to merge the previous BFs. Ideally, only one BF is transmitted in the query result, which is able to indicate the inexistence of an address in thousands of blocks. Since BF suffers from FPM problem, SMT (*Sorted Merkle Tree*) is introduced to deal with FPM [9]. Particularly, two branches in SMT instead of the integrated block are returned to indicate the inexistence. Size of the former is usually much smaller than the latter. Both BMT and SMT reduce the size of query result, which makes the query approach lightweight.

To evaluate our design, we implement the prototype systems of both strawman and LVQ, and conduct multiple experiments based on them. The experimental results demonstrate the lightweight of LVQ. For an address inexistent in the blocks, size of query result in LVQ is only 1.39% of that in the strawman. We also evaluate the effects of BMT and SMT respectively. It is concluded that BMT performs well for the address with few historical transactions, while SMT does a good job for the address with massive transactions.

In summary, our major contributions are as follows:

- The most fundamental approach (i.e., the strawman) to implement the verifiable query is presented in detail, with some challenges being proposed.
- An approach making use of two novel data structures (i.e., BMT and SMT) is elaborated, which deals with the challenges in the strawman and enables the lightweight verifiable query.
- Prototype systems are implemented to evaluate the LVQ approach, whose experimental results demonstrate its lightweight.

The rest of this paper is structured as follows. Section II describes the problem background and gives a rigid definition of the problem. Preliminary knowledges of SMT and BMT are introduced in Section III, which are adopted by the LVQ design elaborated in Section IV. Section V presents how to generate and verify the proofs in the query process. Security analysis and evaluations are performed in Section VI and Section VII respectively. Finally, we review the related works and come to some conclusions in Section VIII and Section IX.

II. Background & Problem

In this section, we introduce the background knowledge of Bitcoin, especially about data structure. Based on the knowledge, we describe the problem to be solved in this paper.

A. Data structure in Bitcoin

From the perspective of data structure, a block in Bitcoin consists of two parts: block header and block body. Size of the former is a constant of 80 bytes, while the latter is limited to be smaller than 1 MB. As time goes by, storing the complete blocks takes up considerable space, more than 250 GB by December 2019. To relieve the storage pressure, the light node is proposed to only store the block headers, whose size is smaller than 50 MB. Comparatively, a node storing the complete blocks is referred to a full node.

All the transactions packaged in a block are organized as a *Merkle Tree* (MT) [10]. A leaf node in MT is set as the cryptographic hash of a transaction, while a non-leaf node is set as the hash of its two child nodes. MT root is stored in the block header. A path from a leaf to the tree root is called a '*Merkle Branch*' (MBr), which can be used to prove the existence of a transaction. However, one deficiency of MBr is that it cannot prove the inexistence of a particular transaction.

B. Problem definition

Transaction history of an address, namely all the historical transactions taking the address as a sender or receiver, can be useful in quite a few scenarios. In particular, by analyzing the transaction history, we can possibly conclude some behavior patterns of an address and further deduce its real-world identity, such as exchange or mining pool [11] [12]. Besides, an address may appear in the input or output of a transaction, thus being a coin sender or receiver. Based on all the inputs and outputs appeared in the transaction history, the balance of an address can be calculated out according to Equation 1:

$$Balance(addr) = \sum_{j=0}^{m} v_j - \sum_{i=0}^{n} w_i \tag{1}$$

where m denotes the number of outputs relevant to $addr$ and v_j represents the value expressed in j-th output, while n and w_i represent the number of inputs and value expressed in i-th input.

Since a light node does not store the block bodies, it has to make requests to full nodes for the transaction history. However, as assumed in the threat model of Bitcoin, no third-party nodes (including the full nodes) should be totally trusted. Therefore, the query results from a full node must be verified to be both correct and complete.

- **Correctness** denotes that all the query results are exactly existed in the blockchain. As a result, data returned by the full node must provide the corresponding *existence proof*. As stated in Section II-A, a MBr can be easily used to prove that a satisfactory transaction is exactly existed in a block.
- **Completeness** represents that no satisfactory results are omitted either intentionally or unintentionally. Particularly, if there are no satisfactory transactions in a block, the full node must provide the *inexistence proof* of this block. In fact, provision of the lightweight *inexistence proof* (also known as *non-membership proof*) is considered as an especially difficult problem [13] [14].

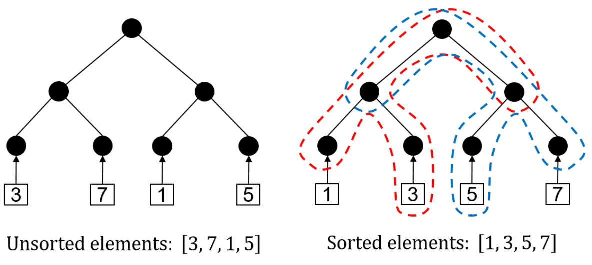
In this paper, we propose an efficient approach to return transaction history of a particular address, from a full node to a light node, whose **correctness** and **completeness** can be verified.

III. Preliminaries

In this section, we give some preliminaries to be used by LVQ, including *Sorted Merkle Tree* (SMT) and *Bloom filter integrated Merkle Tree* (BMT).

A. Sorted Merkle Tree

As mentioned in Section II-A, the original MT is incapable of proving inexistence of an element. Fortunately, SMT makes a wonderful complement to it [15]. Compared to the original


3
7
1
5
Unsorted elements: [3, 7, 1, 5]
1
3
5
7
Sorted elements: [1, 3, 5, 7]

(a) MT (b) SMT

Fig. 1: Comparison between MT and SMT

MT, SMT firstly sorts all the elements, and then adds the elements to the tree. The smaller element is placed in the left position, while the larger is placed on the right. Fig. 1 makes a comparison between MT and SMT. To prove the inexistence of element 4, SMT can generate a proof consisted of two MBrs, as the red and blue paths in Fig. 1(b) shows. Since elements 3 and 5 can be proved to be contained in the tree and placed in adjacent positions, element 4 is able to be proved nonexistent.

B. Bloom filter integrated Merkle Tree

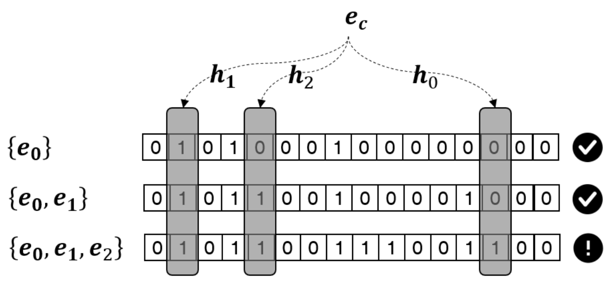
In this section, we introduce a composite data structure, namely BMT, which combines Bloom filter [8] with Merkle Tree together. Before getting into BMT, we take a brief review of Bloom filter and talk about its problem of *false positive matches* (FPM).

1) Bloom filter: Similar to SMT, *Bloom filter* (BF) is also used to prove the inexistence of an element. As a probabilistic data structure, BF takes up less space requirement while suffering from the FPM problem [17]. FPM problem refers to the situation where all the positions corresponding to a particular element are all of value 1, although it is not contained in the set. As a result, there are three cases when checking the existence of an element in BF:

- **Inexistent case**: All the positions corresponding to a particular element are set as 0. In this case, FPM works well and indicates a particular element is inexistent.
- **Existent case**: All the positions corresponding to a particular element are set as 1, and it is exactly contained in the set.
- **FPM case**: All the positions corresponding to a particular element are set as 1, but it is not contained in the set.

The first case can be considered as a **successful check**, while the latter two are **failed check**s. In other words, the BF is unable to distinguish if an element is exactly existent when the latter two cases take place.

In general, the more elements are contained in BF, the higher is the likelihood of **FPM case** [18]. To explain it clearly, we give an example as shown in Fig. 2. e_0, e_1, and e_2 are elements to be added to BF, while e_c is an element to be checked against the BF. Besides, h_0, h_1, and h_2 are three hash functions to map an element to a position in BF. When there is only one element (i.e., e_0) or two elements (i.e., e_0 and e_1) contained, e_c can be checked as being nonexistent



Elements	h_0	h_1	h_2
e_0	1	3	7
e_1	12	1	4
e_2	8	13	9
e_c	13	1	4

Fig. 2: More elements in BF leads to a higher likelihood of FPM

correctly (i.e., **inexistent case**). However, once a new element (i.e., e_2) is added in, all the positions corresponding to e_c are set as 1 and a **FPM case** takes place. From this example, we can find that a larger number of elements in the BF increase the possibility of **FPM case**.

2) BMT: Since a BF is built for one set, it is natural to merge multiple BFs into one, so as to check an element's existence across multiple sets. Combined with the MT's advantage of unforgeability, a novel tree-like data structure (BMT) is proposed. As shown in Fig. 3, each node in BMT is consisted of a hash value and a BF bit vector. BF in the parent node is a bitwise OR of two BFs in its child nodes. Hash value in the parent node is calculated based on the concatenation of the two hash values in the child nodes and the BF in the same node. Specifically, hash value in the leaf node is calculated based on the BF directly. Calculation methods of hash value and BF can be expressed by Equation 2 and Equation 3 respectively.

$$n_p.hash = \begin{cases} H(n_{c_0}.hash, n_{c_1}.hash, n_p.bf) & n_p.l > 0 \\ H(n_p.bf) & n_p.l = 0 \end{cases} \quad (2)$$

$$n_p.bf = \vee(n_{c_0}.bf, n_{c_1}.bf) \quad (3)$$

where n_{c_0} and n_{c_1} represent two child nodes of a parent node n_p, and $n_p.l$ denotes the layer height of a node. In particular, layer height is numbered from the bottom to the top, with the leaf's height as 0.

Since an upper-layer BF is the union of the lower-layer BFs, the upper-layer BF tends to contain more elements. In other words, when checking the existence of an element, **failed check** is more likely to take place in the upper-layer nodes. To deal with the **failed check** in the upper-layers, checks against the lower layers will be made. If there is a **failed check** again, much lower layers will be further checked, until no **failed check** takes place or the bottom layer is reached. The node stopping the check is named as an 'endpoint node'. According to the discussion in Section III-B1, the lower is the layer, the

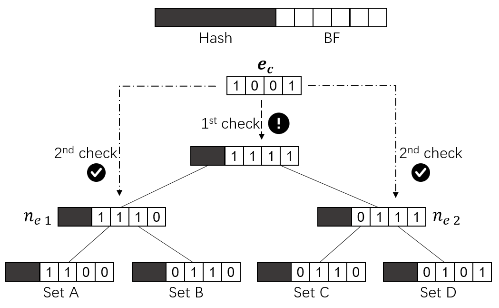
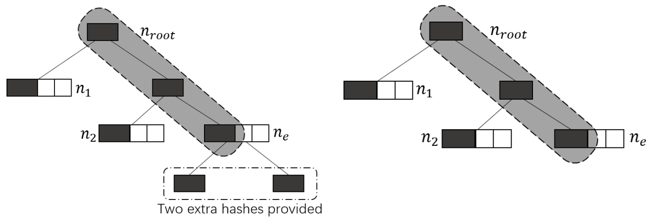


Fig. 3: Bloom filter integrated Merkle Tree



(a) Endpoint node is not a leaf (b) Endpoint node is a leaf

Fig. 4: Composition of a BMT branch

less likely is the FPM. Fig. 3 demonstrates an example of existence check in BMT, where e_c denotes the element to be checked. Each BF in the leaf nodes represents a set (i.e., A, B, C, and D), and BF in the root node represents the union of these four sets. At the first time, e_c is checked against the root node, but a **failed check** takes place. Next, sub-checks are made against two child nodes respectively. Since both of two sub-checks are successful and indicate the inexistence of e_c, we can conclude that e_c is not contained in each of the four sets. It is easy to find that only three BFs are checked, which is smaller than four when BFs in the leaves are checked one by one.

Inexistence proof is created for each endpoint node. To verify the inexistence of an element in multiple sets, the verifier (e.g., a light node) only needs to store the hash value in the root node. The proof provider (e.g., a full node) creates a BMT branch as the inexistence proof. Fig. 4 demonstrates the composition of a BMT branch. From root node to the endpoint node, hash values in the path are required to composite the BMT branch. Besides, both hash values and BFs in 1) nodes alongside the path (e.g., n_1 and n_2 in Fig. 4) and 2) endpoint node (e.g., n_e) are needed. If the endpoint node is not a leaf, two extra hashes in its child nodes must also be provided. As for the example in Fig. 3, Fig. 5 depicts the inexistence proof of e_c in sets C and D. With the BMT branch, a verifier will calculate the hash value in each node from the bottom layer to the top layer and compare the root hash with its holding one. Thanks to hash function's resistance to forgery, the verifier is able to check the validness of this BMT branch. With BF in node n_e, e_c can be proved to be nonexistent in sets C and D. It should be noted that a BMT branch is different from a MT branch in terms of structure. The latter must include a leaf,

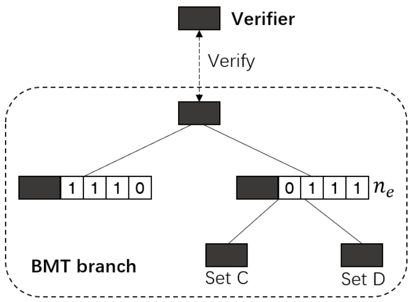


Fig. 5: Inexistence proof of e_c in sets C and D

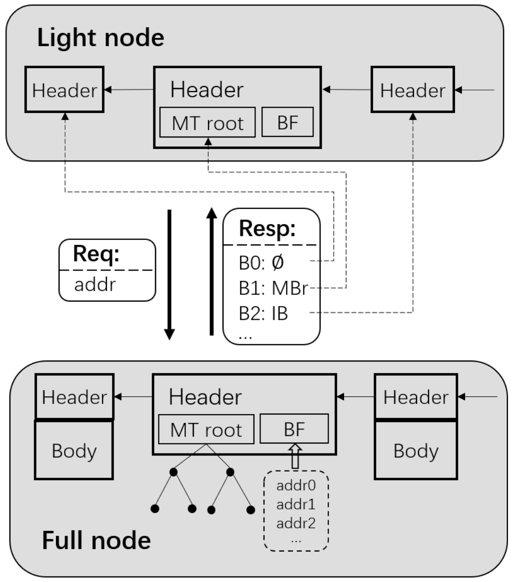


Fig. 6: Query for historical transactions in the strawman design

while the former may not.

IV. LVQ Design

In this section, we start with a strawman design which functionally enables the verifiable query for transaction history. However, it is far from practical use due to some challenges. By tackling these challenges, LVQ is able to implement the efficient verifiable query.

A. Strawman design

In the strawman design, a BF bit vector indicating all the addresses in a block is contained in the block header. As shown in Fig. 6, a light node firstly sends the *requested address* (RA) to the full node. The full node checks its local block bodies one by one, and constructs a response consisted of fragments for each block. A piece of fragment can be in three forms, namely empty (∅), *Merkle branch* (MBr), and *integral block* (IB). Each form corresponds to a case of the BF checking, as presented in Section III-B1. In particular, bit positions of RA are firstly figured out by hash functions. These bit positions are named as '*checked bit positions*' (CBP), which are checked against the BF in the header. If any CBP in the BF is set as 0, a **successful check** takes place and ∅ is returned as a piece

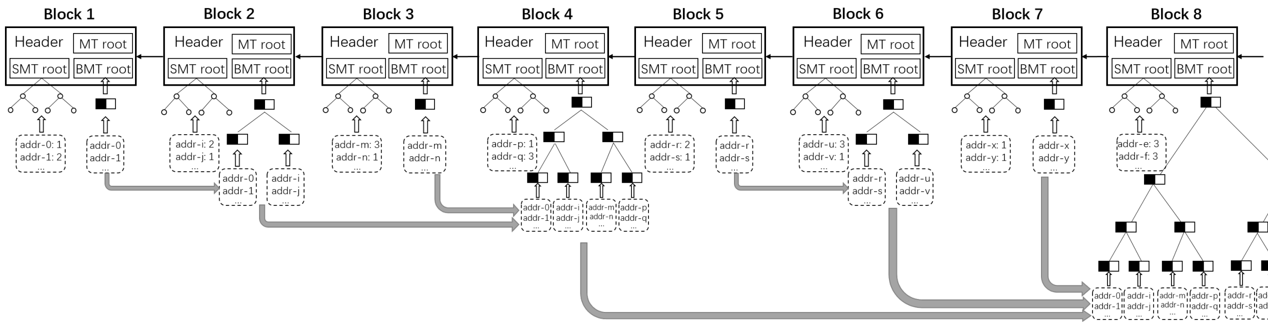


Fig. 7: Data structures in LVQ

of fragment. If all the CBPs in BF are set as 1, there may be either **existent case** or **FPM case**. The full node will further check which case happens based on the block body. If it is the former case, MBrs relevant to RA are returned. Otherwise, the IB is returned. A clear expression of three cases and fragment forms are shown in Equation 4.

$$frag(b_i) = \begin{cases} \varnothing & \exists p \in \mathbb{P} : \text{BF}_p = 0 \\ \text{MBr} & (\nexists p \in \mathbb{P} : \text{BF}_p = 0) \& (\text{RA} \in b_i) \\ \text{IB} & (\nexists p \in \mathbb{P} : \text{BF}_p = 0) \& (\text{RA} \notin b_i) \end{cases} \quad (4)$$

where b_i represents a block to be checked and $\mathbb{P}$ denotes the positions of RA calculated by hash functions.

Since a light node stores all the headers, it can make use of the local BFs, MT roots and query response to verify an address's historical transactions. To be specific, the light node firstly checks the BFs block by block. For the block with BF indicating nonexistent (**successful check**), the corresponding piece of fragment will be verified as valid, if and only if the fragment is $\varnothing$. For the block with BF indicating existent (**failed check**), if the fragment is in format of MBr, the MBr will be compared with the MT root. If the fragment is an IB, all the transactions in the block will be organized as a MT, which will be then compared with the local MT root. Only if all the fragments are verified as valid, can historical transactions returned by the full node be accepted as correct.

Although the strawman is able to work for verifiable query partially, it is far from practical use due to some significant challenges as follows.

1) Larger storage requirement in light nodes: To decrease the possibility of FPM, the size of BF bit vector usually needs to be set much larger than the number of included elements. For example, to make the FPM possibility be lower than 0.01, the ratio of BF size to element number must be larger than 10. Since there are thousands of unique addresses appeared in a block, the size of BF bit vector should be set as thousands of bytes, which is much larger than the size of original header (i.e., 80 bytes). Thus, a light node in the strawman incurs tens or even hundreds of times storage overhead compared with that in the original Bitcoin.

CHALLENGE 1: How to keep the light node from storing a large amount of data in block headers?

2) Large network overhead brought by IB fragments: As stated before, a FPM will lead the full block to be sent as a fragment, whose size may be about 1 MB. Since all the blocks are checked one by one, the larger is the number of blocks, more frequently the FPM takes place. There are already over 600,000 blocks in Bitcoin mainnet by December, 2019. Even the FPM possibility can be reduced to be about 0.001, the expected value of FPM is larger than 600, which makes the size of IBs larger than 600 MB. As a result, a large amount of network overhead may be brought by the IB fragments.

CHALLENGE 2: How to avoid the large network overhead brought by the IB fragments?

3) Incapability of indicating the appearance times: When RA actually appears in a block, the full node returns MBr rather than IB. The reason for it is that size of MBr is usually much smaller than IB. However, we deliberately ignore a problem that the light node cannot check if all the relevant MBrs in the block are included. In other words, both the MBrs and BF cannot indicate how many times RA appears in a block, which makes the completeness verification impossible.

CHALLENGE 3: How to create a proof to indicate the times of RA appeared in a block ?

B. Improvements by LVQ

To deal with the challenges above, multiple improvements are proposed by LVQ, which mainly include a BMT and a SMT.

1) Merge of Bloom Filter as BMT: As shown in Fig. 7, a BMT is created in each block, with the root hash included in the header. All the addresses appeared in a block will be firstly mapped as a BF. Instead of maintaining only one BF of the current block, the BMT also makes an attempt to merge the BFs of previous blocks, with each one as a leaf.

In general, a block with an odd height (e.g., *block 1, 3, 5, or 7* in Fig. 7) builds the BMT with only its own BF. In other words, hash of the BF is directly taken as the BMT root. By contrast, a block with an even height (e.g., *block 2, 4, 6, or 8* in Fig. 7) merges the previous BFs. The number of blocks to be merged is the maximum satisfying two conditions: 1) it is a power of 2, and 2) it is a divisor of the block height. Table I demonstrates several examples for the blocks to be merged. Since the root BF contains all the elements of the merged

TABLE I: Examples for blocks to be merged

Height	#Blocks	Blocks to be merged
1	1	1
2	2	1, 2
3	1	3
4	4	1, 2, 3, 4
5	1	5
6	2	5, 6
7	1	7
8	8	1, 2, 3, 4, 5, 6, 7, 8

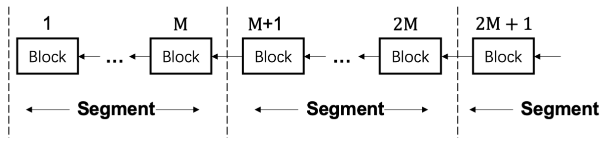


Fig. 8: Division of blocks into segments

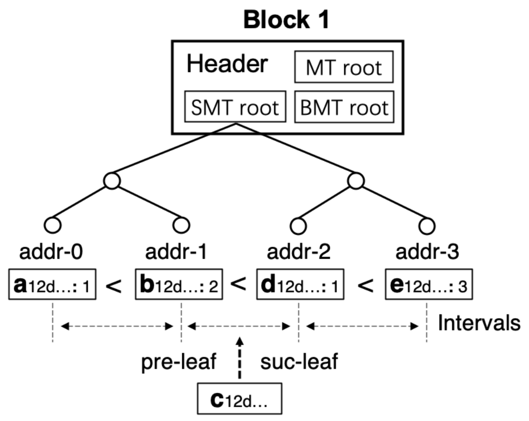


Fig. 9: Predecessor and successor leaf of an inexistent address

blocks, it is necessary to set an upper limit for the number of merged blocks, so as to limit the FPM possibility in root BF. As a result, we set the maximal number of merging blocks as $\mathbb{M}$, and divide all the blocks into several segments, as shown in Fig. 8. Particularly, $\mathbb{M}$ is a power of 2, and the last block in a segment will merge all the blocks in this segment.

For a better demonstration of the merging mechanism, a piece of pseudocode is detailed in Algorithm 1. This pseudocode takes the height of a block and the maximal number of merging blocks as inputs, and returns the block numbers to be merged with the current one. Since only the root hash of BMT is included in the header, whose size is about dozens of bytes, storage requirement in the light nodes can be kept small. Therefore, Challenge 1 can be solved well.

Algorithm 1 Figure out the blocks to be merged

```
Input:
    Height of a block, $h$
    Length of the segment, $\mathbb{M}$
Output:
    Set of blocks to be merged, $\mathbb{B}$
1: $l \equiv h \mod \mathbb{M}$
2: for $i = 0; 2^i < l; i++$ do
3:     $r \equiv l \mod 2^i$
4:     if $r \neq 0$ then
5:         $break$
6:     end if
7: end for
8: $i--$
9: for $j = 0; j < 2^i; j++$ do
10:    $\mathbb{B} = \mathbb{B} \cup \{h - j\}$
11: end for
```

2) Sorted addresses as SMT: Apart from the original MT and BMT, LVQ also builds a SMT for each block, with SMT root included in the header. As shown in Fig. 7, each leaf in the SMT is a combination of an address and its appearance times in the block. All the leaves are sorted in a lexicographical order. It should be noted that SMT only maintains the data contained in the current block, without merging with previous ones.

To deal with a FPM, two SMT branches containing the predecessor and successor leaves can be used to prove an address's inexistence. As shown in Fig. 9, the integral character space is divided into multiple intervals, and an inexistent address falls in a certain one. The lower and higher endpoints of this certain interval are referred to as predecessor and successor leaves respectively. Since only two branches rather than the integral block are returned, Challenge 2 can be tackled.

As depicted in Fig. 9, times of an address appeared in the block is recorded in the SMT, so long as it appears at least once. As a result, the appearance times of an existent address can be proved via a SMT branch, thus resolving Challenge 3.

V. Query in LVQ

A query process in LVQ is similar to that in the strawman. The biggest difference is related to the different formats of proof. Therefore, in this section, we focus on how to generate the proof in the full node and verify the proof in the light node.

As described in Fig. 8, the total blocks are divided into multiple segments. The length of all the segments except the last one is $\mathbb{M}$, which is named as 'complete segment' for short. By contrast, the length of the last segment may be smaller than $\mathbb{M}$. Consequently, we present the details about proof for the complete segment and the last segment respectively.

A. Proof for a complete segment

The proof for a complete segment consists of two aspects: existence proof and inexistence proof, as stated in Section II-B.

1) Existence proof: Existence proof is created block by block, if the *requested address* (RA) exactly appears in these blocks. An existence proof for a particular block usually involves two parts: 1) SMT branch indicating times of RA appeared in this block; 2) MT branches including transactions of RA whose appearance times is consistent with SMT.

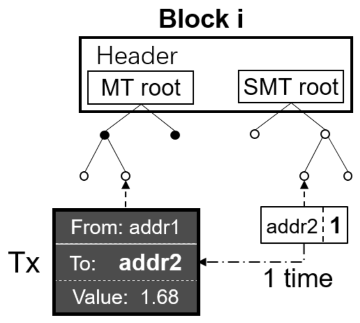


Fig. 10: Existence proof for transactions in a block

Fig. 10 depicts an example of existence proof for block i. On the one hand, the SMT branch shows that $addr2$ appears in this block for only once. On the other hand, the MT branch contains a transaction which transfers 1.68 coins from $addr1$ to $addr2$. The appearance time of $addr2$ in the MT branch is exactly 1, which is consistent with SMT branch. With this existence proof from the full node, the light node is able to validate both the correctness and completeness of query results for block i.

2) Inexistence proof: To build the *inexistence proof* (IEP) for a segment, the full node firstly finds all the endpoint nodes in the last block of this segment. A non-leaf endpoint node always indicates the inexistence of RA in its subtree. However, a leaf endpoint can be in three cases (i.e., **inexistent case**, **existent case**, and **FPM case**), as presented in Section III-B1. For the **existent case** of a leaf endpoint, how to create an existence proof has been described in Section V-A1. Besides, the non-leaf endpoint can be considered as a specialization of the **inexistent case**. Therefore, in this section, we describe how to create the IEP for the former two cases.

For each endpoint node in the **inexistent case**, a BMT branch can be created as an IEP of several blocks, as presented in Section IV-B1. Then, all the BMT branches will be merged for all the blocks in this segment. Fig. 11 demonstrates an example of inexistence proof in this case, when the length of a segment $\mathbb{M}$ is set as 8. Note that we assume all the endpoint nodes are in the **inexistent case**. Firstly, four endpoint nodes are found from the top to the bottom, namely n_{13}, n_{11}, n_7, and n_8. Then, for each endpoint, a BMT branch is created to prove the inexistence in several blocks. For instance, the BMT branch of n_{13} is able to prove the inexistence in blocks $\{b_1, b_2, b_3, b_4\}$. These branches usually share a lot of common data, whose merge can reduce the size of IEP largely. From the merged IEP, we can find that only 4 BFs need to be returned, which is significantly less than the direct return of 8 BFs in all the blocks.

For each endpoint node in the **FPM case**, two SMT branches are created as the EIP of a block, as introduced in Section IV-B2. Based on the BMT branches in the **inexistent case** and SMT branches in the **FPM case**, the light node is able to verify the completeness of query result from a full node.

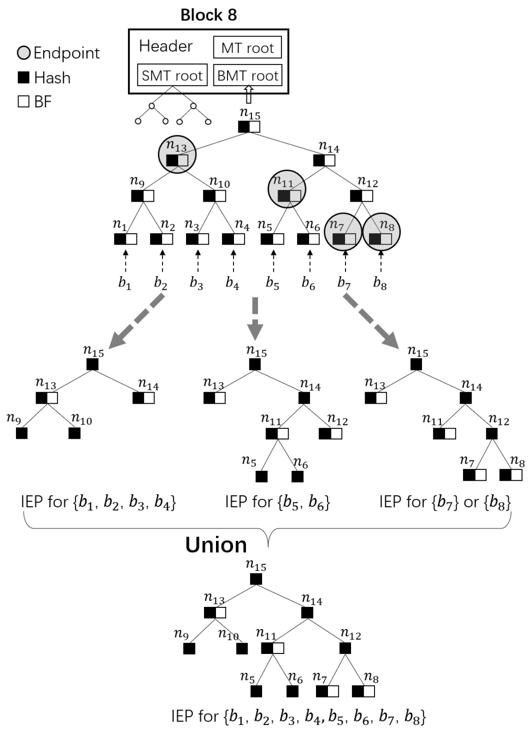


Fig. 11: Inexistence proof for blocks whose BFs indicate the inexistence correctly

TABLE II: Examples for segment division

h_t	Power series	Sub-segments
464	$2^7 + 2^6 + 2^4$	[257,384], [385,448], [449,464]
465	$2^7 + 2^6 + 2^4 + 2^0$	[257,384], [385,448], [449,464], [465]
466	$2^7 + 2^6 + 2^4 + 2^1$	[257,384], [385,448], [449,464], [465,466]

B. Proof for the last segment

Let the height of the latest block be h_t, and $\mathbb{M}$ is a k power of 2 (i.e., 2^k). Length of the last segment l can be calculated as:

$$l \equiv h_t \mod 2^k \tag{5}$$

If l equals 0, the last segment is also a complete segment, which can be dealt with as Section V-A. Otherwise, l can be further expressed in the form of power series:

$$l = \sum_{i=0}^{k-1} a_i 2^i, a_i \in \{0, 1\} \tag{6}$$

As a result, blocks in the last segment can be divided into multiple sub-segments according to Equation 5. Concretely speaking, from the high term to the low term in the power series, if a_i is not zero, a sub-segment of length 2^i is created. Assume that $\mathbb{M}$ is set as 256 and blocks are indexed from 1, examples of segment division for different h_t are shown as Table II.

Length of each sub-segment is a power of 2. According to Algorithm, the last block in each sub-segment exactly merges all the blocks in this sub-segment. For example, block of height

384 exactly merges the blocks from 257 to 384, while block of height 448 merges the blocks from 385 to 448. As a result, each sub-segment can be considered as a complete segment of smaller length, whose proof data can be created just like that in Section V-A.

VI. Security Analysis

In terms of verifiable query, security analysis is mainly performed about the **unforgeability** of query results. As stated in Section V, query results consist of data in three formats: MT branches, SMT branches, and BMT branches. The former two can be used to prove the existence of an address in blocks (correctness verification), while the latter two are able to prove the inexistence (completeness verification). Therefore, in this section, we conduct the unforgeability analysis on these three formats one by one.

First of all, a reasonable and widely accepted assumption is that cryptographic hash functions are secure, whose outputs cannot be forged maliciously. As long as the MT root hash has already been stored by the verifier (e.g., a light node), MT branches cannot be forged, as discussed and proved in [10] [19]. A SMT branch is similar to a MT branch, except the data being hashed in leaves. As a result, SMT branches can be considered to be unforgeable.

A BMT branch is somewhat different from a MT or SMT branch, as each node in BMT consists of a hash value and a bloom filter. Although a hash value is assumed to be unforgeable, a simple bloom filter suffers from being tampered with. To avoid the tampering, calculation of the hash value takes the bloom filter into consideration, as presented in Equation 2. As a result, a BMT branch is also protected from forgeability.

VII. Evaluation

This paper demonstrates a lightweight verifiable query approach LDV in Bitcoin, and in this section we evaluate the prototype systems enabled by the novel SMT and BMT data structure. We focus on the aspect of communication cost and study how various factors influence the performance. Since the communication cost in the query can be mainly reflected by the size of query results, we simply make a comparison of the latter.

A. Implementation

Our prototype systems are based on Btcd[1], a Bitcoin implementation written in Golang[2]. Since Btcd does not support the mode of light node natively, we simply enable it in our prototype by adding an application option, which asks the node to only store headers. The query process is simulated by the RPC (*Remote Procedure Call*) call, whose client and server are run on the light node and full node separately. Our experiments are conducted on two server machines. Each server is equipped with two eight-core Intel Xeon E5-2670 2.60GHz CPUs, 64GB memory, and 8TB disks, whose operating system is CentOS 7.2 distribution.

[1]https://github.com/btcsuite/btcd

[2]https://golang.org/

TABLE III: Number of transactions and blocks relevant to addresses

Index	**Address**	**#Tx**	**#Block**
1	1GuLyHTpL6U121Ewe5h31jP4HPC8s4mLT**s**	0	0
2	1GuLyHTpL6U121Ewe5h31jP4HPC8s4mLT**j**	1	1
3	1JtcMyyQWeTkrkuG22tfHhwXKKgoP9SaDv	10	5
4	1FFraSfgk5sw1jMs9FJR9mYAHZ6oMw26E5	60	44
5	1N6TUnk9YXD9wbkL37RwKk2wXKsaR776oh	324	289
6	1YzZXshuMVZ4Qh6WHvmqxos3vk4jQimdV	929	410

We take the data from the main network of Bitcoin to build up our new chains. For simplicity and without loss of generality, we only query the historical transactions in blocks of a small range, whose block heights are from 204,800 to 208,895, for a total of 4096. A query of larger range can be performed similarly. For the sake of brevity, the phrase 'all the blocks' in the following pages refers to these 4096 blocks. To evaluate our systems under different circumstances, we select 6 addresses whose numbers of relevant transactions and blocks differ largely as listed in Table III.

B. Benefits of LVQ over the strawman

As stated in Section IV-A, the strawman system requires the light nodes to store much larger size of data, which deviates the design merits of light nodes. Therefore, we propose a variable of the strawman as the baseline for query performance, whose light node only stores the hash of BF in the current block. This strawman variable keeps the storage requirement in light nodes being small, which is similar to LVQ. Since two improvements (i.e., BMT and SMT) taken by LVQ are orthogonal to one another, we strip them from LVQ separately, to evaluate the effects of a single improvement. In other words, the strawman variable can be considered as a special case of LVQ without neither SMT nor BMT. In the following parts of this section, we refer to this variable as strawman for short.

In the following, we make comparisons between four different prototype systems, namely strawman, LVQ without BMT, LVQ without SMT, and LVQ. Size of BF in the former two systems without BMT are set as 10KB, while BF and $\mathbb{M}$ in the latter two with BMT are set as 30KB and 4096 respectively. In other words, all the blocks will be merged in the BF of the last block. In addition, we set the number of hash functions in BF simply by default. Without SMT in the first and third systems, an integrated block is returned in the case where BF does not work.

Experimental results are demonstrated in Fig. 12. In general, LVQ performs better than the strawman for all the 6 addresses. Particularly, as for $Addr1$, size of query results returned in LVQ is only 0.57MB, much smaller than 41.12MB in the strawman. As for LVQ without BMT, its result size increases modestly even if the number of transactions and blocks grows largely. In other words, SMT plays an important role in keeping the result size small when RA is relevant with a large number of transactions. By contrast, LVQ without SMT performs well in the case of fewer transactions, while declines dramatically in the case of plentiful transactions. From this

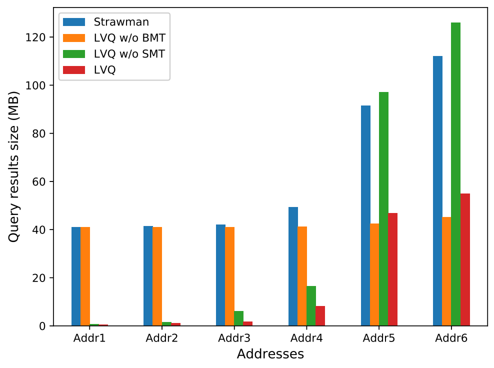


Fig. 12: Benefits of LVQ over the strawman

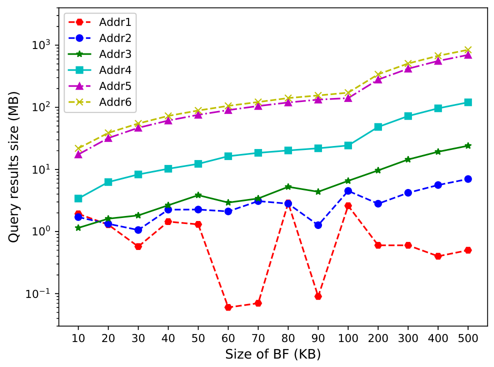


Fig. 13: Impact of BF size

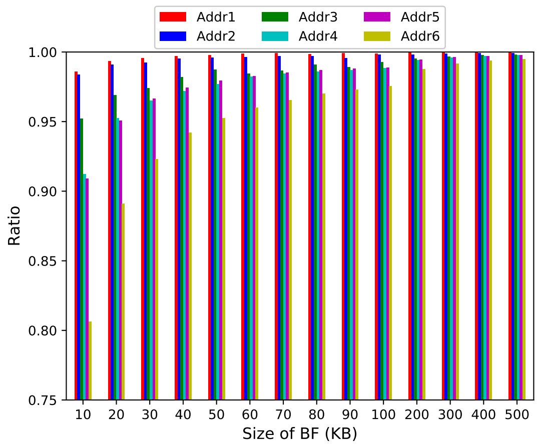


Fig. 14: Size ratio of the BMT branches to the total result

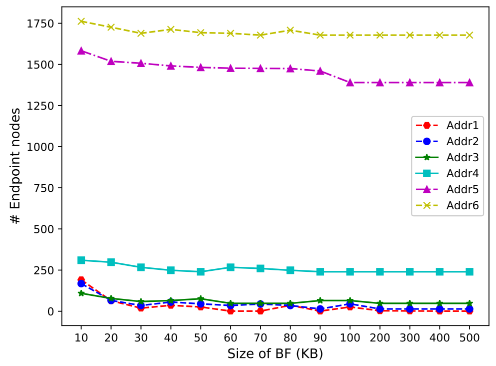


Fig. 15: Number of endpoint nodes for different size of BFs

observation, we can conclude that BMT helps reduce the results size when RA is relevant with a small quantity of transactions. By aggregating both SMT and BMT, LVQ is able to provide lightweight query performance in various cases.

It would be interesting to see that LVQ without BMT maintains a small advantage over LVQ for $Addr5$ and $Addr6$. The reason is that the sizes of BFs returned by LVQ are very large. To decrease the FPM possibilities in the upper-layer nodes of BMT, the size of BFs in LVQ is usually set larger (i.e., 30KB) than that of LVQ without BMT (i.e., 10KB).

C. *Impact of BF size*

BF is known to be a most fundamental structure for the validation of query results, and suffers from FPM problem. Hereby we study the impact of BF size (varying from 10KB to 500KB) on the overall size of query results, as illustrated in Fig. 13. On the one hand, it is easy to find that the result size of $Addr1$ fluctuates within a narrow range and the size of $Addr2$ has a limited growth. On the other hand, as for the other four addresses, there is a considerable increase in the query result size as the BF grows in size. In particular, result size of $Addr6$ increases by roughly 40 folds, from 21.86MB for 10KB BF to 843.22MB for 500KB BF. Taken both hands together, it is preferable to set the size of BF as a small value, such as 30KB in the above section.

Next, in order to analyze how BMT branches influence the overall query performance, we plot the weighted ratio over the total query results in a variety of BF sizes in Fig. 14. We conduct multiple experiments for various addresses. The figure demonstrates that BMT branches take up a very large proportion in the total query result. Noted that the minimal ratio value is over 80%, when size of BF is set as 10KB for $Addr6$.

In retrospect of Fig. 11, size of BMT branches is significantly related to the number of endpoint nodes and size of BFs in these nodes, as the size of a hash value is substantially smaller than that of a BF. As a result, we further conduct multiple experiments to look at the change in number of endpoint nodes as the BF size increases. Fig. 15 depicts the experimental results. From the figure, we can easily find that number of endpoint nodes keeps relatively stable as for each address. In other words, the size of BMT branches are mainly decided by the size of BFs. Therefore, a smaller value of BF size will be preferable to reduce query result size, which is consistent with the conclusion from Fig. 13.

D. *Effects of segment length*

In this section, we analyze the influence brought by different length of segments. We keep the size of BFs in BMT constant

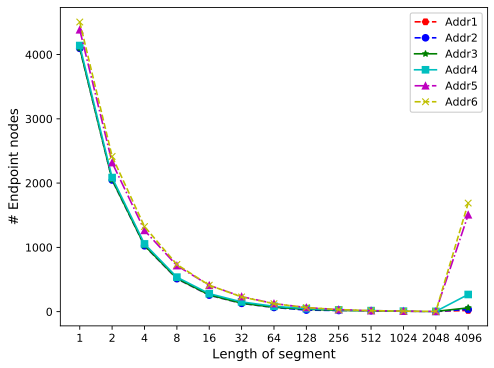


Fig. 16: Change in the number of endpoint nodes when the length of segment varies

as 30KB, while changing the length of segments from 1 to 4096. Review the conclusions in Section VII-C that BMT branches take up a very large proportion in the total query results, and BMT branches are mainly decided by endpoint nodes. Since the size of BFs are fixed, the number of endpoint nodes can reflect the size of total query results well. As a result, we analyze the relationship between the number of endpoint nodes and length of segments instead. The experimental results are shown in Fig. 16. From the figure, we can find that a too small or too large segment can lead to a large number of endpoint nodes, especially for the addresses with plentiful historical transactions. In particular, 1024 or 2048 will be a preferable value to set as the length of segment.

VIII. Related Work

There are already some studies trying to enable efficient or verifiable query in the blockchain systems, which can be broadly classified into two categories: off-chain query and on-chain query. A comparison between these studies and strawman/LVQ is shown in Table IV, which includes the perspectives of categories, methods, correctness validation, completeness validation, and lightweight.

A. Off-chain query

Methods in the off-chain category usually aim to improve the efficiency of query in the blockchain. One of the representatives is EtherQL, which develops an efficient query layer for Ethereum [20]. By reorganizing the data in a query layer, EtherQL enables efficient query for various kinds of queries. In addition to the local query, EtherQL further provides the query for other users with RESTful service. However, it is impossible for other users to verify these query results, which makes the query results unreliable.

The other two attempts are made by UStore [21] and Forkbase [22], which do some optimizations at the layer of storage engine. Both UStore and Forkbase provide query interfaces from the underlying storage layer, which accelerates the query efficiency largely. However, they also fail to support the verifiability on the query results. To sum up, all of these off-chain query methods focus on how to enable efficient and

TABLE IV: Comparison of verifiable query methods in the blockchain systems

Category	**Methods**	**Validation**		**Light-weight**
		Correctness	**Completeness**	
Off-chain	EtherQL	N	N	Y
	UStore	N	N	Y
	ForkBase	N	N	Y
On-chain	VQL	Y	Y	N
	vChain	Y	Y	N
	GEM^2-Tree	Y	Y	N
	LineageChain	Y	N	N
	Strawman	Y	Y	N
	LVQ	Y	Y	Y

lightweight query service, as shown in Table IV. Nevertheless, few of them take verifiability of query results into consideration.

B. On-chain query

In terms of the on-chain query, most of the methods are devoted to provide the verifiable query service. One example is VQL, which aims to provide verifiable data query services for blockchain systems [23]. By including a fingerprint of database in the middleware, VQL is able to ensure that the data in the database is not tampered with. However, it is hard to return a lightweight query result, since the fingerprint can only be used to verify the entire data in the database.

By including an accumulator-based authenticated data structure into the blockchain structure, vChain succeeds in providing batch verification for Boolean range queries [6]. However, size of the public key used by the accumulator is linear to the largest multiset size, which means the size may be very large and it is hard to transmit the key. What's worse, maintaining the SkipLists of large size leads to an expensive overhead for the service providers.

For the account-based blockchain system, Zhang et al. propose GEM^2-tree, a novel structure to ensure the verifiability of query result [7]. By adopting a two-level index, GEM^2-tree is capable of reducing the gas consumption while supporting authenticated range queries. However, the tree structures are maintained on the chain and updated on the fly, which leads to a heavy computational overhead.

To support the provenance query, LineageChain introduces a skip list index in the blockchain [24]. Making use of the block header to verify whether a block is valid, LineageChain ensures the correctness of query result. However, it fails to enable the completeness validation, since it only supports the version-based query for a special state ID.

In conclusion, although these methods are able to provide verifiable query more or less, almost all of them bring about considerable computing or storage overhead. By contrast, we propose LVQ in this paper, which reduces the query overhead and provides a lightweight verifiable query.

IX. Conclusion

Verifiable query for historical transactions of an address is important and challenging in Bitcoin systems. A strawman validation method undergoes problems of incremental storage overhead and significant network overhead. To address this issue, LVQ introduces two novel data structures, namely SMT and BMT, to provide lightweight validation. Extensive analysis and experiments performed on prototype systems show that LVQ works well in reducing both the storage and network overhead.

Acknowledgment

This work is supported by the Technology Innovation Project of Hubei Province of China under grant No. 2019AEA171, Jiang Xiao is the corresponding author of the paper.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf," 2008.

[2] M. Crosby, Nachiappan, P. Pattanayak, S. Verma, and V. Kalyanaraman, "Blockchain technology: Beyond bitcoin," *Applied Innovation*, vol. 2, no. 6-10, p. 71, 2016.

[3] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, "Untangling blockchain: A data processing view of blockchain systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 7, pp. 1366–1385, 2018.

[4] J. Zahnentferner and I. O. HK, "An abstract model of utxo-based cryptocurrencies with scripts." *IACR Cryptology ePrint Archive*, vol. 2018, p. 469, 2018.

[5] H. Pang and K.-L. Tan, "Authenticating query results in edge computing," in *Proceedings of the IEEE 20th International Conference on Data Engineering*. IEEE, 2004, pp. 560–571.

[6] C. Xu, C. Zhang, and J. Xu, "vchain: Enabling verifiable boolean range queries over blockchain databases," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 141–158.

[7] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain," in *Proceedings of the 35th International Conference on Data Engineering*. IEEE, 2019, pp. 842–853.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[9] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse merkle trees," in *Proceedings of the Nordic Conference on Secure IT Systems*. Springer, 2016, pp. 199–215.

[10] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proceedings of the Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[11] A. Biryukov, D. Khovratovich, and I. Pustogarov, "Deanonymisation of clients in bitcoin p2p network," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 15–29.

[12] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," in *Proceedings of the International Conference on Computer Communications*. IEEE, 2018, pp. 1484–1492.

[13] S. A. Crosby and D. S. Wallach, "Authenticated dictionaries: Real-world costs and trade-offs," *ACM Transactions on Information and System Security*, vol. 14, no. 2, p. 17, 2011.

[14] R. Tamassia, "Authenticated data structures," in *Proceedings of the European symposium on algorithms*. Springer, 2003, pp. 2–5.

[15] B. Laurie and E. Kasper, "Revocation transparency," *Google Research, September*, 2012.

[16] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of bloom filters," *Information Processing Letters*, vol. 108, no. 4, pp. 210–213, 2008.

[17] K. Christensen, A. Roginsky, and M. Jimeno, "A new analysis of the false positive rate of a bloom filter," *Information Processing Letters*, vol. 110, no. 21, pp. 944–949, 2010.

[18] M. Szydlo, "Merkle tree traversal in log space and time," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2004, pp. 541–554.

[19] Y. Li, K. Zheng, Y. Yan, Q. Liu, and X. Zhou, "Etherql: a query layer for blockchain system," in *Proceedings of the International Conference on Database Systems for Advanced Applications*. Springer, 2017, pp. 556–567.

[20] A. Dinh, J. Wang, S. Wang, G. Chen, W.-N. Chin, Q. Lin, B. C. Ooi, P. Ruan, K.-L. Tan, Z. Xie, H. Zhang, and M. Zhang, "Ustore: a distributed storage with rich semantics," *arXiv preprint arXiv:1702.02799*, 2017.

[21] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan, "Forkbase: An efficient storage engine for blockchain and forkable applications," in *Proceedings of the International Conference on Very Large Data Bases*, vol. 11, no. 10. VLDB Endowment, 2018, pp. 1137–1150.

[22] Z. Peng, H. Wu, B. Xiao, and S. Guo, "Vql: Providing query efficiency and data authenticity in blockchain systems," in *Proceedings of the IEEE 35th International Conference on Data Engineering Workshops*. IEEE, 2019, pp. 1–6.

[23] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang, "Fine-grained, secure and efficient data provenance on blockchain systems," in *Proceedings of the International Conference on Very Large Data Bases*, vol. 12, no. 9. VLDB Endowment, 2019, pp. 975–988.