An Efficient Query Scheme for Privacy-Preserving Lightweight Bitcoin Client with Intel SGX

Yukun Niu*, Chi Zhang*, Lingbo Wei*, Yankai Xie*, Xia Zhang†, and Yuguang Fang†
*School of Information Science and Technology
University of Science and Technology of China, Hefei, Anhui 230027, P. R. China
†Department of Electrical and Computer Engineering
University of Florida, Gainesville, Florida 32611, USA

***Abstract*—In Bitcoin, lightweight clients outsource most of the storage and computation tasks to full nodes in order to run on resource-limited devices. In the interaction with the full node, the lightweight client leaks considerable information about which address or transaction is relevant to it. The existing schemes to solve this problem do not support efficient yet privacy-preserving transaction search and verification due to the fact that the blockchain is inherently inefficient for transaction query and proposed schemes perform transaction search in a block-by-block manner. Therefore, we propose an efficient transaction query scheme for the privacy-preserving lightweight client with the Intel SGX enclave running on the full node. Our main idea is to leverage the secure enclave to serve transaction-query requests from lightweight clients. However, the usage of secure enclave alone does not achieve our goals. Our scheme reorganizes the blockchain and leverages prefix hash tree to increase transaction-search efficiency. Due to limited capacity, the enclave stores reorganized blockchain data in the untrusted full node. Thus, our scheme integrates prefix hash tree and oblivious searching technologies to simultaneously support efficient transaction search and protect access pattern of externally stored blockchain data for the secure enclave. Security analysis and performance evaluation show that our scheme provides efficient transaction search and verification functionalities for lightweight Bitcoin clients in a privacy-preserving way.**

***Index Terms*—Lightweight Bitcoin client, transaction query, privacy protection, Intel SGX**

I. INTRODUCTION

In Bitcoin, the simplified payment verification (SPV) method allows a lightweight client to outsource most of the transaction-storage and transaction-verification tasks to full nodes in order to run on resource-limited devices, such as smart phones. Currently, a typical Bitcoin client requires more than 200 GB of disk space to store a complete copy of the blockchain. However, it is infeasible to store the entire blockchain on resource-limited devices. To remedy that, the simplified payment verification method, a method for searching and verifying whether a particular transaction is included in a block without downloading the entire block, is proposed by Nakamoto in [1] to support lightweight clients. The lightweight client only downloads and verifies the block headers instead of the blocks, each of which contains a block header and a set of transactions. Each block header is only 80 bytes, so it requires less than 50 MB to store all block headers. In a block, all transactions are hashed into a Merkle tree, and the Merkle tree's root node, called Merkle root, is included in the block header. Given a transaction in a Merkle tree and the Merkle root, the lightweight client can verify whether the transaction is included in the block containing the Merkle root with the help of the Merkle branch which binds the transaction to the block header. To do so, the lightweight client should first download the Merkle branch, also called SPV proof, from the full node which maintains the complete blockchain.

However, a lightweight client leaks considerable information about which address or transaction is relevant to it when retrieving related transactions and their SPV proofs from the full node. The full node can know which Bitcoin addresses in the transactions belong to the lightweight client. Moreover, the full node can link these Bitcoin addresses to the lightweight client's real identity, such as IP address or social account. What was worse, the full node can even deduce a lightweight client's purchase propensity with the help of the public merchant addresses [2].

Some Bitcoin developers proposed to utilize Bloom filter [3] to make a tradeoff between the privacy and the communication overhead of the lightweight client [4]. Bloom filter is a space-efficient probabilistic data structure. In this method, the lightweight client submits a Bloom filter, into which the lightweight client can embed its Bitcoin addresses as keywords, to full nodes. The false positives of the Bloom filter make that the transaction through the Bloom filter is possible not relevant to the lightweight client. In this way, the lightweight client defines an anonymity set to hide its real transactions from the full node. However, Gervais et al. [5] demonstrated that this method offers almost no privacy in practice. To solve the privacy problem, Kanemura et al. [6] proposed a γ-deniability enabled Bloom filter to make sure the transactions are really hidden by the false positives. Osuntokun et al. [7] proposed the full node broadcasts a filter, which is inserted by all transaction keywords (such as Bitcoin addresses) in a block, to the lightweight client. The lightweight client can check whether the transactions that it is interested in are contained in the block, and then downloads the block if the filter matches any transaction keyword.

This work was supported by the National Key Research and Development Program of China under Grant 2017YFB0802202, and by the Natural Science Foundation of China (NSFC) under Grants 61702474 and 61871362.

However, all the filter-based methods cannot simultaneously provide strong privacy protection and low communication overhead for the lightweight client.

Intel's Software Guard Extensions (SGX) technology makes it possible to provide strong privacy protection for lightweight clients with low communication overhead. Intel SGX's enclave (a trusted execution environment) can protect sensitive data and code from attackers even through the attackers control the operating system and the hypervisor of the host machine [8]. Thus, an enclave running on the full node can work as a proxy to perform transaction search and protect search results from the full node. However, the usage of the enclave alone still has privacy leakage problem. This is due to the fact that the enclave has limited capacity (128M [9]). Thus, the blockchain will be stored in the full node. In this case, the access pattern to the blockchain also threats the transaction query privacy of the lightweight client. Matetic et al. [10] have already proposed the usage of the enclave to protect the lightweight client's privacy. They proposed that the enclave performs transaction search through scanning the blockchain in a block-by-block manner to hide the access pattern. However, this means that the total I/O cost in [10] is linear with the size of the scanned blocks when performing transaction search in the enclave.

In this work, we propose an efficient query scheme for lightweight clients to hide the search patterns from the full node with Intel SGX. To achieve this goal, our proposed scheme reorganizes the blockchain into a new transaction database. In the transaction database, a transaction file contains a transaction and its SPV proof, so the lightweight client can still utilize the SPV method to verify whether a transaction is included in the blockchain. Index structures based prefix hash tree are used to support efficient transaction search on the transaction database. We built an index structure for txid-based transaction search and for Bitcoin-address based transaction search, respectively. The txid is the hash of a transaction, and it can be seen as the identity of the transaction.

To protect access pattern of the enclave without scanning all indexes into the enclave, we combine the *oblivious data structure (ODS)* [11] technology with the prefix hash tree. In this way, we provide a sublinear transaction search in an oblivious way. Our query scheme decreases the total I/O cost for the enclave into $O(mlogN)$, where N is the total number of tree nodes in our oblivious index-data structures and m is the height of the prefix hash tree. Similarly, Path ORAM [12], the building block of ODS, is used to load queried transaction data into the enclave in an oblivious way.

Our main contributions are summarized below.

- We devise appropriate data structures to reorganize the blockchain. The transaction index structure is organized by a prefix hash tree, which allows the enclave to perform a sublinear transaction search.
- We combine the *oblivious data structure* technology with the prefix hash tree to support an efficient yet privacy-preserving transaction search without scanning all the transaction indexes.
- We demonstrate that our scheme provides an efficient yet privacy-preserving transaction query service for lightweight clients.

The remainder of this paper is organized as follows. In Section II, we give a short description of the Path ORAM and the oblivious data structures. Section III presents the system model, threat model, and design goals. Proposed scheme is introduced in Section IV. Security analysis and performance analysis are given in Section V and Section VI, respectively. Finally, we conclude this paper in Section VII.

II. PRELIMINARIES

In this section, we introduce the cryptographic building blocks used in our scheme: Path ORAM [12] and oblivious data structures [11].

A. Path ORAM

Path ORAM [12] is an efficient ORAM protocol, which protects access pattern to external (server) memory for a client, with low bandwidth and latency. Data in the external (server) memory is organized as a binary tree with N buckets, each of which contains Z chunks. The client maintains a *position map*, which stores a path from the root to a leaf for every chunk. When reading a chunk, the client retrieves all chunks on the path. Then, the requested chunk is remapped to another leaf, and all the data on the path will be re-encrypted and written back to the external (server) memory.

B. Oblivious Data Structures

Oblivious data structures (ODS) [11] is framework for building a data structure where memory access pattern of it is protected. In general, the oblivious data structure can be expressed as a tree of bounded degree. One way to build an oblivious data structure is to run a classic data structure on Path ORAM. To do so, the client can initialize an classic data structure such as a binary tree containing some nodes and pointers. Then, the client converts the nodes of the binary tree into the ODS nodes through replacing their pointers with ODS pointers. An ODS pointer is the *position map* of a child node in the binary tree. To access a particular node, the client follows ODS pointers from the root of the ODS to the node. The access to every ODS node utilizes the Path ORAM, so the client will update position maps of the accessed ODS nodes to finish the access.

III. PROBLEM STATEMENT

A. System Model

The transaction query scenario in our scheme involves four parties as shown in Fig. 1: a full node, an enclave running on the full node, a lightweight client, and the Bitcoin peer-to-peer network. Both the full node and the lightweight client connect to the Bitcoin peer-to-peer network to synchronize the blockchain. Note that the lightweight client only synchronizes the block headers. The lightweight client can communicate with the enclave through a secure channel which is established through remote attestation.

The lightweight client performs transaction query to verify a payment or track its balance. To do so, the lightweight client sends the payment transaction's hash or the lightweight client's Bitcoin addresses as keywords to the enclave through the secure channel. To serve the lightweight client, the enclave converts the blockchain stored in the full node into new data structures to support efficient and privacy-preserving transaction search. The new data structures are encrypted and stored in the full node by sealing operation of the enclave. To respond to the lightweight client, the enclave performs transaction search on the new data structures using the keywords sent by the lightweight client. If there is any matching, the enclave sends transactions and the corresponding SPV proofs to the lightweight client through the secure channel.

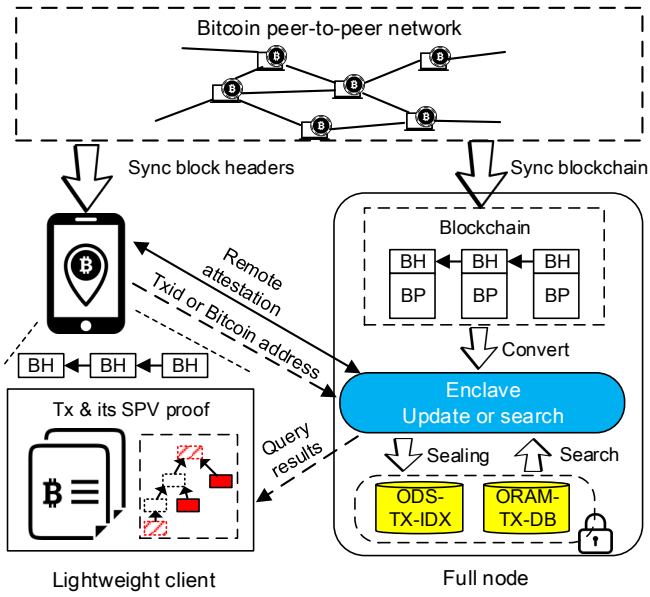

Fig. 1: Transaction query scenario and overview of proposed scheme

B. Threat Model

We consider the full node is curious about the lightweight client's privacy. Due to the fact that the full node controls the operating system and the hypervisor, it can alter the data sealed by the enclave and monitor which data is read by the enclave. Moreover, The full node can launch traffic analysis to the enclave.

As many literatures with the Intel SGX, we assume the full node cannot access keys (e.g. attestation and sealing key) of the enclaves, and it cannot access the enclave's memory which is encrypted and protected by the CPU.

Finally, we assume the cryptographic primitives (such as encryption or signatures) used in the enclave are secure.

C. Design Goals

In light of the above adversary model, the following security requirements are essential to ensure that the enclave serves the lightweight client in an efficient yet privacy-preserving way:

1) Security: The lightweight client cannot be easily fooled into thinking a transaction is in a block when it is not.
2) Privacy: The full node cannot know which transaction is queried by the lightweight client and what the transaction keywords in the transaction query request are.
3) Completeness: The query result should contain all valid transactions relevant to the keywords in the transaction query request.
4) Efficiency: On the one hand, transaction query should have low communication and computation overhead for the lightweight client. On the other hand, transaction query should have low computation overhead for the enclave and the full node.

IV. SYSTEM DESIGN

A. Blockchain Reorganization

In this work, the main objective is to perform an efficient transaction search in an oblivious way. First, the enclave reorganizes the blockchain to support efficient transaction search. In this paper, the lightweight client wants the enclave to respond with the queried transactions and the corresponding SPV proofs. The SPV proofs are used to prove that the transactions was included into the blockchain. Therefore, the enclave stores a transaction and its SPV proof into a transaction file.

Then, the enclave extracts transaction keywords from the transaction files. In this paper, we only consider two types of the transaction keywords: hash of the transaction (also called TxID), and public key hash (denoted by PKH). Note that, the other transaction-keyword types (such as public key, address, or outpoint) can be converted into these two transaction-keyword types.

The efficient transaction search is ensured by using a prefix hash tree to build the transaction index structure. Although both the TxID and the PKH are hash-keywords, they have different length since they are obtained through different hash functions. The TxID is 32 bytes and the PKH is 20 bytes. Thus, we build two prefix hash trees for them. The two prefix hash trees have the similar structure and we just introduce the transaction index structure for TxID in the following. The structure of a prefix hash tree is shown in Fig. 2. Suppose a transaction keyword has n hexadecimal nibbles. That is $Tx\_keyword = \{s_1, s_2, \cdots, s_n\}$. If the transaction keyword is a TxID, $n = 64$. There are two kinds of nodes in the prefix hash tree defined as follow:

- *Branch nodes*: They have seventeen items. The first item stores a single nibble of the transaction keyword. Assume the branch node is in the level i, the first item is s_i. We define that the root node is in the level 0, and its first item is $NULL$ no matter whether the root node is a branch node or a leaf node. The following sixteen items are child nodes. Each of the items contains the hash of the child node and a pointer to the child node.
- *Leaf nodes*: They have at most $B+1$ items. The first item stores s_i if the leaf node is in the level i, and the following items store at most B $(keyword, pointer, hash)$

tuples in which $hash$ is the hash of the chunk containing the transaction file whose keyword is $keyword$. Note that the height of the prefix hash tree is relevant to B.

To hide the access pattern of transaction files, the enclave writes them into the ORAM-TX-DB data structure. Transaction files whose TxID in the same leaf node are written into the same chunk. If too many transaction files are added, the chunk can be split into two and the pointers in the leaf node of the prefix hash tree are updated. The ORAM-TX-DB supports Path ORAM operations on it, so the pointer in the leaf node of the prefix hash tree is a position map. That is, the leaf node stores at most B $(keyword, pos, hash)$ tuples. The position map consists of a chunk identifier and a leaf of the ORAM-TX-DB. Since the ORAM-TX-DB is stored in the full node, the integrity for every access to the ORAM-TX-DB should be protected. We utilize the $hash$ in the above tuple to protect the integrity of the access to the ORAM-TX-DB.

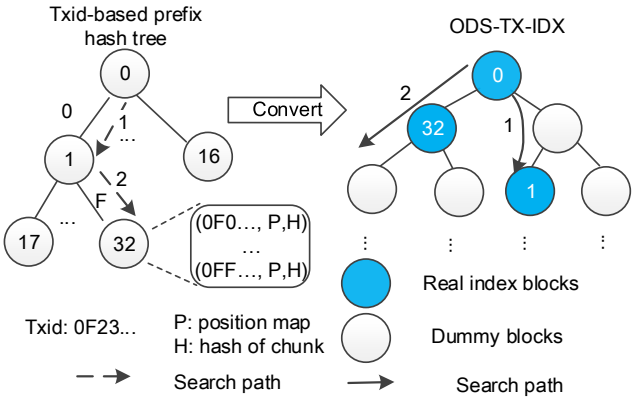

Fig. 2: Illustration of conversion between the prefix hash tree and the ODS-TX-IDX

To hide the search pattern on the index tree without accessing all the tree nodes, the enclave converts the prefix hash tree into the ODS-TX-IDX data structure, as shown in Fig. 2. The enclave randomly places the nodes of the prefix hash tree into a new binary tree, and then replaces the pointers of the prefix-hash-tree nodes with ODS pointers. For a branch node, an ODS pointer is the position map of a child node in the prefix hash tree. With the help of the position map, the enclave can access the child node on the ODS-TX-IDX through a Path-ORAM-read operation.

The blockchain reorganization is performed when the enclave is initialized by the full node or a new Bitcoin block is generated by the Bitcoin network. In the initialization, the enclave scans the whole blockchain from the genesis block to convert the blockchain into the ORAM-TX-DB and the ODS-TX-IDX. The enclave performs sealing operation to encrypt and store the two data structures into the full node, and it only stores the root nodes of them. When there is a new Bitcoin block generated, the enclave verifies the validity of the Bitcoin block and then updates the transaction information into the ORAM-TX-DB and the ODS-TX-IDX data structures. The root nodes of the two data structures are also updated and stored in the enclave. Note that, the complexity of updating a transaction is same as that of performing a transaction search. Thus, updating a new block for the enclave is equate to performing l transaction search, where l is the number of the transactions in the block.

B. Transaction Query Request

When querying a transaction, the lightweight client first performs a remote attestation protocol, provided by Intel SGX, with the enclave to make sure that the enclave is correctly instantiated. After the remote attestation, a secure channel is established between the lightweight client and the enclave. Then, the lightweight client sends a query request message, containing some txids or Bitcoin addresses, through the secure channel.

The size of the query request should be fixed to hide the number of the transaction keywords. Moreover, encryption using in the secure channel is semantically secure. If the size of the transaction keywords is bigger than the size of the query request message, the lightweight client can query several full nodes to hidden its query pattern.

C. Transaction Search

Our scheme provides a sublinear transaction search. That is, we do not need to load all indexes into the enclave to perform a transaction search. We achieve this goal through combining the prefix hash tree and ODS technology. On the one hand, transaction search on the prefix hash tree reads m nodes ($m \leq n$), where m is the height of the prefix hash tree. On the other hand, reading a node in the ODS through a Path ORAM operation which is a sublinear operation. The total I/O cost is $O(mlogN)$, where N is the total number of nodes in the ODS-TX-IDX data structure.

Once receiving a transaction query request, the enclave performs transaction-search operation as shown in Algorithm 1. First, the enclave distinguishes the type of the keyword(s). We consider the transaction search uses the TxID or PKH as the keyword. The other types of the keyword (such as outpoint, or address) can be converted to TxID or PKH. Then, the enclave decides to search on which ODS-TX-IDX based on the type of the keyword, and converts the keyword to a set of nibbles $\{s_1, s_2, \cdots, s_n\}$.

Then, the enclave searches transactions from the root node of the ODS-TX-IDX to a leaf node. A single nibble of the transaction keyword is used to determine which child node will be read, and then the next single nibble is used in the child node. Every node stores the position map of its child nodes. The enclave utilizes the position map to read the child node via a Path-ORAM-read operation. The enclave performs the same operation in the child node until a leaf node is found. After that, the enclave verifies whether there is a $keyword$ in the $(keyword, pos, hash)$ the same as the transaction keyword. That is, queried transaction is in the blockchain or not. If so, the enclave fetches the transaction file through a Path-ORAM-read operation on ORAM-TX-DB

Algorithm 1: Transaction search

Input: The transaction keyword, tx_key; the root nodes of the ODS-TX-IDX;
Output: Position map P and the hash of a chunk H;

```
Determine which root node is used based on the type of tx_key;
Convert tx_key to the set of hexadecimal nibbles {s_1, s_2, ..., s_n};
Push {s_1, s_2, ..., s_n} into a stack S;
while S ≠ NULL do
    if node is a branch node then
        pop s_1 from S;
        access childnode_{s_1} by Path ORAM;
        verify the integrity of the child node;
        reorder S;
    else
        if tx_key is found in a (keyword, pos, hash) then
            P = pos; H = hash; break;
        else
            P = NULL; H = NULL; break;
return P and H;
```

with the pos. Note that, the enclave verifies the integrity of the chunk with the $hash$ in the $(keyword, pos, hash)$. If the position map is NULL, the enclave performs a dummy Path-ORAM-read operation to hide the search pattern. Finally, the enclave updates position map information in the ODS-TX-IDX from the leaf node to the root node.

D. Transaction Query Response

After transaction search, the enclave returns the transaction search results and its local newest block header to the lightweight client. The straightforward way is to send matched transaction files and the block header or a "no matching found" message to the lightweight client. However, the full node could learn whether the queried transaction is written into the blockchain. Moreover, transaction size ranges from more than 100 bytes to 100K bytes. The transaction file size also leaks the search pattern.

In our scheme, the enclave responds to the lightweight client with a fixed-size response message. There is a trade-off between the communication overhead and the search pattern. To mitigate this tradeoff, we obtain the distribution of the transaction size via analyzing the blockchain and choose an appropriate size of the response message. The details are described in Section VI.

V. SECURITY ANALYSIS

Here, we analyze the security of our scheme to verify that the security requirements mentioned in Section III-C are satisfied. Note that, the efficiency requirement will be analyzed in Section VI.

A. Security

The lightweight client cannot be easily fooled into thinking a transaction is in a block when it is not. This is due to the fact that the enclave responds to the lightweight client with transactions and their SPV proofs which cannot be forged. The SPV method allows the lightweight client to make sure that a transaction is in the blockchain if its SPV proof is valid.

B. Privacy

Our scheme protects the lightweight client's privacy from the full node. First, the communication between the lightweight client and the enclave utilizes a secure channel, which means that data transmitted through the channel is encrypted by a session key. Moreover, the communication pattern is protected via fixed-size messages. Second, transaction search is performed in the enclave which protects the confidentiality of data and code in the enclave from the full node. Finally, we use ODS and Path ORAM technologies to protect the access pattern of the enclave when performing transaction search and transaction fetch.

C. Completeness

In our scheme, the completeness is ensured by the verifications of the lightweight client and the enclave. On the one hand, the lightweight client verifies its local block headers and the SPV proofs of the queried transactions. In addition, the lightweight client compares the newest block header in the response message with its local newest block header to ensure completeness of the transaction query. On the other hand, the integrity of transaction search in the enclave is protected through the hash pointer in the prefix hash tree and the page integrity protection mechanism of SGX. Table I compares the security functionality of our scheme with the existing schemes.

TABLE I: Security functionality comparison between our scheme and the existing schemes

	Security	Privacy	Completeness	Efficiency
[4]–[7]	Yes	Weak	No	No
[10]	Yes	Strong	Yes	No
Our scheme	Yes	Strong	Yes	Yes

VI. PERFORMANCE EVALUATION

In this section, we analyze the lightweight client's communication overhead and evaluate the computation overhead of the enclave and the full node.

Note that, scheme in [10] has the best privacy protection and lowest communication overhead for lightweight client. Thus, we compare performance of our scheme with it.

A. Communication overhead of the lightweight client

In our scheme, the lightweight client's communication overhead is relevant to the size of the response message. In order to choose an appropriate size of the response message, we count the distribution of transaction size in the nearest 10,000 blocks[1], as shown in Table II. We can see that more

[1]Block height of the blocks is from 560,000 to 570,000.

TABLE II: Distribution of transaction size in 10,000 blocks

Transaction size less than (byte)	512	1,024	1,536	2,048	2,560	3,072	102,400
The number of the transactions	15,744,682	19,247,789	19,712,715	19,885,362	19,950,945	19,998593	20,258,611
Proportion of the transactions	77.71%	95.01%	97.31%	98.16%	98.48%	98.72%	100%

than 95.01% and 98.72% transactions are less than 1,024 and 3,072 bytes, respectively. In addition, the average size of a SPV proof is 473 bytes[2]. The block header is 80 bytes. Thus, more than 95.01% and 98.72% transaction queries can be responded with a 1,577 and 3,625 bytes message, respectively. In [10], the queried transaction is hid in a set of block headers when synchronizing the block headers. In this way, the communication overhead for querying a transaction can be seen as the transaction's size and the size of its SPV proof. However, the communication overhead should contain the transmitted block headers if the lightweight client has a synchronized block headers. In this case, the size of 144, which is the average number of generated blocks in one day, block headers is already 11,520 bytes.

B. Computation overhead of the enclave

We evaluate the efficiency of our scheme by benchmarking the operations of the Path ORAM leveraging a existing implementation of the ZeroTrace [13]. We implemented and evaluated the performance of Path ORAM on a Dell Optiflex 7050, with a 4 core Intel i7-7700 @ 3.60 GHz CPU, 8 GB RAM, and 512 GB HDD.

In the worst case, the enclave performs m, where m is the height of the prefix hash tree, Path ORAM operations on the ODS-TX-IDX and one Path ORAM operation on the ORAM-TX-DB. Note that, the size of ORAM-TX-DB is larger than that of ODS-TX-IDX. Suppose the chunk size is 256 bytes, the latency of one Path ORAM operation roughly logarithmic increases with the number of chunk. We assume the number of chunk is 10^9, which can contain 256 GB data while the whole blockchain is about 200 GB, for ORAM-TX-DB. In this case, the latency of one Path ORAM operation is about $1.63ms$. Moreover, $m = 9$ is enough for current size of the transaction keywords. Thus, latency of one transaction-search operation is about $1.63*(m+1) = 16.3ms$. While in [10], the response time is linear with the number of scanned blocks. It is about $1s$ to scan 50 blocks and more than $5s$ to scan 300 blocks. Obviously, our scheme is more efficient than scheme in [10].

As described in Section IV-B, updating a new Bitcoin block for the enclave is equate to performing l transaction search, where l is the number of the transactions in the block. The average number of transactions per block is 2,719[3]. Thus, the enclave needs $2,719 * 16.3ms = 44.32s$ to update a Bitcoin block which is generated at the average $10min$.

[2]This result is calculated based on the average size of the block and the average number of transactions in a block. These data are retrieved from https://www.blockchain.com/en/charts at September 19th, 2018.

[3]The data is retrieved from https://www.blockchain.com/en/charts/n-transactions-per-block at April 14th, 2019

VII. CONCLUSION

Transaction query in a privacy-preserving way is very important for the lightweight client in Bitcoin. However, existing schemes do not support efficient yet privacy-preserving transaction search. In this paper, we have proposed an efficient transaction query scheme for the lightweight client in a privacy-preserving way with Intel SGX. The index organized by a prefix hash tree allows the lightweight client to perform transaction search in an efficient way. ODS and Path ORAM technology are used to protect the search pattern. We have shown that our scheme simultaneously provides efficient transaction query and strong privacy protection for the lightweight client. Due to the space limited, we only present transaction search based on the txid. In our future work, we plan to support PKH-based transaction search and UTXO search, and evaluate their performance.

REFERENCES

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

[2] J. D. Nick, "Data-driven de-anonymization in Bitcoin," Master thesis, ETH-zürich, 2015.

[3] 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.

[4] M. Hearn and M. Corallo, "Connection Bloom filtering," 2012. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki

[5] A. Gervaisy, G. O. Karamez, D. Gruber, and S. Capkun, "On the privacy provisions of Bloom filters in lightweight Bitcoin clients," in *Proc. of the 30th Annual Computer Security Applications Conference*, New Orleans, Louisiana, USA, December 2014.

[6] K. Kanemura, K. Toyoda, and T. Ohtsuki, "Design of privacy-preserving mobile Bitcoin client based on γ-deniability enabled Bloom filter," in *Proc. of IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, Montreal, QC, Canada, October 2017.

[7] O. Osuntokun, A. Akselrod, and J. Posen, "Client side block filtering," 2017. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki

[8] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *Proc. of IEEE S&P*, San Francisco, CA, USA, May 2018.

[9] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "HardIDX: Practical and secure index with SGX," in *Proc. of Data and Applications Security and Privacy XXXI*, Philadelphia, PA, USA, July 2017.

[10] S. Matetic, K. Wüst, M. Schneider, K. Kostiainen, G. Karame, and S. Capkun, "Bite: Bitcoin lightweight client privacy using trusted execution," Cryptology ePrint Archive, 2018. [Online]. Available: https://eprint.iacr.org/2018/803.pdf

[11] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *Proc. of ACM CCS*, Scottsdale, Arizona, USA, November 2014.

[12] E. Stefanovy, M. van Dijkz, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proc. of ACM CCS*, Berlin, Germany, November 2013.

[13] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious memory primitives from Intel SGX," in *Proc. of NDSS*, San Diego, CA, USA, February 2018.