Using SGX for Bitcoin Simplified Verification Payment Client

***Abstract*—**

I. Introduction

Over the last few years, we have seen a great interest in public blockchain in the community. Bitcoin blockchain technology offered a way to provide security and privacy for financial transactions. However, with a huge adoption by the community, the size of the blockchain has become too large for small and resource-constrained devices such as personal laptops or mobile devices. As of June 2018, the size of unindexed Bitcoin blockchain is 180 GB.

Bitcoin simplified payment verification (SPV) client has become one of the solutions for storage problem for constrained devices. Nakamoto [14] sketched the idea of SPV clients in the Bitcoin whitepaper, and in the Bitcoin improvement proposal 37 (BIP37) [13], Mike Hearn combines Nakamoto's idea with Bloom filter to standardize the design of Bitcoin SPV clients. This design has become de facto standard and used by other light clients such as BitcoinJ and Electrum.

SPV clients only need to download and verify part of blockchain that is relevant to its addresses. In particular, the SPV client loads its addresses into a Bloom filter and sends the filter to a bitcoin full client. The Bitcoin full client will use that filter to identify if a block contains transactions that are relevant to the SPV client, and once it finds such block, it will send a modified block that only contains relevant transactions along with Merkle proofs for those transactions.

Limitation. Gervais et. al. [9] show that it's possible for a malicious node to learn several addresses loaded in the Bloom filter with high probability. If the adversial node can collect 2 filters issued by a same client, then a considerable number of addresses owned by that client will be leaked. Moreover, full nodes that supports the use of Bloom filter are targeted for Denial-of-Service attacks as the malicious clients can cripple it by making lots requests that cause high CPU usage on the full clients [2].

Our Solution. We propose a design for a centralized system that can handle transaction requests from Bitcoin simplified payment clients while offering strong security and anonymity guarantees agaisnt potentially malicious providers. In particular, our design use standard ORAM scheme with a hardware-based solution to protect client's requests from a potentially malicious server.

II. Related Work

(write this)

III. Preliminaries

A. Oblivious Random Access Memory

Oblivious Random Access Memory (ORAM) concept was first introduced by Goldreich et al [10] for software protection against piracy. The main idea of ORAM is to hide the access patterns resulted by reading and writing accesses on encrypted data. The security of ORAM can be described as

Definition 1: [17] Let $\vec{y} = (\mathsf{op_i}, \mathsf{a_i}, \mathsf{data_i})_{i\in[n]}$ denote a sequence of accesses where $\mathsf{op_i} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{a_i}$ is the identifier, and $\mathsf{data_i}$ denotes the data being written. For an ORAM scheme Σ, let $\mathsf{Access}_\Sigma(\vec{y})$ denote a sequence of physical accesses pattern on encrypted data produced by $\vec{y}$. We say:

1) The scheme Σ is secure if for any two sequences of accesses $\vec{x}$ and $\vec{y}$ of the same length, $\mathsf{Access}_\Sigma(\vec{x})$ and $\mathsf{Access}_\Sigma(\vec{y})$ are computationally indistinguishable.
2) The scheme Σ is correct if it returns on input $\vec{y}$ data that is consistent with $\vec{y}$ with probability $\geq 1 - \mathsf{negl}(|\vec{y}|)$

Path-ORAM. Path-ORAM is an efficient tree-based ORAM construction proposed by Stefanov et. al. [17]. In Path ORAM, the client encrypts his database into N different encrypted data blocks, and uses a binary tree of height L to obliviously store those data blocks into a server's untrusted storage. Each node of the tree is a *bucket* that can store at most Z blocks. The client uses a *position map* to keep track of the location for each real data block. For any read and write accesses performed on the encrypted database, the client requires to perform a read, an eviction, and an update procedure on the tree and on the position map. In particular, in order to perform an access on a block, the client uses the position map to look up the block's position in the binary tree to obtain its path, and the client retrieves the path from the server, decrypts, and stores the whole path in a local *stash* of size $O(\lambda)$ blocks where the parameter λ denotes the security level which defines the overflow probability and security level of the path ORAM. The client looks up the targeted block in the stash, and perform access on it. Finally, the client assigns the target block with a new random path, and push all blocks in stash back to the tree according to the position map. Figure III-A describes in detail a Path-ORAM access.

Using same notations in [17], we denote $\mathcal{P}(x)$ to be the path from the leaf x to the root. $\mathcal{P}(x, \ell)$ denote the bucket at level ℓ along the path $\mathcal{P}(x)$. $\mathsf{ReadBucket}(\mathsf{b})$ denotes decrypting and reading all blocks from bucket b. Similarly, $\mathsf{WriteBucket}(b, \mathsf{Block}')$ denotes encrypting and writing block Block' into bucket b. $\mathsf{Position}[\cdot]$ denotes the position map, and $\mathsf{Position}[\mathsf{a}]$ outputs the path identifier that block with a is stored. Finally, S is the stash that is stored on client side.

Recursive ORAM. In the non-recursive Path-ORAM construction, the client has to store a position map of the size $O(N)$ bits. This is not suitable for a resource-constrained client. Stefanov et. al [17] and Shi et. al. [16] present path ORAM construction that reduces the size of the position map to $O(1)$. The main idea of those constructions is to store a position map as another ORAM tree in the server side, and the client only keeps the position map of the new ORAM. The client keeps compressing the position map into another ORAM tree until the size of position map is small enough to be kept client's storage. These constructions come with the cost of an increase in communication between client and server. In this work, we consider the Intel SGX as the client in the ORAM protocol; therefore, the total size of the position map and stash is limited by the size of EPC which is around 128MB. Thus, recursive path ORAM might be a suitable solution for a large dataset.

Algorithm 1 $\mathsf{Path\text{-}ORAM.Access}(\mathsf{op}, \mathsf{a}, \mathsf{data}^*)$

```
1:  $x \leftarrow \mathsf{Position}[a]$
2:  $\mathsf{Position}[a] \overset{\$}{\leftarrow} \{0, \ldots, 2^L - 1\}$
3:  for $i \in \{0, \ldots, L\}$ do
4:      $S \leftarrow S \cup \mathsf{ReadBucket}(\mathcal{P}(x, \ell))$
5:  end for
6:  data $\leftarrow$ read block $a$ from $S$
7:  if op $=$ write then
8:      $S \leftarrow (S - \{(\mathsf{a}, \mathsf{data})\} \cup \{(\mathsf{a}, \mathsf{data}^*)\})$
9:  end if
10: for $\ell \in \{0, \ldots, L\}$ do
11:     $S' \leftarrow \{ (\mathsf{a}', \mathsf{data}') \in S: \mathcal{P}(x, \ell) = \mathcal{P}(\mathsf{position}[\mathsf{a}'], \ell)\}$
12:     $S' \leftarrow$ Select $\min(S', Z)$ blocks from $S'$
13:     $S \leftarrow S - S'$
14:     $\mathsf{WriteBucket}(\mathcal{P}(x, \ell, S'))$
15: end for
16: return data
```

Fig. 1. a Path-ORAM access

B. Intel SGX

Intel SGX is a set of hardware instructions introduced with the 6th Generation Intel Core processors. We use Intel SGX as a trusted execution environment for the execution of ORAM controller on the untrusted server.

In this project, we focus on these main properties of Intel SGX:

- **Enclave:** is the trusted execution unit that is located in a dedicated portion of the physical RAM. The Intel SGX makes sure that all other software on the system cannot access the enclave memory.
- **Remote Attestation and Key Exchange:** SGX allows a remote client to perform a remote attestation to verify the correct creation of an enclave. More importantly, the remote attestation provides an authentication needed for an authenticated key exchange protocol [5]. In particular, after a remote attestation, a client can use Diffie-Hellman Key Exchange [8] to agree on a shared secret.
- **Local Attestation:** Local attestation allows two enclaves on the same system to authenticate each other and exchange data in a secure manner.

Limitations. There are several weaknesses of the Intel SGX

- **Side Channel Attacks:** While Intel SGX provides security guarantees against direct memory attacks, it suffers from various side-channel attacks [19], [18], [4], [11], [12].
- **Enclave Page Cache Limit:** The size of EPC is limited to around 128MB [3].

a) Secure Oblivious Operations: discussion on oblivious comparision, oblivious assignment here. oevict, oreadpath in zerotrace.

C. Bitcoin Unspent Transction Output Database

Each Bitcoin transaction contains inputs and outputs. Thus, a Bitcoin client can determine its balance by summing up all values in those outputs that have not been spent. These outputs are called Unspent Transaction Output (UTXO). Moreover, Bitcoin full nodes maintain a separate database that keeps track of all unspent transaction outputs. This approach allows a full client to fast validate unconfirmed transactions before relaying it to other nodes. More importantly, the UTXO set stores all information needed for client with knowledge of a secret key to create a new transaction. Therefore, we realize that:

- **Privacy**: If the server can provide the SPV client oblivious accesses to the UTXO set, the privacy of the SPV client is preserved.

IV. Proposed System

In this section, we describe how to use Intel SGX as trusted execution unit and Path ORAM to allow a remote client to access the UTXO set in an oblivious manner.

A. Overview

There are 3 components of this system: a client, a untrusted server, and the Bitcoin network. In the untrusted server, we require that there exist two secure enclaves: one for read and one for update.

- **Client** is the simplified payment verification node that remotely connects to the secure enclave on the untrusted server to perform oblivious searches on the UTXO set.
- **Server** is the untrusted entity that is made up of three components: an untrusted OS, a secure trusted read enclave, and a secure trusted update enclave. In particular, the secure update enclave communicate to the Bitcoin network in order to initialize and update the ORAM tree. Moreover, SPV clients can connect to the read secure enclave to invoke ORAM read operations on the ORAM tree to obtain its unspent output.
- **Bitcoin Network** is a set of nodes that maintain the Bitcoin blockchain and relays the new Bitcoin block validated by miners.

The system will works as follow:

- **Initialization:** Since there is no protection from the SGX on system calls such as Network I/O, we expect that the Bitcoin client that runs in the untrusted OS will perform the communication with the Bitcoin network to obtain the Bitcoin blockchain data. Fortunately, Bitcoin uses proof-of-work for its consensus protocol, so we argue that the Bitcoin client reside untrusted OS cannot fool the enclave into using different blockchain because it will have to perform enormous amount of computation power to reproduce a blockchain of same length. First, the enclave will first initialize an empty ORAM tree. Then, for each of Bitcoin block, the enclave will verify the proof of work of the block before using ORAM update operations to populate the ORAM tree. This operation might take several hours. However, once the SGX catches up with the current state of the Bitcoin blockchain, we expect that the enclave only has to perform a batch of update accesses on the ORAM tree every 10 minute.
- **Search Query:** In order to obtain its unspent outputs, the client first performs the remote attestation to the enclave. As mentioned in the previous section, the remote attestation process allows the client to verify if the program is correctly loaded into the enclave or not. More importantly, during the remote attestation, the client can share a secret session key with the enclave to establish a secure connection using *mbed-tls* in SGX. Next, after establishing a secure channel, the client will send a list of hash preimages to the enclave. The read enclave will use a mapping function that map those the preimages to the ORAM block identification and perform ORAM read-only operations on the ORAM tree. In particular, those read-only operations do not involve the eviction procedure which requires reencrypting and remapping ORAM block. We will describe the intuition and the security of this operation in section IV-C.
- **Update Query:** It has been reported that the throughput of Bitcoin network is 3-7 transactions per second [6]. Every input of a transaction implies a deletion operation or an ORAM update operation with dummy data, and every new output means another update operation on the ORAM tree. We estimate that there are an 14 inputs and outputs on average in a Bitcoin transaction (as of January 2018). Thus, an ORAM implementation using Intel SGX can easily handle 14 ORAM operations per second. However, due to the Bitcoin proof-of-work consensus protocol, the Bitcoin network generates a new block approximately 10 minutes. The system will expect a batch of update operations every 10 minutes instead of every 14 operations every seconds. Therefore, in order to prevent updating operations from being blocked by read requests from client, we require that there must be two copies of the ORAM trees: a read-only tree and a shadow-copy tree, and there is an dedicated update enclave that performs a batch of write accesses every 10 minute. Then, when a batch updating operations are finished, we update the position map and the read-only tree with the shadow-copy tree.

B. Oblivious Storage of the UTXO set

In this work, we only consider 2 types of transaction: `Pay-to-Pubkey-Hash` transaction and `Pay-to-Script-Hash` transaction. According to [7], these two types of transaction make up of 99.1% of the UTXO set. Also, one can assume that the `Pay-to-Pubkey-Hash` transaction is a special version of the `Pay-to-Script-Hash` transaction because both transaction types require the spender to know the preimage of the hash digest before being able to spend those outputs. Each of those output contains a hash digest of length 160 bit.

In this system, those outputs belong to `Pay-to-Pubkey-Hash` and `Pay-to-Script-Hash` will be partitioned into different data blocks and stored as a tree structure in the memory of the untrusted server. Thus, there must be a way to map the hash digest in each output into ORAM block identification.

If we consider each unspent output as one ORAM block in the ORAM tree and use the hash digest as the ORAM block identification, the position map will become too big for the EPC memory because the size of each identification key is 20 bytes. Another way is to use a standard hash function that provides a mapping between public key hash and ORAM block identification (e.g. $H : \{0,1\}^{160} \rightarrow \{0,1\}^{32}$). However, directly hashing public key hash into ORAM block identification is not suitable because an attacker can generate several public keys which the hash digests map to the same ORAM block, then fills those public key with small amount of bitcoin. The block will become overflow, and those unspent outputs of legitimate users will be discarded.

To avoid such attack, in our design, during the initialization, we require the program inside the enclave to use a keyed hash function to map the public key hash to ORAM block identification. The secret key of the hash function is stored inside the enclave. In other word, the mapping between the public key hash and the ORAM block identification is known only to the SGX. We define the mapping as follow:

- $\mathsf{bid} \leftarrow blockMapping(\mathsf{hd}, K_b)$: It takes as input a hash digest hd and a secret key K_b, it outputs the block identification number $\mathsf{bid} \in \{0, \ldots, N-1\}$. The value m will determine the maximum number of unspent outputs packed in one block and the size of the position map. We will analyze and show how to choose an appropriate value of m later.

This approach, however, does not entirely prevent an attacker from populating a single address with lots of unspent outputs which also causes an ORAM block to overflow. Fortunately, after analyzing the UTXO set, we find out that around 95% of all bitcoin addresses owned less than 5 unspent outputs. Thus, if we limit the number of unspent outputs for each public key to 5, we can accommodate 95% of all bitcoin users and avoid the above attack. Figure 2 shows us the distribution of unspent outputs and bitcoin address.

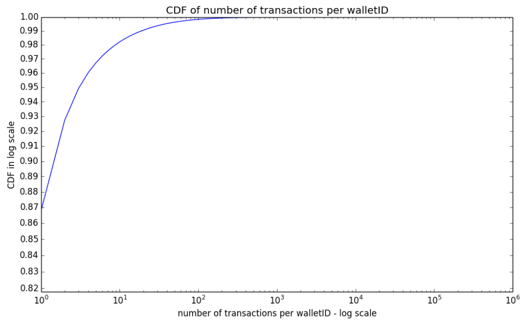

Fig. 2. Number of transactions per wallet id

The hashing approach gives us some flexibility between the size of an ORAM blocks and the size of the position map stored inside the enclave. However, it also comes with the cost of additional storage in the server. The following claim gives us a loose upper bound of the number of addresses can be stored inside an ORAM block.

Claim 1: (Addresses per ORAM block) Let m be the number of public keys, N be the number of ORAM blocks. If the $blockMapping(\cdot,\cdot)$ acts a truly random function, then the maximum load of each ORAM block is smaller than $e \cdot m/N$ with probability at least $1 - 1/N$

a) Proof:: This is a standard bin and ball analysis.

Thus, if we limit each ORAM block to contain the outputs of at most $e \cdot m/N$ addresses, then the probability that every address is included is at least $1 - 1/N$. We let δ to be the number of unspent outputs each address can have and α be the size of each unspent output in bytes. The size of each ORAM block is $em\delta\alpha/N$ bytes. Table I shows us the trade-off between the choice of δ and the required storage of N ORAM blocks and the unspent output database coverage.

δ	1	2	3	4	5
UTXO coverage	86%	91%	92%	94%	95%
Storage, $\alpha = 68$ bytes	3GB	6GB	9GB	12GB	15GB

TABLE I
TRADE-OFF

Remark 1: The choice of N does not affect the data size of the ORAM tree. However, it determines the size of the ORAM block and the height of the tree. Thus, it decides the size of the position map for a non-recursive ORAM construction, and the number of recursion levels for a recursive ORAM construction We will show the experiment result for the optimal choice of N.

Remark 2: Notice that the system requires large extra storage in order to increase the upper limit of unspent outputs for each address. The reason is because in most ORAM schemes, the size of ORAM block is fixed;

therefore, we need to pad dummy data to each ORAM block. However, Roche et. al. [15] proposed an ORAM scheme that allows to have different size block. Their construction is a direct extension of non-recursive Path ORAM. Therefore, we believe that it is possible to overcome the storage requirement by using Roche's ORAM scheme. However, in this work, we only use path ORAM as the proof-of-concept for the design of our system.

Finally, in this system, the untrusted server stores two ORAM trees: a read-only tree and a shadow-copy tree:

- **Read-only Tree** serves as a dedicated storage to handle client requests. While the structure of the tree is identical to the standard ORAM tree, the read-only access does not perform the eviction procedure on the tree like in a standard ORAM setting. In other word, the read-only tree remains the same for the time interval of 10 minutes, and this approach limits the client to perform at most one read per public key every 10 minutes.
- **Shadow-copy Tree** is where all standard ORAM eviction operations are performed. In particular, when the read enclave receives a request from the client, it signals the update enclave to perform standard ORAM reads on the shadow-copy tree by sending a batch of ORAM block IDs via a secure channel; this process can be done after a local attestation between two enclaves. As a new Bitcoin block arrives, the update enclave perform all update operations on the shadow-copy tree, and once the update is finished, it signals the read enclave to use the shadow-copy tree as the new read-only tree.

C. UTXO Oblivious Read and Update Protocols

In this protocol, the SPV client is the party who performs read accesses, and the Bitcoin network is the party who performs write accesses. The SGX in the server is the one that performs both of those accesses on behalf of the client and the Bitcoin network.

1) Oblivious Read Protocol: We will describe how a remote client can perform a read access on the UTXO set. We denote K_s to be the session key, K_o to be the ORAM key, K_b to be the block mapping key, R to be the response stash, bid to be ORAM block id and PK_{bid} be the set can be uniquely identified by the bid. Figure 3 gives an overview of this process.

1) **Establish Secure Channel:** First, the client performs a remote attestation with the secure read enclave and agrees on a session key, K_s.
2) **Prepare Read Query:** The client encrypts its list of public keys or preimages of the hash digests and sends the encrypted query to the server to be passed to the enclave. For simplicity, we assume that the plaintext only contains a list of public keys that the client is interested in, $C \leftarrow Enc_{K_s}(pk_1, \ldots, pk_n)$
3) **Identify and Forward ORAM Block ID:** After decrypting the ciphertext $(pk_1, \ldots, pk_n) \leftarrow Dec_{K_s}(C)$, the read enclave uses $blockMapping(\cdot, \cdot)$ function to learn which block may contain those public key, $(\mathsf{bid}_j)_{j \in [m]} \leftarrow blockMapping(pk_i, K_b)$ for each $i \in \{1, \ldots, n\}$ where K_b is the secret key stored in enclave for mapping purpose. After obtain a list of ORAM $(\mathsf{bid})_{j \in [m]}$, the read enclave forward the list to the update enclave via a secure channel.
4) **Perform ORAM read-only accesses on read-only tree:** Based on the given blockID and the public key, the enclave uses the K_o to perform standard ORAM read accesses on the ORAM tree. If the block contains the unspent output that is related to the public key, we add it into the response stash R. Figure 4 describes the read-only operation. Also, we note that in this operation the read enclave only bring a copy of the path into the stash. This allows different enclave to handle different requests from different clients.
5) **Perform ORAM read accesses on shadow-copy tree:** After obtaining the list of block identifica-

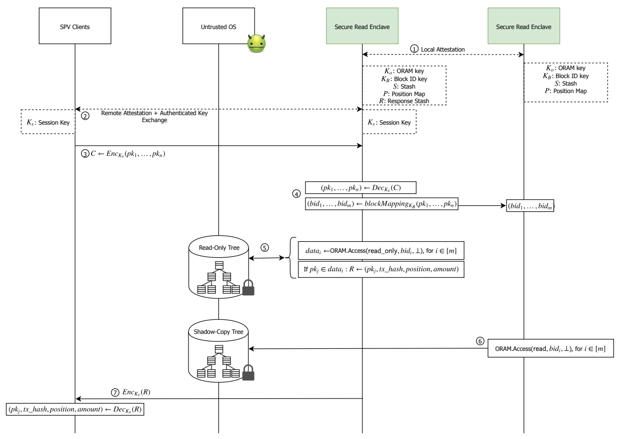
SPV Clients
Untrusted OS
Secure Read Enclave
Secure Read Enclave
① Local Attestation
K_o: ORAM key
K_B: Block ID key
S: Stash
P: Position Map
R: Response Stash
K_o: ORAM key
K_B: Block ID key
S: Stash
P: Position Map
② Remote Attestation + Authenticated Key Exchange
K_s: Session Key
K_s: Session Key
③ $C \leftarrow Enc_{K_s}(pk_1, \ldots, pk_n)$
④
$(pk_1, \ldots, pk_n) \leftarrow Dec_{K_s}(C)$
$(bid_1, \ldots, bid_m) \leftarrow blockMapping_{K_B}(pk_1, \ldots, pk_n)$
$(bid_1, \ldots, bid_m)$
Read-Only Tree
⑤
$data_i \leftarrow$ORAM.Access(read_only, bid_i, $\perp$), for $i \in [m]$
If $pk_j \in data_i : R \leftarrow (pk_j, tx\_hash, position, amount)$
Shadow-Copy Tree
⑥
ORAM.Access(read, bid_i, $\perp$), for $i \in [m]$
⑦ $Enc_{K_s}(R)$
$(pk_j, tx\_hash, position, amount) \leftarrow Dec_{K_s}(R)$

Fig. 3. Oblivious Read Operation

Algorithm 2 $\mathsf{ORAM.Access}_{K_o}(\mathsf{read\_only}, \mathsf{a}, \perp)$

```
1: $x \leftarrow \mathsf{Position}[a]$
2: for $i \in \{0, \ldots, L\}$ do
3:     $S \leftarrow S \cup \mathsf{ReadBucket}(\mathcal{P}(x, \ell))$
4: end for
5: data ← read block $a$ from $S$
6: return data
```

Fig. 4. a read-only access

tion, the update enclave will perform a standard ORAM read accesses on the shadow-copy tree.

6) **Response to Client:** The enclave encrypts the response stash using the session key K_s then sends it to the client.

Figure 5 explains the detail of step 2, 3, 4.

Algorithm 3 ObliviousRead(C, K_o, K_b, K_s)

```
1:  $(pk_i)_{i \in [n]} \leftarrow Dec_{K_s}(C)$
2:  $\mathcal{B} = \{\}$
3:  for $i \in \{1, \ldots, n\}$ do
4:      bid $\leftarrow mappingBlock(pk_i, K_b)$
5:      $\mathcal{B} = \mathcal{B} \cup bid$
6:      $PK_{\mathsf{bid}} = PK_{\mathsf{bid}} \cup pk_i$
7:  end for
8:  send $\mathcal{B}$ to the update enclave
9:  for bid $\in \mathcal{B}$ do
10:     blockData $\leftarrow$ ORAM.Access$_{K_o}$(read_only, bid, $\perp$)
11:     for $\mathsf{pk_i} \in PK_{\mathsf{bid}}$ do
12:         if blockData contains UTXO$_{pk_i}$ then
13:             $R = R \cup$ UTXO$_{pk_i}$
14:         end if
15:     end for
16: end for
17: return $R$
```

Fig. 5. Enclave Read: step 2,3,4

2) Oblivious Update Protocol: In Bitcoin network, a new block is generated on average every 10 minute. Therefore, when the server receives a new block from the Bitcoin network, the update enclave has to perform oblivious write accesses on the shadow-copy ORAM tree, and once these update operations are finished, the update enclave will signal the read enclave to use the new ORAM tree by sending it new position map and pointer to the shadow copy tree via a secure channel. We denote btcBlock to be the Bitcoin block.

1) **Signal Read Enclave to queue read requests:** Once a bitcoin block arrives, the update enclave will signal the read enclave to queue up the read requests from the client. This can be done by appending those requests as an encrypted database in the untrusted memory region.
2) **Verify new Bitcoin block:** Once a bitcoin block arrives to the system from the network, the secure enclave can obtain it from the Bitcoin client. However, since the client runs outside the enclave, the enclave needs to verify the integrity of the new block by computing the Merkle root and verify the proof of work. For the detail of these computations, we refer readers to [1]. Moreover, we make sure that the enclave keeps a separate block headers chain with integrity check outside the enclave just for this verification purpose. Once the Bitcoin block passed the verification, the enclave brings the block inside the trusted region, and removes all unnecessary data. In particular, for each output, the enclave on keep the public key hash or script hash (20 bytes), the amount (8 bytes), the transaction hash (32 bytes), the output position (8 bytes), and for each input, we use the scriptSig field to compute the public key hash or script hash in order to use it to identify the ORAM block. For simplicity, we define this procedure as follow:
   - EVerifyAndPrune(btcBlock, HeaderChain): takes as input a new Bitcoin block btcBlock and a chain of bitcoin header ChainHeader and outputs a pruned set of input $\mathcal{I}$ and output $\mathcal{O}$.

   Also, once the new Bitcoin block arrived, we require that the update enclave signals the read enclave to queue up those read requests from clients for later eviction procedure. In particular, for each client request, the read enclave will encrypt those ORAM block identifications in the untrusted memory.
3) **Map ORAM Block ID:** After pruning the Bitcoin block, the enclave can use the $blockMapping$ function to identify which block needs update on the ORAM tree from the inputs and outputs. This step is identical to step 3 of the read operation.
4) **Perform ORAM write accesses:** In this step, each input will incur at most one write operation on the ORAM. This write operation removes old UTXO from the tree. Moreover, each output requires at most one write operation in order to add new UTXO into the tree. Thus, for each new bitcoin block, we expect at most $14,000$ write accesses on the ORAM tree. As mentioned before, during this process, the read enclave queues up all read requests from the client and stores those requests in a untrusted memory
5) **Signal Read Enclave to use the new ORAM tree** Once the update enclave finishes all update queries, it signals the read enclave to update to the new ORAM tree and sends the read enclave an updated position map.
6) **Perform ORAM read accesses:** After the read enclave updates its read-only tree, the write enclave makes a copy of the read-only tree as a new shadow-copy tree. The write enclave will perform an eviction procedure for read requests that is stored in the untrusted region. Then, once the eviction is finished, the update enclave can signal the read enclave to update the read-only tree again.

Algorithm 4
ObliviousUpdate(BTCBlock, HeaderChain, K_o, K_b, K_s)

```
 1: $(\mathcal{I}, \mathcal{O}) \leftarrow$ EVerifyAndPrune(BTCBlock, HeaderChain)
 2: for input $\in \mathcal{I}$ do
 3:     bid $\leftarrow mappingBlock$(input.$pk$, $K_b$)
 4:     blockData $\leftarrow$ ORAM.Access$_{K_o}$(read, bid, $\perp$)
 5:     if blockData contains UTXO$_{intput.pk}$ then
 6:         blockData = blockData $-$ UTXO$_{intput.pk}$
 7:     end if
 8:     ORAM.Access$_{K_o}$(write, bid, blockData)
 9: end for
10:
11: for output $\in \mathcal{O}$ do
12:     bid $\leftarrow mappingBlock$(output.$pk$, $K_b$)
13:     blockData $\leftarrow$ ORAM.Access$_{K_o}$(read, bid, $\perp$)
14:     if blockData is not full then
15:         blockData = blockData $\cup$ UTXO$_{output.pk}$
16:     end if
17:     ORAM.Access$_{K_o}$(write, bid, blockData)
18: end for
```

Fig. 6. Enclave Write Step 1, 2, 3

Remark 3: We note that step 13 and 17 in the algorithm are just for a demonstration purpose. In the implementation, there is no need to perform 2 ORAM operations, it's possible to change the logic of $ORAM$ write access to update the block data in one operation. We try to stress that the design is not limited to Path ORAM scheme.

D. Security

Since our prototype system is built based on the design use in ZeroTrace, the security of our system is inherited from their design. In particular,

- **Private Information Retrieval via ORAM.**
- **Security Against known side channel attacks.** As we mentioned before, using enclave does not come without limitations. The security of the system against side

V. Performance

(write this)

VI. Conclusion

(write this)

References

[1] bitcoin developer reference, https://bitcoin.org/en/developer-reference

[2] Could spv support a billion bitcoin users? sizing up a scaling claim (2017), https://www.coindesk.com/spv-support-billion-bitcoin-users-sizing-scaling-claim/

[3] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: SCONE: Secure linux containers with intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 689–703. USENIX Association, Savannah, GA (2016)

[4] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.: Software grand exposure: SGX cache attacks are practical. CoRR abs/1702.07521 (2017), http://arxiv.org/abs/1702.07521

[5] Costan, V., Devadas, S.: Intel sgx explained. Cryptology ePrint Archive, Report 2016/086 (2016), https://eprint.iacr.org/2016/086

[6] Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün Sirer, E., Song, D., Wattenhofer, R.: On scaling decentralized blockchains. In: Clark, J., Meiklejohn, S., Ryan, P.Y., Wallach, D., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security. pp. 106–125. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)

[7] Delgado-Segura, S., Prez-Sol, C., Navarro-Arribas, G., Herrera-Joancomart, J.: Analysis of the bitcoin utxo set. Cryptology ePrint Archive, Report 2017/1095 (2017), https://eprint.iacr.org/2017/1095

[8] Diffie, W., Hellman, M.: New directions in cryptography. IEEE Trans. Inf. Theor. 22(6), 644–654 (Sep 2006), http://dx.doi.org/10.1109/TIT.1976.1055638

[9] Gervais, A., Capkun, S., Karame, G.O., Gruber, D.: On the privacy provisions of bloom filters in lightweight bitcoin clients. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 326–335. ACSAC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2664243.2664267

[10] Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing. pp. 182–194. STOC '87, ACM, New York, NY, USA (1987), http://doi.acm.org/10.1145/28395.28416

[11] Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B.: Hacking in darkness: Return-oriented programming against secure enclaves. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 523–539. USENIX Association, Vancouver, BC (2017)

[12] Lee, S., Shih, M.W., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 557–574. USENIX Association, Vancouver, BC (2017)

[13] Mike Hearn, M.C.: Connection bloom filtering (2012), https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki

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

[15] Roche, D.S., Aviv, A., Choi, S.G.: A practical oblivious map data structure with secure deletion and history independence. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 178–197 (May 2016)

[16] Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious ram with o((logn)3) worst-case cost. In: Lee, D.H., Wang, X. (eds.) Advances in Cryptology – ASIACRYPT 2011. pp. 197–214. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[17] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: An extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security. pp. 299–310. CCS '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2508859.2516660

[18] Weichbrodt, N., Kurmus, A., Pietzuch, P., Kapitza, R.: Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.)

Computer Security – ESORICS 2016. pp. 440–457. Springer International Publishing, Cham (2016)

[19] Xu, Y., Cui, W., Peinado, M.: Controlled-Channel attacks: Deterministic side channels for untrusted operating systems. In: S&P (Oakland) (2015)