A Tale of Two Trees: One Writes, and Other Reads

Optimized Oblivious Accesses to Large-Scale Blockchains

Duc V. Le
Purdue University

Lizzy Tengana Hurtado
National University of Colombia

Adil Ahmad
Purdue University

Mohsen Minaei
Purdue University

Byoungyoung Lee
Seoul National University

Aniket Kate
Purdue University

Abstract—The Bitcoin network has offered a new way of securely performing financial transactions over the insecure network. Nevertheless, this ability comes with the cost of storing a large (distributed) ledger, which has become unsuitable for personal devices of any kind. Although the simplified payment verification (SPV) clients can address this storage issue, a Bitcoin SPV client has to rely on other Bitcoin nodes to obtain its transaction history and the current approaches offer no privacy guarantees to the SPV clients.

This work presents T^3, a trusted hardware-secured Bitcoin full client that supports efficient and scalable oblivious search/update for Bitcoin SPV clients without sacrificing the privacy of the clients. In this design, we leverage the trusted execution and attestation capabilities of a trusted execution environment (TEE) and the ability to hide access patterns of oblivious random access memory (ORAM) to protect SPV clients' requests from a potentially malicious server. The key novelty of T^3 lies in the optimizations introduced to conventional oblivious random access memory (ORAM), tailored for expected SPV client usages. In particular, by making a natural assumption about the access patterns of SPV clients, we are able to propose a two-tree ORAM construction that overcomes the concurrency limitation associated with traditional ORAMs. We have implemented and tested our system using the current Bitcoin Unspent Transaction Output database. Our experiment shows that the system is highly efficient in practice while providing strong privacy and security guarantees to Bitcoin SPV clients.

I. INTRODUCTION

Over the last few years, we have seen a great interest in public blockchain in the community. The Bitcoin blockchain offered a way to provide security and privacy for financial transactions. However, due to the huge adoption by the community, the size of the Bitcoin blockchain has become too large for small and resource-constrained devices such as personal laptops or mobile phones, raising not only performance but also privacy concerns in the community. As of October 2018, the size of the unindexed Bitcoin blockchain is 230 GB.

To this end, Bitcoin's simplified payment verification (SPV) client has become a widely-adopted solution to resolve a storage problem for constrained devices. Nakamoto [36] sketched the idea of SPV clients in the Bitcoin whitepaper, and in the Bitcoin improvement proposal 37 (BIP37) [34], Mike Hearn combines Nakamoto's idea with the use of Bloom filters to standardize the design of Bitcoin SPV clients. This design has become de facto standard and been used by other SPV clients such as BitcoinJ [4] and Electrum [5].

The core of SPV clients is in only downloading and then verifying part of the blockchain that is relevant to the SPV client itself. In particular, the SPV client loads its addresses into a Bloom filter and sends the filter to a Bitcoin full client, and 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 the block, it will send a modified block that only contains relevant transactions along with Merkle proofs for those transactions.

However, the current SPV solution relied on Bloom filters raises security and privacy concerns to the SPV clients when communicates with potentially malicious nodes. In particular, Gervais et al. [25] show that it is possible for a malicious node to learn several addresses of the client from the Bloom filter with high probability. Moreover, if the adversarial node can collect two filters issued by the same client, then a considerable number of addresses owned by the client will be leaked.

To provide a strong privacy guarantee for SPV clients, one needs a solution that can hide wallets/addresses queried by the SPV clients. While such a system can be built using private information retrieval (PIR) primitive, the existing cryptographic PIR solutions [30] are not been practical to scale to handle millions of Bitcoin users. On the other hand, to gain more efficiency, one can use ORAM and trusted execution environment to propose generic PIR systems [28], [23], [42]. However, as it becomes apparent in the later in this paper, naively combining ORAM scheme as it is with TEE makes the practicality of those generic systems questionable when used in a large network like Bitcoin due to the lack of concurrency in ORAM as well as the limitation of TEE with restricted memory.

Our Contribution. This work aims not only to design a system that provides SPV clients with privacy-preserving access to the Bitcoin blockchain data but also to consider other practical aspects on how to scale such a system to handle client requests in a large-scale. Our contributions can be summarized as follows:

Firstly, we present a novel design for a system that can handle up to thousands of requests per minute from Bitcoin SPV clients based on a *restricted access* Oblivious Random Access Memory (ORAM) and the trusted execution capabilities of TEE. In particular, one of the main contributions of our design is the optimization access in the prominent tree-based ORAM schemes that allow those ORAM schemes to

support concurrent accesses which is essential for handling SPV clients' requests. In this design, the access privacy guarantee is still maintained because of our natural assumption that the rational Bitcoin SPV clients should only query for their particular transaction *once* before the arrival of a new Bitcoin block. Nevertheless, we later show that even when the SPV clients are irrational then the privacy for such clients is only compromised for a short period of time. The security guarantee of T^3 also relies on the trusted execution capabilities of TEE that allows SPV clients to perform ORAM operations securely and remotely. Our generic design works with other blockchains, any tree-based ORAM schemes [43], [44], [49], and any TEE with attestation capability.

Secondly, we implemented a prototype of T^3 and evaluated its performance to demonstrate the practicality of our approach. More specifically, we extracted the unspent transaction outputs set of Bitcoin in October 2018 and used it to measure the performance of the system when handling clients' requests. The implementation of T^3 also adopts standard techniques (i.e., oblivious operations using cmov [42], [10], [40]) to be secure against known side-channel attacks [32], [31], [51]. Moreover, the use of recursive ORAM constructions in T^3 makes the system much more suitable for TEE with restricted trusted memory like Intel SGX. We then show that the running time of the ORAM read access decreases linearly with the number of the threads used (e.g, up to $8\times$ performance gained with 4 threads).

Finally, we conclude that putting natural restrictions on the access patterns on oblivious memory can lead to significant performance improvement and better ORAM design. While the applicability of T^3 in cryptocurrencies beyond Bitcoin is apparent, we believe our work will also motivate further research on oblivious memory with restricted access patterns.

Concurrent Work. The soon-to-be published BITE system [9] also employs the Oblivious Database construction for SPV client privacy. The main idea of the BITE construction is to combine the use of non-recursive PATH-ORAM [44] construction and TEE (such as Intel SGX) to propose a generic system that offers SPV client with oblivious access to the database. However, BITE did not address several shortcomings of using PATH-ORAM as it is and TEE with restricted memory in practice. In particular, the BITE design did not consider use recursive ORAM constructions to reduce the trusted memory usage; therefore, the efficiency of the system will be degraded once the size of the database gets too large. Moreover, due to the inherent lack of concurrency in tree-based ORAM such as PATH-ORAM, naively using Path-ORAM makes BITE unsuitable for handling thousands of Bitcoin client's requests per minute as well as thousands of updates every fixed period of times (e.g., 10 minutes for Bitcoin). In this work, we investigate the use of both recursive PATH-ORAM and recursive CIRCUIT-ORAM to understand the actual performance and the actual storage overhead put on the server. Importantly, we propose a two-tree ORAM design to further enhance the performance of standard ORAM accesses as well as to allow concurrent requests from the SPV client.

II. DESIGN GOALS AND SOLUTION OVERVIEW

In this section, we define the system components, outline our security goals, and give an overview of how our system works.

A. System Components

There are three key components of this system: the Bitcoin network, a client, and a untrusted server. The **Bitcoin network** is a set of nodes that maintains the Bitcoin blockchain, and the network validates and relays the new Bitcoin block produced by miners. A **client** is a Bitcoin simplified payment verification node that remotely connects to the secure TEE on the untrusted server to perform oblivious searches on the unspent transaction output (UTXO) set. The client is also able to connect to the Bitcoin Network to obtain other network metadata such as the latest Bitcoin block header. A **server** is the untrusted entity made up of two components: an untrusted server and several trusted TEEs (i.e., the *managing*, *reading*, and *writing* TEEs). Moreover, the untrusted server stores three encrypted databases which are the *read-once* ORAM tree, the *original* ORAM tree, and the Bitcoin header chain. The untrusted server hosts a potentially malicious bitcoin client (e.g., bitcoind) that handles the communication with the Bitcoin Network.

B. Design Goals

The goal of our system is to leverage the trusted execution capabilities of Trusted Execution Environment (TEE) with attestation to design a public Bitcoin full node that supports oblivious search and update on the current Bitcoin unspent transaction output database. Our system aims to provide data confidentiality and privacy to Bitcoin SPV clients in a large scale by using standard encryption and Oblivious RAM techniques on the current set of unspent transaction outputs. The main goals that T^3 tries to achieve are:

1) **Privacy.** T^3 aims to provide privacy and confidentiality to SPV clients' requests. In particular, the system allows SPV clients to obliviously search its relevant transactions without revealing their addresses to potentially malicious providers by using TEE to encrypt the data and using ORAM schemes to eliminate known side channel leakages [40], [10], [28], [42].
2) **Validity.** In our design, the SPV client should be able to obtain valid information based on the provided addresses, and a malicious adversary should not able to tamper the Blockchain data with invalid transaction outputs.
3) **Completeness.** The system should provide clients with access to most of its relevant transactions in order to determine balance or to obtain essential information to form new transactions.
4) **Efficiency.** The system should be practical to deploy. More specifically, the system should be efficient enough to handle different concurrent SPV clients' requests without compromising the privacy of the clients.

C. Solution Overview

The idea of using ORAM schemes and trusted execution environment to construct database systems that support oblivious accesses has been investigated by the research community [28], [23], [42]. However, the efficiency and scalability of those systems are hampered by the lack of concurrency of traditional ORAM schemes [44], [49].
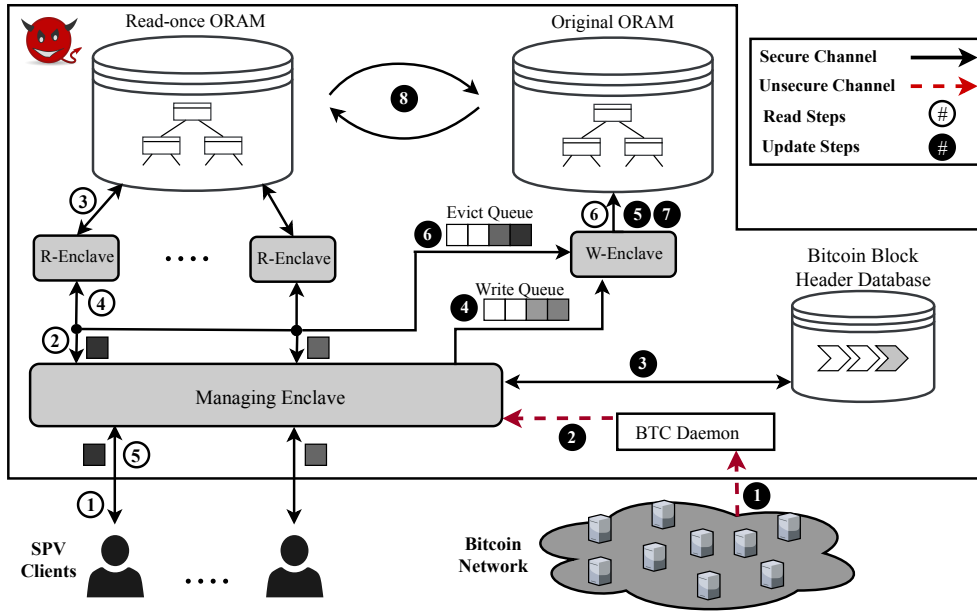
Fig. 1: An overview of T^3 workflow. The encrypted ORAM databases are stored in server untrusted memory region. Steps ①-⑥ describe the flow of the request sent from SPV clients. Steps ❶-❽ describe the flow of the update procedure when T^3 receives new Bitcoin block

In this work, we design T^3 to overcome the limitations of efficiency and concurrency plaguing existing systems. Our design is motivated by the following observations. The first observation is that each ORAM access in a standard tree-based ORAM settings is a combination of two operations: a *read-path* operation and an *eviction* operation. By separating the effects two operations into two different trees: a *read-once* ORAM tree and an *original* ORAM tree, one can use read-path operation on the *read-once* ORAM tree to handle clients' requests simultaneously while performing a non-blocking eviction operation on the *original* ORAM tree sequentially. This design is also independently investigated by ConcurORAM [14]; however, their design is not suited for TEEs with limited trusted memory (such as Intel SGX). We elaborate on this in the coming sections.

The second observation is that the access privacy guarantee of this approach relies the characteristic of the Bitcoin blockchain. In particular, the Bitcoin network generates new Bitcoin block on average of 10 minutes, and if we require T^3 to periodically synchronize these the two trees, then the privacy of clients' queries are preserved. Moreover, if we assume that upon receiving transactions belonged to its addresses, the rational client should not query same transactions *again* until the next block arrives, the proposed approach on the separation of *read-path* and *eviction* procedure not only does not affect the privacy guarantees of ORAM access but also allows T^3 to handle much more clients' requests. More importantly, we also argue that even when the SPV clients are irrational by submitting requests for the same transaction more than one, the privacy of those clients is only compromised for a short period of time (i.e., 10 minutes for the Bitcoin network) because T^3 will always synchronize the old instance of the *read-once* ORAM tree with the more updated instance of the *original* ORAM tree. With the intuition of T^3 described above, we outline the workflow of our design:

Server Initialization ❶-❽: Initially, the managing TEE will initialize a *writing* TEE that creates an empty ORAM tree. For each of Bitcoin block obtained from the network, the managing TEE verifies the proof of work of the block before passing relevant update data to the *writing* TEE in order to populate the ORAM tree. With the current size of the Bitcoin blockchain, this operation might take several hours. However, once the TEEs catch up with the current state of the Bitcoin blockchain, we expect that the TEE only has to perform a batch of update accesses on the ORAM tree every 10 minute. When the initialization is completed, the *managing* TEE creates two copies of the ORAM tree which are the *read-once* ORAM tree and the *original* ORAM tree.

Oblivious *read-once* Protocol ①-⑥: In order to obtain its unspent outputs, the client first performs the remote attestation to the *managing* TEE. The remote attestation mechanism allows the client to verify the correctness of program execution inside the TEE. More importantly, after a successful attestation, the client can use standard key exchange mechanism (i.e. Diffie-Hellman's key exchange) to share a secret session key with the TEE in order to establish a secure connection with the *managing* TEE. Upon receiving client's connection requests, the *managing* TEE creates a *reading* TEE with its own copies of the ORAM position map and the ORAM stash to handle client subsequent requests. Next, after having a secure channel, the client will send his Bitcoin addresses along with the proof of ownership of those addresses to the TEE (e.g., the knowledge of the public key along with a signature to a random nonce or the preimage of the public key hash). The *reading* TEE will use a mapping function to map Bitcoin addresses into the ORAM block identification number and performs *read-once* ORAM access on the ORAM tree. In particular, those *read-once* ORAM access do *not* involve the eviction procedure which requires re-encrypting and remapping the ORAM block. The eviction procedure will be performed on the *original* ORAM tree by the *writing* TEE.

Oblivious Write Protocol ❶-❽: The T^3 requires to update

the ORAM tree via batch of write accesses every 10 minutes on average. In particular, T^3 will rely on a standard Bitcoin client to handle the communication with the Bitcoin network to obtain blockchain data[1]. Thus, T^3 needs to verify the block relayed by a potentially malicious Bitcoin client before updating the ORAM tree. More specifically, in the design, T^3 stores a separate Bitcoin header chain to verify the proof of work and the validity of all transactions inside a Bitcoin block. After the verification, the *managing* TEE forms a batch of ORAM updates and delegates those updates to the *writing* TEE. Once those updates are finished, the *managing* TEE will queue up read requests from SPV clients in order to allow the *writing* TEE to finish the eviction requests from the *read* TEEs during the updating interval. As soon as the *writing* TEE finishes performing those eviction requests, the *managing* TEE updates the position map and *stash*, and makes the ORAM tree used by the *writing* TEE become the new ORAM tree used by *reading* TEE. At this point, the *reading* TEE can use the new tree instance to respond to clients' requests while the *writing* TEE performs the eviction procedure on another copy of the same ORAM tree.

III. PRELIMINARIES AND THREAT MODEL

A. Trusted Execution Environment

The design of T^3 relies on a trusted execution environment (TEE) to prove the correctness of the computations. In particular, TEE is a trusted hardware that provides both confidentiality and integrity of computations as well as offer an authentication mechanism, known as *attestation*, for the client to verify computation correctness. In this work, we chose Intel SGX [18] to be the building block of our system. However, with minor modifications, the design of our system can be extended to any TEE with *attestation* capabilities such as Keystone-enclave [7] and Sanctum [19] as other trusted execution environments might not have the same strengths/weaknesses as Intel SGX.

Intel SGX is a set of hardware instructions introduced with the 6th Generation Intel Core processors. We use Intel SGX as a TEE for the execution of an ORAM controller on the untrusted server. The relevant elements of SGX are as follows. **Enclave** is the trusted execution unit that is located in a dedicated portion of the physical RAM called the enclave page cache (EPC). The SGX processor makes sure that all other software components on the system cannot access the enclave memory. Intel SGX supports both **local and remote attestation** mechanisms to allow remote parties or local enclaves to authenticate and verify if the program is correctly executed within an SGX context. More importantly, attestation protocols provide the authentication required for a key exchange protocol [18], i.e., after a successful attestation, the concerned parties can agree on a shared session key using Diffie-Hellman Key Exchange [22] and create a secure channel.

Limitations. Intel SGX comes with various limitations which have been uncovered by the academic community over the past few years. In particular, some of the limitations are:

- **Side Channel Attacks:** While Intel SGX provides security guarantees against direct memory attacks, it does not provide systematic protection mechanisms against side channel attacks such as page table-based [51], [31], cache-based [13], and branch-prediction-based [32]. Through page table and cache attacks, a privileged attacker can observe cache-line-granular (i.e., 64B) memory access patterns from the enclave program. On the other hand, the branch-prediction attack can potentially leak all the control-flow taken by the enclave program.
- **Enclave Page Cache Limit:** The size of the Enclave Page Cache (EPC) is limited to around 96MB [11]. Although Intel SGX alleviates this limitation by supporting page-swapping between trusted memory region and untrusted memory region, this operation is expensive due to encryption and integrity verification [18], [11].
- **System Calls:** Intel SGX programs are restricted to ring-3 privileges and therefore rely on the untrusted OS for ring-0 operations such as file and network I/O. There are various previous works which try to solve this problem using library OSes [47] and/or other techniques [28].

Oblivious Operations inside the Enclave. Several techniques [40], [28], [42], [37] have been introduced to mitigate side-channel attacks on the SGX. In this work, we built our system based on the implementations of both Zerotrace [42] and Obliviate [10]. Therefore, our system inherited standard secure operations from both of these libraries. In particular, their implementations use an oblivious access wrapper by using the x86 instruction cmov as introduced by Raccoon [40]. Using cmov, the wrapper accesses every single byte of a memory object while reading or writing only the required bytes in memory. From the perspective of an attacker (which can only observe access-patterns), this is the same as reading or modifying every byte in memory. We refer readers to [42], [10], [40] for detailed description of these oblivious operations.

B. Oblivious Random Access Memory

Oblivious Random Access Memory (ORAM) was first introduced by Goldreich et al [26] for software protection against piracy. The core 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 follows.

Definition 1: [44] Let $\vec{y} = (\mathsf{op}_i, \mathsf{bid}_i, \mathsf{data}_i)_{i \in [n]}$ denote a sequence of accesses where $\mathsf{op}_i \in \{read, write\}$, bid_i is the identifier, and 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: (a) 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. (b) The scheme Σ is correct if it returns on input $\vec{y}$ data that is consistent with $\vec{y}$ with probability $\geq 1 - \mathsf{negl}(|\vec{y}|)$ i.e negligible in $|\vec{y}|$

Tree-based ORAM schemes. One strategy of designing an ORAM scheme is to follow the tree paradigm proposed by Shi et al. [43] and Stefanov et al. [44]. In tree based ORAM, the client encrypts their database into N different encrypted data blocks and obliviously stores those data blocks in a binary tree of height $\lceil \log_2(N) \rceil$. Each node in the tree is called a *bucket*, and each *bucket* can contain up to Z blocks. The client also maintains a *position map*, to indicate which path a data block

---

[1]This ability can be easily included in the future implentation of T^3.

Algorithm 1 $ORAM.\text{Access}(op, \text{bid}, \text{data}^*)$

1: $S \leftarrow \text{ReadPath}(\text{bid})$ //scan the whole stash
2: $\text{data} \leftarrow$ block bid from S
3: **if** $op = write$ **then**
4: $\quad S \leftarrow (S - \{(\text{bid}, \text{data})\}) \cup \{(\text{bid}, \text{data}^*)\}$
5: **end if**
6: $p' \xleftarrow{\$} \{0, \ldots, N-1\}$ //the random eviction path is selected
7: $S \leftarrow \text{Evict}(S, p')$
8: **return** data

Fig. 2: a standard tree-based ORAM *read/write* access

resides on. Finally, the client needs to have a *stash* to store a path retrieved from the server.

We follow the same generalization of a tree-based ORAM access described in [28]. Each access in both ORAM schemes requires two operations: a ReadPath operation and an Evict operation. Intuitively, ReadPath takes as input the ORAM block identifier, bid, accesses the position map, and retrieves the path that block bid resides onto the stash, S. After performing ORAM access (i.e. read/write) on the identified block, the block is assigned to a different path and pushed back to the tree via the Evict operation. In general, the Evict operation takes a stash and the assigned path as input, writes back blocks from stash to the assigned path, and update the position map. Figure 2 gives an overview of how tree-based ORAM access works.

PATH-ORAM/CIRCUIT-ORAM scheme. In this work, we consider two popular tree-based constructions of ORAM which are PATH-ORAM [44] and CIRCUIT-ORAM [49]. While PATH-ORAM offers simple ReadPath and Evict operations, CIRCUIT-ORAM offers a smaller circuit complexity for the Evict procedure. Thus, CIRCUIT-ORAM is more efficient when implemented with Intel SGX. As noted in [42], [28], [49], CIRCUIT-ORAM can operate with $Z = 2$ compared to $Z = 4$ as in PATH-ORAM; therefore, the server storage overhead is significantly reduced. Moreover, the size of *stash* in CIRCUIT-ORAM is smaller compared to the size of *stash* in PATH-ORAM; this allows a more efficient performance when scanning the stash as one needs to scan the whole path and stash to avoid side-channel leakage.

Recursive ORAM. In a non-recursive tree-based ORAM setting, the client has to store a position map of the size $O(N)$ bits. This approach, however, is not suitable for a resource-constrained client. Stefanov et. al [44] presented a technique 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 the position map is small enough to be saved on the client's storage. One main drawback of those constructions is the increased cost in the communication between a client and the server. Fortunately, in our setting, this cost can be safely ignored because the communication between client and server becomes the I/O access between TEE and the random access memory. Thus, it is more reasonable to use recursive constructions because it reduces the memory stored in the trusted region (e.g., Processor Reserved Memory).

C. Blockchain

The Bitcoin blockchain is a distributed data structure maintained by a group of nodes. In this work, to simplify the structure of the Bitcoin blockchain, we denote the network as a single party that maintains a growing database of Bitcoin blocks. On average of 10 minutes, the network outputs a Bitcoin block which is a combination of Bitcoin transactions and a block header. Each block header contains relevant information about the Bitcoin block such as Merkle root, nonce, network difficulty. The Merkle root can be used to verify the membership of Bitcoin transactions, and the nonce and difficulty are used to check the proof of work. Each Bitcoin transaction contains a set of inputs and outputs where transaction inputs are unused outputs of previous transactions.

● **Unspent Transaction Output Database.** In the Bitcoin network, the balance of a Bitcoin address is determined by values of those outputs that have not been used in other transactions. These outputs are called Unspent Transaction Outputs (UTXO). Moreover, in the implementation of common Bitcoin nodes such as Bitcoin core [1], Bitcoin nodes maintain a separate database that keeps track of all unspent transaction outputs and other metadata of the Bitcoin blockchain. This database is known as the UTXO set. Intuitively, a client with the knowledge of the secret key and the commitment value can query the UTXO set directly to obtain essential information such as transaction hash, position, and value to form new valid transactions. Therefore, in this work, we realize that if a full node can securely update and maintain the integrity of the UTXO set via while provides SPV clients with oblivious accesses to the UTXO set, the privacy of the SPV client is preserved.

● **Bitcoin transaction types.** In the Bitcoin blockchain, transactions are classified based on the structure of the input and output scripts. In particular, there are five types of standard script templates which are *Pay-to-Pubkey* (P2PK), *Pay-to-PubkeyHash* (P2PKH), *Pay-to-ScriptHash* (P2SH), *Multisig*, and *Nulldata*. Intuitively, scripting in Bitcoin provides a way to prove the ownership of the coins. In particular, a challenge script (*scriptPubkey*) is included as a part of the transaction output to specify the condition for its redemption, and a response script (*scriptSig*) is part of the transaction input to reveal the condition needed to redeem the bitcoins from other output.

In this work, we only consider two types of transaction: *Pay-to-PubkeyHash* (P2PKH) transaction and *Pay-to-ScriptHash* (P2SH) transaction. According to [21], [35], these two types of transaction make up of 97-99% of the UTXO set. Also, one can assume that the *Pay-to-Pubkey-Hash* transaction is one variant of the *Pay-to-Script-Hash* transaction because both transaction types require the spender's knowledge of the preimage of the hash digest before being able to spend those outputs. For simplicity, from this point on, we assume that the only information needed to obtain the unspent output is the public key hash, pkh. Moreover, all other transaction types such as *Multisig* and P2PK can be easily supported in the future.

● **Block creation interval.** The block creation time in Bitcoin is the time that the network takes to generate a new block, and block creation time is specified to be 10 minutes on average by the network. We call the waiting period between

the most recent block and a new block, *block creation interval*. In this work, we discretize time as *block creation intervals*.

D. Threat Model

We assume that SPV *clients* are honest and rational which means that before during the *block creation interval*, a SPV *client* should not request the server for transaction outputs of a same public key hash more than once. The underlying remote attestation service provided by Intel is secure and trusted. The local attestation between enclaves is secure. The server and its programs are assumed to be untrusted except for programs running within an enclave.

We assume that the adversary who controls the operating system can read/inject/modify encrypted messages sent by enclaves. The adversary also can observe memory access patterns of both trusted and untrusted memory. Also, the computation power of the adversary is assumed to be limited. In particular, during *block creation interval*, the adversary should not have enough computation power to forge a new Bitcoin block that satisfies the current Bitcoin network difficulty. As the time of writing, the network difficulty [3] is around 6×10^9; therefore, the expected number of hashes to mine a Bitcoin block is roughly 2^{72}.

The server's attacks on availability are out of scope. More specifically, denial of service (DoS) attacks by system admin and untrusted operating system are out of the scope. Otherwise, such adversary can prevent the enclaves from receiving new bitcoin block by shutting down the communication channel between the enclave and the Bitcoin network as the enclave has to rely on the untrusted OS to perform system calls such as file and network I/O.

IV. PROPOSED SYSTEM

In this section, we first describe how T^3 stores the UTXO set by exploring different mappings between the unspent transaction outputs and the ORAM blocks. We see that naive mapping may not be secure for blockchain applications as it may lead to denial of services attacks. Next, we demonstrate how Intel SGX can be considered as a trusted execution unit to access ORAM and perform read/write operations in an oblivious manner. Finally, we will describe how the system handles clients' requests during a write operation.

A. Storage Structure of the UTXO set

In the first step, we show how to map the public key hash to ORAM block identification and then describe the storage requirements in T^3.

1) Bitcoin unspent transaction output mapping: In the design of T^3, we assume that the SPV clients only know his/her addresses (i.e., the public key hashes); therefore, to return the outputs belonging to the client's address, the enclave needs to know the mapping between the address and the ORAM block identification.

In this work, we propose two simple mappings to store unspent outputs in the ORAM tree. More specifically, both approaches use standard cryptographic hash functions along with a secret key generated by the enclave. The first approach is to map a single Bitcoin address into a single ORAM block,

and the second approach is to map a Bitcoin address into multiple ORAM blocks. We will later explain the trade-off between these two approaches. Intuitively, the first approach is more efficient in terms of performance and can be more expensive in terms of storage overhead. The second approach gives some flexibility in terms of storage overhead; however, to offer strong privacy to every address, this approach can be more expensive in terms of performance because it may incur more ORAM calls.

Single address into Single ORAM block. In this 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 generated and known only by the enclave. In other words, the mapping between a Bitcoin address to an ORAM block identification is known only to the SGX. We define the mapping as follow:

- bid $\leftarrow$ OBlockMap(pkh, k_b): the function takes as input a 20-bytes hash digest pkh and a secret key k_b, it outputs the block identification number bid $\in \{0, \ldots, N-1\}$.

The key-hashing approach offers some flexibility when deciding the size of an ORAM blocks and the size of height of the ORAM tree. These two factors affect the size of the position map (resp. number of recursive levels) for non-recursive (resp. recursive) ORAM constructions. However, since the output domain of OBlockMap$(\cdot, \cdot)$ is limited to the size of the ORAM blocks, there will exist collisions. The following claim gives us a loose upper bound on the number of addresses that should be stored inside an ORAM block.

Claim 1: (Addresses per ORAM block) Let m be the number of public key hashes, N be the number of ORAM blocks. If the OBlockMap() acts as a truly random function, then the maximum number of addresses in each ORAM block is smaller than $e \cdot m/N$ with a probability $1 - 1/N$.

Proof: This is a standard max-load analysis. We refer readers to [20] for detailed analysis. We note that there exists a tighter bound, but we use $e \cdot m/N$ bounds to simplify the equation. ∎

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 $1 - 1/N$. Figure 3 gives us a high level overview of this approach.
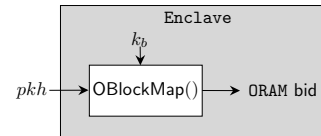

Fig. 3: Single address into Single ORAM block

Single address into Many ORAM blocks. Mapping a single address into a single ORAM block incurs less work on the server as it requires a single ORAM access for an address. However, if we want to allow each address to have more than one output, using the first approach implies that the storage overhead increase linearly because the first approach distribute unspent outputs based on its addresses. Therefore, we have to pad dummy data for those addresses that contain 1 outputs. Thus, we need a different mapping without linear increasing

in storage overhead. To fix this shortcoming, the system needs to assign unspent outputs into ORAM block uniformly. One method is to allow a client to specify the number of ORAM accesses to obtain all of its unspent outputs as long as the number of requests does not exceed certain threshold. We define the mapping as follows:

- $\{bid_i\}_{i\in\{0,...,\delta-1\}} \leftarrow$ OBlockMap(pkh, k_b, δ): the function takes as input a 20-bytes hash digest pkh, a secret key k_b, and a number δ where the maximum value of δ is specified by the system. It outputs a set of block identification numbers $\{bid_i\}_{i\in\{0,...,\delta-1\}} \subseteq \{0,...,N-1\}$.

This approach also introduces some leakage as some addresses may contain more unspent outputs than others. Alternatively, the system can fix the value of δ ORAM accesses for all addresses with the expense of performance (i.e., one address incurs constant ORAM accesses). Similarly, the storage overhead of T^3 can be computed using the following claim:

Claim 2: (UTXO per ORAM block) Let m be the number of unspent outputs, N be the number of ORAM blocks. If the OBlockMap acts as a truly random function, then the maximum number of outputs in each ORAM block is smaller than $e \cdot m/N$ with probability at least $1 - 1/N$

The proof is identical to proof of claim 1.
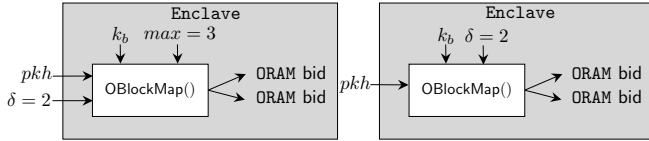
Figure 4 offers an overview of the both approaches.

Fig. 4: Single Address into Many ORAM blocks

2) Storage: In this system, we require the untrusted server to store three separate databases which are the *read-once* ORAM tree, the *original* ORAM tree, and the blockheader chain. In particular, **Read-Once ORAM Tree** serves as a dedicated storage to handle clients' requests. The structure of the tree is identical to the standard ORAM tree **Original ORAM Tree** is where all standard ORAM eviction operations are performed. In this work, we also require the enclave to maintain the **Bitcoin Header Chain** to verify the proof of work of the bitcoin block sent by other bitcoin client. The header chain is stored in the untrusted memory with integrity check.

B. Oblivious Read and Write Protocols

In T^3, the SPV client is the party who invokes read accesses, and the Bitcoin network is the party who invokes write accesses. The TEE in the server is the one that performs both of those accesses on behalf of the client and the Bitcoin network.

1) Server System Components: Before explaining how oblivious read and write accesses work, we first start outlining the different components of our design. The server is initialized with different enclaves: **Managing Enclave** $\mathcal{E}_m$ coordinates other enclaves and to handle requests from the clients. The *managing* enclave also handles the communication with other Bitcoin client or local Bitcoin client (bitcoind) via
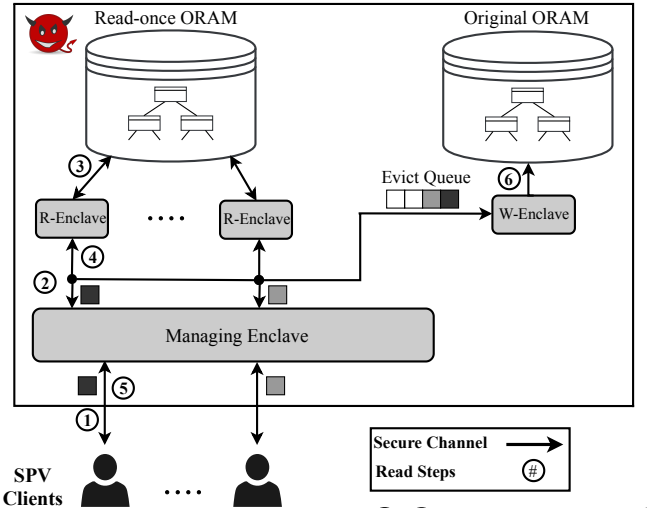

Fig. 5: The read protocol. Steps ①-⑤ describes how T^3 receives and responds to the client, and for each request, the *writing* enclave performs the Eviction procedure of ORAM on the *original* ORAM tree during step ⑥.

request procedure calls (RPC) to obtain Bitcoin blocks. Upon receiving the Bitcoin block, the *managing* enclave also verifies the integrity of the block using a separated header chain. **Reading Enclave** $\mathcal{E}_r$ is a dedicated enclave initialized by the *managing* enclave. It has a copy of ORAM position map and its own stash. The *reading* enclave operates on the *read-once* ORAM tree. Also, the *reading* enclave only performs ORAM ReadPath operations to obtain data while ORAM Eviction operations will be handled by the *writing* enclave. **Writing Enclave** $\mathcal{E}_w$ performs Eviction procedure for each read request, and performs ORAM writing accesses when a new Bitcoin block arrives from the Bitcoin network.

2) Oblivious read-once Protocol: Here, we describe how a remote client can perform a read access on the UTXO set.

Notation. First, let's denote K_b to be the block mapping key, bid to be the ORAM block identification. We let (Enc, Dec) denote an authenticated encryption scheme. We assume that the the server has already been initialized with a *writing* enclave, $\mathcal{E}_w$ and a *managing* enclave, $\mathcal{E}_m$. The *managing* enclave has a similar copy of the position map as the map in the *writing* enclave. Figure 5 presents the oblivious read protocol of T^3. The oblivious read protocol can be described as follows:

1) **The client establishes a secure channel with the *managing* enclave** ①: First, the client performs a remote attestation with the secure *managing* enclave, $\mathcal{E}_m$, and agrees on a session key, K_s. The client encrypts his address along with the proof of ownership of that address, and sends the encrypted query to the server to be passed to $\mathcal{E}_m$. For simplicity, we assume that the plaintext only contains a public key hash, pkh, that the client is interested in, and the proof of ownership of the pkh is ϕ, $C \leftarrow$ Enc$_{K_s}(pkh, \phi)$. Note that there are different ways to prove the ownership of public key hash/addresses. In Bitcoin, if the public key is never revealed before, the proof of ownership can simply be the public key (i.e. $\phi = pk$ such that $H(pk) = pkh$). Alternatively, the system can enforce a client to provide the signature and the public key to prove the ownership of the public key hash.

2) **The *managing* enclave initializes a *reading* enclave** ② : after receiving a client's request, $\mathcal{E}_m$ initializes a dedicated *reading* enclave, $\mathcal{E}_r$ to handle the client's future requests. Also, we require that the enclaves authenticate each other, and the existence of a secure channel between enclaves. Moreover, the *reading* enclave has its copy of the position map, its own stash, the block mapping key K_b, and the agreed session key K_s.

3) **The *managing* enclave identifies and forwards ORAM Block ID to both *reading* and *writing* enclaves** ② : After decrypting the ciphertext $(pkh, \phi) \leftarrow \text{Dec}_{K_s}(C)$, $\mathcal{E}_m$ verifies the proof ϕ and pkh, then uses $\text{OBlockMap}(\cdot, \cdot)$ [2] function to learn the ORAM block ID, $\text{bid} \leftarrow \text{OBlockMap}(pkh, K_b)$ where K_b is the secret key generated by the enclave during initialization for mapping purposes. After obtaining the ORAM id, bid, the *managing* enclave forwards bid to the *writing* enclave for the eviction procedure, and forwards the (pkh, bid) to the *reading* enclave.

4) **The *reading* enclave performs *read-once* ORAM access on the *read-once* ORAM tree** ③ : Based on the given bid, the *reading* enclave performs ORAM read only accesses on the ORAM tree to obtain the block. If the block contains the unspent output that belongs to the public key pkh, the *reading* enclave adds outputs into the response R. To mitigate the size leakage, the response R is padded with dummy data if there is no UTXO found.

5) **The *reading* enclave responds to the Client** ④-⑤ : The enclave encrypts the response, R, using the session key K_s then sends it to the client.

6) **The *writing* enclave performs the eviction procedure on the *original* ORAM tree** ⑥ : After obtaining the bid from the *managing* enclave, the update enclave will perform a standard ORAM read accesses on the *original* ORAM tree. The goal of this procedure is to use the Eviction procedure inside standard ORAM operation to rerandomize the location of the actual block. No actual data is return in this step.

3) Oblivious Write Protocol: In the Bitcoin network, miners generate a new Bitcoin block on average every 10 minutes. When the server receives a new block from the Bitcoin network, the *managing* enclave can obtain it from the bitcoin client. The $\mathcal{E}_m$ verifies the integrity of the block by computing the Merkle root from transactions, then verifying the proof of work, and the *writing* enclave has to perform an update on the *original* ORAM tree; however, in the mean time, the system should be able to handle clients' requests during updates. We will explain how T^3 handles oblivious write accesses while handling clients' requests as follow:

1) **The *managing* enclave verifies a new Bitcoin block** ❶-❸ : Once a bitcoin block arrives to the system from the Bitcoin network, the *managing* enclave $\mathcal{E}_m$ 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 verifying the proof of work to make sure that the block has not been tampered by the untrusted OS. For the detail of these computations, we refer readers to [2]. Moreover, as discussed
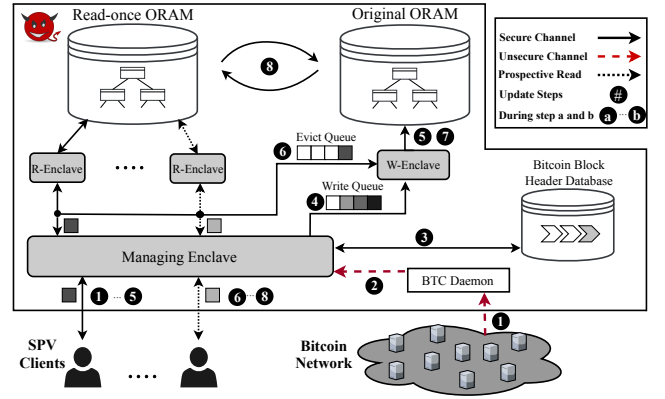
²for simplicity, we assume that the one-to-one mapping is used here

Fig. 6: Oblivious write protocol. During steps ❶-❺, the *managing* enclave receives and responds to SPV client request as usual. During steps ❻-❽, read requests from clients are queued up, and the *managing* enclave resume these requests after updating the *read-once* ORAM tree

in section IV-A, to verify a newly arrived block, the system is required to keep a separate block headers chain with integrity check in the untrusted memory. Once $\mathcal{E}_m$ verifies the bitcoin block, $\mathcal{E}_m$ starts pruning the transactions to obtain relevant information of the transactions' inputs and outputs. Then, $\mathcal{E}_m$ uses $\text{OBlockMap}(\cdot, \cdot)$ to find the ORAM block identification to queue up ORAM write requests to the *writing* enclave. During this process, the oblivious read protocol performs as normal on the *read-once* ORAM tree.

2) **The *managing* enclave sends write requests to the *writing* enclave** ❹ : Once the pruning process completes, the $\mathcal{E}_m$ starts sending write requests based on data extracted from the bitcoin block to the *writing* enclave, $\mathcal{E}_w$. On otherhand, for each eviction request resulted from SPV client's requests, $\mathcal{E}_m$ starts queuing up those eviction requests.

3) **The *writing* enclave performs write accesses on the *original* ORAM tree** ❹-❺ : Upon receiving writing requests from $\mathcal{E}_m$, the $\mathcal{E}_w$ performs all writing requests in the writing queue on the *original* ORAM tree.

4) **The *writing* enclave finishes all eviction requests queued up on the *original* ORAM tree** ❻-❼ : Once finished updating the tree, the $\mathcal{E}_w$ signals $\mathcal{E}_m$ to start queuing up clients' requests and performs all eviction requests incurred by SPV clients' read requests during update interval. Finally, when it finishes, it signals the $\mathcal{E}_m$ to update the *read-once* ORAM tree and make a copy of the position map.

5) **The *managing* enclave performs an update the *read-once* ORAM tree and the *original* ORAM tree and enclave metadata** ❽ : In particular, $\mathcal{E}_m$ discards the current copy of the *read-once* ORAM tree, and makes 2 identical copies of the most updated *original* ORAM tree. One is used as *read-once* ORAM tree, and the other is used as *original* ORAM tree. Also, the new position map and new stash are updated for the *managing* enclave. Once this process is finished, $\mathcal{E}_m$ starts answering SPV clients' requests again.

Figure 6 gives us an overview of the oblivious write protocol.

Discussion The core idea of the oblivious update protocol is to minimize the downtime of T^3 during the update process.

More specifically, during step ❶-❺, T^3 still allows SPV clients to query the system while from step ❻-❽, T^3 stops accepting clients' requests in order to synchronize both trees. This approach introduces some delay; however, the system downtime is minimized to the same amount of time it takes for the *writing* enclave to finish all eviction requests.

V. EVALUATION AND COMPARISON

In this section, we describe our configuration, our experimental results, and the storage overhead of the system based on the analysis of the UTXO set on the Bitcoin blockchain. Moreover, we give a comparison between T^3 and the current existing SPV solution in term of performance and communication overhead. Finally, we address the capabilities of T^3 compared to other related works.

A. Configuration

Software. We implemented our system with C++ using Intel SGX SDK v2.0. The implementation of the ORAM controller is built on top the Zerotrace [42] implementation. In order to handle the communication with the Bitcoin network, we have used `libjson-rpc-cpp` [6] framework to build C++ wrapper functions to communicate with the Bitcoin daemon (`bitcoind` [1]) from inside the enclave through JSON-RPC calls. For extracting the UTXO database, we used the `bitcoin-tool` implementation proposed in [21]. This allows us to save time during the initialization phase. Finally, we used `python-bitcoinlib` [8] to compare the performance of T^3 with the current existing SPV solution.

Database. To reduce the time of initializing both ORAM trees from the *genesis* block, we used `bitcoin-tool` implementation proposed in [21] to extract 3.2GB of the Bitcoin UTXO set in February 2019.

We have downloaded a snapshot of the Bitcoin blockchain including block 0 to $551, 731$, containing a total of $58, 156, 895$ Unspent Transaction Outputs (UTXO). Figure 7 shows the distribution of the unspent transaction outputs per address. We see that more than 90% of the addresses have less than three UTXOs. In our prototype, we considered at most two UTXOs per wallet ID. This results in covering more than 92% of all the UTXOs per wallet ID. Also, as discussed in section IV, by using different mapping, one can cover more percentage of Bitcoin addresses.
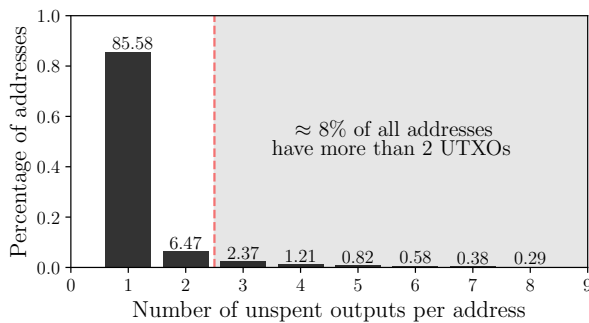

Fig. 7: Number of transactions per wallet ID. By allowing each address can have up to 2 UTXO, T^3 can cover approximate 92% of the UTXO set.

Hardware. We evaluated the performance of T^3 on a desktop which is equipped with Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz, 128GB RAM. Since Intel(R) Xeon(R) silver 4116 is not SGX-enabled CPU, we obtain the performance results by running our implementation in the simulation mode. However, we expect to not have much of a performance difference when executing in the two different modes. More specifically, we have tested the performance of T^3 using a smaller ORAM tree in the hardware mode on a commodity desktop equipped with SGX-enabled Intel Core i7. Comparing the hardware and simulation mode results (i.e., simulation on the Intel Core i7 CPU), we see no noticeable difference in the running time of both *read-once* and standard ORAM accesses.

B. Experimental Results

We have implemented T^3 using multiple threads. As reported in [27], [45], as long as the total amount of memory used by all threads does not exceed the EPC limit, the performance gain should be similar to the use of different enclaves. In this work, we implemented all functionalities in one single enclave, and we used multiple threads to concurrently accesses the ORAM trees.

System parameters We tested our system with both recursive PATH-ORAM and recursive CIRCUIT-ORAM using different tree size $N = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$. We allow each Bitcoin address to have up to 2 unspent transaction outputs, and we use the single address into single ORAM block mapping approach described in section IV-A to map addresses into ORAM block. Finally, we use claim 1 to determine the size of each ORAM block.

Performance of *read-once* and standard ORAM accesses. In T^3, the *reading* enclave performs *read-once* accesses to handle client's requests in an efficient manner. Table I presents an overall performance of a standard ORAM access as well as the performance of a *read-once* access for both CIRCUIT-ORAM and PATH-ORAM. For this experiment, we took the average running time of 10000 accesses.

As shown in the results, ORAM constructions with smaller block sizes provides a better performance in both schemes. The reason is that oblivious operations like oblivious comparisons and cmov-based stash scan are more efficient because of a smaller size stash. Moreover, CIRCUIT-ORAM gives a better performance compared to PATH-ORAM, as it can operate on a smaller block compared to PATH-ORAM, and this requires much smaller stash size allowing much faster oblivious execution.

Parallelization. Since there is no race condition in the *read-once* accesses, the design of T^3 allows different threads to concurrently perform *read-once* accesses on the *read-once* ORAM tree. Compared to other oblivious system like BITE [42], T^3 is able to handle bursty client read requests concurrently while the eviction requests are distributed sequentially during the *block creation interval*. To measure this performance gain, we used multiple threads to access the *read-once* enclave and perform *read-once* access simultaneously on a tree of size $N = 2^{24}$ and ORAM block of size 544 bytes. Table II shows the performance of T^3 implemented using multiple threads for both CIRCUIT-ORAM and PATH-ORAM.

N	Block Size	T^3 (PATH-ORAM, $Z = 4$)		T^3 (CIRCUIT-ORAM, $Z = 2$)	
		read-once Access	Standard ORAM access	*read-once* Access	Standard ORAM access
2^{20}	6528 bytes (96 utxos)	16.34 ms	30.40 ms	2.13 ms	6.45 ms
2^{21}	3264 bytes (48 utxos)	9.24 ms	16.58 ms	1.27 ms	3.76 ms
2^{22}	2176 bytes (32 utxos)	7.56 ms	12.42 ms	1.05 ms	2.92 ms
2^{23}	1088 bytes (16 utxos)	4.12 ms	7.78 ms	0.72 ms	2.09 ms
2^{24}	544 bytes (8 utxos)	2.43 ms	5.89 ms	0.64 ms	1.70 ms

TABLE I: Performance of two different types of PATH/CIRCUIT-ORAM accesses on different block size.

Comparison to current SPV solutions. We give a comparison in term of performance and communication overhead over several number of requests to the existing SPV client's solution and to BITE [9] Oblivious database.

1) *Performance*: Figure 8 gives us an overview of the performance of T^3 compared to the performance of the current existing SPV with Bloom filter solution and the performance of Bite Oblivious database. In particular, it shows the response latency from the client's perspective. In this comparison, a request for the SPV solution with Bloom filter solution means the time the server takes to scan one Bitcoin block, and a request for T^3 and BITE means the time it takes to perform an ORAM access on the ORAM tree. For the current SPV clients with Bloom filter, we set the false positive rate of the Bloom filter to 1.0% and 5.0% respectively. For T^3, we used $N = 2^{24}$ and block of size 544 bytes for both PATH-ORAM with $Z = 4$ and CIRCUIT-ORAM with $Z = 2$. For BITE database, based on our understand of their construction, we re-implemented BITE using non-recursive construction of PATH-ORAM, and we used the same ORAM block of size 32kB which leads to the number of block is $N = 2^{17}$. Also, we also provide an additional construction of BITE which is implemented using recursive PATH-ORAM and suggested parameters for T^3 where the tree is of size 2^{24} and block of size 544B. Figure 8 gives us the overall performance of three existing solutions.

The performance of T^3 outperforms the SPV with Bloom filter solution. The reason is that in T^3, the system relies on the TEE to handle the integrity checking of the Bitcoin block before updating the ORAM tree while in the current SPV solution, the full client needs to scan the Bloom filter every time and detect the relevant transactions and recompute the Merkle path for each of those transactions.

Also, T^3 performs much better than BITE oblivious database as the BITE system does not consider the use of recursive ORAM construction. Another reason is that the size of the ORAM block used in Bite is large; hence, the cost of oblivious operation like cmov-based stash scan becomes more expensive. Thus, we envision and realize an improved construction of BITE using recursive construction of PATH-ORAM to demonstrate the practical impact of using recursive ORAM

Number of threads	T^3 (PATH-ORAM)	T^3 (CIRCUIT-ORAM)
1	2.43 ms	0.64 ms
2	1.40 ms	0.58 ms
3	0.90 ms	0.43 ms
4	0.73 ms	0.35 ms

TABLE II: Performance gain of multiple-thread *read-once* access on Path/CIRCUIT-ORAM with $N = 2^{24}$ block size = 544 bytes.

construction on TEE with restricted memories.

2) *Communication Overhead*: In term of communication between client and server, T^3 offers much lower communication overhead compared to the existing solution for SPV clients. T^3 does not need to provide the SPV clients with the Merkle proofs to its relevant transactions because all those proofs are validated by the Intel SGX before being added the ORAM tree. Thus, one can reduce both the amount of work that the full node needs to perform and the amount of data that it needs to send to the SPV clients. Moreover, T^3 prunes all other information of transactions to extract only relevant data needed for client to determine balance and form new transactions, while in the current SPV solution, due to the false positive rate used in the Bloom filter, the full client may send additional irrelevant information to the SPV client. Figure 9 shows an overview of the communication cost of T^3 compared to the current solution. To give an estimation of the communication cost of the current SPV solution, we assumed that each request requires a separate Merkle proof. Hence, for each request, the size of the proof is at least: $\log_2(\text{NoTXs}) \cdot 32$ bytes where NoTXs is the number of transactions in one block. Moreover, the size of the transaction data is approximately $\text{fpr} \cdot \text{BlockSize}$ bytes where the fpr is the false positive rate and the BlockSize is the size of the Bitcoin block. To compute the overhead cost we used block 551731, as an example, which has block size of 1149 KB and contains 3017 transactions. However, in practice, we would expect the Bitcoin blocks to have different sizes; resulting, the communication cost to be different across blocks. Therefore, the results in fig. 9 is only an estimation on the communication overhead using the current SPV solution. We omit the comparison to the communication overhead of BITE because both T^3 and BITE return a fixed amount a data to the SPV client which is the output itself.

Storage Overhead As noted in the previous section, using ORAM incurs a constant size blow up of the storage of the
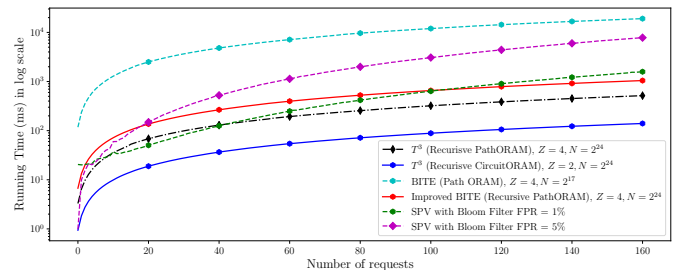

Fig. 8: Performance of T^3 using PATH/CICRUIT ORAM with block of size 544B, the current SPV with Bloom filter, Original BITE oblivious database block of size 32kB, and improved BITE with block of size 544B. For the SPV client with Bloom filter, we used the false positive rate of 1% and 5%.
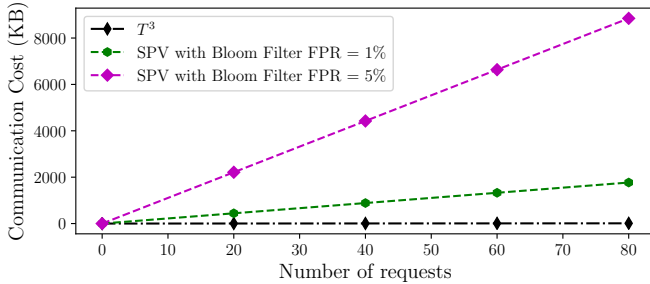
Fig. 9: Communication cost of T^3 and the current SPV solution. Since both systems return the information of unspent outputs to the client, the communication overhead of BITE will be equal to the communication overhead of T^3.

UTXOs (e.g., $\approx 3 - 4\times$ for CIRCUIT-ORAM, $6 - 8\times$ for PATH-ORAM). In particular, for PATH-ORAM with $Z = 4$, the storage cost of ORAM trees is about $\approx 51GB$, and for CIRCUIT-ORAM with $Z = 2$, the storage cost of two ORAM tree is around $\approx 26GB$. For the EPC memory usage, we need to consider the size of the position map and the stash used by each enclave. In this work, since we use recursive ORAM constructions for both schemes, the size of the position map can be as small as possible at the cost of storing more recursive ORAM trees in the untrusted memory region. Precisely, in the prototype of our implementation, each thread uses 8KB for the position map and a stash of size $2 \cdot \log(N) \cdot Z \cdot \mathsf{BlockSize}$ bytes (e.g., for a tree of size $N = 2^{24}$, we use $\approx 0.62MB$ bytes for PATH-ORAM, and ≈ 0.31 MB for CIRCUIT-ORAM). Thus, as long as the total memory usage by all threads/enclaves does not exceed the EPC limit (e.g., 96MB), the performance of the system will not suffer from the expensive swapping operations discussed in section III-A. Moreover, for integrity protection, T^3 only requires the server to store the Bitcoin header chain which is approximately 44MB instead of storing a complete Bitcoin blockchain. Thus, in the future work, if T^3 can handle the communication with the Bitcoin network without the reliance on the existing Bitcoin client, T^3 reduces the need of storing the 230 GB of Bitcoin blockchain.

C. Comparison with Other Oblivious Systems

Here we provide a comparison between T^3 and other generic oblivious systems. We compare our work with BITE [9] Oblivious Database that also uses ORAM and TEE to provide a generic PIR system for Bitcoin client, CONCURORAM [14] that provides concurrency access to ORAM clients, OBLIVIATE [10] that prevents leakage from file system accesses, and ZEROTRACE which proposes an efficient generic oblivious memory access primitives. In particular, section V-C compares those systems based on the capabilities of supporting concurrency access, enabling recursive construction, and preventing side-channel leakage.

For generic trusted hardware-based systems like BITE oblivious database and OBLIVIATE, while providing protection against side-channel leakage, those systems do not consider the use of recursive ORAM construction to reduce the EPC memory usage. Hence, the performance of their systems will degrade once the database becomes too large. Other works that harnesses the use of recursive ORAM construction are ZERO-TRACE; however, concurrency is not supported in the current

version of ZEROTRACE. Thus, without concurrency support, such systems will not scale well to handle Bitcoin SPV clients. CONCURORAM is a recent ORAM construction that offers concurrency accesses from the clients; however, due to more optimized eviction strategy and complex synchronization schedule, the recursive construction of CONCURORAM introduces implementation challenges. Nevertheless, we believe that it can be an interesting future work to use CONCURORAM in the design of T^3.

VI. SYSTEM ANALYSIS

A. Security Claims

In order to prove the security properties of T^3's design, we put forth six claims, each of which represents the security of a major component of T^3 in term of privacy goal.

Claim 1. The managing enclave does not leak user-related information to an attacker. The managing enclave is responsible for three tasks — (a) converting wallet IDs to UTXOs, (b) creating and managing threads which will perform read operations on the *read-once* ORAM tree, and (c) handle the updates to be performed on the *original* ORAM tree.

Firstly, the conversion of wallet IDs to their respective UTXOs is private since the channel between clients and the *managing* enclave is secured by the shared key during the remote attestation process. More importantly, when receiving addresses from a client, the *managing* enclave uses blockmapping function (described in IV-A1) to map each address to a fixed number of ORAM blocks. This does not reveal information about the number of outputs belonging to an address. Secondly, each read thread performs the same operations irrespective of the wallet ID provided to it, i.e., each thread simply retrieves an ORAM block using ORAM accesses implemented with cmov-based oblivious executions. Lastly, the only thing revealed by the update process of T^3 is the number of blocks updated into the Write Tree. However, this is public information and T^3 does not try to hide it. Each update is performed using an ORAM access which ensures that the attacker is unaware of the final position of each block.

Claim 2. The optimized read operations on *read-once* ORAM tree do not leak information. As explained in section IV-B2, the *read-once* ORAM tree is accessed using an optimized read operation which chooses not to shuffle and write-back the retrieved path to the *read-once* ORAM tree. However, this is secure since each path corresponding to a UTXO can only be accessed once during a read interval and will be shuffled before the next interval.

Claim 3. The write operations performed on the *original* ORAM tree do not leak information. There are two specific

System	Capabilities		
	Concurrency	Recursive Construction	Side-channel Protection
CONCURORAM [14]	✓	✗	- [a]
OBLIVIATE [10]	✗	✗	✓
ZEROTRACE [42]	✗	✓	✓
BITE Oblivious Database [9]	✗	✗	✓
T^3	✓	✓	✓

[a] CONCURORAM does not aim to provide side-channel protection for TEE. Hence, we omit this comparison.

TABLE III: Comparison between T^3 and other oblivious systems.

operations performed on the *original* ORAM tree— (a) the UTXOs are updated based on the updated bitcoin block, and (b) the previously accessed ORAM blocks are shuffled. However, all of these updating accesses are standard ORAM operations implemented in a side-channel-resistant manners as previously done by [42], [10]. Therefore, all write operations reveal no information about a user's UTXO.

Claim 4. The data fetched from the untrusted world to the TEE is correct. There are two major sources of data transferred from the untrusted to the trusted world — (a) the updated block fetched from the bitcoin daemon after a fixed interval and (b) the ORAM tree blocks which are fetched from the untrusted world into the TEE. As mentioned in IV-B3, Bitcoin blocks are fetched from outside the enclave. However, T^3 verifies the integrity of the Bitcoin block based on the proof of work and the header chain, and since the cost of producing a valid block is expensive, we argue that T^3 should be able to obtain valid block from the Bitcoin network. Also, T^3 maintains a Merkle Hash Tree (MHT) of the ORAM trees and therefore prevents malicious tampering by verifying all encrypted data fetched from the untrusted memory using the MHT. All encrypted data fetched from the untrusted memory is verified using the MHT.

Claim 5. The multiple threads involved do not create synchronization issues. Here, it is worth-noting that multiple threads are only involved while accessing the Read Tree of T^3. Thanks to the optimized read operation, T^3 does not run into synchronization bugs since there is no memory region that could be simultaneously written to by more than one thread. In particular, each thread shares the position map but only reads from the position map. Each thread contains its own stash memory which is written to separately by each thread.

Claim 6. The memory interactions within the enclave are side-channel-resistant. The design of T^3 incorporates defenses against the side-channel threats [51], [31], [32] plaguing Intel SGX. In particular, we used ORAM operations to hide all data access patterns on the untrusted memory region, and we incorporated similar oblivious operation techniques introduced in [40], [10], [42] to prevent operations inside the enclave from leaking sensitive information. Finally, the implementation of T^3 is also secure against branch-prediction attacks since each individual operation (e.g., accessing Read Tree, updating Write Tree etc.) takes the same sequence of branches and therefore reveals no information to the attacker, from the accessed branches.

B. Denial of Service Attacks from Malicious Clients

While the design of T^3 is pratical, a malicious client can still incur a large processing time on the server by creating lots of addresses and sending large number of requests for those requests. One way to mitigate such attack is to apply fees on users of the service. Another approach to mitigate denial of service attack is to use a cuckoo filter [24] to load all addresses from the UTXO set. Upon receiving requests from client, the managing enclave can verify if the address matches the filter as well as the proof of ownership of that address before performing ORAM accesses. Moreover, since Cuckoo filter data structure supports deletion operation, the system can add and remove addresses when performs updating. In other word, in order to perform the denial of service attack, clients need both the proof of ownership as well as a certain amount of Bitcoin in each address. Hence, it will cost more for the client to perform such attack.

C. Other Goals Achieved by T^3

In this subsection, in addition to the **Privacy** goal describe in section VI-A, we explain how T^3 achieves the other goals mentioned in subsection II-B.

Validity. Under the assumption that the adversary does not have enough computational power to form a new Bitcoin block, the system will only obtain valid transaction by verifying the Merkle root and the proof of work of the Bitcoin block.

Completeness. By offering different ways of mapping between Bitcoin addresses and ORAM block id, we can offer services to $92-96\%$ of all clients with some trade-off between storage overhead and performance.

Efficiency. Our contribution to efficiency is threefold. First, our system is able to handle bursty requests from client concurrently. The core idea is to separate the effect of a standard ORAM access into different enclaves. Thus, the multiple reading enclave can concurrently perform read operations at the same time that the writing enclave can perform a non-blocking Evict procedure on the other tree. Second, by having two ORAM trees, we minimize the downtime of the system by having the writing enclave performed updates on one tree and reading enclave handled clients' requests on the other tree. The server downtime depends on the number of requests that the system receives when the writing enclave performs ORAM updates on the original ORAM tree. Finally, by enforcing clients to provide the proof of ownership of the address and assuming that a honest client is rational, we prevent other clients from querying addresses that do not belong to them.

VII. Related Work

General SGX Systems. Haven [12] is a pioneering work on SGX computing enabling native application SGX porting on windows. Graphene [15] provides a linux-based LibOS for SGX programs. Ryoan [29] retrofits Native Client to provide sandboxing mechanisms for Intel SGX. Eleos [38] provides a user-space extension of enclave memory using custom encryption. T^3 uses some concepts from Eleos especially in the way we store the ORAM tree using custom encryption outside the SGX enclave.

SGX Side-channels. There are three main memory-based side-channel vulnerabilities disclosed within Intel SGX, namely, page table-based attacks [51], cache-based attacks [13], and branch-prediction attacks [31]. Furthermore, since SGX relies on the untrusted OS for system-call handling, it is also vulnerable to IAGO attacks [16]. Leaky Cauldron [48] presents an overview of the possible attack vectors against SGX programs. T^3 is secure against all disclosed memory-based side-channels since it uses oblivious RAM (ORAM) to protect the access-patterns. Furthermore, T^3 uses oblivious memory primitives to secure the runtime ORAM operations as well as its library.

Oblivious Systems. Raccoon [40] provided a technique to protect a small part of a user program against all digital

side-channels. OBLIVIATE [10] and ZEROTRACE [42] used ORAM-based operations to protect files and data arrays respectively inside Intel SGX. Thang Hoang et al. [28] proposed a combination of TEE and ORAM to design oblivious search and update platform for large dataset. Eskandarian et a. [23] leveraged Intel SGX and Path ORAM to propose oblivious SQL database management system.

Recently, Chakraborti et al. proposed a new parallel ORAM scheme called ConcurORAM [14]. Similar to the T^3 design, ConcurORAM also uses two-tree structure to propose a non-blocking eviction procedure, and the system periodically synchronizes two trees to maintain the privacy of the user's access pattern. In ConcurORAM, the scheme requires the client to download the query log and the result log to learn about ongoing queries before requesting ORAM accesses. Hence, if we combine ConcurORAM along with Intel SGX to design this system, the use of query and result logs introduces additional storage overhead to the limited storage capacity of the Intel SGX. More importantly, the author also noted that ConcurORAM cannot be trivially extended to a recursive ORAM construction because of concurrent data structure accesses. However, if ConcurORAM can be implemented into a recursive ORAM construction, we believe that ConcurORAM can be an interesting alternate solution for the ORAM scheme used in the design of T^3.

Another interesting parallel ORAM construction is TaoStore [41]. TaoStore assumes a trusted proxy that handles concurrent client's requests, and the proxy runs a scheduler to make sure that there are no conflicting queries while preventing no information leakage. However, similar to ConcurORAM, the implementation of TaoStore is limited to the non-recursive construction of Path ORAM which is not suitable when combining with TEE with limited trusted memory capacity. This work aims to design a simpler design that is suitable for any flavor of tree-based ORAM schemes.

TEE for cryptocurrencies. The research community has investigated different ways of combining TEE with blockchain to both improve privacy and scalability of blockchains. Obscuro [46] is a Bitcoin transaction mixer implemented in Intel SGX that addresses the linkability issue of Bitcoin transactions. Teechan [33] is an off-chain payment micropayment channel that harnesses TEE to increase transaction throughput of Bitcoin. Bentov et al. proposed a new design that uses Intel SGX to build a real-time cryptocurrency exchange. Another example is the Towncrier system [52] that uses TEE for securely transferring data to smart contract. Another prominent example is Ekiden [17] which proposed off-chain smart contract execution using TEE. Finally, ZLite [50] system is another example which used ORAM and TEE to provide SPV clients with oblivious access. However, similar to BITE, ZLite employed non-recursive PATH-ORAM as it is, and thus, the scalability and efficiency of the system is inherently limited due to the non-concurrent accesses.

Osuntokun et al. [39] recently present a new proposal for Bitcoin SPV clients. This proposal is the building block for systems like Neutrino. In particular, each block will have its own Bloom filter. The SPV client first fetches the filter from the full client and decides to download the block from another client if transactions of interested are in the block. This approach, however, introduces an additional communication overhead to the client. In particular, a client with lots of transactions scattered among different blocks needs to download lots of full blocks, and performing verification of block can be expensive for the resource-constrained client. This approach does not necessarily provide more privacy for the SPV client as the full client still learn the block that the addresses belong to.

VIII. CONCLUSION

In this paper, we developed a system design that supports a large-scale oblivious search on unspent transaction outputs for Bitcoin SPV clients while efficiently maintains the state of the Bitcoin UTXO set via an oblivious update protocol. Our design leverages the TEE capabilities of Intel SGX to provide strong privacy and security guarantees to Bitcoin SPV client even with the presence of a potentially malicious server. Moreover, by putting reasonable assumptions on the accessing frequency of the SPV clients, we present novel ORAM construction that offers both privacy and efficiency to the clients. We showed that the prototype of the system is much more efficient than the use of standard ORAM construction as it is. In particular, due to the use of two ORAM trees in the design of T^3, we improve the performance of an ORAM access by two time and allow the system to handle concurrent client's requests. Also, our implementation shows one order of magnitude performance gain when combining recursive ORAM construction the current existing construction to stress the importance of using recursive ORAM construction in TEE with restricted memory. Finally, while the applicability of T^3 in cryptocurrencies beyond Bitcoin is apparent, we believe our work will motivate further research on oblivious memory with the restricted access patterns.

REFERENCES

[1] Bitcoin core. https://bitcoin.org/en/bitcoin-core/. Accessed in Nov 2018.

[2] Bitcoin Developer Reference. https://bitcoin.org/en/\developer-reference. Accessed in Nov 2018.

[3] Bitcoin difficulty and network hash rate. https://bitcoinwisdom.com/bitcoin/difficulty.

[4] Bitcoinj. https://bitcoinj.github.io/. Accessed in Oct 2018.

[5] Electrum Bitcoin Wallet. https://electrum.org/. Accessed in Oct 2018.

[6] json-roc-cpp. https://github.com/cinemast/libjson-rpc-cpp. Accessed in Oct 2018.

[7] Key stone project. https://keystone-enclave.org/. Accessed in Oct 2018.

[8] python-bitcoinlib. https://github.com/petertodd/python-bitcoinlib. Accessed in April 2019.

[9] BITE: Bitcoin lightweight client privacy using trusted execution. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.

[10] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *NDSS*. The Internet Society, 2018.

[11] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, 2016. USENIX Association.

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, 2014. USENIX Association.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[14] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client ORAM. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[15] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, 2017. USENIX Association.

[16] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *SIGARCH Comput. Archit. News*, 41(1):253–264, March 2013.

[17] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *CoRR*, abs/1804.05141, 2018.

[18] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[19] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, 2016. USENIX Association.

[20] Artur Czumaj. Lecture notes on approximation and randomized algorithms. http://www.ic.unicamp.br/~celio/peer2peer\/math/ czumaj-balls-into-bins.pdf.

[21] Sergi Delgado-Segura, Cristina Pérez-Solà, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. Analysis of the bitcoin utxo set. Cryptology ePrint Archive, Report 2017/1095, 2017. https://eprint.iacr.org/2017/1095.

[22] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.

[23] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017.

[24] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, New York, NY, USA, 2014. ACM.

[25] Arthur Gervais, Srdjan Capkun, Ghassan O. Karame, and Damian Gruber. On the privacy provisions of bloom filters in lightweight bitcoin clients. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 326–335, New York, NY, USA, 2014. ACM.

[26] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.

[27] Danny Harnik, Eliad Tsfadia, Doron Chen, and Ronen I. Kat. Securing the storage data path with SGX enclaves. *CoRR*, abs/1806.10883, 2018.

[28] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A. Yavuz. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset (to appear). In *In Proceedings on Privacy Enhancing Technologies Symposium (PoPETs)*, 2019.

[29] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 533–549, Savannah, GA, 2016. USENIX Association.

[30] Angela Jäschke, Björn Grohmann, Frederik Armknecht, and Andreas Schaad. Short paper: Industrial feasibility of private information retrieval. In *SECRYPT*, pages 395–400, 2017.

[31] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, 2017. USENIX Association.

[32] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association.

[33] Joshua Lind, Ittay Eyal, Peter R. Pietzuch, and Emin Gün Sirer. Teechan: Payment channels using trusted execution environments. *CoRR*, abs/1612.07766, 2016.

[34] Matt Corallo Mike Hearn. Connection bloom filtering, 2012.

[35] Mohsen Minaei, Pedro Moreno-Sanchez, and Aniket Kate. R3c3: Cryptographically secure censorship resistant rendezvous using cryptocurrencies. Cryptology ePrint Archive, Report 2018/454, 2018. https://eprint.iacr.org/2018/454.

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

[37] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multiparty machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, Austin, TX, 2016. USENIX Association.

[38] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 238–253, New York, NY, USA, 2017. ACM.

[39] Olaoluwa Osuntokun, Alex Akselrod, and Jim Posen. Client side block filtering.

[40] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., 2015. USENIX Association.

[41] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 198–217, 2016.

[42] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[43] Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 197–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[44] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[45] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *International Conference on Learning Representations*, 2019.

[46] Muoi Tran, Loi Luu, Min Suk Kang, Iddo Bentov, and Prateek Saxena. Obscuro: A bitcoin mixer using trusted execution environments. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 692–701, New York, NY, USA, 2018. ACM.

[47] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[48] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. *CoRR*, abs/1705.07289, 2017.

[49] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 850–861, New York, NY, USA, 2015. ACM.

[50] Karl Wüst, Sinisa Matetic, Moritz Schneider, Ian Miers, Kari Kostiainen, and Srdjan Capkun. Zlite: Lightweight clients for shielded zcash

transactions using trusted execution. Cryptology ePrint Archive, Report 2018/1024, 2018. https://eprint.iacr.org/2018/1024.

[51] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel attacks: Deterministic side channels for untrusted operating systems. In *S&P (Oakland)*, 2015.

[52] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 270–282, New York, NY, USA, 2016. ACM.