BITE: Bitcoin Lightweight Client Privacy using Trusted Execution

Sinisa Matetic
ETH Zurich

Karl Wüst
ETH Zurich

Moritz Schneider
ETH Zurich

Kari Kostiainen
ETH Zurich

Ghassan Karame
NEC Labs

Srdjan Capkun
ETH Zurich

Abstract

Blockchains offer attractive advantages over traditional payments such as the ability to operate without a trusted authority and increased user privacy. However, the verification of blockchain payments requires the user to download and process the entire chain which can be infeasible for resource-constrained devices like mobile phones. To address this problem, most major blockchain systems support so called lightweight clients that outsource most of the computational and storage burden to full blockchain nodes. However, such verification leaks critical information about clients' transactions, thus defeating user privacy that is often considered one of the main goals of decentralized cryptocurrencies.

In this paper, we propose a new approach to protect the privacy of light clients in Bitcoin. Our main idea is to leverage the trusted execution capabilities of commonly available SGX enclaves. We design and implement a system called BITE where enclaves on full nodes serve privacy-preserving requests from light clients. However, as we will show, naive processing of client requests from within SGX enclaves still leaks client's addresses and transactions. BITE therefore integrates several private information retrieval and side-channel protection techniques at critical parts of the system. We show that BITE provides significantly improved privacy protection for light clients without compromising the performance of the assisting full nodes.

1 Introduction

Since its inception in 2008, Bitcoin has fueled considerable interest in decentralized currencies and other blockchain applications. The main goals of blockchains include a distributed trust model and increased user privacy. Several other blockchain platforms, such as Ethereum [4], leverage the same open or permissionless model as Bitcoin, while platforms like Hyperledger [15], Ripple [10] and R3 [9], enable closed or permissioned blockchains. Most blockchains implement a decentralized time-stamping mechanism that ensures eventual consistency of *transactions* by collecting them from the underlying peer-to-peer (P2P) network, verifying their correctness, and including them in connected blocks. This process imposes heavy requirements on bandwidth, computing, and storage resources of blockchain nodes that need to fetch all transactions and blocks issued in the blockchain, locally index them, and verify their correctness against all prior transactions. For instance, a typical Bitcoin installation requires more than 200 GB of storage today, and the sizes of popular blockchains are growing fast [12, 5]. Therefore, users operating resource-constrained clients like mobile devices cannot afford to run their own full node.

Lightweight clients and privacy. To address such heavy resource requirements, most open blockchain platforms support *lightweight clients*, targeted for devices like smartphones, that only download and verify a small part of the chain. As a matter of fact, according to [24], in $73-85\%$ of $5.8-11.5$ million active Bitcoin wallets users control keys. Since there are $\sim 10,000$ full nodes [11], estimated 4.2-9.8 million wallets are lightweight clients. For example, Bitcoin provides the BitcoinJ [2], PicoCoin [8] and Electrum [3] clients implementing the Simple Payment Verification (SPV) mode [44], where the clients connect to a full node that has access to the complete chain and assists the client in transaction confirmation. Transactions contain inputs and outputs that are bound to *addresses* owned by users. As the full node has to learn all transactions issued and received by the requesting client to confirm them, such payment verification obviously violates user privacy.

To improve user privacy, several clients support *filters* (e.g., Bitcoin's BIP37 [31] and Ethereum's LES [6]). The goal of filters is to allow the client to define an anonymity set in an attempt to hide its real addresses from the full node. For instance, BIP37 supports Bloom filters [18] that allow the client to define a set of transactions, with false positives, that are requested from the full node. Essentially, this approach presents a trade-off between communication efficiency and privacy: a filter that returns many false positives provides a

larger anonymity set but requires more communication. Although such filters can be configured to be efficient, recent studies have shown that in practice they offer almost no privacy [25]. Ergo, none of the current light clients provides adequate privacy with practical performance overhead.

Our solution. Our goal is to improve the privacy of Bitcoin lightweight clients without compromising the performance of the assisting full nodes. The starting point of our solution is to leverage the commonly available trusted computing capabilities of SGX enclaves [23] on full nodes. We propose BITE (for ***BI**tcoin lightweight client privacy using **T**rusted **E**xecution*), a solution in which a potentially untrusted entity runs a full node with an SGX enclave that serves transaction confirmation requests from clients. Since SGX provides code integrity and data confidentiality for enclaves, such a solution can preserve privacy (confidentiality) and completeness (integrity) of client requests.

Unfortunately, simple usage of trusted computing is not sufficient to solve our problem. While SGX prevents an adversary that controls malicious software from directly accessing enclave's memory, secret-dependent access patterns to external storage, such as transaction databases, can reveal the client's address. SGX is also susceptible to side-channel attacks, where malicious software on the same platform infers secret-dependent enclave data access patterns or control flow by monitoring shared resources like caches [20, 43, 27, 50]. Thus, the simple usage of SGX would still leak the client's addresses to malicious full node.

Given such limitations of SGX, *the primary research problem and contribution of this paper is how to design and implement a solution that enables private processing of light client request in the presence of enclave leakage without compromising the system's overall performance.* To address this non-trivial challenge, we carefully select and apply known private information retrieval (PIR) and side-channel protection techniques and combine them into a novel solution that meets our performance requirements. We emphasize that in our application the assisting full node needs to process a large blockchain database to serve client requests, and thus straightforward usage of generic SGX side-channel protection systems, such as Raccoon [47], Cloak [28] or ZeroTrace [49], would result in either excessive performance overhead or imperfect side-channel protection. Instead of using such systems directly, we pick low-level primitives and apply them at critical points in our system to achieve more complete protection and better performance.

We design two variants of our solution. Our first variant, *Scanning Window*, is similar to the current SPV clients that verify transactions using block headers and Merkle paths received from the full node. To prevent leakage from file accesses and message sizes, we design a customized chain access mechanism that hides the client's transactions and the relationship between the size of the response and the number of read blocks. Our second variant, *Oblivious Database*, allows the client to verify the amount of coins associated with its addresses by querying a specially-crafted version of the unspent transaction output (UTXO) database. To prevent leakage from database accesses, we leverage a well-known Oblivious RAM (ORAM) algorithm [52]. (Prior to us, usage of ORAM from enclaves has been proposed in systems like ZeroTrace [49].) This variant allows even *lighter* clients that no longer need to download and verify Merkle paths.

To prevent software-based side-channels, we adopt further protections from recent SGX research. The basic building block for our control-flow hiding is the `cmov` instruction [7] that enables building oblivious execution of branches. (We adopt this technique from the Raccoon system [47].) To prevent leakage from data access patterns we apply additional defenses, such as iterating over the entire data structure when an element is accessed based on the protected client address.

Results. We show that our solution provides strong privacy protection. In both of our variants, the external data access patterns are independent of the protected client address. The side-channel protections in the Oblivious Database variant also make the enclave's memory accesses (both code and data) independent of the address, thus preventing leakage caused by known SGX side-channels [20, 43, 27, 50, 58, 36]. While similar protections can also be used for the Scanning Window variant, they impose a high overhead, which is why we recommend using Oblivious Database if side-channels are a concern. Our solutions also fail gracefully: even if the used SGX processor would be completely broken (e.g., through a physical attack), the adversary cannot double spend or steal users' coins or wallets.

In terms of performance, our solution is comparable to the SPV scheme. The Oblivious Database variant increases the full node's *storage* moderately (e.g., additional 4 GB). The required *communication* is significantly lower (e.g., 12 KB instead of 17 MB per client request). The *processing* cost for incoming client requests is reduced (e.g., 0.5s instead of 1.1s), but the processing cost for new blocks is higher (79s instead of 2s). Even compared to SPV without privacy protection, our solution adds no processing time or communication overhead (in fact, BITE's processing is faster by 0.1s and the response size is 2kB smaller). The full node can be easily made responsive for incoming client request during block updates by using two enclave instances in parallel. The Scanning Window variant requires no additional storage and its communication cost is lower than in SPV. The processing cost is also comparable when full side-channel protection is not used.

We argue that BITE emerges as the *first* practical solution that provides strong privacy protection for lightweight Bitcoin clients. Our solution can be integrated into existing full nodes and lightweight clients with minor modifications to the existing software. While BITE is designed for Bitcoin,

we stress that it finds direct applicability in various other blockchain platforms as well.

Contributions. In summary, in this paper we make the following contributions:

• *Novel approach.* We propose leveraging commonly available trusted execution capabilities of SGX enclaves for improved lightweight Bitcoin client privacy.

• *New system.* We design and implement a system called BITE that carefully combines a number of PIR and side-channel protection techniques to prevent leakage.

• *Evaluation.* We show that BITE significantly improves client privacy without compromising full node performance. We argue that BITE is the first practical way to provide strong privacy for lightweight Bitcoin clients.

The remainder of this paper is organized as follows. Section 2 describes our problem and Section 3 outlines our approach. Section 4 explains the details of our system BITE. Section 5 covers security analysis and Section 6 provides performance results. We provide discussion in Section 7, review related work in Section 8, and conclude in Section 9.

For readers unfamiliar with SGX and ORAM, we provide brief introductions in Appendices A and B.

2 Problem Statement

In this section, we provide background on Bitcoin lightweight clients, explain the limitations of known approaches and define requirements for our solution.

2.1 Bitcoin Lightweight Clients

Bitcoin [44] is the first and still most popular cryptocurrency based on blockchain technology. It enables users to perform payments by issuing *transactions* that transfer Bitcoins (BTC) from one or more transaction inputs to one or more outputs. Each of the outputs is bound to an *address* that is derived from a user's public key. A user that knows the corresponding private key is able to spend the Bitcoin contained in the transaction output.

When a user wants to perform a payment, she creates a transaction that contains inputs, outputs, and the signatures that allow her to spend the inputs. Subsequently, the transaction is propagated to all nodes using a peer-to-peer network created by the system's participants. Miners, a special type of nodes, collect valid transactions into blocks and solve Proof-of-Work (PoW) puzzle to make the contained transactions hard to revert. A miner that successfully finds a valid PoW, broadcasts the block to all other nodes, who then verify its correctness and include it in their copy of the chain.

To verify transactions, Bitcoin users, or clients, need to store the full history of all Bitcoin transactions. This approach puts a heavy load on client implementations in terms of network and storage, and as a consequence, makes transaction confirmation on mobile clients infeasible. To address this concern, the original Bitcoin paper proposed a solution called *Simplified Payment Verification* (SPV) [44]. In this technique, light clients store only block headers, check their PoW puzzles and then request their own transactions and the Merkle paths that are needed to verify their presence in the blocks from a full node that stores the entire chain.

Improvement proposal BIP 37 [31] introduced Bloom filters [18] that allow a light client to request a subset of all transactions to preserve some privacy without needing to download all transactions for each block. A Bloom filter [18] is a probabilistic data structure that consists of a set of hash functions and a bit array where each bit is set to one if one of the hash functions hashes one of its inputs to the index of the bit in the array. This allows checking if a value is contained in the filter by hashing the value with each of the hash functions and checking whether the corresponding bit is set. If this is not true, the value was not an input. If it is true, however, the value *might* have been an input or a false positive. The false positive rate can be set by the creator of the filter.

In Bitcoin light clients, Bloom filters are used to encode transactions or addresses, and allow a full node to determine which transactions to send to a lightweight client without letting the full node know the exact addresses. A lightweight client prepares a Bloom filter to which she adds all of her addresses and sends it to the full node. The full node then checks for incoming (or past, if requested) transactions whether they match the Bloom filter. If they match, she sends them to the client together with the Merkle path needed for verification. The client can adjust the false positive rate to increase her privacy. If the false positive rate is higher, the client will receive more irrelevant transactions, in an attempt to hide her true addresses with a larger anonymity set.

2.2 Limitations of Known Solutions

The use of Bloom filters to receive Bitcoin transactions from an assisting full node inherently creates a trade off between performance and privacy. If a client increases the false positive rate she receives more transactions which provides increased privacy, as any of the matching addresses could be her real addresses, but it also means that she needs the network capacity to download all of these transactions. In the extreme cases, the filter matches everything, i.e., the client downloads the full blocks, or the filter only matches the client's addresses, i.e., she has no privacy at all.

Gervais et al. [25] have shown that using Bloom filters in Bitcoin light clients leaks more information than was previously thought. In particular, if the Bloom filter only contains a moderate number of addresses, the attacker is able to guess addresses correctly with high probability. For example, with 10 addresses the probability for a correct guess is 0.99. They also show that, even with a larger number of

addresses, the attacker is able to correctly identify a client's addresses with high probability if she is in possession of two distinct Bloom filters from the same client (e.g., due to a client restart). Hearn [30] later expanded on why solving these issues is hard (e.g., need for resizing). Furthermore, it is likely that an attacker using additional de-anonymization heuristics, such as the ones described in [16, 41], could further increase the probability to guess correctly.

Finally, a lightweight client cannot be sure that she receives *all* transactions that fit her filter from a full node. While the full node cannot include faulty transactions in the response, as this would be detected by the client when recomputing the Merkle root, the client cannot detect whether she has received all requested transactions. This problem can be solved by requesting transactions from multiple nodes, which again imposes more network load on the client.

Another solution would be to run the SPV protocol with Bloom filters over a network anonymity mechanism such as Tor. While this would prevent the full node from learning the IP address of the client, the full node could still correlate queries from the same client based on the addresses that leak from Bloom filters. We argue that query unlinkability is a useful privacy property in systems like Bitcoin and can be considered similar to the transaction unlinkability in systems like Zcash (that provide more advanced privacy protection).

2.3 Requirements

The high-level goal of this paper is to develop a solution that provides better privacy for lightweight clients without compromising the system performance. More precisely, our solution should meet the following requirements:

(**R1**) **Privacy.** Lightweight clients should be able to verify that their transactions are confirmed on the blockchain or check the amount of coins associated with their addresses without revealing their addresses to the potentially untrusted entity that controls the assisting full node. The full nodes should not be able to link queries from the same light client that could allow them to incur additional information regarding the client's transactional pattern or behavior.

(**R2**) **Completeness.** The verification process should guarantee that no valid transactions have been omitted.

(**R3**) **Performance.** The performance of the system should be comparable to or better than current light client schemes.

3 Our Approach

The main idea behind our approach is to leverage commonly available Trusted Execution Environments (TEEs) such as Intel's SGX enclaves [33, 23] running within full nodes to provide a privacy-preserving verification service to light clients. Besides increased privacy, TEEs can enable better

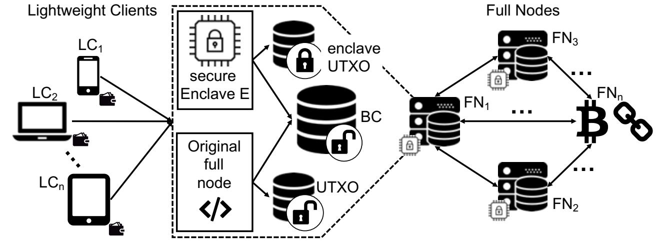


Figure 1: **System model.** Lightweight clients request transaction verification from enclaves hosted on full nodes.

performance in terms of reduced processing and bandwidth, and guarantee completeness of received responses.

In short, SGX provides a set of security enhancements in the processor that allow creation of small applications, called *enclaves*, whose data confidentiality and code integrity is protected from any malicious software running on the same platform, including the privileged OS.

A simple way to leverage SGX would be a solution where the light client sends its wallet private key to an enclave on the assisting full node. Using that key, the enclave can perform any operation on behalf of the user, including transaction verification. However, such simple solution has a critical drawback. If the used enclave is compromised, the adversary can steal all user's coins. Such approach might give the owners of full nodes an undesirable economic incentive to break their own SGX processors, e.g., using physical attacks.

To avoid such incentives, we choose a different approach. In our solution, when a client needs to verify a transaction or check the amount of coins associated with the user's addresses, the client connects to one of the full nodes that supports our service. The client performs remote attestation and establishes a secure channel to the enclave. Then, the lightweight client sends the addresses that the user is interested in to the enclave. The enclave obtains all the required verification information from the locally stored blockchain or custom unspent transactions database (UTXO) and sends back a response to the client that can verify it. Importantly, the client's private key is never shared with the enclave which enables safe adoption of our solution.

We envision two types of deployment for our system. In the first example deployment, a well-recognized company could provide such a verification service. In the second example, any volunteer currently running a Bitcoin full node could adopt our extension and start providing the service to lightweight clients. In both cases, to incentivize deployment by the full nodes, the service could be run in exchange for some small renumeration (i.e., verification fees).

3.1 System Model

Figure 1 shows our system model that consists of full nodes $FN_1...FN_m$ and lightweight clients $LC_1...LC_n$. When a lightweight client LC_i wants to acquire information about its

transactions or addresses, it can connect to any full node FN_j that supports our service and hosts an enclave E_j. Full nodes download and store the entire blockchain (BC) locally and based on that maintain a database that contains all unspent transaction outputs (UTXO). Our system additionally maintains a specially-crafted version of the UTXO, called *enclave UTXO*, in an encrypted (sealed) form.

In SGX, enclave memory is limited to 128MB. Although swapping memory pages is supported (swapping requires expensive encryption and integrity verification [17]), the complete blockchain (BC) and the database of unspent transaction outputs (UTXO) are significantly larger (as of Jan 2019, 200GB [12] and 2.8GB [13] or more, respectively) than the enclave's memory limits . Therefore, these databases are stored on the local persistent storage.

3.2 Adversary Model

We consider an adversary who controls the OS and any other privileged software on the full node. For example, the adversary could be a malicious administrator or an external attacker who has remotely compromised the OS on the full node. Since the adversary controls the OS, she can schedule and restart enclaves, start multiple instances, and block, delay, read, or modify all messages sent by enclaves, either to the OS itself or to other entities over the network. We assume that the adversary cannot break the hardware security enforcements of Intel SGX. That is, the adversary cannot access processor-specific keys (e.g., attestation or sealing key) and she cannot access enclave runtime memory that is encrypted and integrity-protected by the CPU. (Although we consider SGX trusted, in Section 5 we discuss enclave compromise and show that our solution can handle it without any financial loss.) Finally, we assume that common cryptographic primitives like encryption or signatures are secure.

3.3 Challenges

Secure and practical realization of our approach under the defined attacker model involves several technical challenges.

Leakage through external accesses. Since the adversary controls the OS, she can observe access patterns to any *external* resources, such as files or databases stored on the disk. Although externally stored data can be sealed (encrypted by the CPU such that only the same enclave can decrypt), the OS can infer information about the accessed element by observing access patterns to individual records, such as files or database entries. In a simple implementation of our approach, the adversary could infer the client's addresses by observing which entries the enclave reads from a (sealed) UTXO database when processing a client request.

Similarly, enclaves rely on the OS to perform communication operations which allows it to infer information about the enclave's communication patterns. Even if messages are encrypted by the enclave, the message sizes, frequency and destination can leak information. In our case, the adversary could determine how many transactions are included to the response by observing response sizes.

Leakage through side channels. The SGX architecture is also susceptible to *internal* leakage. Numerous, recently demonstrated side-channel attacks against SGX show that internal leakage is a relevant concern. For example, by monitoring CPU caches the OS can infer secret-dependent data and code accesses inside the enclave's memory [20, 43, 27, 50]. The OS can also infer enclave's secrets by monitoring the memory pages that the enclave requests [58]. Researchers have also demonstrated side-channel attacks using the CPU's branch prediction functionality [36]. In a simple implementation of our approach, the adversary can monitor address-dependent branching in the enclave's control flow and data accesses and thus determine the client's addresses.

4 BITE System

In this section we present a system called BITE that realizes the above approach securely and addresses the aforementioned challenges. In particular, we present two variants of the same approach that serve slightly different purposes.

Our first variant, *Scanning Window*, can be seen as an extension to the current SPV verification mode, but without reliance on bloom filters. Based on the client request, an enclave on the full node *scans* the blockchain and replies with a set of Merkle paths that the client can use to verify its transactions using downloaded block headers. This variant allows the client to check that each of its transactions are confirmed on the blockchain. As Bitcoin provides only eventual consensus, the client may want to additionally verify that the blocks where its transactions are placed have been extended with a sufficient number of valid blocks (e.g., six).

Our second variant, *Oblivious Database* is a completely new verification mode for lightweight clients. In this variant, the enclave on the full node maintains a specially-crafted version of the unspent transaction outputs (UTXO) database and when a client sends a verification request, it checks for the presence of client's outputs in this database using oblivious database access (ORAM [52]) and responds accordingly. Such verification allows the client to check how many coins are currently associated to its addresses, with significant performance improvements over SPV.

In both variants, the client performs remote attestation and establishes a TLS connection to the enclave. We note that current light clients communicate with the full nodes without encryption. Existing full node functionality, such as participation in the P2P network and mining, remain unaffected. Therefore, our system can be seen as a simple add-on to ex-

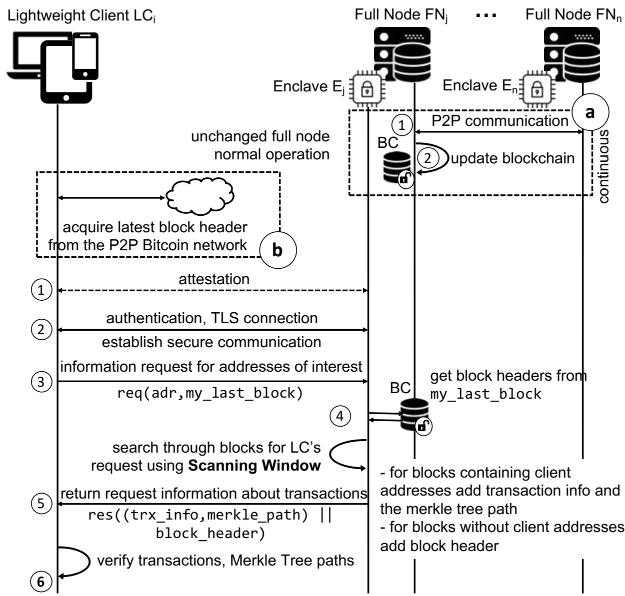


Figure 2: **Scanning Window operation.** Light client creates a secure connection to an enclave on full node and sends a request with its address and last known block. The enclave scans the locally stored chain and prepares a response with the size proportional to the number of scanned blocks.

isting full nodes. For clients, payment execution remains unchanged. Payment verification requires minor additions (attestation and TLS) when Scanning Window is used or slightly bigger changes with Oblivious Database variant.

4.1 Scanning Window Variant

In our first variant, we want to improve the privacy of the current SPV verification mode. When a client needs to verify transactions, it constructs a request that specifies the addresses of interest and the last block that it has in its internal state and sends that to the secure enclave residing on the full node. The enclave reads the locally stored blockchain database using a custom scanning technique that normalizes the relationship between response sizes and actually accessed data to hide the data/block access patterns and ensure client privacy. Figure 2 shows the operation of this variant, and we describe the details as follows:

Initialization and continuous operation.

(**a**) On initialization the Full Node FN_j connects to the Bitcoin network (**a-1**) and downloads the full blockchain (**a-2**). Similarly, the locally stored blockchain database is updated for each new block that is appended to the chain (i.e., as new blocks are received over the P2P network).

(**b**) The lightweight client installation package includes a checkpoint block header from a recent date. When the client is started for the first time, it downloads all newer block headers from the peer-to-peer network and verifies that (i) they all have correct Proof of Work and (ii) the hash chain of the downloaded headers leads to the checkpoint. Once the client's internal state it synchronized with the peer-to-peer network, it stores a small number of the newest headers (e.g., six blocks from the head of the chain to handle shallow forks). The client can update its internal state by downloading newest block headers periodically or before each transaction verification request. The network and storage requirements of this process are minor and easily met even by clients with severe resource constraints.[1]

Client request handling.

(**1**) The Lightweight Client LC_i performs attestation with the secure Enclave E_j residing on the full node FN_j.

(**2**) If the attestation was successful, the Lightweight Client LC_i establishes a secure communication channel to the Enclave E_j using TLS.

(**3**) The Lightweight Client LC_i sends a request containing the addresses of interest and a block number that specifies how deep in the chain transactions should be searched for verification. Typically, this number would be saved from the previous interaction with a full node or in the case of the first transaction verification the number could roughly match the date when the client started using Bitcoin.

(**4**) The Enclave E_j starts *scanning* its locally stored copy of the blockchain (BC) for the requested address and range of blocks using a scanning technique described in detail below.

(**5**) In preparation of the response, the Enclave E_j does the following: for blocks containing client addresses it adds the full transaction information and the corresponding Merkle tree path to the response, while for blocks without client addresses it only adds the block header.

(**6**) The Lightweight Client LC_i verifies that (i) the received block headers match its internal state and (ii) the received transactions and Merkle Tree paths match to the block headers. The client considers such received transactions as confirmed (assuming that they are sufficiently deep in the chain). The client updates its internal state regarding the latest verified block number and closes the connection to the enclave.

Block scanning details. As explained in Section 3.3, enclave execution can leak information in various ways. For example, if our solution would simply return each matching transaction (and the corresponding Merkle Tree) in the specified range of blocks, based on the size of the response the adversary could deduce how much information of interest

[1] For example, obtaining block headers for a checkpoint that is one month old, would require 300 kB of downloaded data (one-time operation) and updating the block headers once per day would require 10 kB of communication per day. Storing the latest six headers takes less than 1 kB of storage.

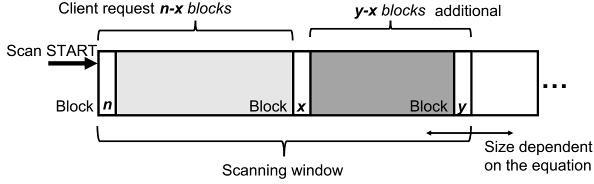


Figure 3: **Block reading in Scanning Window**. Depending on the number of requested blocks (up to x) and the number of matching transaction in them, we read potentially extraneous blocks (up to y) to keep the ratio between the read blocks and the response message size constant.

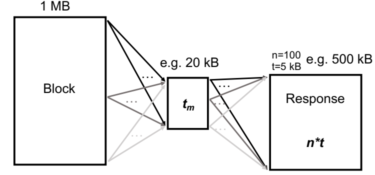


Figure 4: **Oblivious copying in Scanning Window.** The data is copied in an oblivious fashion from the block to a temporary array, i.e., every transaction is conditionally moved using `cmov` to every possible destination. The data contained in the temporary array is then copied to the response in an oblivious fashion, again using `cmov` to conditionally copy everything to all possible locations in the response.

for the client was contained within the scanned blocks. Over a period of time, by tracking requests and response sizes, the adversary could gain significant information about the client's addresses and transactions.

We address such leakage by using a custom-made block scanning scheme. The main goal of the scheme is to fully hide the ratio between the response size (that indicates the number of transactions returned to the client) and the number of scanned blocks. When this ratio is constant, the adversary cannot deduce any meaningful information.

Figure 3 depicts the details of our scanning scheme. The newest block in the blockchain observed by the Bitcoin network is n. A client's request contains an addresses of interest and the number block x indicating how deep the chain should be scanned. The enclave starts scanning from n and moves towards x. It stores intermediate responses and when it reaches block x it performs a check. The total size of the response, r, is divided by the threshold size, t. The threshold indicates the maximum response size per block such that if we are to scan $n-x$ blocks, the maximum response size for the client can be $r=(n-x)*t$. If the given response size r is greater, then the enclave has to scan up to block y (or $y-x$ more blocks), such that $r=(n-y)*t$. If the response size is smaller, i.e., if after scanning $n-x$ blocks $r \leq (n-x)*t$, we pad the response size such that $r=(n-x)*t$. The exact size of the threshold is empirically determined in Section 6.

Side-channel protection. The scanning technique described above prevents external leakage through response sizes and disk accesses. However, if the adversary is able to mount high-granularity digital side-channel attacks (e.g., one that allows her to observe execution paths with instruction-level granularity), she will be able to determine the transactions that were accessed, and thus infer the client's addresses.

To make our system more robust against such attacks, we optionally add side-channel protections at the expense of performance (cf. Section 6). To protect against timing leakage we compute the Merkle path for all transactions in each of the scanned block in contrast to only computing the path for the transactions of client's interest. For protection against control-flow side channels we make use of the `cmov` assembly instruction to hide execution paths. `cmov` is a conditional move such that "*If the condition specified in the opcode (cc) is met, then the source operand is written to the destination operand. If the source operand is a memory operand, then regardless of the condition, the memory operand is read*" [7]. We use the `cmov` instruction in form of a wrapper (originally presented in [47]) that allows us to remove branches from our code resulting in the same control flow with no leakage.

The same technique is also used in previous side-channel protection solutions like Raccoon [47]. However, since using such a general purpose side-channel defense system directly would incur an extremely high performance overhead in our particular setting (due to large amounts of accessed data), we customize these techniques to our setting. Specifically, we apply the following modifications, as per Figure 4:

(i) Instead of continuing to scan the chain if the size of the response exceeds the threshold, we stop scanning after the specified number of blocks. If not all transactions fit in the response, the client does not receive all transactions and is informed of this through a flag in the response. This allows the allocation of a response array that does not change size during processing. The client can request the remaining transactions in a new query (potentially from a different node).

(ii) For each block, we allocate a temporary array of size t_m (see Figure 4), where t_m is a threshold that specifies the maximum data per block, as opposed to the threshold t that specifies the average data per block. While the block is parsed, each transaction is moved to the temporary array in an oblivious fashion, i.e., we use the `cmov` instruction to conditionally move each word of each transaction to every entry in the array. This means that for every transaction we access every entry in the array and since the same instruction is used for each possible copy – independent of whether the data is actually copied – even an attacker with an instruction level view of the control flow cannot determine which data is actually copied. After processing the block, the temporary array is traversed and all entries are copied to the response array (see

Figure 4). This is again done in an oblivious fashion, i.e., each entry is copied conditionally using the `cmov` method to every possible position in the response array.

This method of copying transactions from the block to the response is required to efficiently keep the data accesses oblivious. Specifically, for a block of size m, a temporary array of size t_m and n requested blocks, this method requires $\mathcal{O}(m \cdot t_m + t_m \cdot n \cdot t)$ instead of $\mathcal{O}(m \cdot n \cdot t)$ operations when naively copying the data obliviously from the block to the response. Since t_m is usually much smaller than m and $n \cdot t$, this method is in practice orders of magnitude faster.

4.2 Oblivious Database Variant

In our second variant, we focus on reducing the load of lightweight clients in terms of computation and network while offering even better privacy preservation (namely, the block number that specifies how deep the chain should be searched does not leak). The main idea behind this variant is to allow lightweight clients to send requests containing addresses of their interest and directly receive information regarding unspent outputs, without the need to verify block headers and Merkle tree paths.

In order to achieve such verification, a new indexed database of unspent transactions (denoted as *enclave UTXO*) is created and searched for every client request using an Oblivious RAM algorithm. Figure 5 shows the operation of this variant, and we describe the details as follows:

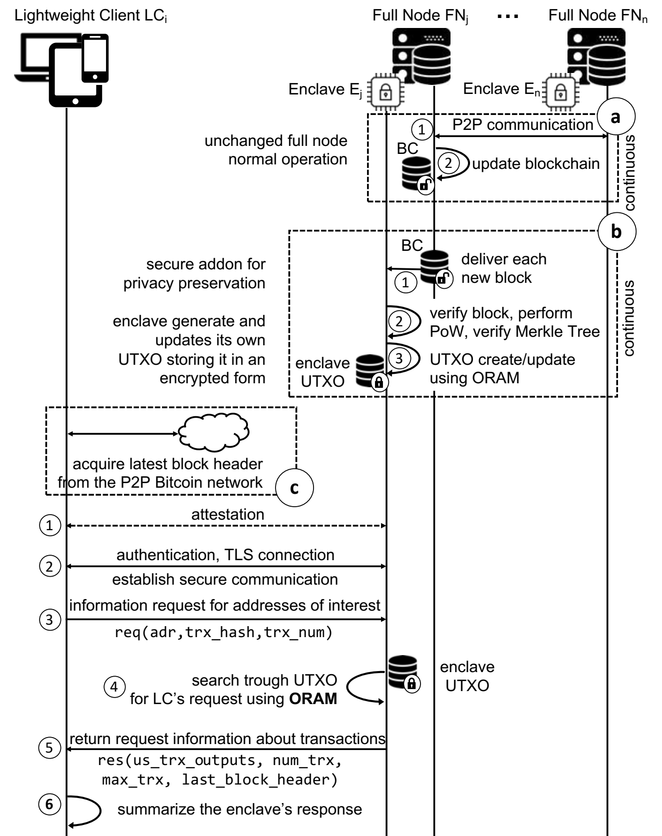


Figure 5: **Oblivious Database operation.** Lightweight client sends a request containing its address and the last transaction to an enclave on full node. Enclave queries a specially-constructed UTXO database using ORAM and provides a response back to the client.

Initialization and continuous operation.

(**a**) Similar to a standard full node, on initialization the full node FN_j connects to the peer-to-peer network and downloads and verifies the entire blockchain. After initialization, when new blocks are available in the peer-to-peer network, FN_j downloads and verifies them.

(**b**) During initialization Enclave E_j reads the locally stored blockchain and verifies each block. The enclave builds its own *enclave UTXO* database that is a special version of the original structure present in standard full nodes. In particular, this UTXO set is encrypted on the disk as sealed storage, indexed for easy and fast access depending on the client request, and accessed using ORAM to prevent information leakage through disk accesses. After initialization, the enclave updates this UTXO using ORAM when new blocks are available in the locally stored blockchain.

(**c**) As in the Scanning Window variant, the client obtains the latest block headers from the peer-to-peer network.

Client request handling.

(**1**) The Lightweight Client LC_i performs an attestation with the secure Enclave E_j residing on the full node FN_j.

(**2**) LC_i establishes a secure communication channel to the Enclave E_j using TLS.

(**3**) LC_i sends a request containing the addresses of interest, along with the hash and number of the latest transaction known to the client. The last two parameters are needed in case the number of unspent outputs contained by an address is larger than the maximum size of the message. For example, LC_i receives the first response containing x transaction outputs with an indication that there is more, and in a consequent request specifies the same address as in the first request along with the x–th transaction hash and transaction number. This gives an indication to the enclave to respond with the second batch of outputs starting from that transaction. The process repeats (possibly with a different node) until the client is satisfied. To prevent information leakage through the message sizes, requests are always of constant size, i.e., the client pads shorter requests and splits up larger queries. The size is defined to accommodate the majority of requests. Since a lightweight client can choose any available node to connect to, she can choose to send requests to

different nodes to hide the number of sent requests.

(**4**) The Enclave E_j reads the enclave UTXO database to get the unspent transaction output information in respect to the client's request. E_j uses ORAM and the previously created index to access the enclave UTXO in an oblivious fashion.

(**5**) In preparation of the response, E_j includes the relevant information as explained in step **(3)**, which encompasses the currently included and maximum number of unspent transactions found for a specific address. When these numbers match, the LC_i knows that she has received all the unspent outputs of a specific address. The enclave additionally includes the block hash of the last known block from the local blockchain (longest chain). With this information the client can deduce whether the enclave has been served with the latest block and that the enclave's database is fully updated. Responses are always of constant size, i.e., shorter responses are padded and if a response is too large, the client is informed of missing outputs, such that she can later retrieve the rest of the outputs (e.g., from a different node). The size of the response is chosen such that it accommodates the majority of responses.

(**6**) The Lightweight Client LC_i can summarize the unspent transaction outputs received from the Enclave E_j. The enclave guarantees completeness in terms of transaction confirmation and the current state of the chain, so the client does not have to perform any additional checks by herself. Successful update of the client's internal state results in the connection termination between the enclave and the client.

Oblivious Database details. In this variant, we use an ORAM algorithm called Path ORAM [52] to protect data access patterns of our enclaves. For readers unfamiliar with this algorithm, a brief description is in Appendix B.

Database Initialization. The ORAM database is initialized by creating dummy buckets on disk and filling the *position map* with randomized entries. The *stash* is also filled with dummy chunks. After that the ORAM database is fully initialized and can be used to add new unspent outputs from the blockchain. To ensure that the enclave always uses the latest version of the sealed UTXO database, SGX counters or rollback-protection systems such as ROTE [39] can be used.

Database Update. When a new Bitcoin block is added, the enclave first verifies the proof of work. It then extracts all transaction inputs and outputs and bundles them by address. For each address found in the block, the UTXO database entry is requested and then updated with the new information. If too many entries are added, resulting in the chunk getting too big, the chunk is split into two and the index is updated to reflect the changes made to the UTXO database. All accesses are performed using the ORAM algorithm and, therefore, do not leak any information about the access patterns.

Database Access. Accesses to the ORAM database follow the normal procedure described in [52] and in Appendix B.

Side-channel protection. While the usage of ORAM protects against all external leakage, side-channel attacks, and thus, internal leakage remains a challenge. If we consider the most powerful attacker that can perform all digital side-channel attacks (see Section 3.3), this variant would be forfeit due to the leakage of the code access patterns, specifically, execution paths in the *if* statements when the stash, indexes and the position map is being accessed. This would leak the exact address which is used to search for the unspent transactions in the internal database.

To remedy internal leakage, we deploy several mechanisms that protect our code and execution. First, when accessing the security critical data structures, specifically, the position map, stash, and the indexes containing information about which chunks contain unspent transactions of a certain Bitcoin address we pass over them entirely in the memory to hide the memory access pattern. Second, to hide the execution paths we remove all branching in the code that accesses these data structures and deploy the `cmov` assembly instruction (see Section 4.1). Observation of the control flow and memory access does not leak whether the operation performed by the enclave was a read or a write, and since there is a single control flow without creating multiple branches depending on the condition, we effectively hide the execution and thus protect this variant from internal leakage in full.

5 Security Analysis

In this section, we provide an informal security analysis. First, we analyze our solution with respect to our adversary model where SGX security enforcements cannot be broken. In particular, we show that our solution ensures confidentiality of the requested client addresses, as the attacker cannot infer the requested address from disk access patterns, response sizes, side-channels, or a combination thereof. Second, we discuss implication of potential SGX compromise and show that our solution can handle such cases gracefully.

5.1 External Leakage Protection

Scanning window. This variant scans complete blocks from the blockchain database, instead of accessing individual transactions within them, and thus prevents direct information leakage from disk access patterns. The constant ratio of response size to scanned blocks prevents information leakage from the response size. The adversary may only infer the number of blocks that are accessed and not which addresses are sent by the client or how many transactions are returned.

Oblivious Database. To protect against information leakage attacks on the disk access, our second variant utilizes the well-studied Path ORAM [52] algorithm. Our setting is slightly different than the typical client-server model considered in ORAM. In our case, the enclave corresponds to

the client. Because the adversary can run the enclave freely, she can use it as an oracle, i.e., she can influence the data that is written (by delivering blocks to the enclave) and can query for values himself. Regardless of that, due to the unlinkability property of ORAM, the attacker learns nothing about what is accessed and the probability to guess correctly which ORAM block was accessed is equal to that of a random guess, as shown in [52]. Also, the adversary learns nothing from responses as they are of constant size.

5.2 Side-channel Protection

Most known side-channel attacks on SGX provide imperfect data-access or control-flow traces and require many repetitions to filter out noise [20, 43, 27, 50]. In BITE, queries from legitimate clients cannot be replayed due to the authenticated TLS channel and since the enclave is either stateless across power cycles or protected against rollback. The adversary can create his own client and send requests to the enclave, but this will not result in any advantage against legitimate clients. For these reasons, mounting side-channel attacks against BITE is more challenging than performing side-channel attacks against enclaves in general. To analyze our solution against future adversaries that may be able to mount more precise attacks, below we consider the worst case scenario, i.e., side-channel attacks that obtain perfect data access and control flow traces from enclave's execution.

Scanning Window. To harden our Scanning Window variant against side-channels, we provide optional protections that incur significant performance penalty. When the enclave scans through both the temporary array and the final response array in their entirety, it performs `cmov` operations for all possible transactions. This allows replacing branches in our code with a few instructions resulting in the same control flow with no leakage to the attacker since all data is accessed and the same operation is executed every time.

Oblivious Database. For our Oblivious Database variant we always include side-channel protections to our solution, since the performance overhead is negligible. When accessing the security critical data structures such as stash, indexes and the position map, we pass over them entirely to hide the memory access pattern. Second, to hide the execution paths, we remove all branching in the code that accesses these data structures and replace them with `cmov` assembly instructions (see Section 4.2). Observation of the control flow and memory access does not leak whether the operation performed by the enclave was a read or a write, and since there is a single control flow without creating multiple branches depending on the condition, we effectively hide the execution path and thus protect this variant from internal leakage in full.

The usage of `cmov` for protecting against digital side-channel and internal leakage was previously studied in Raccoon [47] and with respect to protecting ORAM-based systems it was studied in other SGX-related works [49, 14]. These works show the effectiveness of `cmov` in protecting against internal leakage. Our solution uses the same techniques, and thus directly inherits the security guarantees that successfully protect against the same type of attacks, i.e., those based on digital side-channel leakage.

5.3 Completeness

In the Scanning Window variant, the client herself performs the verification of the block headers, Merkle paths and transactions. Since the client can retrieve block headers from the P2P network and the enclave returns all transactions from its view of the chain, the client can ensure completeness of the response by checking that she received data from the longest chain. In the Oblivious Database variant, the enclave performs all verifications for the client. To ensure completeness, the client can compare the latest block hash from received response to information from other sources.

An adversary that controls the OS of the full node server can deliver incomplete blocks to BITE enclave or decide to not deliver specific new blocks to the enclave. However, this would be noticed by the light clients. Remember that light clients are required to obtain the latest block hash from an alternative source in order to verify the completeness of BITE responses. (Another approach to solve this would be to use systems such as TownCrier [59] or TLS-N [48], that enable the enclave to get an authenticated feed that could confirm the correctness of the blocks received from the full node.)

5.4 Implications of a Full SGX break

Our adversary model assumes that side-channel leakage from enclave's execution may happen, but the adversary cannot fully break SGX, i.e., the adversary cannot read all enclave's secrets and modify its control flow arbitrarily. However, SGX was never intended to provide tamper resistance against physical attacks and recent research has demonstrated that platform vulnerabilities like Spectre [35] and Meltdown [38] can be adapted to extract attestation keys from SGX processors [21, 54]. Therefore, it becomes relevant to ask how BITE handles a full SGX compromise.

In the Scanning Window variant, the client only loses the privacy protections provided by our system and all of his funds remain secure. Since the client still performs SPV, the security is otherwise not affected and our system provides the same guarantees as current light clients, i.e., a node may omit transactions, but cannot steal funds or make a client falsely accept a payment.

In the Oblivious Database variant, a compromised enclave could make the client accept false payments by sending invalid UTXOs. However, we argue that this will not be a realistic threat since it would require the client to sell some

System	Our implementation		Libraries	Total
	Bitcoin[1]	Network[2]	*mbed-tls*	
Scanning Window	1'876	1'613	53'831	57'320
Oblivious Database	4'117	1'613	53'831	59'561

[1] Processing the Bitcoin blockchain.
[2] Parsing responses from the client over TLS.

Table 1: Trusted Computing Base in LOC.

goods or service to the provider of the node, i.e. this is not a realistic issue for most users. Merchants that see a full break of SGX as a realistic threat can instead use the Scanning Window variant. Additionally, such an attack would be easily detectable after the fact and result in loss of reputation of the provider of our service and would thus likely only be profitable for high value transactions for which most merchants would probably run a full node.

We conclude that BITE can provide as much security and privacy as traditional lightweight clients even given a full break of SGX. This is in contrast to the naive solution of storing the clients' private keys in the enclave and using it as a remote wallet. Lastly, we emphasize that our approach and BITE as a solution are not limited to SGX. Our main ideas could most likely be applied to other TEEs as well, such as the open-source Keystone TEE [1], thus reducing the reliance on SGX (and thereby Intel) even further.

6 Performance Evaluation

In this section, we describe our implementation and provide performance evaluation results.

6.1 Implementation Details

The centerpiece of our system is an original blockchain parser. For TLS connections we use the *mbed-tls* library from ARM [37]. Table 1 shows the trusted computing base.

Scanning Window. The implementation of Scanning Window is very small since it only involves scanning the blockchain and does not have to keep state. The network code including the *mbed-tls* library contributes the most to the TCB with over 96%. The same work for matching and non-matching transactions is performed in order to keep the scanning time per block constant for all requests.

The response size per block allows for around 5 transactions. We believe this is a reasonable choice that satisfies common usage patterns for light clients. For n included and N total transactions in the block, an upper bound for the Merkle path size is $n * \log(N)$ and each entry is 32 bytes long. This results in an approximate upper bound of 2.2kB for $N = 4000$, the current limit in Bitcoin. As of today (November 2018) the average transaction size is around 500 bytes, therefore, a response size per block of 5kB is enough to fit around 5 transactions ($5 * 500\text{B} + 2200\text{B} < 5\text{kB}$). If more or larger transactions are found, following from Section 4, the enclave scans more blocks of the blockchain until the response can fit all requested transactions.

Oblivious Database. The implementation of Oblivious Database is more complex than Scanning Window and the enclave has to keep state and store a large UTXO set on disk. At the time of writing, the UTXO size (indexed by Bitcoin address) is around 3GB while our ORAM overhead accounts for 2 times the original size, totaling around 6GB.

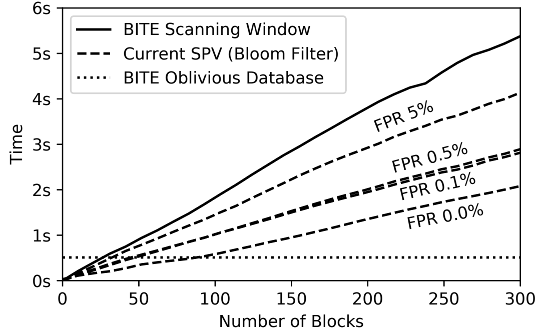
We use Path ORAM to store the UTXO set and have evaluated various chunk sizes for the implementation. The chosen chunk size accounts for 32kB, meaning a single chunk can fill up to 32kB with outputs from one address. If an address has more unspent outputs, the outputs are stored in multiple chunks. Assuming an average output size of 100B, one ORAM read can return up to 320 outputs for one address. The outputs are grouped by the receiving address and then ordered alphabetically. This is necessary in order to keep the size of the index small enough to fit in the enclave's memory. In the worst case the maximum index size involves the lower and upper limits for addresses (20B) and transaction hashes (32B) for every ORAM block resulting in a maximum of $(8\text{GB}/32\text{kB}) \cdot (32\text{B} * 2 + 20\text{B} * 2) \approx 19.5\text{MB}$.

To set the response size, we analyzed the typical unspent outputs per active address in the Bitcoin network. Our results show that 95% of all addresses have 5 or fewer unspent outputs and 98% have fewer than 12 outputs. Based on this data, we settled on 12 average outputs per request, resulting in around 1.2kB.

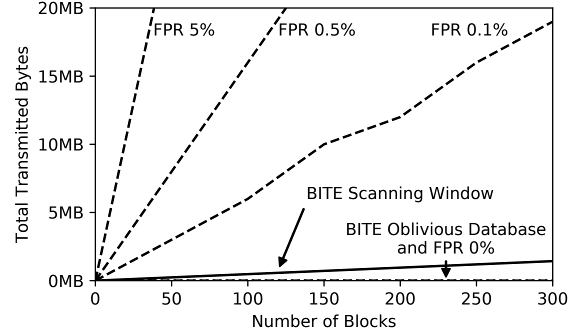
6.2 Performance Results and Comparison

In this section, we evaluate both variants of BITE and compare them to the current SPV performance using *python-bitcoinlib* [53]. The focus is put on three different metrics: processing time, communication overhead, and storage requirements. Processing time encompasses both the request handling from the client to the enclave as well as the time needed for the enclave to update the UTXO for new blocks. Communication overhead is evaluated through the response size, thus directly affecting the client's necessary bandwidth. Lastly, we report the necessary storage requirements on the full nodes that these system need for operation. A summary of all reported results can be found later on in Table 3.

Note that in all our data points, the TLS handshake times are omitted. Matetic et al. [40] report around 100ms for a new handshake and <10ms for TLS session resumption using *mbed-tls* in SGX. We do not evaluate the performance of a client since the client-side storage and network overhead are insignificant. We tested our implementation on an Intel i7-8700k with a Samsung 960 SSD for local storage.



(a) **Processing cost (client request)** for Scanning Window, Oblivious Database and current SPV protocols using Bloom filters.



(b) **Communication cost** for Scanning Window, Oblivious Database and current SPV protocols using bloom filters.

Figure 6: Performance evaluation of Scanning Window and Oblivious Database.

		t_m		
		5kB	10kB	20kB
Blocks	100	0.7s (± 0.2s)	1.3s (± 0.5s)	2.7s (± 0.9s)
	200	0.7s (± 0.2s)	1.4s (± 0.5s)	2.8s (± 0.9s)
	300	0.7s (± 0.2s)	1.5s (± 0.5s)	3.0s (± 0.9s)

Table 2: Processing time per block with oblivious execution for Scanning Window depending on the number of requested blocks and the temporary size, averaged over 100 blocks.

Processing. Figure 6a shows the processing cost to filter blocks for BITE and current SPV protocols. Note that the measurements in Figure 6a do not account for the network speed. For client update requests over the last 100 blocks, the current SPV mode takes 0,62s, 1.06s, 1.06s, 1.5s, with the false positive rates of Bloom filters set to 0.0% 0.1%, 0.5% and 5%, respectively. Note that the numbers regarding standard SPV with the Bloom filter false positive rate of 0.0% actually indicates a solution with no privacy, e.g. the light client sends only his addresses in the request without any masquerading.

For the Scanning Window variant without side-channel protections we report 1.9s, corresponding to an 81% overhead compared to the SPV with FPR 0.1% and 0.5%. If the side-channel protection is added to Scanning Window, the oblivious execution and memory access adds a significant overhead. Table 2 shows the time per block for various requests and t_m size. Higher t_m allows to cope with high variance of relevant activity within the requested blocks. Note that the blocks vary in size, and thus the time per block fluctuates a lot leading to a high standard deviation. Synchronizing 100 blocks with $t_m = 5$kB takes around 73 seconds corresponding to an overhead of approximately 40x. Note that the oblivious Scanning Window variant is not shown in Figure 6a due to its size.

In our Oblivious Database variant, the unspent outputs are directly fetched from the enclave UTXO and the individual blocks are not scanned. Thus, the performance does not depend on the client's last known block, but only on the ORAM database access times. A request that fetches the information regarding 10 client addresses accounts only for 0.5s and is completely independent on the number of requested blocks, thus making it even faster than the standard SPV mode used without any privacy protections.

Contrary to the Scanning Window, in the Oblivious Database variant, the enclave needs to update its UTXO set after each new block arrives in the ORAM database which takes 78.5s. To reach permanent availability we propose to use 2 systems in parallel which update with an offset between each other. If a user requests the result from a node that is not fully up to date, the remaining blocks can be scanned by utilizing oblivious Scanning Window. The number of clients that can be served by a single SGX enclave can be estimated by using around 120s (pessimistic estimate) for updating the state and then the remaining 8 out of 10 minutes (Bitcoin block interval) to continuously answer client requests, leading to an approximate 10000 clients per enclave.

Communication. Figure 6b shows the bandwidth comparison between all discussed protocols. Our variants use significantly smaller response sizes compared to SPV since they do not need to hide relevant information with false positives. A device with a decent 4G connection that operates at 100Mbit/s additionally requires around 1.4s to retrieve 100 blocks (17MB) with the current SPV protocol and a 0.5% false positive rate while Scanning Window only takes 0.04s (500kB). The Oblivious Database variant reduces the communication overhead even more and accounts only for 0.0001s (only 12kB), which is insignificantly small since only unspent outputs are included and not the entire transaction information along with the Merkle paths. The SPV with no privacy protections performs slight less effective than the Oblivious Database of BITE as it was the case when the processing performance was compared.

	Processing		Communication	Storage		
	Request	UTXO Update	Response	Blockchain	UTXO	Leakage Protection
Scanning Window[1]	1.9s	-	500kB	200GB	0	✓/✗[4]
Oblivious SW[1]	73s	-	500kB	200GB	0	✓
Oblivious Database[3]	0.5s	78.5s	12kB	50MB[5]	6GB	✓
Stan. SPV FPR 0.5%[1]	1.1s	≈2s	17MB	200GB	2.8GB	✗
Stan. SPV FPR 0.0%[1,2]	0.6s	≈2s	14kB	200GB	2.8GB	✗

[1] For 100 blocks. [2] SPV with no privacy protection. [3] For 10 addresses.
[4] Protects against external leakage but not side-channels.
[5] Only the block headers need to be stored.

Table 3: Performance comparison and requirements on the full node for supporting light clients.

Storage. The SPV mode has to store both the whole blockchain (200GB) and the UTXO set (2.8GB), while our Scanning Window variant only needs to store the blockchain. Moreover, our Oblivious Database variant does not need the whole blockchain (except during initialization) but only the block headers (50MB in total) and the special enclave UTXO stored in the ORAM database. This database accounts to 6GB, a 100% overhead compared to the regular UTXO set, due to the ORAM algorithm requirements. It is clear that both our variants require less storage, and our Oblivious Database variant's requirements are insignificant compared to all mentioned solutions.

Comparison of BITE variants. Table 3 shows a performance comparison between all our variants and the standard SPV mode from the full node's perspective. The performance of Scanning Window is heavily dependent if side-channels are a concern. The original Scanning Window offers a slightly worse performance than the standard SPV but offers increased privacy, protecting against external leakage, and requires significantly less bandwidth. Adding protection from side-channels greatly increases the processing time, while the communication load stays the same. Oblivious Database, on the other hand, offers the same full privacy guarantees as the oblivious Scanning Window, and has the smallest footprint in both the processing time and the network overhead. The enclave UTXO does require regular updating affecting the uptime. However, the previously mentioned solution of having two parallel enclaves with operation offset effectively removes this limitation. In conclusion, we have shown that our variants offer comparable or better performance with increased end client's privacy.

Comparison to side-channel protection systems. Finally, we compare the performance and security of BITE to previous SGX side-channel protection systems. For our comparison we use Raccoon [47] that addresses internal leakage due to secret-dependent memory accesses and

	Leakage			Performance Overhead
	External	Internal	Response Size	
Raccoon[47]	✗	✓	✗	~ 100x[1]
Obliviate[14]	✓	✗	✗	> 4x[1]
Raccoon[47] + Obliviate[14]	✓	✓	✗	$100x - 400$x[2]
BITE Scanning Window [3]	✓	✓	✓	40x
BITE Oblivious Database	✓	✓	✓	1x

[1] Based on the performance evaluation of [47] and [14].
[2] Combination of the two primitives can yield an overhead in this range.
[3] Fully oblivious Scanning Window variant.

Table 4: Performance overhead and security comparison between existing primitives and BITE.

Obliviate [14] that addresses external leakage due to file accesses. We note that ZeroTrace [49] also provides similar external leakage protection as Obliviate, but since the ZeroTrace paper does not report performance overhead numbers suitable for comparison, we exclude it from our discussion.

Table 4 summarizes our comparison. By applying Raccoon to the target enclave code, the performance overhead of the enclave's execution can range up to 1000x depending on the complexity of the original code that is made oblivious. In our case, the complexity of the original code matches the examples that report the overhead of around 100x. Applying techniques from Obliviate can cause a performance overhead of >4x. Neither Raccoon nor Obliviate alone provide full leakage protection, as Raccoon prevents only from internal leakage and Obliviate protects only against external leakage. Neither of these two systems protects against leakage from response sizes. The combination of Raccoon and Obliviate would protect both internal and external leakage, but still not leakage from response sizes. We estimate that the combined overhead of these two protection tools would amount to 100x-400x.

Our fully oblivious Scanning Window variant has a performance overhead of 40x while the Oblivious Database variant has practically no overhead. More importantly, both of the BITE variants protect against external leakage, internal leakage, and leakage from the response sizes, and therefore achieve more complete protection than any of the previous solutions.

7 Discussion

Usage and long-term privacy. Lightweight clients can use BITE in different ways and the chosen usage model can have implications on the clients' long-term privacy. For example, in what we consider *non-recommended usage*, the client (i) performs payment verification requests only when the payment appears in the ledger, (ii) always uses the same full node for verification, and (iii) only uses a single or few Bitcoin address. If all of the above conditions are met, although the adversary controlling the full node does not learn the client's address from a single verification request, he might be able to *correlate* the timing of the verification re-

quest events and the Bitcoin addresses visible in the ledger at roughly the same time, and thus construct a set of candidate addresses that may belong to the served client. We acknowledge that our solution cannot eliminate this type of correlation completely. However, we stress that such correlation would require long-term tracking of verification requests from the adversary and that the same limitation applies to any light client payment verification scheme.

In *recommended* usage of BITE, the client (i) uses different full nodes for payment verification, (ii) regularly uses fresh Bitcoin addresses (e.g., using an HD wallet [57]), and (iii) introduces unpredictability to the timing pattern of payment verification requests like a small number of extra requests at random time points. Following such a usage model, the above mentioned correlation becomes very difficult.[2]

Large responses. Some client requests might result in a larger response than our defined threshold for message size. As our performance analysis shows, the number of these requests is almost negligible. However, our mechanism still allows these types of request with the distinctive factor that the client would have to request them in batches. For example, if a client in the Scanning Window variant requests transactions for 10 of his addresses from the last 300 blocks using the full-side-channel protection, there might be more transaction data then the $300 * t$ kB message size. In this case, the enclave sets a flag indicating there is more information to be delivered. After receiving the response, the client can repeat the request with the defined flag and receive the rest of the information. The protocol operates in the same way, thus no distinction between these two requests can be observed by the attacker. However, the attacker can see the repeated request and infer that the specific client has more transactions of interest in the designated blocks. To mitigate this problem one could wait a period of time before requesting the rest of the response, obfuscate the IP address or change to a completely different service provider (another enclave) for finishing the request.

Denial of service. A malicious user might attempt DoS by asking for a very long scan window, incurring large processing times for full nodes and making the service momentarily unavailable for other clients. DoS (and spam) are common in systems where there is no significant cost involved (e.g., sending 1M emails is practically free). In our setting, one could easily remedy such denial of service attacks by applying fees based on the nature of the request. Large balance updates for lightweight clients would incur higher costs than just frequent updates, thus limiting the attacker from performing "free" DoS attacks. On the other hand, a malicious node can easily block all enclave messages or interrupt enclave execution, thereby preventing the enclave to access the blockchain, update its UTXO or serve client requests. This however falls in a domain which is impossible to fully prevent. If this would occur, the light client can just send its request to another enclave hosted by another entity.

[2]To quantify how accurately the adversary can correlate the client's addresses with these best practices, would be an interesting direction for future work. As building an accurate model would require collecting significant amount data about the behavioral patterns of light clients, we consider this task a research project on its own and outside our scope.

Unbounded enclave memory. The performance of our system is mostly bounded by the slower disk operations. However, if future versions of Intel SGX would allow more enclave memory (i.e., currently the limit is 128MB without the expensive page swapping) ranging up to the RAM limit on the residing platform, one could keep the UTXO database and all other security critical data in the memory and not on the disk, similar to recently proposed SGX-based in-memory database systems like EnclaveDB [46].

8 Related Work

Lightweight client privacy. The idea of light clients for Bitcoin was already included in the Bitcoin paper by Satoshi Nakamoto [44] in the form of *Simple Payment Verification* (SPV). Hearn and Corallo later introduced Bloom filters [18] in BIP 37 [31] that allow a client to probabilistically request a subset of all transactions in a block to mask which addresses are owned by the client. Gervais et al. later showed that the information leaked by the use of Bloom filters in Bitcoin can in many cases enable the identification of client addresses [25]. Hearn later expanded on these issues and discussed the difficulties of solving them [30].

Osuntokun et al. recently proposed modifications to Bitcoin nodes and lightweight clients that move the application of the filter to the client [45]. Full nodes create a filter (with a low false positive rate) for the set of all transactions in a block. A lightweight client then fetches the filter from one or more full nodes and can then check whether the block contains transactions that she is interested in. If that is the case, the client will request the full block from any node.

This approach suffers from a number of shortcomings. First, the gained privacy largely depends on the client behavior and how well the client is connected to distinct entities. If the client does not request the filter headers from multiple entities and then requests the blocks from a different one, she can be easily tricked into revealing her addresses by using forged filters: A node prepares a filter matching half of all addresses and sends it to the client. If the client requests the block, at least one of her addresses lies within that set, otherwise all of her addresses lie in the other half. The node can then further reduce the possible set using binary search by sending modified filters for the following blocks, allowing bitwise recovery of client addresses. Second, depending on how often a transaction is of interest to the client, she might

end up downloading the full blockchain after all. Since the client always either requests the full block or nothing at all, she will download almost every block if a large fraction of blocks contain at least one transaction that is of interest.

Other research on Bitcoin privacy shows that using different heuristics, large parts of the Bitcoin transaction graph can be deanonymized [16, 41]. These techniques are orthogonal to the problem of light client privacy and out of our scope.

Lastly, there exist alternative solutions that tackle limited computation abilities of light clients, such as VerSum [55]. The main idea is that the complex computation is outsourced to a set of remote servers. Even though these solutions do not focus on privacy preservation directly, they do offer alternative ways to construct support systems for light clients that do not require the creation of UTXO type databases for proving correctness.

SGX Leakage Protection. During the last few years, the research community has studied information leakage from SGX enclaves extensively and proposed a number of defenses. In this section we explain why none of the existing systems solves our problem directly and which prior systems use similar protective primitives as our solution.

Raccoon [47] addresses both internal and external information leakage for both code and data accesses. For control-flow obfuscation, Raccoon uses taint analysis to determine execution paths that should be hidden and transforms enclave code such that it executes extraneous decoy paths to hide the enclave's actual control flow. The basic building block for such control-flow obfuscation is the `cmov` instruction that we use as well. Raccoon also uses Path ORAM to hide external secret-dependent data accesses and "streaming" over data structures (i.e., accessing every element) in the internal enclave memory. The main difference between Raccoon and our solution is that by tailoring our implementation, we avoid the need for taint analysis and extra decoy paths enabling a more efficient solution.

Other related systems include Cloak [28] that prevents cache leakage using hardware-based transactional memory features in processors; ZeroTrace [49] and Obliviate [14] that provide a library for data structures protected using ORAM; DR.SGX [19] that randomizes and periodically re-randomizes all data locations in enclave's memory with cache-line granularity; and, T-SGX [51] and Deja Vu [22] that detect and prevent side-channel attacks based on repeated interrupts. The main limitation of Cloak is that it requires hardware features that are not available on all SGX CPUs and it only prevents cache-based leakage. ZeroTrace and Obliviate are limited to data access protection and does not prevent leakage from secret-dependent control flow. DR.SGX is also limited to data accesses and imposes a high performance overhead when configured to prevent all leakage. T-SGX and Deja Vu are limited to attacks that perform repeated interrupts (subset of known attacks).

Oblix [42] presents a new ORAM algorithm tailored to SGX. We use Path ORAM, but our solution is agnostic to the used ORAM algorithm and we could easily replace it.

9 Conclusion

Improved user privacy is one of the main goals of decentralized currencies like Bitcoin. However, payment verification requires downloading and processing the entire chain which is impossible for most mobile clients. Therefore, all popular blockchains support simplified verification modes where lightweight clients can verify transactions with the help of full nodes. Unfortunately, such payment verification does not preserve user privacy and thus defeats one of the main benefits of using systems like Bitcoin. In this paper, we have proposed a new approach to improve the privacy of lightweight clients using trusted execution. We have shown that our solution provides strong privacy protection and additionally improves performance of current lightweight clients. We argue that BITE is the first practical solution to ensure privacy for light clients, such as mobile devices, in Bitcoin.

Acknowledgments

The research work leading to these results has been supported by Zurich Information Security and Privacy Center (ZISC). We would also like to thank our shepherd Rob Jansen for his insightful comments.

References

[1] Keystone: Open-source Secure Hardware Enclave.

[2] BitcoinJ, 2018. `https://bitcoinj.github.io/`.

[3] Electrum, 2018. `https://electrum.org/#home`.

[4] Ethereum, 2018. `https://www.ethereum.org/`.

[5] Etherscan.io, 2018. `https://etherscan.io`.

[6] Light Ethereum Subprotocol (LES), 2018. `https://github.com/zsfelfoldi/go-ethereum/wiki/Light-Ethereum-Subprotocol-%28LES%29`.

[7] OpCodes: CMOV, 2018. `http://www.rcollins.org/p6/opcodes/CMOV.html`.

[8] PicoCoin, 2018. `https://github.com/jgarzik/picocoin`.

[9] R3, 2018. `https://www.r3.com/`.

[10] Ripple, 2018. `https://ripple.com/`.

[11] Bitnodes, 2019. `https://bitnodes.earn.com/`.

[12] Blockchain.info, 2019. `https://blockchain.info`.

[13] Statoshi.info, 2019. `https://statoshi.info`.

[14] AHMAD, A., KIM, K., SARFARAZ, M. I., AND LEE, B. OBLIVIATE: A Data Oblivious File System for Intel SGX. In *NDSS* (2018).

[15] ANDROULAKI, E., BARGER, A., BORTNIKOV, V., CACHIN, C., CHRISTIDIS, K., DE CARO, A., ENYEART, D., FERRIS, C., LAVENTMAN, G., MANEVICH, Y., ET AL. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference* (2018), ACM.

[16] ANDROULAKI, E., KARAME, G. O., ROESCHLIN, M., SCHERER, T., AND CAPKUN, S. Evaluating User Privacy in Bitcoin. In *International Conference on Financial Cryptography and Data Security* (2013), Springer.

[17] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M., ET AL. SCONE: Secure Linux Containers with Intel SGX. In *11th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI)* (2016).

[18] BLOOM, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[19] BRASSER, F., CAPKUN, S., DMITRIENKO, A., FRASSETTO, T., KOSTIAINEN, K., MÜLLER, U., AND SADEGHI, A. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization, 2017.

[20] BRASSER, F., MULLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT)* (2017).

[21] CHEN, G., CHEN, S., XIAO, Y., ZHANG, Y., LIN, Z., AND LAI, T. H. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *Computing Research Repository (CoRR), arXiv abs/1802.09085* (2018).

[22] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *Proceedings of the 12th ACM ASIA Conference on Computer and Communications Security (ASIACCS)* (2017).

[23] COSTAN, V., AND DEVADAS, S. Intel SGX explained. In *Cryptology ePrint Archive, Report 2016/086* (2016).

[24] FOR ALTERNATIVE FINANCE, C. C. Global Cryptocurrency Benchmarking Study, 20187. `https://goo.gl/7B99Ev`.

[25] GERVAIS, A., CAPKUN, S., KARAME, G., AND GRUBER, D. On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM.

[26] GOLDREICH, O., AND OSTROVSKY, R. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[27] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security* (2017), ACM.

[28] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security* (2017).

[29] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM 52*, 5 (2009), 91–98.

[30] HEARN, M. Bloom Filter Privacy and Thoughts on a Newer Protocol, 2015. `https://groups.google.com/forum/#!msg/bitcoinj/Ys13qkTwcNg/9qxnhwnkeoIJ`.

[31] HEARN, M., AND CORALLO, M. Connection Bloom Filtering. *Bitcoin Improvement Proposal 37* (2012). `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`.

[32] INTEL. Intel SGX, Ref. No.: 332680-002, 2015. `https://software.intel.com/sites/default/files/332680-002.pdf`.

[33] INTEL. Intel Software Guard Extensions - Developer Zone - Details, 2017. `https://software.intel.com/en-us/sgx/details`.

[34] KAUER, B. OSLO: Improving the Security of Trusted Computing. In *USENIX Security* (2007).

[35] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)* (2019).

[36] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *USENIX Security* (2017).

[37] LIMITED, A. mbedTLS (formerly known as PolarSSL), 2015. `https://tls.mbed.org/`.

[38] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security* (2018).

[39] MATETIC, S., AHMED, M., KOSTIAINEN, K., DHAR, A., SOMMER, D., GERVAIS, A., JUELS, A., AND CAPKUN, S. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security* (2017).

[40] MATETIC, S., SCHNEIDER, M., MILLER, A., JUELS, A., AND CAPKUN, S. DELEGATEE: Brokered Delegation Using Trusted Execution Environments. In *USENIX Security* (2018).

[41] MEIKLEJOHN, S., POMAROLE, M., JORDAN, G., LEVCHENKO, K., MCCOY, D., VOELKER, G. M., AND SAVAGE, S. A Fistful of Bitcoins: Characterizing Payments among Men with No Names. In *Proceedings of the 2013 conference on Internet Measurement Conference* (2013), ACM.

[42] MISHRA, P., PODDAR, R., CHEN, J., CHIESA, A., AND POPA, R. A. Oblix: An Efficient Oblivious Search Index. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)* (2018).

[43] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. Cachezoom: How SGX Amplifies the Power of Cache Attacks. In *International Conference on Cryptographic Hardware and Embedded Systems* (2017), Springer.

[44] NAKAMOTO, S. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[45] OSUNTOKUN, O., AKSELROD, A., AND POSEN, J. Client Side Block Filtering. *Bitcoin Improvement Proposal 157* (2017). `https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki`.

[46] PRIEBE, C., VASWANI, K., AND COSTA, M. EnclaveDB: A Secure Database using SGX. IEEE.

[47] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *USENIX Security* (2015).

[48] RITZDORF, H., WÜST, K., GERVAIS, A., FELLEY, G., ET AL. Tls-n: Non-repudiation over tls enabling ubiquitous content signing. In *Network and Distributed System Security Symposium (NDSS)* (2018).

[49] SASY, S., GORBUNOV, S., AND FLETCHER, C. ZeroTrace: Oblivious memory primitives from Intel SGX. In *NDSS* (2017).

[50] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2017), Springer.

[51] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS* (2017).

[52] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2013).

[53] TODD, P. python-bitcoinlib, 2018. `https://github.com/petertodd/python-bitcoinlib`.

[54] VAN BULCK, J., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security* (2018).

[55] VAN DEN HOOFF, J., KAASHOEK, M. F., AND ZELDOVICH, N. Versum: Verifiable computations over large public logs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1304–1316.

[56] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking SMM Memory via Intel CPU Cache Poisoning. *Invisible Things Lab* (2009).

[57] WUILLE, P. Hierarchical Deterministic Wallets. *Bitcoin Improvement Proposal 32* (2012). `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`.

[58] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)* (2015).

[59] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., AND SHI, E. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016).

A Intel SGX

Intel's SGX [23, 32] entails a security enhancement for new Intel CPUs in form of a TEE for security-critical applications in commodity PC platforms. The SGX architecture enables protected applications, called *enclaves* that are *isolated* from software running outside of the enclave. This isolation protects the integrity and confidentiality of the enclave's execution from any malicious software running on the same system, including BIOS, OS and hypervisor, or even malicious peripherals such as compromised network cards [56, 34, 29]. Enclave memory is handled in plaintext only inside the processor and is encrypted by the processor whenever it leaves the CPU (e.g., to DRAM) to ensure that neither the OS nor malicious hardware can access it.

Even though the OS is untrusted, it is responsible for starting and managing enclaves. To protect the integrity of the execution, the CPU securely records all initialization actions to create a *measurement* that records the code and initial state of the enclave. This can be later used by a third party to verify that the correct code is running on the system supported by SGX. This process is called *remote attestation*. A system service called Quoting Enclave signs the attestation statement – which contains the mentioned measurements – for remote verification. Using an online attestation service run by Intel, the verifier can check that signature. An enclave can attach data to the attestation statement, such as a public key, that it sends to the verifier. This can be used to establish a secure communication channel to an enclave.

In addition, SGX enables enclaves to store data for persistent storage in an encrypted form through a process called *sealing*. The processor provides a sealing key that can only be accessed by the same enclave running on the same platform, i.e. only the enclave that sealed data can later unseal it. This provides confidentiality and integrity for the stored data, but it does not protect from so called rollback attacks [39] when the enclave is restarted. Finally, enclaves cannot execute system calls and do not have access to secure peripherals. For this reason, software using SGX has to be split into two parts, a protected enclave and an unprotected component that runs in normal user space and handles communication with the OS, i.e. operations concerning networking and file accesses. For further details, we refer the reader to [23, 32].

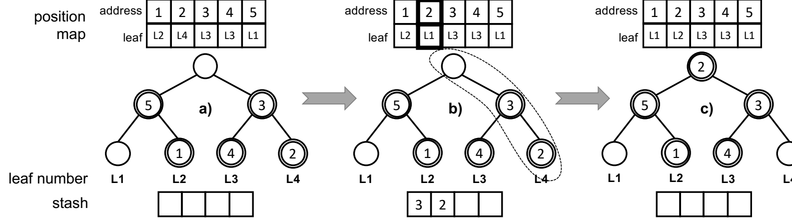


Figure 7: a) The client wants to access the chunk 2 that is stored in Path ORAM. b) The position map specifies that the chunk 2 is on the path to leaf 4. Therefore, the server reads all entries on the path into the stash and re-randomizes the position map entry of the requested chunk. c) The server writes back as many chunks as possible on the previously read path.

B Oblivious RAM

Oblivious RAM (ORAM) [26], is a well-known technique that hides access patterns to an encrypted storage medium. A typical ORAM model is one where a trusted client wants to store sensitive information on an untrusted server. Encrypting each data record before storing it on the server provides confidentiality, but access patterns to stored encrypted records can leak information, such as correlation of multiple accesses to the same record. The intuition behind the security definition of ORAM is to prevent the adversary from learning anything about the access pattern. In ORAM, the adversary does not learn any information about which data is being accessed and when, whether the same data is being repeatedly accessed (i.e., unlinkability), the pattern of the access itself, and lastly the purpose, type of the access (i.e., write or read). However, one should note that ORAM techniques cannot hide access timing.

In this work, we use a popular and simple algorithm called Path ORAM [52] that provides a good trade-off between client side storage and bandwidth. The storage is organized as a binary tree with buckets containing Z chunks each. The position of each chunk is stored in a *position map* that maps a database entry to a leaf in the tree, and for every access the leaf of the accessed entry is re-randomized. A small amount of entries is stored in a local (i.e., memory) structure – *stash*.

Every access involves reading all buckets of a path from the root to a leaf into the *stash* and then writing back new or old re-randomized data from the *stash* to the same path resulting in an overhead of $O(\log N)$ read/write operations. If the requested chunk is already in the stash, an entire path still gets read and written. The summary of ORAM operations is:

(**1**) get leaf from *position map* and generate new random leaf for the database entry. Insert it into the *position map*, read all buckets along the path to the leaf and put them into the *stash*

(**2**) if access is a write, replace the specified chunk in the stash with the new chunk

(**3**) write back some chunks from the *stash* to the path. Chunks can only be put into the path if their leaf from the *position map* allows it. Chunks are pushed down as far as possible into the tree to minimize *stash* capacity.

(**4**) return requested chunk