Bite: Bitcoin Lightweight Client Privacy using Trusted Execution

Sinisa Matetic[1], Karl Wüst[1], Moritz Schneider[1], Kari Kostiainen[1], Ghassan Karame[2], and Srdjan Capkun[1]

[1] ETH Zurich, Switzerland
firstname.surname@inf.ethz.ch
[2] NEC Labs, Germany
firstname.surname@neclab.eu

Abstract. Decentralized 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, such as mobile phones. To address such concerns, most major blockchain systems support lightweight clients that outsource most of the computational and storage burden to full blockchain nodes. However, such payment verification methods leak considerable information about the underlying clients, thus defeating user privacy that is considered one of the main goals of decentralized cryptocurrencies.
In this paper, we propose a new approach to protect the privacy of lightweight clients in blockchain systems like Bitcoin. Our main idea is to leverage commonly available trusted execution capabilities, such as SGX enclaves. We design and implement a system called Bite where enclaves on full nodes serve privacy-preserving requests from lightweight clients. As we will show, naive serving of client requests from within SGX enclaves still leaks user information. Bite therefore integrates several privacy preservation measures that address external leakage as well as SGX side-channels. We show that the resulting solution provides strong privacy protection and at the same time improves the performance of current lightweight clients.

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 [1], leverage the same open or permissionless model as Bitcoin, while platforms like Hyperledger [2], Ripple [3] and R3 [4], enable closed or permissioned blockchains.

Most blockchains implement a decentralized time-stamping mechanism that ensures eventual consistency of data, such as *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 160 GB of storage today, and the sizes of popular blockchains are growing fast (e.g., Bitcoin's blockchain grew over 60% in the last 14 months) [5,6]. This makes usage of blockchains infeasible on resource-constrained clients like mobile devices.

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. E.g., the Bitcoin community provides the BitcoinJ [7], PicoCoin [8] and Electrum [9] clients implementing the Simple Payment Verification (SPV) mode [10], where the clients connect to a full node that has access to the complete blockchain and can assist the client in transaction confirmation. Transactions contain inputs and outputs that are bound to *adresses* owned by users. As the full node has to learn all transactions issued and received by the requesting client to verify their correctness, such action obviously violates user privacy.

To improve user privacy, several clients support *filters* (e.g., Bitcoin's BIP37 [11] and Ethereum's LES [12]). 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, Bitcoin's BIP37 supports Bloom filters [13] 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 Bloom 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 [14]. Consequently, none of the current lightweight clients provides adequate privacy protection with practical performance overhead.

Our solution. The main goal of this work is to improve the privacy of Bitcoin lightweight clients without compromising their performance. To reach this goal, we combine techniques from several separate fields, including trusted computing, private information retrieval and side-channel protection. We stress here that a naive composition would result in a poor trade-off between privacy and performance. The primary problem that we solve in this paper is how to combine known and new techniques such that the resulting solution provides strong privacy, good performance and easy adoption at the same time.

The starting point of our solution is to leverage commonly available trusted computing capabilities, such as Intel SGX enclaves [15], on full nodes. SGX enables the execution of protected applications, called *enclaves*, in isolation from any untrusted software such as the OS and protects the integrity of enclave execution and the confidentiality of enclave data.

We propose a new solution that we call BITE (for ***BI****tcoin lightweight client privacy using* ***T****rusted* ***E****xecution*), in which a potentially untrusted entity runs a full node with an SGX enclave that serves transaction verification requests from clients. However, the usage of trusted computing alone does not solve our problem. SGX prevents an adversary that controls malicious software, such as the OS, on the full node from directly accessing enclave's memory or reading sealed storage. However, secret-dependent access patterns to external storage, such as transaction databases, can reveal the client's address to the adversary. Additionally, recent research has shown that SGX can be susceptible to software-based side-channel attacks, where malicious software on the same platform infers secret-dependent data access patterns or control flow in the enclave by monitoring usage of shared hardware-resources, such as caches [16,17,18,19], or memory management events like paging [20].

With such enclave leakage in mind, we design two Bite variants. Our first variant is called *Scanning Window* and its operation is similar to the current SPV clients that verify transactions using block headers and Merkle paths received from the full node. To prevent leakage through external data access patterns, 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 is called *Oblivious Database* and it 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 [21] to hide access patterns to an encrypted storage. This second variant allows even *lighter* lightweight clients that no longer need to download and verify Merkle paths.

To prevent software-based side-channels, we adopt protections from recent SGX research. The basic building block for our control-flow hiding is the `cmov` instruction [22] that enables building oblivious execution of branches. To prevent leakage from data accesses we adopt additional defenses, such as iterating over the entire data structure when an element is accessed based on the protected client address. In our use case, full nodes need to process large blockchain databases to serve client request, and thus straightforward usage of known SGX side-channel protection systems, such as [23,24,25,26], would result in either excessive performance overhead or imperfect side-channel protection. Instead of using such systems directly, we carefully pick low-level primitives and apply them at critical points in our system.

Results. We show that our solution improves both the privacy and performance of current lightweight clients. 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 [16,17,18,19,20,27]. 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 solution leverages trusted execution in a way that makes its adoption safe for Bitcoin's users. In particular, our solution can be used such that even if specific SGX processors would be completely broken (e.g., through physical attacks), our solution provides no improvement in users' privacy, but importantly also no degradation in users' security. That is, usage of BITE on fully broken SGX does not enable double spending or stealing of users' wallets or coins.

In terms of performance, the Oblivious Database variant reduces both processing times and bandwidth consumption drastically. For example, transaction search for 10 addresses accounts only for 750ms of processing time and a 12 kB message size. In Scanning Window, processing times are comparable to the current SPV mode, but bandwidth is reduced significantly. For 200 blocks, Scanning Window accounts for 3.9s of processing time and 0.95MB message size, while current SPV mode accounts for 2.1s of processing time and 35.01MB message size.

We argue that BITE emerges as the *first* practical solution that provides strong privacy protection for lightweight Bitcoin clients like mobile devices. 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. We plan on releasing the full implementation of our solution online as open source.

Contributions. To summarize, in this paper we make the following main 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 known and new private information retrieval and side-channel protection techniques to prevent information leakage.
- *Evaluation.* We show that our solution significantly improves both privacy and performance of current clients. 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 discuss directions for future work 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, respectively.

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 [10] is the first and still most popular cryptocurrency based on blockchain technology. It enables users to perform payments by issuing transactions. While Bitcoin enables execution of a simple scripting language, regular Bitcoin transactions generally 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 will then be 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 a hash-based Proof-of-Work puzzle to make the contained transactions hard to revert. A miner that successfully finds a valid Proof-of-Work for a candidate block, broadcasts the block to all other nodes, who then verify its correctness and include it in their copy of the blockchain if valid.

In order 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) [10]. In this technique, lightweight clients store only block headers, check their Proof-of-Work 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 [11] introduced Bloom filters [13] that allow a lightweight client to request a subset of all transactions to preserve some privacy without needing to download all transactions for each block. A Bloom filter [13] 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 data structure 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 the case, the value was not an input. If it is the case, however, the value *might* have been an input or it could be a false positive. The false positive rate can be set by the creator of the filter.

In Bitcoin lightweight 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 towards the full nodes.

In addition, Gervais et al. [14] have shown that the use of Bloom filters in Bitcoin lightweight clients leaks more information than was previously thought. In particular, they showed that 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 [28] 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 [29,30], 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 re-computing 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.

2.3 Requirements

The high-level goal of this paper is to develop a solution that provides better privacy for lightweight clients without compromising their 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.

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 lightweight 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 [31,15] running within full nodes to provide a privacy-preserving verification service to lightweight clients. Besides increased privacy, trusted execution can enable better 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 execution integrity is protected from any malicious software running on the same platform, including the OS. In particular, all enclave memory is encrypted by the CPU whenever it leaves the processor package which prevents the OS from directly accessing enclave's memory (see Appendix A for more details).

A simple way to leverage SGX would be a solution, where the lightweight client sends its wallet private key to an enclave on the assisting full node. Using the wallet 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 ran in return of 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

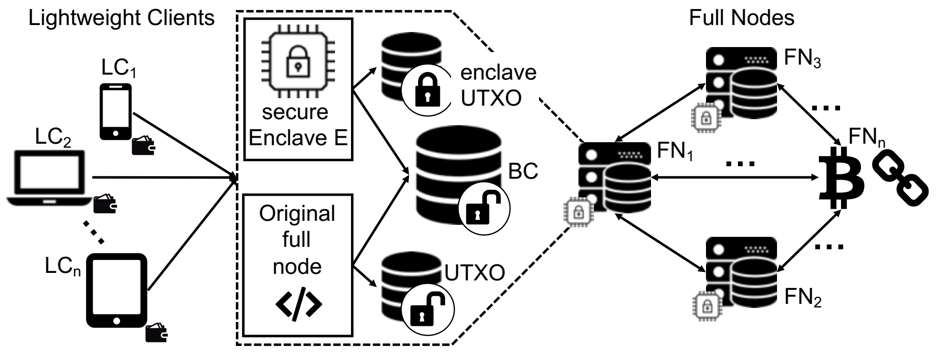


Fig. 1: **System model.** Lightweight clients request transaction verification service from enclaves hosted on full nodes.

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 [32]), the complete blockchain (BC) and the database of unspent transaction outputs (UTXO) are significantly larger than the enclave's memory limits (160GB or more). Therefore, these databases are stored on local persistent storage such as disk outside enclave's memory.

3.2 Adversary Model

We consider an adversary who controls the OS and any other privileged software on the full node. For example, when the verification service is provided by a company, the adversary could be a malicious administrator or an external attacker who has remotely compromised the OS on the full node. If the service is provided by an unknown volunteer, the adversary could be a malicious volunteer.

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's 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 gracefully.)

Finally, we assume that common cryptographic primitives are secure, e.g., the adversary cannot break cryptographic primitives such as encryption or signatures.

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 may be able to infer information about the accessed element by observing access patterns to individual records, such as files or database entries.

Similarly, enclaves rely on the OS to perform communication operations which allows it to infer information about the communication patterns of the enclave. Even if messages are encrypted by the enclave, the message sizes, frequency and destination can leak information to the OS.

Leakage through side channels. The SGX architecture can also be susceptible to *internal* leakage. While Intel acknowledges the possibility of side-channel attacks on enclaves [33], they consider it out of scope for the SGX adversary model. However, recent research shows that such attacks are practical and need to be taken into account. For example, by monitoring usage of shared hardware resources, such as CPU caches, the OS may be able to mount software-based side-channels and infer secret-dependent data and code accesses inside the enclave's memory [16,17,18,19]. In SGX, the memory management is left to the untrusted OS, and therefore the OS may also be able to infer enclave's secrets by monitoring the memory pages that the enclave requests from the OS [20]. Researchers have also demonstrated side-channel attacks using the CPU's branch prediction functionality [27] and speculative execution (the Spectre attack) [34].

Known side-channel attacks can be classified with respect to which memory content is targeted. Code monitoring can identify secret-dependent execution paths, that is, control flow. Data access monitoring can identify secret-dependent data object usage. Branch prediction attacks [27] target execution paths, while most demonstrated cache attacks target data accesses [16,17,18,19], although cache attacks can target control-flow as well.

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 is called *Scanning Window* and it 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 is called *Oblivious Database* and it can be seen as a completely new verification mode for lightweight clients. In this variant, the enclave on 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 [21]) and responds accordingly. Such verification allows the client to check how many coins is currently associated to its addresses, with significant performance improvements over SPV.

In both variants, the client performs remote attestation on the used enclave and establishes a TLS connection to it. We note that current lightweight 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 existing full nodes. For clients, payment execution remains unchanged. Payment verification requires minor additions (attestation and TLS) when Scanning Window variant is used or slightly bigger changes when Oblivious Database variant is used.

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 P2P Bitcoin network (**a-1**) and downloads the full blockchain (**a-2**). Similarly, the locally stored blockchain database is updated for each new blocks 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

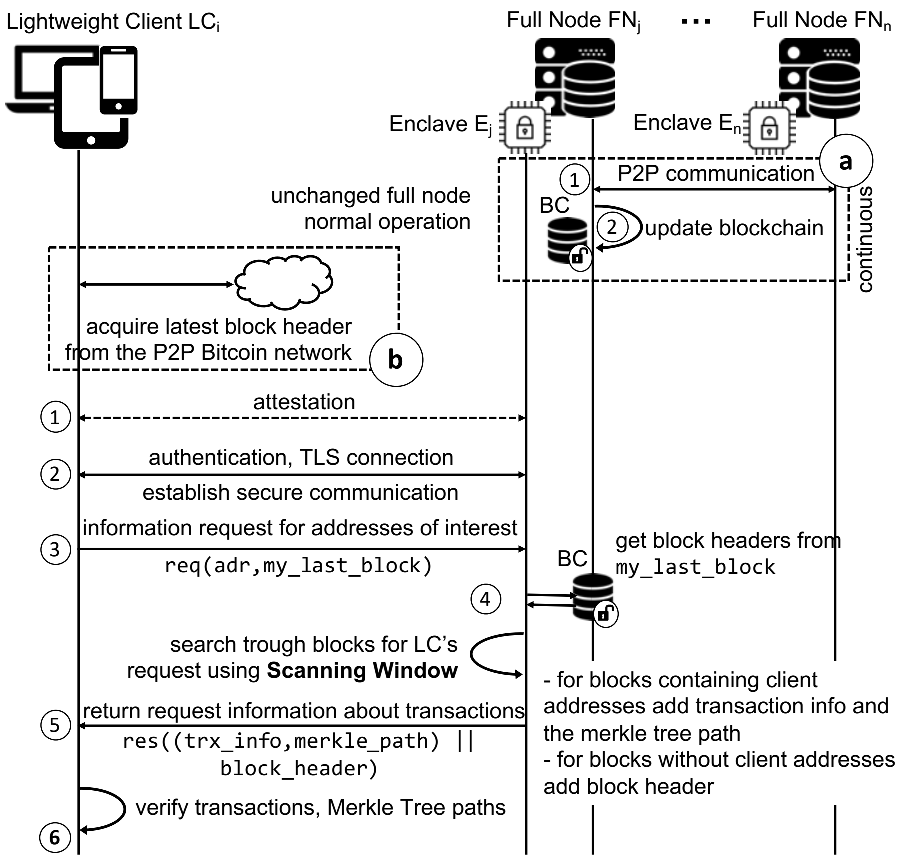

Lightweight Client LC_i
Full Node FN_j
Full Node FN_n
Enclave E_j
Enclave E_n
a
1 P2P communication
BC
2 update blockchain
continuous
unchanged full node
normal operation
acquire latest block header
from the P2P Bitcoin network
b
1 attestation
2 authentication, TLS connection
establish secure communication
3 information request for addresses of interest
req(adr,my_last_block)
BC
get block headers from
my_last_block
4
search trough blocks for LC's
request using Scanning Window
- for blocks containing client addresses add transaction info and the merkle tree path
- for blocks without client addresses add block header
5 return request information about transactions
res((trx_info,merkle_path) || block_header)
6 verify transactions, Merkle Tree paths

Fig. 2: **Scanning Window operation.** Lightweiwght client establishes a secure connection to an enclave on full node and sends a request that contains its address and last known block. The enclave scans a number of blocks from the locally stored complete chain and prepares a response whose size is proportional to the number of scanned blocks.

of this process are minor and easily met even by clients with severe resource constraints.[3]

Client request handling. Clients perform transaction verification as follows:

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

[3] 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 latest six headers takes less than 1 kB of storage.

(**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 underlying 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 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 tailor-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 from the response size.

Figure 3 depicts the details of our scanning scheme. The newest block in the blockchain observed by the Bitcoin network is n. A clients 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,

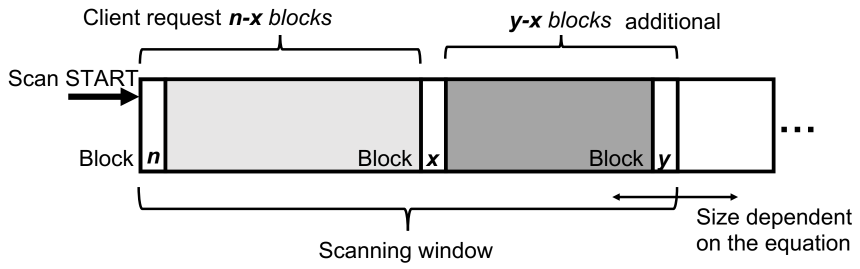


Fig. 3: **Block reading in Scanning Window**. Depending on the number of requested blocks (up to x) and the number of matching transaction in them, our scanning technique read potentially extraneous blocks (up to y) to keep the ratio between the read blocks and the response message size constant.

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 leakage from externally observable response sizes. However, if the adversary is able to mount a high-granularity digital side-channel attacks (e.g., one that allows her to observe execution paths with instruction-level granularity), the adversary 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*" [22]. This allows us to replace 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 [23]. However, since using such a side-channel defense system directly would incur an extremely high performance overhead in our particular setting (due to large amounts of accessed blockchain 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

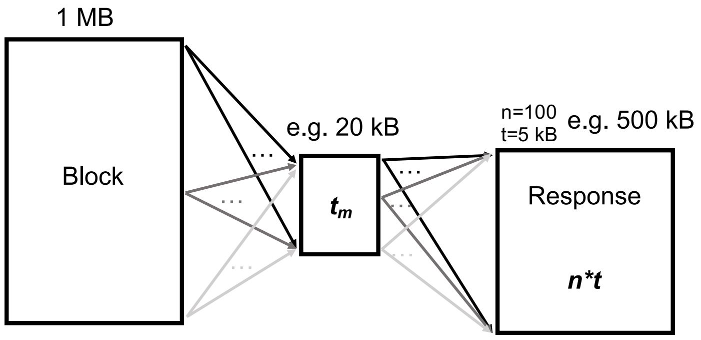


Fig. 4: **Oblivious copying in Scanning Window.** The data is copied in an oblivious fashion from the block to a temporary array, i.e., every transaction is conditionally moved using `cmov` to every possible destination. The data contained in the temporary array is then copied to the response in an oblivious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

request the remaining transactions in another request (from the flagged block until the end of the initially specified block).

(**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` instruction 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 $\mathcal{O}(m \cdot t_m + t_m \cdot n \cdot t)$ operations instead of $\mathcal{O}(m \cdot n \cdot t)$ operations when naively copying the data in an oblivious fashion from the block to the response directly. Since t_m is usually much smaller than m and $n \cdot t$, this method is in practice orders of magnitude faster, in relation to the data copy in oblivious fashion.

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:

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. Clients perform transaction verification as follows:

(**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

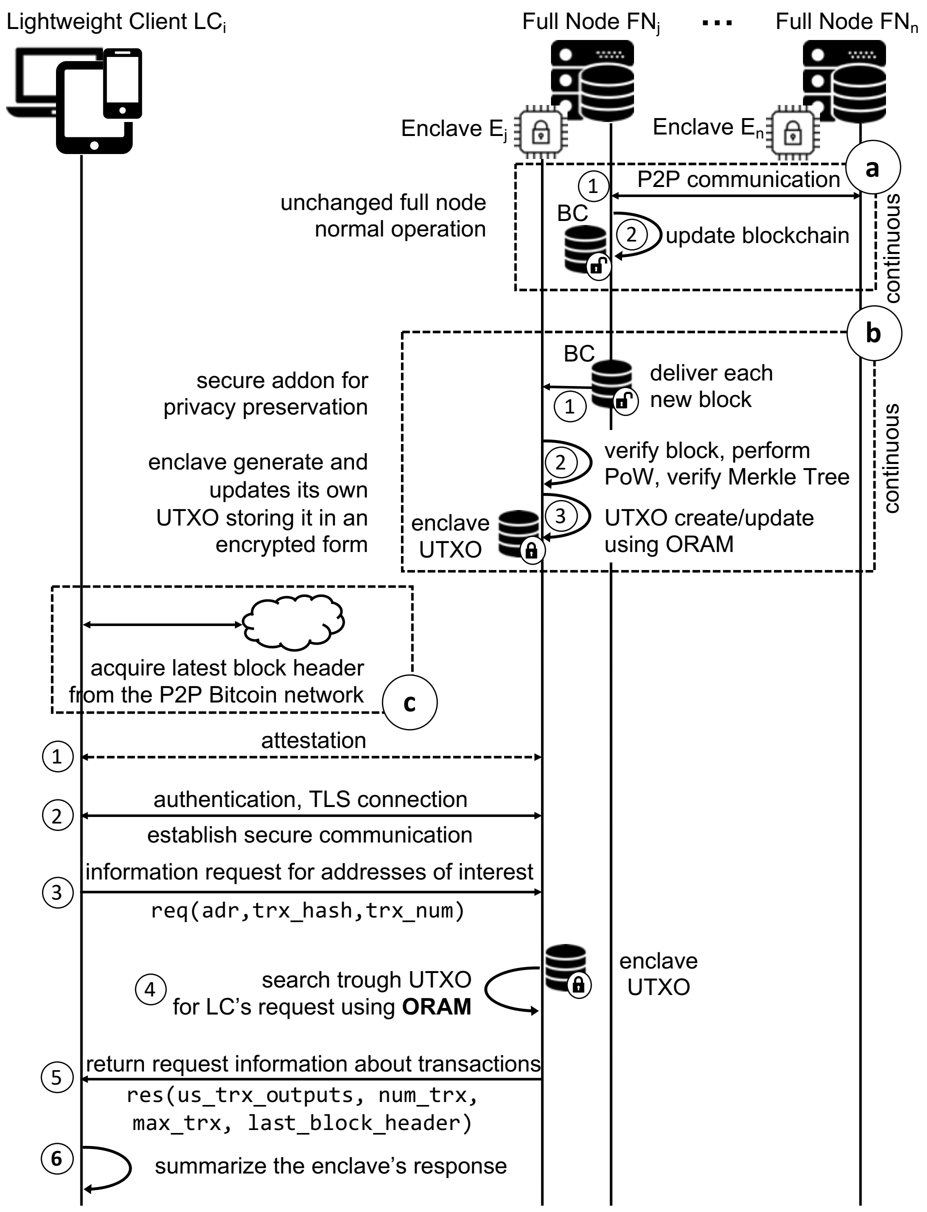

Lightweight Client LC$_i$
Full Node FN$_j$
Full Node FN$_n$
Enclave E$_j$
Enclave E$_n$
a
1 P2P communication
unchanged full node normal operation
BC
2 update blockchain
continuous
b
BC
secure addon for privacy preservation
deliver each new block
1
enclave generate and updates its own UTXO storing it in an encrypted form
2 verify block, perform PoW, verify Merkle Tree
3 UTXO create/update using ORAM
enclave UTXO
continuous
acquire latest block header from the P2P Bitcoin network
c
1 attestation
2 authentication, TLS connection
establish secure communication
3 information request for addresses of interest
req(adr,trx_hash,trx_num)
4 search trough UTXO for LC's request using ORAM
enclave UTXO
5 return request information about transactions
res(us_trx_outputs, num_trx, max_trx, last_block_header)
6 summarize the enclave's response

Fig. 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 version of the UTXO database using ORAM and provides a response back to the client.

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 [21] to protect data access patterns of our enclaves. For readers unfamiliar with this algorithm, we provide a brief description 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 enclave UTXO database, SGX counters or rollback-protection systems such as ROTE [35] 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 [21] 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. This protection mechanism has negligible performance overhead (see Section 6).

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 [21] 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 [21]. As the responses always have a constant size, the adversary cannot learn anything from response sizes either.

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 [16,17,18,19]. 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 side-channels, 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 us to replace branches in our code with a single instruction 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 [23] and with respect to protecting ORAM-based systems it was studied in other SGX-related works [25,36]. 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 blocks, Merkle paths and transactions based on the information received from the full node and can compare the hash of the latest block to its local view of the chain to ensure completeness of the response. In the Oblivious Database variant, the enclave performs all verifications for the client. To ensure completeness, the client can compare the received response to its local view of the chain.

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 [37] and Meltdown [38] can be leveraged to extract attestation keys from SGX processors [39]. 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 Simple Payment Verification, the security is otherwise not affected and our system provides the same guarantees as current lightweight 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 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.

6 Performance Evaluation

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

Table 1: Trusted Computing Base in LOC.

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.

6.1 Implementation Details

The centerpiece of the Scanning Window and Oblivious Database system variants is an original blockchain parser. For TLS connections we use the *mbed-tls* library from ARM [40]. A comparison between the two systems in terms of Trusted Computing Base is shown in Table 1. We differentiate between the code that is used for communication (*Network* in Table 1) and the code for processing the blockchain (*Bitcoin* in Table 1).

Scanning Window. The implementation of Scanning Window is very small (around 3.5k lines of code without *mbed-tls*) 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%. In order to keep the scanning time per block constant for all requests, the enclave does the same work for matching and non-matching transactions.

The message size per block is calculated to allow for around 5 transactions per block. We believe that this is a reasonable choice that satisfies common usage patterns for lightweight client users. 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 in the Merkle path 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 (March 2018) the average transaction size is around 500 bytes, therefore, a message 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 message size is big enough.

Oblivious Database. The implementation of Oblivious Database is more complex than Scanning Window and contains around 5.7k lines of code without *mbed-tls*. Contrary to the Scanning Window variant, the enclave has to keep state and store a large UTXO set on disk. At the time of writing, the size of the UTXO set (indexed by Bitcoin address) is around 4GB. The bucket size for ORAM is set to $Z = 4$ so one bucket contains 4 chunks.

We evaluated the ORAM performance for 32kB, 64kB and 128kB chunk sizes in Table 2 and settled on 32kB chunk size which then implies a tree height of 16, i.e. 2^{16} buckets. The total resulting file size on disk for the ORAM database amounts to around 8.5GB. With the selected chunk size of 32kB, a single chunk can fill up to 32kB with outputs from one address. If an address has more unspent

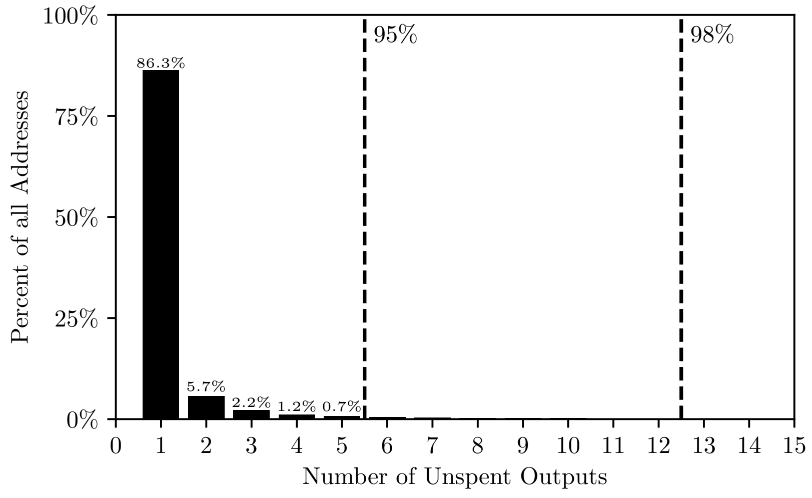


Fig. 6: Distribution of the number of unspent transaction outputs per active address in the Bitcoin network.

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 enclaves memory. In the worst case we store the lower and upper limits for addresses (20B) and transaction hashes (32B) for every ORAM block resulting in a maximum of $2^{18} * (32B * 2 + 20B * 2) \approx 26MB$.

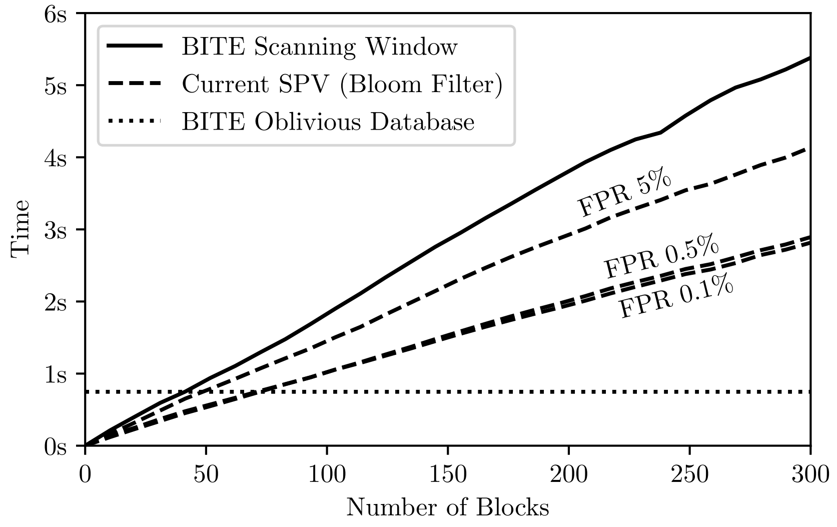
To set the message size, we analyzed the typical unspent outputs per active address in the Bitcoin network (Figure 6) and settled on 12 average outputs per request, resulting in around 1.2kB. This size is big enough to accommodate for more than 98% of all Bitcoin addresses currently in use.

We plan to release the code of our implementation online.

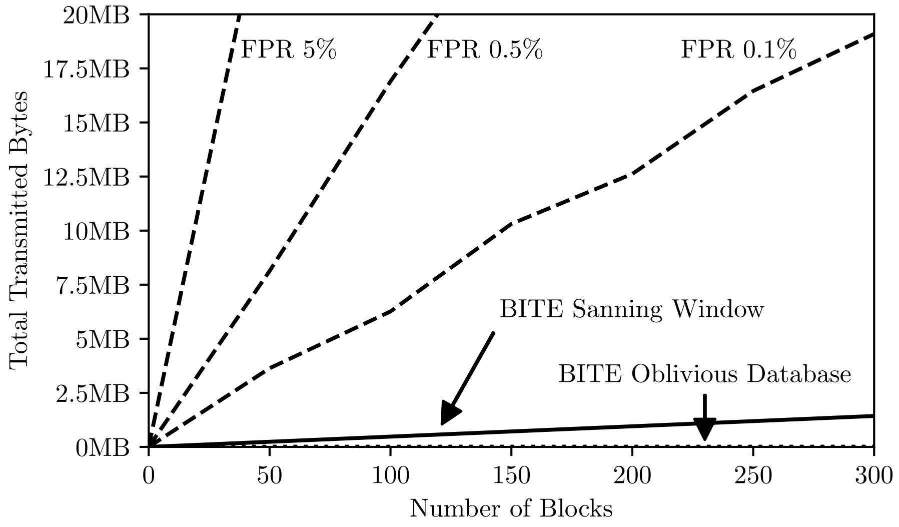
6.2 Performance Results and Comparison

In this section, we evaluate both variants of Bite and compare them to current SPV protocols. Note that in all our data points, the TLS handshake times are omitted. Matetic et al. [41] 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 Core i7-8700k processor clocked at 3.70 Ghz. The blockchain and the ORAM database were stored on a Samsung 960 Pro 512GB SSD. To compare with current SPV clients we used *python-bitcoinlib* [42].



(a) Server time comparison between Scanning Window, Oblivious Database and current SPV protocols using bloom filters. Note that the connection speed is assumed to be



(b) Bandwidth comparison between Scanning Window, Oblivious Database and current SPV protocols using bloom filters.

Fig. 7: Performance evaluation of Scanning Window and Oblivious Database.

Table 2: ORAM access times for various chunk sizes and the corresponding size (number of entries) needed to store the entire UTXO set. Time measurement averaged over 1000 runs.

Chunk size	Size	ORAM
32kB	2^{16}	74.77 ms (±15.52 ms)
64kB	2^{15}	108.26 ms (±33.91 ms)
128kB	2^{14}	172.01 ms (±38.98 ms)

Table 3: 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.

	t_m		
	5kB	10kB	20kB
Nr of Blocks 50	0.6s (± 0.2s)	1.2s (± 0.5s)	2.6s (± 0.9s)
100	0.7s (± 0.2s)	1.3s (± 0.5s)	2.7s (± 0.9s)
150	0.7s (± 0.2s)	1.4s (± 0.5s)	2.7s (± 0.9s)
200	0.7s (± 0.2s)	1.4s (± 0.5s)	2.8s (± 0.9s)
250	0.7s (± 0.2s)	1.4s (± 0.5s)	2.9s (± 0.9s)
300	0.7s (± 0.2s)	1.5s (± 0.5s)	3.0s (± 0.9s)

Scanning Window. In the Scanning Window variant, our approach is similar to the original SPV procedure. Both systems let the client request filtered blocks which results in scanning the blockchain. Previous work show that Intel SGX imposes significant overhead for copying buffers (reading files) across the trust boundaries [32].

Figure 7a shows the time needed to filter blocks by Scanning Window and current SPV protocols. We report an overhead of around 100% (in total the time is 5.3s) in comparison to Bloom filters with a false prositive rate of 0.1% (2.65s) to 0.5% (2.7s). Note that the measurements in Figure 7a do not account for the network speed. A device with a decent 4G connection that operates at 100Mbit/s requires additionally around 5s to retrieve 300 blocks with the current SPV protocol and a 0.1% false positive rate. For even higher false positive rates, i.e., 0.5% (the default value of *BitcoinJ*), an SPV client synchronizes in additionally around 8s. Our systems are not significantly impacted by the limited bandwidth of 4G. The synchronizing time for Scanning Window rises by 0.1s to around 5.4s (and Oblivious Database stays constant at 0.5s). Our systems can reduce the required bandwidth because no false positives have to be included to fool the attacker.

The scanning time is impacted significantly by the need to recompute the merkle tree (approx. 6.5ms). Storing the merkle tree for every block could lead to better performance but the required disk space would grow significantly.

Table 4: Update time for the ORAM of Oblivious Database solution averaged over 100 measurements.

Blocks	ORAM update time
1	78.5s (±13.6s)
3	112.5s (±19.8s)
6	146.7s (±19.8s)

Scanning Window with side-channel protection. Oblivious execution and memory access adds a significant overhead to Scanning Window. All branches have to be taken and all in-memory structures have to be touched in their entirety to hide access patterns. The merkle tree has to be recomputed for every block aswell but since generating a merkle tree for the average bitcoin block takes 6.5ms this does not contribute significantly to the runtime of the side-channel free Scanning Window.

Table 3 shows the time per block for various number of blocks requested and t_m size. Higher t_m allows to cope with some blocks that have a lot of relevant transactions while others do not, since it limits the amount of transactions of a single block that can be included in a response. Note that the blocks vary in size, and thus the time per block fluctuates a lot leading to a high standard deviation. Synchronizing 300 blocks with $t_m = 10$kB takes around 7.3 minutes corresponding to an overhead of approximately 100x.

Oblivious Database. In this variant, the unspent outputs are directly fetched from the enclave UTXO. Therefore, the time needed is independent of the number of requested blocks, yet only on the ORAM database access times. Figure 7a shows the Oblivious Database variant response time for a request containing 10 addresses and a ORAM block size of 32kB. Table 4 shows the time an update to the ORAM database takes for various blocks at a time. In order to reach permanent availability we propose the usage of at least 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 amount of clients that can be served by a single SGX enclave can be estimated by using around 120s for updating the state and then the remaining 8 minutes to continuously answer client requests, leading to an approximate 10000 clients per enclave. The message size is significantly lower than all other variants, since only unspent outputs are included and not the entire transactions and the corresponding partial Merkle path. The message size for a request of 10 addresses amounts to $10 * 1.2$kB $= 12$kB. Figure 7b shows bandwidth comparison between all discussed protocols.

7 Discussion

Usage models 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 request events and the Bitcoin addresses visible in the ledger at roughly at 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 lightweight client payment verification scheme.

In *recommended* usage of our solution, the client (i) uses different full nodes for payment verification, (ii) regularly uses fresh Bitcoin addresses (e.g., using an HD wallet [43]), and (iii) introduces unpredictability to the timing pattern of payment verification requests like a small number of extra requests at random points in time. If the client follows such a usage model, the above mentioned correlation becomes very difficult.[4]

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 and represents truly a minority of the complete set of transactions in the blockchain. 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 transactional 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 that the request is repeated and infer that the specific client has more transactions of interest in the designated blocks. To mitigate this problem one could obfuscate the IP address or change to another enclave for finishing the request.

[4] To quantify how accurately the adversary can correlate the client's addresses and how difficult to such correlation becomes with the above discussed 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 lightweight clients, we consider this task a research project on its own and outside the scope of this paper.

Deployment models. We consider two deployment models. In the first model, a verifiable company can run our solution as a service offering lightweight clients privacy. In the second model, any full node operator, or a volunteer, can operate our solution. In both deployment models we presume that multiple, or even all nodes, will support this solution. and the scalability depends on that number. One single node can only support around 10000 lightweight clients, and the overall success depends on the supply, demand and acceptance of such service. Such different deployments can enable different authentication models. In the first option, the company could use a PKI which would allow the lightweight clients to recognize which specific enclave they are communicating with. In the second option the clients only know that they are connecting to a correctly attested enclave, but they cannot make distinction between different enclaves. Since successful attestation guarantees expected enclave execution, our solution's privacy properties hold in both cases, unless Intel SGX is broken. We discuss SGX compromise in Section 5.

Denial of service. A malicious user might attempt denial of service (DoS) by asking for a very long scan window — incurring large processing times for full nodes and thus 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) and hard to prevent when introduction of fees is hard. 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.

Unbounded enclave memory. The performance of our system is mostly bounded by the slower disk operations. However, in case that future versions of SGX architecture 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 [44].

8 Related Work

In this section, we review related work that can be classified into two main categories: Bitcoin lightweight client privacy and SGX information leakage protection.

8.1 Bitcoin Lightweight Client Privacy

The idea of light clients for Bitcoin was already included in the Bitcoin paper by Satoshi Nakamoto [10] in the form of *Simple Payment Verification* (SPV). Hearn and Corallo later introduced Bloom filters [13] in BIP 37 [11] that allow a client to probabilistically request a subset of all transactions in a block to mask which addresses are in fact owned by the client. Gervais et al. later showed that the information leaked by the use of Bloom filters in Bitcoin poses a serious privacy risk and can in many cases enable the identification of client addresses [14]. Hearn – who introduced Bloom filters to Bitcoin – later addressed the issues [28], expanded on them, and discussed the difficulties of solving them.

To overcome this privacy issue, Osuntokun et al. recently proposed modifications to Bitcoin nodes and lightweight clients that move the application of the filter to the client [45]. In their protocol, 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.

While this approach likely provides more privacy than the protocol using Bloom filters, it still suffers from a number of shortcomings. First, the gained privacy largely depends on the client behavior and how well the client is connected to different entities. If the client does not request the filter headers from multiple entities[5] and uses another entity to then request the blocks, she can be easily tricked into revealing her addresses by using forged filters as follows: A node can prepare a filter that matches half of all addresses and send 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 continue reducing the possible set using a binary search approach by sending modified filters for the following blocks, which allows bitwise recovery of all 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 [29,30]. These techniques are, however, orthogonal to the problem of lightweight client privacy and thus out of scope for our work.

8.2 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.

The previous work that is probably closest to our solution is a system called Raccoon [23] that 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

[5] Even if the client connects to multiple different nodes to receive the filters, she cannot verify that they are not under the control of the same entity.

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 [24] that prevents cache leakage using hardware-based transactional memory features in processors; ZeroTrace [25] that provides a library for data structures that are protected using ORAM; DR.SGX [26] that randomizes and periodically re-randomizes all data locations in enclave's memory with cache-line granularity; and, T-SGX [46] and Deja Vu [47] 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 protection cache-based leakage. ZeroTrace is limited to data access protection and it does not prevent leakage from secret-dependent control flow. DR.SGX is also limited to data accesses and imposes 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).

Recently published Oblix [48] presents a new ORAM algorithm that is designed specifically for SGX. We use well-known Path ORAM, but our solution is agnostic to the used ORAM algorithm and we could easily replace Path ORAM with another algorithm.

9 Conclusion

Improved user privacy is one of the main benefits 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 lightweight clients, such as mobile devices, in Bitcoin.

References

1. Ethereum, 2018.
2. Elli Androulaki, Artem Barger, and et. al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. *arXiv preprint arXiv:1801.10228v1*, 2018.
3. Ripple, 2018.
4. R3, 2018.
5. Blockchain.info, 2018.
6. Etherscan.io, 2018.
7. BitcoinJ, 2018.
8. PicoCoin, 2018.
9. Electrum, 2018.
10. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
11. Mike Hearn and Matt Corallo. Connection bloom filtering. *Bitcoin Improvement Proposal*, 2012.
12. Light Ethereum Subprotocol (LES), 2018.
13. Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.
14. Arthur Gervais, Srdjan Capkun, Ghassan O Karame, and Damian Gruber. On the privacy provisions of bloom filters in lightweight bitcoin clients. In *Computer Security Applications Conference*. ACM, 2014.
15. Victor Costan and Srinivas Devadas. Intel SGX explained. In *Cryptology ePrint Archive*, 2016.
16. Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*. USENIX, 2017.
17. Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems*. Springer, 2017.
18. Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *European Workshop on Systems Security*. ACM, 2017.
19. Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks, 2017. http://arxiv.org/abs/1702.08719.
20. Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *SP*. IEEE, 2015.
21. Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *CCS*. ACM, 2013.
22. OpCodes: CMOV, 2018.
23. Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.
24. Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*, 2017.
25. Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. In *NDSS*, 2017.
26. Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization.

27. Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *USENIX Security*, 2017.
28. Mike Hearn. Bloom filter privacy and thoughts on a newer protocol, 2015.
29. Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in bitcoin. In *Financial Cryptography and Data Security*. Springer, 2013.
30. Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Internet Measurement Conference*. ACM, 2013.
31. Intel. Intel Software Guard Extensions - Developer Zone. Website, 2017.
32. Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, 2016.
33. Intel. Intel Software Guard Extensions developer guide, 2016.
34. Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.
35. Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. *USENIX Security*, 2017.
36. Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A Data Oblivious File System for Intel SGX, 2018.
37. Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
38. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.
39. Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.
40. ARM Limited. mbedTLS (formerly known as PolarSSL), 2015.
41. Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. DELEGATEE: Brokered Delegation Using Trusted Execution Environments. In *USENIX Security*, 2018.
42. Peter Todd. python-bitcoinlib, 2018.
43. Pieter Wuille. Hierarchical deterministic wallets. *Bitcoin Improvement Proposal*, 2012.
44. Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *SP*. IEEE, 2018.
45. Olaoluwa Osuntokun, Alex Akselrod, and Jim Posen. Client side block filtering. *Bitcoin Improvement Proposal*, 2017.
46. Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.
47. Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *Asia CCS*, 2017.

48. Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *SP*. IEEE, 2018.
49. Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. *Invisible Things Lab*, 2009.
50. Bernhard Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security*, 2007.
51. J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM*, 2009.
52. Intel SGX, Ref. No.: 332680-002, 2015.
53. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 1996.

A Intel SGX

Intel's SGX [15,**?**] 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 [49,50,51]. 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 [35] 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 [15,52].

B Oblivious RAM

Oblivious RAM (ORAM) [53], 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

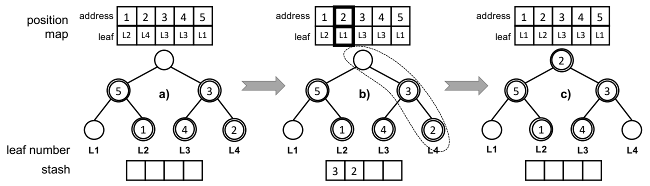


Fig. 8: a) The client wants to access the chunk 2 that is stored in Path ORAM. b) The position map specifies that the chunk 2 is on the path to leaf 4. Therefore, the server reads all entries on the path into the stash and re-randomizes the position map entry of the requested chunk. c) The server writes back as many chunks as possible on the previously read path.

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 [21] 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*
2. generate new random leaf for the database entry and insert it into the *position map*, then read all buckets along the path to the leaf and put them into the *stash*
3. if access is a write, replace the specified chunk in the stash with the new chunk
4. 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.
5. return requested chunk

Figure 8 shows an example of data access in Path ORAM.