ZLiTE: Lightweight Clients for Shielded Zcash Transactions using Trusted Execution

Karl Wüst[1⋆], Sinisa Matetic[1⋆], Moritz Schneider[1], Ian Miers[2], Kari Kostiainen[1], and Srdjan Čapkun[1]

[1] Deparment of Computer Science, ETH Zurich
[2] Cornell Tech

Abstract. Cryptocurrencies record transactions between parties in a blockchain maintained by a peer-to-peer network. In most cryptocurrencies, transactions explicitly identify the previous transaction providing the funds they are spending, revealing the amount and sender/recipient pseudonyms. This is a considerable privacy issue. Zerocash resolves this by using zero-knowledge proofs to hide both the source, destination and amount of the transacted funds. To receive payments in Zerocash, however, the recipient must scan the blockchain, testing if each transaction is destined for them. This is not practical for mobile and other bandwidth constrained devices. In this paper, we build ZLiTE, a system that can support the so called "light clients", which can receive transactions aided by a server equipped with a Trusted Execution Environment. Even with the use of a TEE, this is not a trivial problem. First, we must ensure that server processing the blockchain does not leak sensitive information via side channels. Second, we need to design a bandwidth efficient mechanism for the client to keep an up-to-date version of the witness needed in order to spend the funds they previously received.

1 Introduction

Decentralized cryptocurrencies offer the potential to revolutionize payments. By providing transparent means to audit transactions, they reduce the need to rely on trusted incumbents and allow new innovation on financial applications. But this same transparency renders nearly all cryptocurrencies completely unsuited for wide-scale adoption: all transaction are broadcast publicly in a manner that can be readily linked to real world identities [26,4], raising issues with government surveillance, harassment and stalking, and the viability of business competition when competitors can see all cash flow.

A variety of protocols have been proposed, such as Solidus [2,9], Cryptonote [37], Zerocoin [27] and Zerocash [5], that, with varying effectiveness [29,22,21], alleviate these issues. For example, in Cryptonote the destination address is always a newly generated one-time public key derived from the receiver's public key and some randomness from the sender. Zerocoin functions as an overlay on Bitcoin, where users mint a zerocoin and issue a transaction to transfer the

⋆ The first two authors contributed equally to this work.

funds to its commitment. The coin can be further spent by using zero-knowledge proofs. The most promising of these protocols, Zerocash, removes all information, such as sender/recipient identity, value, and linkability through the use of a zero-knowledge proof that there exists some past transaction which gave the user the funds they are spending. Zerocash is deployed in the cryptocurrency Zcash.

Payment Notification. Unlike in traditional means of payment, like credit cards and cash payments, in nearly all cryptocurrencies, including Bitcoin, Ethereum, and Zcash, it is possible to send money to a recipient's address without direct interaction or communication with the recipient. The recipient is paid, but only learns this next time when she is online. This raises the problem of *payment notification*, that is, the recipient must find out they were paid. Some cryptocurrencies, like Ethereum, use an account model where there is a single, well-defined location for payments. As a result, the recipient and, more significantly, anyone else, can see when and for how much someone is paid. Other cryptocurrencies, including Bitcoin and Zerocash, eschew this approach for improved privacy, storing payments individually as unspent transaction outputs (UTXOs) in unpredictable locations. In such systems, there must be some mechanism for users to discover a UTXO belongs to them. The simplest way to do this is to scan the blockchain and check each transaction.

Payment notification is a particular problem for privacy-preserving systems like Zcash. Transactions in Zcash, consist of an opaque commitment, a ciphertext, a serial number to prevent double spending, and a zero-knowledge proof of the transaction's correctness and the existence of funds to spend. In particular, there is no metadata to identify the sender or recipient. The only way for a client to identify if a payment is directed to them is by trial decryption of the ciphertext associated with a transaction: each transaction contains a ciphertext under the recipient's public key. To monitor for payments, clients must, therefore, conduct a trial decryption for *every* transaction on the blockchain. While this is completely feasible for well-resourced clients, running on platforms like standard desktop PCs, it is not desirable, nor often feasible, for resource-constrained clients like mobile devices where both power and bandwidth are major constraints. In this paper, we focus on such resource-constrained clients.

Light Client Model. Several cryptocurrencies address this problem with a model, exemplified by Bitcoin's Simplified Payment Verification (SPV) scheme [30], where "light clients" entrust a server (*full node*) to respond to queries about payments to a given address. The SPV protocol reveals to the server which (pseudonymous) addresses belong to a client and thus links multiple addresses together and potentially to real world identities, reducing user privacy. Directly applying the same model to Zcash is not possible without revealing the client's decryption key to the server so that it can perform the trial decryption for transactions, and thus completely breaking the privacy properties of Zcash.

Another challenge for resource-constrained clients is that simply notifying users that they received funds is not sufficient to use them for new payments in Zcash. To spend funds sent to them in a previous transaction tx in block n, users must prove that there exists a path (called *witness*) w from the root of a

Merkle tree (called *note commitment tree*) to tx. Moreover, this information is not static and it needs to be updated as new transactions are added to the tree.

Our Contribution. In this paper, we introduce ZLiTE, a system that enables efficient *privacy-preserving light clients for Zcash.* Our approach follows the common "light client and server" model, thus minimizing the client bandwidth and computation requirements by offloading processing to the server. To tackle the privacy problem of client queries, we leverage trusted execution, namely Intel SGX [19], on the server. This approach allows the server to perform the trial decryption for transactions without learning the client's key.

Although this approach is conceptually simple, realizing it securely requires overcoming technical challenges. First, *external* reads and writes from the SGX enclave to the blocks stored on the server or to response buffers can leak which transactions belong to the client. Second, SGX enclaves are susceptible to side-channel attacks [7,8,28,15,34,39] that can leak their *internal* memory access patterns. Secret-dependent code and data accesses can enable a malicious server to infer the used client's key. Third, our system also needs to ensure that the residing platform cannot mount a combination of eclipse attacks [18,38] on the blockchain and replay client messages to identify queried transactions. And fourth, we need to efficiently provide the client with up-to-date Merkle tree witnesses needed to spend funds from a given transaction without leaking any private information.

To address these challenges, ZLiTE combines, in a novel way, a number of known techniques from private information retrieval and side-channel resilient trusted execution, making the processing of client queries oblivious towards a powerful adversary controlling the supporting server. We also design a new commitment tree update mechanism that allows the client to obtain efficiently from the server all the needed information to spend the received funds.

Parallel Work. Finally, we note that, in parallel work, a similar solution has been suggested for privacy protection in Bitcoin [25]. While our overall approach is similar, the technical challenges that we address are specific to Zcash, and thus different. We review such parallel work in more detail in Section 7.

2 Background

Transactions in Cryptocurrencies. Many cryptocurrencies operate in the so called Unspent Transaction Output (UTXO) model. In this model, a transaction consists of a set of outputs, each with a numerical amount of money and an *address*, and a set of inputs each of which references the output of a previous transaction. For a transaction to be valid, the following conditions have to be met: (1) referenced outputs must exist, (2) inputs must be signed by the key specified in the referenced output address, (3) the $\sum$(output amounts) must be $\leq$ to the $\sum$(input amounts), and (4) referenced outputs must not be spent by a previous transaction.

In Bitcoin, this is accomplished by directly identifying the referenced outputs, checking that they are not referenced by any other transactions, and then checking the sum inputs and outputs. If a transaction validates, then the outputs it references are removed from the UTXO set and the outputs it generates added. Transactions in Bitcoin and most cryptocurrencies are validated via a peer-to-peer network and assembled into blocks (e.g., every 10 minutes), that are broadcast to the network.

In Zcash there are two types of transactions: transparent and shielded transactions. The transparent kind is directly derived from Bitcoin and will not be considered for the rest of this paper.

Shielded transactions also take some inputs and create new outputs, but the similarities end there. Outputs, also called notes, are created by so called *joinsplits* and are a commitment to an amount and the address it belongs to. A *joinsplit* takes a transparent input and up to two notes as input and creates one transparent output and up to two notes as output. However this information is encrypted and can only be inspected by the receiver. Additionally a Merkle tree is constructed over all notes on the blockchain forming the *note commitment tree*. A zero-knowledge proof forms the second part of the transaction and shows that conditions (1)-(3) hold with respect to that Merkle tree root. Because the "outputs" that a shielded transaction spends are not revealed, they cannot be removed from the UTXO set. Instead, a unique serial number, sometimes called a *nullifier*, is produced by the transaction that ensures the referenced outputs cannot be used again. This prevents double spending.

To perform operations in Zcash, each user has two keys associated to his shielded address. First, the *spending key* that is used during the creation of a zero-knowledge proof allowing the users to prove ownership of the received funds. Second, the *viewing key* that is used to decrypt the shielded transaction in the blocks and verify if each transaction belongs to the user.

Full Nodes and Light Clients. To interact with a cryptocurrency, one must have a client. In both Bitcoin and Zcash, the default client is a *full node*, which receives and validates every block, and contains the full state of the blockchain. Full nodes do not need to trust other entities, provided the system functions as assumed, e.g., for Proof-of-Work systems the majority of the network's computational power is honest and messages disseminate without problems. While full nodes offer the best security and privacy, they entail considerable resource usage. The computation and network resources necessary to maintain a full node are a major impediment and in some cases, e.g. mobile devices, simply prohibitive.

In contrast, *lightweight clients* are nodes that have smaller resource footprint. They were originally proposed for Bitcoin [30] as the *Simplified Payment Verifications* (SPV) scheme. In this proposal, clients store only the header of each block instead of the entire blockchain. This is sufficient to check the Proof-of-Work on each block and verify the presence of transactions by checking their inclusion in the Merkle tree whose root is contained in the block header: clients must merely request both a transaction and the witness to its inclusion in the Merkle tree from a full node.

The reduced resource usage of SPV clients comes at a major cost: privacy. As the light client must request individual transactions from a full node, it reveals which transactions and addresses belong to the requesting client. In Bitcoin, this allows multiple addresses to be linked together. In Zcash, this effect is far more pronounced since the client completely loses privacy: without such queries, no shielded transaction can be linked together, i.e., an adversary learns nothing.

ORAM. Encryption provides data confidentiality but access patterns can leak information possibly leading to reconstruction of the content itself. Oblivious RAM (ORAM) [14] is a popular scheme that hides access patterns and achieves fully oblivious data accesses. Most ORAM algorithms use randomized encryption and shuffling techniques to build a fully oblivious database. Intuitively ORAM hides the address, access patterns, whether the same data access is repeated and the type of operation, i.e. read or write. Note that ORAM operations still leak timing information related to the frequency of access operations themselves.

3 Our Approach

3.1 Requirements

The main goal of this paper is to design a solution that enables privacy-preserving light clients assisted by full-node servers for Zcash. More precisely, we specify the following requirements for our solution:

R1 **Privacy.** ZCash light clients should be able to privately retrieve all transaction related data without revealing sensitive information (e.g., viewing key, transaction count, blocks containing transactions) to the server.

R2 **Integrity.** The server that is assisting the light client should not be able to steal funds or make a client falsely accept a payment.

R3 **Completeness.** The retrieval of transactions should guarantee that the light client receives *all* data necessary for spending the funds they received.

R4 **Performance.** The solution should have minimal bandwidth and processing requirement for the client. The server's processing should be in the same order of magnitude as the normal full node operation.

3.2 Main Idea

Our main idea is to leverage commonly-available Trusted Execution Environments (TEEs) and apply them to full nodes (servers) to enable privacy-preserving light clients for Zcash. In particular, we use Intel's SGX [19] which provides isolated execution of security-critical application code, called *enclaves*, such that enclave data confidentiality and execution integrity remains protected from untrusted software such as other applications, the OS, hypervisor. In SGX, the CPU enforces that untrusted software cannot access enclave memory. For space reasons we omit details regarding Intel SGX. We refer readers unfamiliar with the technology to a more detailed SGX introduction [12,20].

Similar to SPV in Bitcoin, we assume deployments where the light Zcash clients may be assisted by *any number of* servers (full nodes) that support TEEs. Some of the servers could be run by well-known companies as commercial services where light clients may have to pay a small fee for the service. Other servers could be run by private individuals, like members of the cryptocurrency community, as a free service. As in SPV, the light clients are free to choose which servers to use, if any. In this regard, our solution retains the decentralized nature of Zcash.

3.3 Controversy and Challenges

The use of TEEs is often controversial. TEEs rely on a trusted authority to design a secure processor and issue some form of certification for it. Attestations from the TEE can be forged either via exploiting design flaws or by corrupting the provider and falsely claiming that an attestation came from a genuine piece of hardware. The hardware and software are frequently closed source and the manufacturers opaque. These kind of trust assumptions are frequently an anathema, especially for cryptocurrencies. Moreover, usage of TEEs often seems like lazy systems design choice, since, if one assumes fully trusted TEEs (e.g., none of the enclaves can be compromised, no side-channel leakage, full resilience on physical attacks etc.), solving many problems becomes relatively easy.

However, current TEEs including SGX enclaves have noteworthy limitations such as side-channel attacks that leak information and no resilience to physical attacks. We argue that the real research challenge is to leverage TEEs such that one can enable improved performance and privacy, but at the same time address the limitations of TEEs such as side-channel leakage. In the (unlikely) case that TEEs are fully broken (e.g., a new severe processor vulnerability is discovered), the system should fail gracefully. One example of graceful failure is that the affected clients' privacy may be reduced, but integrity of the system is preserved, i.e., in a cryptocurrency, no money is lost or stolen.

3.4 Adversary Model

In this paper, we consider the standard SGX adversary model where the attacker controls the OS and all other system software in the supporting server. In practice, the adversary could be a malicious administrator in a company that provides the full node service, an external attacker that has compromised the OS on the full node server, or a malicious individual operating a free server.

The adversary is able to perform digital side-channel attacks [7,8,15]. We assume that he is able to *perfectly* observe the enclave's control flow with instruction-level granularity and its data accesses with byte-level granularity (best known attacks are cache-line granularity). We overestimate the attacker capabilities, as all current side-channel attacks suffer from significant noise and cannot extract perfect traces in pratcice. By assuming such an adversary, we design our solution for future attacks that may be able to mount more precise side-channels. Additionally, the adversary has full control over the communication and can thus read, modify, block or delay all messages sent by the enclave.

The adversary *cannot* break the hardware protections of SGX along with cryptographic primitives such as encryption schemes and signatures. More specifically, the adversary cannot access SGX's processor-specific keys and the enclave's encrypted runtime memory protected by the CPU.

Finally, even if full compromise of SGX is outside our adversary model, we consider this possibility in our system design and discuss how our solution handles such worst case scenario in Section 5.3.

3.5 Strawman Solutions

We propose to leverage TEEs to protect the privacy of Zcash light clients. If client privacy relies on TEEs, it becomes natural to ask if one needs a complicated solution like Zcash and if anonymous payments can be realized through a much simpler solution using TEEs. To answer this question, we consider the limitations of a few strawman solutions.

Our first strawman solution is that clients send all transactions in an encrypted format to a set of authorized TEEs that process them privately. Such a solution would protect user privacy, but in case the enclaves get broken, the adversary can perform unlimited double spending on all users. Additionally, such a solution would not be decentralized.

Our second strawman solution is to use pseudonymous transactions that are published to a permissionless ledger, similar to Bitcoin, and mix them in one or more TEEs for improved privacy. Such a solution would prevent double spending, ensuring security for all users, even in the event that TEEs are broken. However, such a solution does not provide the same strong privacy protection, namely *unlinkability*, as Zcash, since the anonymity set for a transaction output only consists of the inputs of the mixed transaction. An adversary controlling the OS on the mixing service can further reduce anonymity by blocking incoming transactions or injecting his own.

Our third strawman solution is to use the Zcash system, due to its strong privacy properties, but allow light clients to offload their complete wallets to TEEs that perform new payments and notification of received payments for them. The main drawback of this approach is that if the TEE would be compromised, it would incur direct monetary loss for a high number of clients.

Our goal is to design a solution that enables light clients for Zcash, and thus benefits from its sophisticated privacy protections, but avoids the above discussed limitations of simple TEE-based solutions.

3.6 Solution Overview

In our solution, when a light clients wants to be notified about received funds or make new payments, she connects to one of the TEE-enabled full node servers, performs remote attestation of the server's SGX enclave, and establishes a secure channel to it.

To enable payment notification, the client sends its *viewing keys* for the addresses that she owns to the enclave and indicates from which point on (e.g., the latest known block to the client) she wishes to update the light client's state. The enclave obtains the data and information from the locally stored blockchain and processes it in a *side-channel oblivious* manner based on the client request and sends back the response to the client.

To enable new payments by the client, the server also prepares a witness for each new transaction of the client, as well as the note commitment tree update and sends them to the client. Given this information, the client can efficiently create new transactions, and the associated zero-knowledge proofs, using the received funds, without revealing his *spending key* to the enclave.

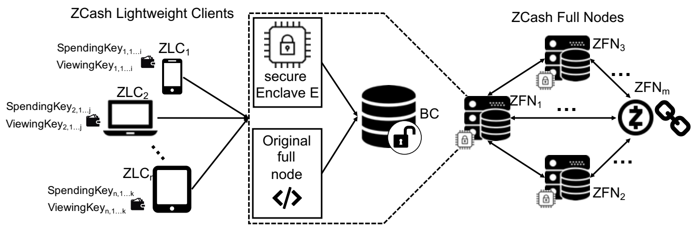


Fig. 1. System model. Lightweight clients request transaction verification and payment issuing service from enclaves hosted on full Zcash nodes.

4 ZLiTE System Design

4.1 System Model and Operation Overview

Figure 1 presents our system model. The main stakeholders in the system are Zcash Lightweight Clients $ZLC_1...ZLC_n$ and Zcash Full Nodes $ZFN_1...ZFN_m$. A lightweight client ZLC_a connects to any full node ZFN_b that supports our service by hosting an enclave E when she wants to acquire information regarding transactions and addresses that belong to the client or to issue new transactions towards another Zcash client. ZLC_a can own one or more addresses in her wallet that are also characterized by the $SpendingKey_{a,1...c}$ and the $ViewingKey_{a,1...c}$.

Full nodes maintain the local version of the blockchain (BC) as usual, appending each new confirmed block to the longest chain they have. The blockchain is maintained outside the secure environment, either on the disk or memory of the platform where the node resides. SGX enclave memory is limited (128MB) and is only suitable for smaller storage related to the currently executed task.

A client that wants to retrieve transactions, performs remote attestation for the ZLiTE enclave and then establishes a secure connection (TLS), through which she sends her viewing key and the height h of the last known block B_h. The enclave then scans the blockchain for transactions for this viewing key starting from B_h and obliviously moves them to a temporary *response ORAM* (rORAM) to hide which transactions are of the client's interest. Additionally, the ORAM structure is obliviously serialized in the response buffer sent to the client.

Oblivious Scanning. All processes that rely on secret data, i.e. the clients viewing key, must be performed in an oblivious fashion to prohibit any leakage of sensitive information (see Section 5). Finding the transactions that match the clients viewing key clearly depends on the client's secrets. To make block scanning oblivious to a side-channel observer (see the adversary model in Section 3.4), processing of each transaction should produce the same side-channel trace. A naive way to solve this is to do a fake copy of each non-matching transaction (viewing key does not result in a valid decryption) to the response buffer as well. However, in that case the response buffer is as big as the scanned blocks (no performance improvement). To improve the performance we use a *response ORAM* to hold all relevant transactions of the current client. The rORAM allows us to perform one ORAM operation per transaction while still hiding if this operation is a write (relevant transaction) or a read (irrelevant transaction). This is achieved by constant-time branchless code using the `cmov` instruction [32]. In conclusion, the enclave performs the following operations for each transaction:

(**1**) check if the viewing key manages to decrypt the transaction

(**2**) calculate the Merkle tree

(**3**) perform an ORAM operation (write or read transaction into the rORAM depending on the outcome of **(1)**)

Together with the transactions stored in the rORAM, ZLiTE delivers the corresponding Merkle paths, all block headers since B_h, and the note commitment tree update for the requested interval (see Section 4.3). Below we first describe the details of the ZLiTE operation and the retrieval of transactions and then describe how a lightweight client using our system can create new shielded transaction.

4.2 Transaction Retrieval

The operation of the synchronization protocol (see Figure 2) works as follows:

Initialization and continuous operation.

(**a**) On initialization the Full Node ZFN_j connects to the P2P network (**a-1**) and downloads the full ZCash blockchain (**a-2**). This locally stored blockchain is continuously updated as new blocks are received from the network.

(**b**) When the lightweight client is installed, it contains a checkpoint block header (this can be from a recent date or the genesis block). The client then downloads all newer block headers from the P2P network and verifies them (i.e. the client checks the PoW and that their hash chain leads to the checkpoint). All but a small number of the most recent block headers (to handle shallow forks) can be deleted afterwards. This state is later updated during the synchronization process that the client performs with a ZLiTE node in order to check for received transactions or before sending transactions (see below). This is similar to the operation of existing lightweight clients for other blockchains (e.g. Bitcoin).

Synchronization of Transactions. Clients synchronize with a ZLiTE enclave as follows:

(**1**) The ZCash Lightweight Client ZLC_i performs attestation with the secure Enclave E_j residing on the full node ZFN_j.

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

(**3**) The Lightweight Client ZLC_i sends a request containing its viewing key and the number of the latest known block.

(**4**) The Enclave E_j creates a temporary in-memory *response ORAM* (rORAM) to store the transactions that will be sent to the client. E_j then *scans* its locally stored copy of the blockchain (BC) starting at the block number specified by the client and decrypts the transactions with the specified viewing key. The decryption will either result in garbage or in a valid plaintext transaction. If the decryption is successful, E_j moves the transaction and the corresponding Merkle paths

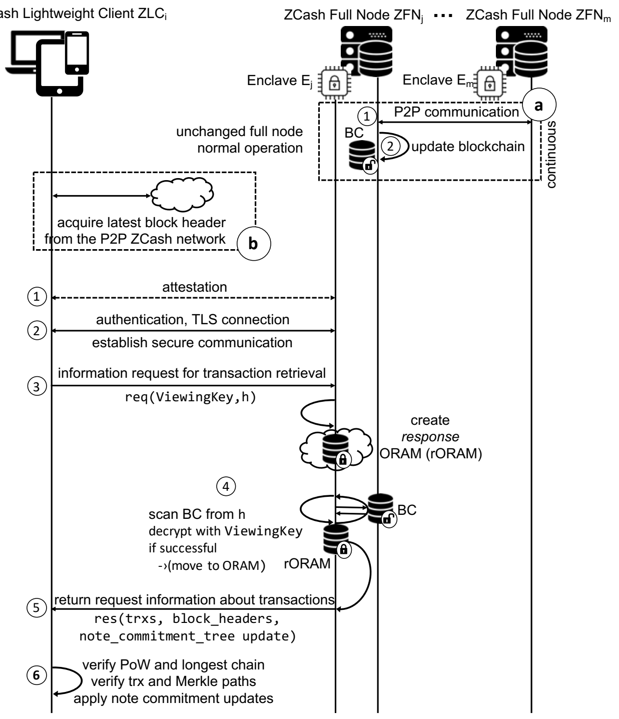

Zcash Lightweight Client ZLC$_i$
ZCash Full Node ZFN$_j$ ••• ZCash Full Node ZFN$_m$
Enclave E$_j$
Enclave E$_m$
a
P2P communication
unchanged full node normal operation
BC
update blockchain
continuous
acquire latest block header from the P2P ZCash network
b
attestation
authentication, TLS connection
establish secure communication
information request for transaction retrieval
req(ViewingKey,h)
create response ORAM (rORAM)
scan BC from h
decrypt with ViewingKey
if successful
->(move to ORAM)
rORAM
return request information about transactions
res(trxs, block_headers, note_commitment_tree update)
verify PoW and longest chain
verify trx and Merkle paths
apply note commitment updates

Fig. 2. Synchronization. The lightweight client establishes a secure connection to an enclave on a full node and sends a request that contains its viewing key and latest known block to perform the retrieval of all of her transaction information.

(for the transaction and for the note commitments) to the response ORAM, and if it is not successful, E_j performs a read operation, thereby performing the move obliviously using the `cmov` technique mentioned in Section 4.1 to replace conditional statements.

(**5**) After the scanning operation has finished, the rORAM is serialized by moving the entries to a fixed-size (dependent on the request, i.e., number of requested blocks for update) response array that is then sent to the client. In addition, the response contains all of the block headers and the note commitment tree updates (see Section 4.3 for details).

(**6**) The ZCash Lightweight Client ZLC_i verifies that the received block headers have a valid proof of work, create a chain to its latest known header and that the chain is the heaviest chain advertised in the P2P network. For every received transaction, it checks whether the recomputed Merkle root, given the received path, matches the corresponding block header. The client then updates the witnesses for all transactions with the received note commitment tree update and finally deletes old block headers that no longer need to be stored.

4.3 Transaction Creation

The lightweight client receives all information necessary to create shielded Zcash transactions from our system. Namely, for every output he wants to spend, he requires the witness (at the time of creating the new transaction) of the corresponding note commitment (i.e., its Merkle path in the note commitment tree).

These witnesses could be retrieved from a ZLiTE node at the time of spending. However, this would require the node to retrieve the witness in an oblivious fashion on request, which becomes computationally expensive as the commitment tree gets larger. Instead, when scanning the chain for a client, we additionally supply the witness of a note at the block height where it was created (see Section 4.2). When synchronizing, the client then also receives *commitment tree updates*, which allow him to update witnesses for any previous note commitment. In this case, there is no need for oblivious computation since the update only depends on the block height and not on the transaction relevant to the client.

Given a note commitment tree at time t_1 and a note commitment tree at time t_2, to compute the commitment tree update, the enclave starts with an empty list U_{ct} to store the update. Let cm_i be the latest note commitment in the tree at time t_1, i.e., it is the rightmost non-empty leaf. Then, in the tree at time t_2, for every node on the path from cm_i to the root of the tree, add the right child to U_{ct}. A client in the possession of a witness at time t_1 for some note, can then apply the update by replacing any node on the witness with the corresponding node from U_{ct}, if these two nodes have the same location in the Merkle tree. We present a proof in Appendix A that this construction results in a correct witness for the note commitment tree at time t_2.

5 Security Analysis

In this Section, we provide an informal security analysis of ZLiTE. We first discuss protection against information leakage, then discuss the completeness of responses, and finally consider the worst-case scenario, i.e. a full break of SGX.

5.1 Protection Against Information Leakage

Since ORAM reads and writes are indistinguishable, an adversary observing memory access patterns is not able to determine which transactions were written. For ORAM accesses, when accessing the stash, indexes or the position map, every location is accessed to hide memory access patterns.

To protect against side channels (e.g. [8,28,15,34]), conditional statements that depend on transactions (e.g. during the process of moving transactions to the response ORAM) are replaced using the `cmov` instruction. Since this results in the same control flow independent of the transaction, protection against leakage even against an adversary that can observe the control flow with instruction level granularity is guaranteed. The `cmov` instruction has been previously used to protect against side channels by Raccoon [32], Zerotrace [33] and also Obliviate [3] in the context of providing secure ORAM access using SGX. This prior research shows that `cmov` can effectively protect against digital side channels.

Finally, the response size only depends on the number of scanned blocks, i.e. it is independent of how many (or if any) transactions are in the response, and thus does not leak any information about a client's viewing key or transactions.

5.2 Integrity and Completeness

The ZLiTE node delivers the requested information along with all block information needed for simple payment verification. The client herself then verifies the block headers using the Merkle paths for her transactions. Similar to SPV in Bitcoin lightweight clients [30], this ensures that the server cannot make a client falsely accept payments for which the transactions are not included in the chain. As the client can also check the proof of work and gossips with the P2P network to receive block headers, she can ensure that she receives information from the longest chain. Thus, the server does not have stronger capabilities to eclipse a lightweight client than against a full node.

In contrast to standard SPV (as e.g. in Bitcoin [30]), where the client cannot be sure to have received all of her transactions, the usage of a TEE makes sure that the received response contains all of her transactions for the scanned interval given the ZLiTE node's view of the blockchain.

5.3 Impact of Full SGX Compromise

While our adversary model considers side-channel attacks, we do not consider a full compromise of SGX, i.e., forged attestations, arbitrary control flow change or enclave secrets reading. However, recent research has shown that secrets can be read even from the quoting enclave allowing an adversary to extract attestation keys [10,36] which makes it necessary to discuss such a worst-case scenario.

While it is obvious that the privacy provided by ZLiTE can no longer hold, if the adversary can read all secrets, or a client connects to a server that uses a forged attestation to impersonate an SGX enclave, such a breach cannot lead to loss of funds. In addition to the loss of privacy, a client also loses completeness, since a node may omit payments. However, because the client's spending key is never sent to a ZLiTE node and the client performs Simple Payment Verification for all of his transactions, a node is not able to steal coins from the client or make him falsely accept a payment.

5.4 Trust Assumptions Comparison

In terms of security properties like double-spending protection, Zcash relies on the following two trust assumptions: First, there must be an honest majority of mining power. Second, the dissemination of messages broadcast to the peer-to-peer network must be sufficiently good, i.e., no eclipse attacks. ZLiTE relies on the same trust assumptions as Zcash for its security properties.

For privacy, Zcash relies on securely-generated public parameters and hardness of numeric cryptographic assumptions. ZLiTE requires the same assumptions and additional trust in TEEs.

6 Performance Evaluation

6.1 Implementation Details

Our implementation of ZLiTE is based on the protocol specification of Zcash. It consists of a blockchain parser, an oblivious Path ORAM implementation [35] and it makes use of some bundled cryptographic libraries. We support the current Zcash protocol specification including the 'overwinter' protocol update.

The Trusted Computing Base (TCB) of our implementation can be split up into a network part that is responsible for the communication with a client (around 1.5k LoC) and the blockchain relevant part (around 3.7k LoC). Additionally we use well reviewed crypto libraries like *mbedTLS* (53k LoC) and small libraries that provide crypto primitives: sha256 , blake2b, ripemd160, ChaCha20Poly1305 and ed25519 totaling to around 2.2k LoC. All of the included crypto primitives come from well reviewed sources. We will not go into details on the TLS library *mbedTLS* [23] and refer the interested reader to [40,24] for implementation details and performance results.

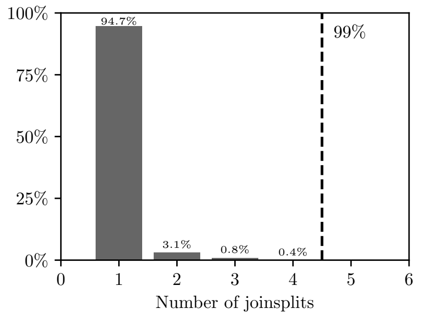
6.2 Performance

ZLiTE measurements were done on an i7-8700k processor with an SSD. Note that all the reported timing results are without the additional TLS latency. All measurements are according to the blockchain activity as of August 2018.

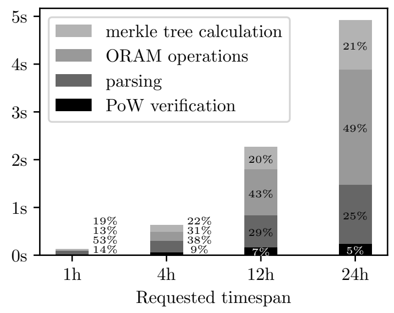
Lower Bound. Any node that wants to check for new transactions needs to parse the new blocks and test its viewing keys against all transaction in the blocks. This is part of the Zcash specification and implies a lower bound for any full node. Testing viewing keys is computationally intensive because it involves a key exchange based on an elliptic curve for each transaction and viewing key. Our implementation manages to parse blocks of an entire day and test a single viewing key against the transactions within 1.24s compared to fully oblivious operation of ZLiTE which takes around 5s. We have to retrieve the Merkle paths and perform at least one ORAM operation per shielded transaction while non oblivious solutions can skip this for all non-relevant transactions.

Average Transaction Size. We measured the average number of *joinsplits* in a shielded transaction and show a histogram in Figure 3a. Around 95% of all shielded transactions only contain one *joinsplit*, thus they have at most 2 shielded inputs and 2 shielded outputs. Every *joinsplit* occupies around 2KB of data. We also have to store the *commitment tree update* (see Appendix A) which is around 1KB in size. The average shielded transaction thus requires an ORAM operation for around 3KB of data. These measurements allow us to chose optimal ORAM block size for our response ORAM of 3KB.

Latency. We measured the time required to fetch various amounts of blocks and show a comparison between different expected client data per hour in Table 1. Note that the time per block rises when a client requests a longer time period because the response ORAM is chosen accordingly and a big ORAM database leads to slower accesses. Additionally, slower responses are observed when the client expects a lot of activity and requests a lot of client data.



(a) Joinsplit distribution in all shielded transactions up until block 350000.



(b) Enclave latency for various request sizes (with 24576B of client data/hour).

Fig. 3. Performance measurements.

Table 1. Total time for various request and response sizes. (100 runs)

		Client data per hour		
Time	Blocks	6144B	12288B	24576B
24h	576	4187ms ±504ms	4382ms ±510ms	4967ms ±617ms
12h	288	1875ms ±315ms	2122ms ±364ms	2317ms ±397ms
4h	96	541ms ±75ms	583ms ±92ms	631ms ±104ms
1h	24	123ms ±21ms	129ms ±21ms	130ms ±21ms

Figure 3b shows the latency for a request with 24576B of client data per hour and various requested time spans. The latency is further divided in the four main contributors to the total: parsing the block, proof of work verification, ORAM operations and generating the merkle tree. Note that the ORAM operations start to take the lions share of the latency as soon as longer time spans are requested.

Bandwidth. The required bandwidth can be split into a static part (not dependent on the number of blocks requested) and a dynamic part. The dynamic part is composed of the blockheader (1487B) and the private data per block that is used to return transactions to the client. For reasonable usage we estimate a lightweight client to have (at most) one transaction every hour occupying 12kB. This results in 1024B of private data per block and the total dynamic bandwidth accumulates to 2511B per block. The static part only consists of the *commitment tree update* and is therefore $29 * 32B = 928B$ large. A client that requests one day of blocks from our system gets a response of 1.38MB.

Increased Blockchain Activity. As of August 2018 shielded transactions are not very common on the Zcash blockchain (only 1.5 shielded txs/block). With single steps measurements we estimated ZLiTE performance with increased future activity. For 100 shielded transactions per block, a daily request would take 112s, while with an hourly one the latency would shrink to around 750ms.

7 Related Work

Privacy for Lightweight Clients. Nakamoto introduced SPV in [30] in order to enable light clients for Bitcoin. The straight forward application of SPV trivially sacrifices client privacy, which is why BIP 37 [17] introduced Bloom filters [6] to somewhat hide the client's addresses in requests. Gervais et al. showed that this only marginally improves privacy [13]. Recently, Bitcoin protocol changes were proposed where full nodes publish a filter for all transactions in a block and clients download the block if the filter matches one of his addresses [31].

Most closely related to our work, Matetic et al. recently used SGX to provide privacy to Bitcoin lightweight clients in a system called Bite [25]. While the main challenge was to efficiently protect privacy in a system that already provides light clients, we tackle the problem of enabling light clients in a system that provides privacy, but until now does not support operation of light clients. One notable difference between [25] and our work is that in Zcash spending previously received funds requires the witness to the transaction's inclusion in the Merkle tree of all transactions, and therefore client must obtain, in efficient manner, an up-to-date version of this witness to spend the funds.

Zcash Scalable Clients. Several proposals aim to lower the resource requirements for clients in Zcash. While protocol upgrades [1] have reduced the computational resources required to generate a transaction, they have not substantially changed the bandwidth or verification requirements.

Bolt [16] proposes privacy preserving payment channels in which clients conduct most transactions off chain in a fully private manner. However, the current version requires clients to either monitor the blockchain for channel closure using a full-node, or entrust a third party to do so. While this does not violate privacy, failures by the third party can result in monetary loss. No such risk of theft exists with ZLiTE even if TEE integrity is violated. Moreover, Bolt requires payers to have an existing relationship with the recipient or an intermediate payment hub. While promising, Bolt is not a full solution for bandwidth limited clients.

In [11], Chiesa et al. explore the use of probabilistic micro-payments as a way of increasing throughput. In this setting, a sequence of, e.g., 100 micro-payments for one cent, is approximated by paying \$1 with probability $\frac{1}{100}$. Thus only $\frac{1}{100}$ of transactions are actually issued. However, this is only suitable for small and frequently repeated payments. Moreover, it is unclear if it will reduce the total volume of transactions or simple free up capacity for even more transactions.

8 Conclusion

Zcash provides strong privacy for its users. Shielded transactions, however, require clients to download and process every block which is impractical for devices like smartphones, and consequently no mobile client that supports shielded transactions exists in the market. In this paper we have developed a new solution that enables light clients to create and receive shielded payments by leveraging a supporting server and a commonly available TEE. Usage of trusted execution, obviously, changes the original trust model of Zcash, but we argue that such a solution strikes a balance between the best possible privacy and the range of scenarios where Zcash can be used in practice. Thanks to our solution, development of mobile clients that support shielded transactions becomes possible and more users can benefit from the sophisticated privacy protections of Zcash.

References

1. Sapling (2018), `https://z.cash/upgrade/sapling.html`
2. Abraham, I., Malkhi, D., Nayak, K., Ren, L., Spiegelman, A.: Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. CoRR, abs/1612.02916 (2016)
3. Ahmad, A., Kim, K., Sarfaraz, M.I., Lee, B.: OBLIVIATE: A Data Oblivious File System for Intel SGX (2018)
4. Androulaki, E., Karame, G.O., Roeschlin, M., Scherer, T., Capkun, S.: Evaluating user privacy in bitcoin. In: International Conference on Financial Cryptography and Data Security. pp. 34–51. Springer (2013)
5. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: IEEE Symposium on Security and Privacy. pp. 459–474. IEEE Computer Society (2014)
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
7. Brasser, F., Capkun, S., Dmitrienko, A., Frassetto, T., Kostiainen, K., Müller, U., Sadeghi, A.: DR.SGX: hardening SGX enclaves against cache attacks with data location randomization, `http://arxiv.org/abs/1709.09917`
8. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software Grand Exposure: SGX Cache Attacks Are Practical. In: 11th USENIX Workshop on Offensive Technologies,WOOT 2017. USENIX (2017)
9. Cecchetti, E., Zhang, F., Ji, Y., Kosba, A.E., Juels, A., Shi, E.: Solidus: Confidential distributed ledger transactions via PVORM. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 701–717 (2017). https://doi.org/10.1145/3133956.3134010, `http://doi.acm.org/10.1145/3133956.3134010`
10. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H.: Sgxpectre attacks: Leaking enclave secrets via speculative execution. arXiv preprint arXiv:1802.09085 (2018)
11. Chiesa, A., Green, M., Liu, J., Miao, P., Miers, I., Mishra, P.: Decentralized anonymous micropayments. In: Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II. pp. 609–642 (2017). https://doi.org/10.1007/978-3-319-56614-6_21, `https://doi.org/10.1007/978-3-319-56614-6_21`
12. Costan, V., Devadas, S.: Intel SGX explained. In: Cryptology ePrint Archive (2016)
13. Gervais, A., Capkun, S., Karame, G.O., Gruber, D.: On the privacy provisions of bloom filters in lightweight bitcoin clients. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 326–335. ACM (2014)
14. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) **43**(3) (1996)
15. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache attacks on intel sgx. In: Proceedings of the 10th European Workshop on Systems Security. p. 2. ACM (2017)
16. Green, M., Miers, I.: Bolt: Anonymous payment channels for decentralized currencies. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 473–489 (2017). https://doi.org/10.1145/3133956.3134093, `http://doi.acm.org/10.1145/3133956.3134093`

17. Hearn, M., Corallo, M.: Connection bloom filtering. Bitcoin Improvement Proposal **37** (2012), `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`
18. Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on bitcoin's peer-to-peer network. In: USENIX Security Symposium. pp. 129–144 (2015)
19. Intel: Intel Software Guard Extensions, `https://software.intel.com/en-us/sgx`
20. Intel: Software Guard Extensions Tutorial Series (2016), available at: `https://software.intel.com/en-us/articles/introducing-the-intel-software-guard-extensions-tutorial-series`
21. Kappos, G., Yousaf, H., Maller, M., Meiklejohn, S.: An empirical analysis of anonymity in zcash. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 463–477 (2018), `https://www.usenix.org/conference/usenixsecurity18/presentation/kappos`
22. Kumar, A., Fischer, C., Tople, S., Saxena, P.: A traceability analysis of monero's blockchain. In: ESORICS (2). Lecture Notes in Computer Science, vol. 10493, pp. 153–173. Springer (2017)
23. Limited, A.: mbedTLS (formerly known as PolarSSL) (2015), `https://tls.mbed.org/`
24. Matetic, S., Schneider, M., Miller, A., Juels, A., Capkun, S.: Delegatee: Brokered delegation using trusted execution environments. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association (2018)
25. Matetic, S., Wúst, K., Schneider, M., Kostiainen, K., Karame, G., Capkun, S.: BITE: Bitcoin Lightweight Client Privacy using Trusted Execution. IACR Cryptology ePrint Archive **2018**, XXXX (2018)
26. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of bitcoins: characterizing payments among men with no names. In: Proceedings of the 2013 conference on Internet measurement conference. pp. 127–140. ACM (2013)
27. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous distributed e-cash from bitcoin. In: Security and Privacy (SP), 2013 IEEE Symposium on. pp. 397–411. IEEE (2013)
28. Moghimi, A., Irazoqui, G., Eisenbarth, T.: Cachezoom: How sgx amplifies the power of cache attacks. In: International Conference on Cryptographic Hardware and Embedded Systems. Springer (2017)
29. Möser, M., Soska, K., Heilman, E., Lee, K., Heffan, H., Srivastava, S., Hogan, K., Hennessey, J., Miller, A., Narayanan, A., Christin, N.: An empirical analysis of traceability in the monero blockchain. PoPETs **2018**(3), 143–163 (2018)
30. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
31. Osuntokun, O., Akselrod, A., Posen, J.: Client side block filtering. Bitcoin Improvement Proposal **157** (2017), `https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki`
32. Rane, A., Lin, C., Tiwari, M.: Raccoon: Closing digital side-channels through obfuscated execution. In: USENIX Security Symposium (2015)
33. Sasy, S., Gorbunov, S., Fletcher, C.: Zerotrace: Oblivious memory primitives from intel sgx. In: Symposium on Network and Distributed System Security (NDSS) (2017)
34. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware Guard Extension: Using SGX to Conceal Cache Attacks (2017), http://arxiv.org/abs/1702.08719

35. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 299–310. ACM (2013)
36. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In: Proceedings of the 27th USENIX Security Symposium. USENIX Association (2018)
37. Van Saberhagen, N.: Cryptonote v 2.0 (2013), `https://cryptonote.org/whitepaper.pdf`
38. Wüst, K., Gervais, A.: Ethereum eclipse attacks. Tech. rep., ETH Zurich (2016)
39. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: Security and Privacy (SP), 2015 IEEE Symposium on. pp. 640–656. IEEE (2015)
40. Zhang, F., Cecchetti, E., Croman, K., Juels, A., Shi, E.: Town Crier: An Authenticated Data Feed for Smart Contracts. In: CCS (2016)

A Commitment Tree Updates

As described in Section 4.3, the commitment tree update U_{ct} for the interval between time t_1 and t_2 consists of the right child of the path from cm_i to the root at time t_2, where cm_i is the rightmost non-empty leaf at time t_1.

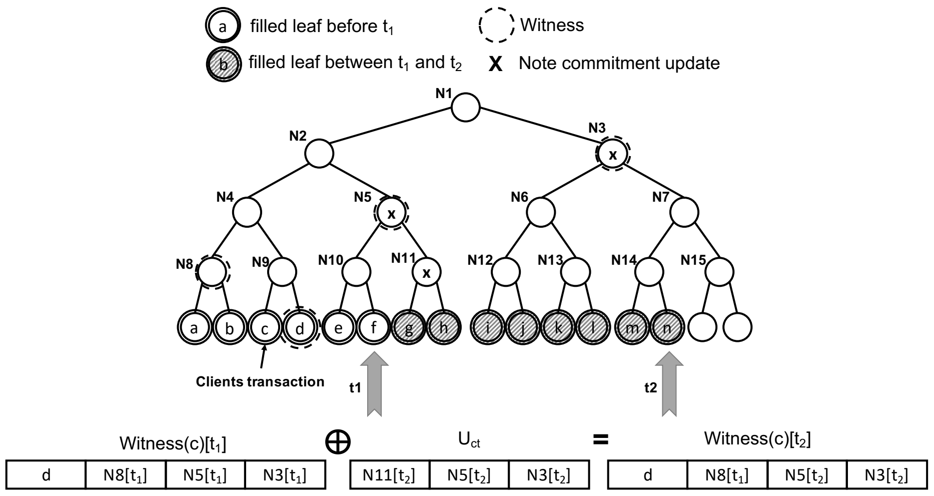

a filled leaf before t_1
b filled leaf between t_1 and t_2
Witness
X Note commitment update
N1
N2
N3
N4
N5
N6
N7
N8
N9
N10
N11
N12
N13
N14
N15
a b c d e f g h i j k l m n
Clients transaction
t1
t2
Witness(c)[t_1]
U_{ct}
Witness(c)[t_2]
d | N8[t_1] | N5[t_1] | N3[t_1]
N11[t_2] | N5[t_2] | N3[t_2]
d | N8[t_1] | N5[t_2] | N3[t_2]

Fig. 4. At a time t_1 the note commitments Merkle tree is fully updated up to the latest block. A specific client holds a transaction with a note commitment c and knows the witness (i.e. the Merkle path) for it (d, N8, N5, and N3 nodes). After some time the blockchain is updated and new transactions added, thus, the Merkle Tree is updated accordingly (t_2). In order for the client to update the witness of her commitment c, she only needs the updated information from nodes (N11, N5, N3).

In Figure 4, we show an example for the commitment tree update. In this example, the leaf f is the rightmost non-empty leaf at t_1, i.e. it corresponds to

cm_i, which means that the commitment tree update consists of the values of the nodes N11, N5, N3 at time t_2. In the example, the update is applied to the witness of the leaf c (consisting of the nodes d, N8, N5, and N3). In this case, the values of the leaf d and node N8 do not change between time t_1 and t_2, the values of N5 and N3 do, however, and thus the values are contained in the commitment tree update and updated from there.

We now show that given a witness at time t_1 for a commitment cm_j (where $j < i$, i.e. cm_j was added to the tree before cm_i) and the commitment tree update U_{ct}, a client can compute the witness for cm_j at time t_2.

Let A_{ji} be the lowest common ancestor node of cm_j and cm_i in the commitment tree, i.e. cm_j is in the left subtree of A_{ji} and cm_i is in the right subtree. Any node in the left subtree of A_{ji} remains unchanged between t_1 and t_2, i.e. any node from that subtree which is part of the witness for cm_j also remains unchanged. Since none of these nodes changes through the update process, updating the witness with U_{ct} results in the correct values.

Similarly, any node of the witness for cm_j that is a left child of a node on the path from A_{ji} to the root remains unchanged in the Merkle tree at time t_2, since all leafs in any left subtree are already fixed at time t_1 and thus all node values are already final. Since our update process does not change any left children in the tree, it also leaves these values unchanged and thus results in the correct values.

Finally, any node of the witness for cm_j that is a left child of a node on the path from A_{ji} to the root may change in the Merkle tree at time t_2. Since A_{ji} is an ancestor of cm_i, any such node is included in U_{ct}, i.e. these nodes on the witness are updated in our update process. These values are therefore changed to the correct values from the note commitment tree at time t_2.

It follows that the witness at time t_2 for cm_j can be constructed correctly given the witness at time t_1 and the commitment tree update U_{ct}.