

RouTEE: A Secure Payment Network Routing Hub using Trusted Execution Environments

Junmo Lee
Seoul National University
jmlee@altair.snu.ac.kr

Seongjun Kim
Seoul National University
sjkim@altair.snu.ac.kr

Sanghyeon Park
Seoul National University
lukepark@altair.snu.ac.kr

Soo-Mook Moon
Seoul National University
smoon@snu.ac.kr

Abstract—Cryptocurrencies such as Bitcoin and Ethereum have made payment transactions possible without a trusted third party, but they have a scalability issue due to their consensus mechanisms. Payment networks have emerged to overcome this limitation by executing transactions outside of the blockchain, which is why these are referred to as off-chain transactions. In order to establish a payment channel between two users, the users lock their deposits in the blockchain, and then they can pay each other through the channel. Furthermore, payment networks support multi-hop payments that allow users to transfer their balances to other users who are connected to them via multiple channels. However, multi-hop payments are hard to be accomplished, as they are heavily dependent on routing users on a payment path from a sender to a receiver. Although routing hubs can make multi-hop payments more practical and efficient, they need a lot of collateral locked for a long period and have privacy issues in terms of payment history.

We propose *RouTEE*, a secure payment routing hub that is fully feasible without the hub’s deposit. Unlike existing payment networks, *RouTEE* provides high balance liquidity, and details about payments are concealed from hosts by leveraging trusted execution environments (TEEs). *RouTEE* is designed to make rational hosts behave honestly, by introducing a new routing fee scheme and a secure settlement method. Moreover, users do not need to monitor the blockchain in real-time or run full nodes. They can participate in *RouTEE* by simply verifying block headers through light clients; furthermore, having only one channel with *RouTEE* is sufficient to interact with other users. Our implementation demonstrates that *RouTEE* is highly efficient and outperforms Lightning Network that is the state-of-the-art payment network.

I. INTRODUCTION

Decentralized blockchains such as Bitcoin [1] and Ethereum [2] have been playing an important role by providing a payment infrastructure for mutually distrusting parties. But their lack of scalability is one of the biggest limitations to overcome. Bitcoin, for instance, can execute a maximum of only 7 transactions per second, which is absolutely insufficient to replace existing payment systems. However, previous work [3] proved that it is not that effective to solve this inherent problem by simply changing parameters of Bitcoin (e.g., block size).

Payment networks such as Lightning Network [4] and Raiden [5] have been proposed to process payments with off-chain transactions, achieving higher transaction throughput. These transactions are not written in blockchains, so their performance is not limited by blockchain protocols. To execute *off-chain* payments between two users, they open a payment

channel by broadcasting a transaction (i.e., an *on-chain* transaction) to the blockchain network and locking their deposits. If users are connected within the payment channel network, routing users between a sender and a receiver can convey the sender’s balance to the receiver through their channels, and collect routing fees from the sender, which is called a *multi-hop payment*.

However, payment networks suffer from various shortcomings. Once a payment channel is created, there is no way to withdraw some of the balance or to add additional deposits: users have to close the channel and open another one. Furthermore, payment channels should be settled at the latest balance state, but malicious users could try to settle channels at a past state. Constant surveillance is thus required for the underlying blockchain to prevent such attacks. In addition, various conditions must be satisfied to complete the multi-hop payment. Senders should keep tracking how the channel network topology changes to find proper payment paths to receivers. Intermediary routing nodes must be on-line, of course, and all channels on the payment path must possess a greater balance than the sender’s payment amount.

A star graph with one central hub node is one of the most efficient topologies for payment networks [6]. Users need only one channel with the hub and do not have to struggle to find payment paths. In addition, hubs would reduce the length of the multi-hop payment path and thus the total routing fees that a sender must pay. Despite this efficiency, these central nodes are impractical to employ. There is a privacy issue in that they can obtain most of the private information about payments that occurred in the network. Moreover, to deal with numerous transactions, hub nodes should lock a tremendous amount of deposits in their numerous channels.

A key idea in this work is to overcome these flaws by utilizing trusted execution environments (TEEs). An example of a TEE is Intel Software Guard Extensions (SGX) [7], [8], a prominent TEE product operated by extended instruction set architecture in recent Intel CPUs. SGX supports hardware-based memory encryption for application codes and data in memory in order to securely isolate them from adversaries. Sensitive data is stored in independent regions of memory, called *enclaves*, which are inaccessible to other processes even with higher privilege levels.

We describe *RouTEE*, a secure payment routing hub, providing a new payment network system. It makes multi-hop

payments efficient and confidential, and does not require hosts to stake their assets. Users who have the channel with RouTEE can easily execute multi-hop payments via RouTEE, with less cost for verification. It is very cost-efficient because senders pay a routing fee only to RouTEE, and users do not have to open multiple channels, unlike existing payment networks. Due to SGX, the RouTEE protocol becomes concise and prevents rational hosts from misbehaving.

This work's main contributions are as follows:

- **Secure Payment Hub.** Only one channel is enough to fully utilize RouTEE's features and payment details are securely protected by a TEE. Basically, a user can open a channel with RouTEE through an on-chain transaction, similar to other payment networks. But in RouTEE, there is a way to create the channel and get balance by executing off-chain transactions. In addition, users can put additional deposits into their channels and withdraw part of their balance, which means that RouTEE provides high balance liquidity. It's noteworthy that these channels do not contain any assets of RouTEE's host (i.e., only users' collateral), making it easy for multi-hop payments to succeed with fewer routing fees.
- **Less Burden on Users.** RouTEE only requires users to run light clients to verify block headers, and this verification does not need to be processed in real-time. Precisely speaking, users might need to check whether a specific block is included in the blockchain when other users want to pay them. Furthermore, there is no additional data (such as penalty transactions) that users must store in other payment networks, and users execute multi-hop payments easily without knowing the network state (i.e., the network topology, the channel's balance state).
- **New Routing Fee Scheme.** It is important not to let the host unexpectedly abort operating RouTEE as RouTEE manages a lot of deposits and should store them safely. So we have introduced *pending routing fees*, which motivate incentive-driven hosts to keep running RouTEE and settle users' balances properly. In other words, hosts gain their entire pending routing fees only after every user has settled their whole balances.
- **Secure Settle Protocol.** RouTEE collects all of its deposits together to make secure on-chain settlement transactions, which we call *spend-all-settlement*. In this way, RouTEE can prevent feeding fake chain attacks, where adversaries insert fake blocks not included in the main chain to deceive RouTEE as if they actually sent their assets. *spend-all-settlement* also improves anonymity since it can be regarded as a mixing service.

We implement the RouTEE prototype using SGX for Bitcoin. We note that only one TEE is required to establish the RouTEE network. With only one RouTEE hub node, RouTEE can deal with more than 18,000 payments per second. Its throughput can be highly improved by batching payments.

II. BACKGROUND

A. Blockchains

Since Bitcoin [1] appeared in 2008, there have been many other cryptocurrencies. In Bitcoin, which is the most famous example of *proof of work* (PoW) [9] based blockchains, nodes are connected over peer-to-peer networks. Users make transactions when they want to pay other users and broadcast them with their cryptographic signatures to verify whether the owner of this asset has made this transaction or not. Then miners collect these transactions and try to solve a computationally expensive puzzle to make a valid block. A valid block means that it has a lower hash value than a target hash value. The target value is adjusted every 2,016 blocks (i.e., about two weeks) so it takes about 10 minutes on average to solve the puzzle. A block with a lower hash value than other blocks is considered to have a higher block difficulty. When a miner finds a valid block, it is broadcast to the blockchain network and appended to the chain of blocks that is append-only. Due to this hard work, blockchain data is difficult to revert.

When blocks with the same block number are made around the same time, it is described as a *fork*. In Bitcoin, the *longest chain rule* determines which block is valid. Each miner chooses a chain to continue mining, then one chain will become the longest chain and have the largest total difficulty, due to the difference in mining power for each chain. At that time, miners who were mining for other shorter chains move to the longest chain, and this chain becomes a *main chain*.

Many cryptocurrencies, including Bitcoin, use an *unspent transaction output* (UTXO) model. Transactions consist of inputs and outputs. Outputs represent an asset that can be spent freely by the owner. To make a transaction, senders select some of their unspent outputs as inputs of the transaction. Then they create new outputs for receivers, which means that the asset ownership has been moved to them. An output is locked with a lock called *ScriptPubKey*, which can be unlocked with a key called *ScriptSig* that only an owner of the output can create using its private key. These locks and keys are implemented in Bitcoin's Script language. Furthermore, all unspent outputs cannot be used more than once to prevent a double-spending attack (i.e., sending the same asset to several different receivers).

If a user wants to verify transactions, it has to run a full node to download all blockchain data including block headers and transactions, consuming plenty of network and storage resources. Then a user needs to verify every block header and transaction, which costs a lot of time and computation. A light client is suggested to deal with this problem. It only saves block headers, which require much less storage, and checks that they have actually solved PoW puzzles in a very short amount of time. Now users are able to verify that a certain transaction is included in a certain block with the block's header and the transaction's Merkle proof, which is called *simplified payment verification* (SPV) [1].

B. Payment Networks

Payment networks such as Lightning Network [4] and Raiden [5] have emerged to solve the low scalability problem of blockchains. As on-chain payment transactions are very slow and expensive to execute, payment networks process payments privately with off-chain transactions through payment channels. This method massively improves their transaction throughput as their performance is not limited by the blockchain anymore. Payment networks also reduce the blockchain's storage because off-chain transactions are not written to blockchains. Only two transactions are recorded to open and close a payment channel (in normal situations where there is no dispute).

Users need to establish payment channels first to pay through payment networks. For instance, two users, Alice and Bob, make a funding transaction, which will lock their deposits, and broadcast it to peers in the blockchain network. Once they confirm that their funding transaction is included in the blockchain, they are considered to have opened a payment channel. Balances in the channel are determined by the amount of the deposit. Then, when they want to pay, they make a new off-chain payment transaction that changes their balance states and exchange their signatures for it.

If there are channels between Alice and Bob, and Bob and Charlie, then Alice cannot directly pay Charlie because there is no channel between them. To make such payments possible, payment networks offer multi-hop payments. A sender can execute multi-hop payments when a receiver is connected to the sender through multiple channels. For the above example, Alice first sends her balance to Bob with a routing fee for him. Next, Bob passes over the same amount of balance to Charlie through their channel. In this case, Bob is acting as a routing node for the multi-hop payment, and there could be more routing nodes. It should be noted that these channel state changes occur atomically, meaning that there is no circumstance where only some of the channels in the payment path are modified. This property is achieved by a Bitcoin's feature called *hashed time-lock contracts* (HTLCs) in the Lightning Network.

When neither user (neither Alice nor Bob) wants to pay any longer, they can bilaterally settle their channel by broadcasting a settlement transaction that distributes their on-chain deposit to them immediately. Even if Alice is against closing the channel, Bob can unilaterally settle it by broadcasting any one of the off-chain payment transactions. In that case, Alice and Bob will be settled after a certain time limit if he broadcast the latest transaction. But if Alice detects Bob's malicious settlement attempts to broadcast a previous transaction rather than the latest one, she must broadcast the penalty transaction within the time limit in order to forfeit all channel deposits.

C. Disadvantages of Payment Networks

Even though payment networks can handle a lot more payments, other problems arise. Once a user's balance is exhausted, the user is no longer able to make payments through that channel. All the user can do is close the channel and create

a new one, which incurs two expensive on-chain transaction fees. Likewise, withdrawing a fraction of the balance or transferring it to other channels is impossible, which implies that payment networks have low liquidity.

As mentioned above, any user can unilaterally close channels with the state at their own advantage. Therefore, users must keep downloading new blocks and watching all transactions to find malicious settlement transactions (i.e., running blockchain full node continuously). In other words, payment networks cost massive storage and network resources to achieve high scalability.

Although multi-hop payments seem convenient as they allow paying users without a direct payment channel, there are many restrictions. Users should broadcast and collect information about opening and closing channels to find out how payment networks are connected. Then, they need to search for proper payment paths in which all channels have a balance greater than the payment amounts. Unfortunately, to protect privacy, channel balance information is not provided to other users, so it is hard to know which path could complete the multi-hop payment. There is no choice but to try the multi-hop payment through various paths until it succeeds. Users may have to split the payment amount within several multi-hop payments in order to complete the payment. Moreover, considering that the mean shortest payment path length is about three in the Lightning Network [10], and that a multi-hop payment through the shortest path could fail, senders might end up paying for two routing nodes on average, or even far more. We note that users open seven channels on average in the Lightning Network [10], which means that users pay a lot of channel creation costs and that they need to split their assets.

Moreover, there are privacy and security issues. Recent research [11], [12] shows that there is a way to figure out how much balance each user in the channel has, which can lead to payment information leakage (e.g., who the sender or receiver was, which routing nodes were used, how much amount to pay). (e.g., who was the sender, the receiver, or routing nodes, how much amount to pay). A *wormhole attack* [13] allows two attackers on a payment path to intercept routing fees for other routing nodes between them. In addition, it has been proven that adversaries are able to interrupt other routing nodes to monopolize routing requests, by holding the balances of other nodes for a long period [14] and exhausting their channels deliberately [15].

III. DESIGN

A. Design Overview

RouTEE performs as a payment routing hub which makes multi-hop payments more practical, keeping private data confidential within SGX. RouTEE can support any UTXO-based blockchain such as Bitcoin or Litecoin as it only utilizes UTXO's features and simple transactions that just transfer coins.

There is a set of users who participate in RouTEE, and a host who runs the RouTEE platform using an SGX-enabled

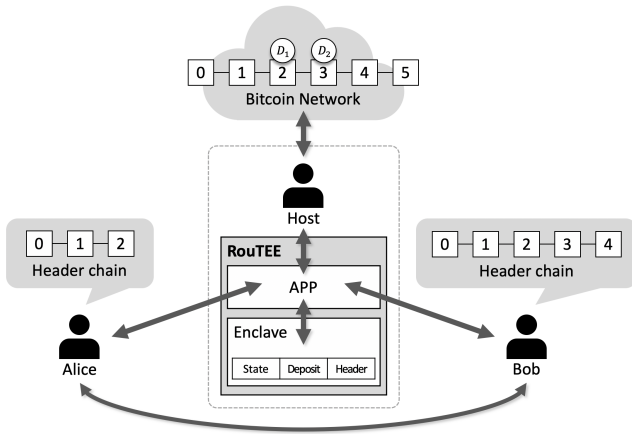


Fig. 1. The RouTEE system overview.

machine and feeds blockchain data into SGX. We note that users are able to verify that the RouTEE program is actually executed in SGX and build secure communication sessions through the *remote attestation* properties of SGX [8], [16]. First, SGX makes a *measurement* which consists of program codes, the enclave state, and additional data (e.g., a public key to establish secure sessions). Then it signs the measurement with its private key concealed within it. After that, users can verify this signature through Intel’s online attestation service, having confidence that the signature is truly created by SGX and that it is executing the correct RouTEE program.

It is important to separate channel creation from deposit locking in RouTEE, meaning that users can build channels even with off-chain transactions that do not require expensive on-chain transaction fees for blockchain miners. To be more specific, like existing payment networks, users can open channels and add balance at the same time via on-chain transactions, or they can simply open channels that contain no balance through off-chain transactions. This separation also allows users to add more deposits to their channels or withdraw their balance partially without the channel closing.

We also note that RouTEE makes multi-hop payments way more achievable. For users, tracking the network topology is no longer necessary to find proper payment paths, since every user is connected via RouTEE. This implies that RouTEE is a unique routing node and that senders have fewer routing fees to pay for their multi-hop payments. Moreover, users can determine whether their multi-hop payments are valid or not with a simple block header verification. For RouTEE, it just delivers the senders’ assets to receivers, meaning that RouTEE’s asset is not required. Accordingly, there is no case where multi-hop payments fail as RouTEE has not enough balance, and the host can operate RouTEE without any collateral.

Fig. 1 describes RouTEE’s overall system. The host first initializes RouTEE by feeding blockchain data until it reaches the latest block. Then users must verify that RouTEE is initialized successfully within SGX and establish secure sessions through remote attestation. All important data is kept safe inside SGX’s

enclave. The host and users interact with SGX through an interface process that is not protected by SGX. This means that the host has full controls of the process and can manipulate it. Users should maintain their own header chains, but their perspectives on the blockchain (e.g., what is the latest block number, which block is correct) might be different.

In order to interact with RouTEE, users need one of the three types of channels: *send-only channels*, *receive-only channels*, and *bidirectional channels*. There being no balance in RouTEE at first, users who want to be multi-hop payment payers broadcast on-chain *deposit transactions*, opening send-only channels and gaining balance in their channels. Deposit transactions will be included in some blocks, which we call *source blocks*. These transactions are automatically applied to user states inside SGX, as the host should insert every newly created block and transaction. Users who are going to be payees set their *boundary blocks* by simple off-chain transactions, building receive-only channels. A boundary block is the latest block among blocks that a user believes valid. That is, users cannot receive from other users who have balance derived from newer source blocks than their boundary blocks. To sum up, users with more than a 0 balance and users who set their boundary blocks can be considered as having send-only channels and receive-only channels, respectively. Users who meet both conditions have bidirectional channels. Fig. 2 shows RouTEE’s network example with various types of channels.

When both the payer and the payee are ready, multi-hop payments can be executed. Senders pay RouTEE *pending routing fees* for their multi-hop payments. Pending means that these routing fees are not yet in the host’s possession. To prevent the host from abnormally quitting RouTEE operations, pending routing fees are confirmed only after users settle their balance successfully.

Users can receive on-chain assets (e.g., BTC in Bitcoin) by settling their balances in RouTEE. RouTEE makes *settlement transactions* that use all deposits it owns inside SGX to securely settle a batch of users who requested a settlement. This is called *spend-all-settlement*, which effectively protects on-chain assets from misbehaving hosts who try to steal them by exploiting their fake deposits. Then rational hosts will honestly broadcast settlement transactions to the blockchain network to confirm their pending routing fees.

B. Adversary model and assumptions

Using TEEs is essential to building the RouTEE system. SGX is one of the well-known products of a TEE, but it is not perfect in terms of security. Recent works revealed several vulnerabilities of SGX [17]–[21]. This means that, in practice, some of SGX’s properties such as integrity and confidentiality might not be guaranteed. However, a great deal of research has been conducted to surmount these limitations [22]–[26], and any other TEE (e.g., ARM TrustZone [27]) can also be employed to implement RouTEE. So in this paper, we assume that there is no security issue with SGX itself.

Since RouTEE is operated atop of the blockchain, some basic properties of the blockchain should be satisfied. First,

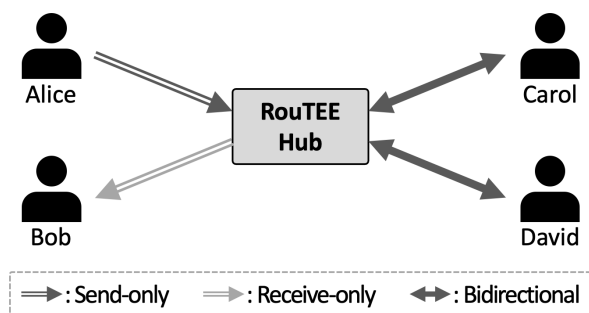


Fig. 2. An example of the RouTEE network. It forms a star graph which has RouTEE as a unique intermediary node. Each user has one of three kinds of channels with RouTEE, but channels between users are not necessary.

any valid transactions with a proper amount of transaction fee are included within the reasonable upper-bounded time period. Though adversaries could bribe miners not to contain certain transactions in their blocks, it must cost a lot to keep miners corrupt for a long period. Second, blocks with more than k -confirmations are not removed from the blockchain. In other words, a block followed by more than $k-1$ blocks can be considered immutable. k can be any integer bigger than 0, but commonly k is 6 in Bitcoin. This assumption also indicates that there is no attacker who can change the main chain of the blockchain with tremendous hash power.

We assume all participants are rational and incentive-driven. Their goal is to maximize their own interests. This assumption is also applied to adversaries (i.e., *rational adversaries* [28]), so they never choose strategies that result in financial losses for themselves, even if the strategy brings greater losses to victims. There might be *byzantine adversaries* [29] who do not care about their payoff and act irrationally (e.g., the host could turn off the RouTEE’s power, abandoning all its accumulated pending routing fees). However, they are quite unrealistic so we are not concerned about them. If we attempt to mitigate this kind of vulnerability of a hub struct, we could separate the hub into several nodes, backup every payment and state, and introduce more complicated protocols. Still, this dilutes the benefits of the hub, which implies that there is a trade-off between security and performance. So we concentrate on inducing rational adversaries to behave honestly.

C. Design Challenges

Misbehaving Host. There are two types of participant in RouTEE: a user and a host. The host has more authority and attack strategies than users, considering that it can directly access RouTEE and be the user. For example, the host is actually located between RouTEE and the users, passing messages between them. Although the host cannot find out message contents as messages are encrypted, the host can deliberately delay or drop them. Furthermore, blockchain data is important for operating RouTEE correctly, and the host plays the role of data feeding by running a blockchain full node client. Though basic block verification is performed inside RouTEE’s SGX, the host with moderate mining power and

sufficient time can make and insert a fake blockchain because there is no other chain in SGX to compare for determining which chain is valid based on the longest chain rule. That is, the host can deceive RouTEE as if he had staked the deposit and can get invalid balance inside RouTEE. Then by paying or settling the balance, the host will gain illegal profits. Executing all full node codes inside the SGX could be the solution, but it increases the size of the *trusted computing base* (TCB) too much, making the solution impractical as it would provide various attack vectors to adversaries. Besides, if aborting RouTEE costs nothing (i.e., there is no asset of the host inside RouTEE), the rational host could quit running RouTEE, and unsettled balances inside RouTEE would totally disappear. To protect RouTEE from irresponsible hosts, we have introduced an appropriate incentive model to make honesty the best strategy along with secure protocols to prevent illegal payments and settlements.

Verifying Blockchain Data. In existing payment networks, all users must observe every transaction to prevent invalid settlements due to their mutual distrust. However, it is a huge waste running full node clients to detect illegal attempts, because blockchain read/write operations are expensive and most transactions are irrelevant to users. To resolve this inconvenience, we leverage SGX and light clients. Since payments and settlements are performed securely inside SGX, no one can manipulate user states or produce invalid settlement transactions. Thus, users do not have to keep watching the blockchain in real-time. Instead, light-weight verification with block header data from light clients can efficiently preclude malicious payments. All users have to do is simply verify block headers and update their boundary blocks, before they become multi-hop payment payees. We emphasize that this process is not required for every payment.

IV. PROTOCOL

This section describes RouTEE protocol details, including how hosts initialize RouTEE and feed blockchain data, along with how users create channels, add deposits, execute multi-hop payments, and settle their balances.

A. Initialization

There are a few things that the host has to do to start RouTEE. The host must register the host’s public key to prevent users from executing host-only operations such as inserting Bitcoin blocks. Then the host sets a minimum amount of routing fee per multi-hop payment. Other users cannot modify the amount since that operation needs the host’s cryptographic signature.

Next, block headers need to be inserted inside SGX until it reaches the latest block. Although the header chain verification is a much lighter process than the full blockchain verification, it takes a lot of time to check every block from the genesis block (e.g., In Bitcoin, there are about 655,000 blocks as of November 2020). To shorten this procedure, it can start from one of the *checkpoint blocks*, which are hard-coded into standard blockchain clients and widely accepted as valid

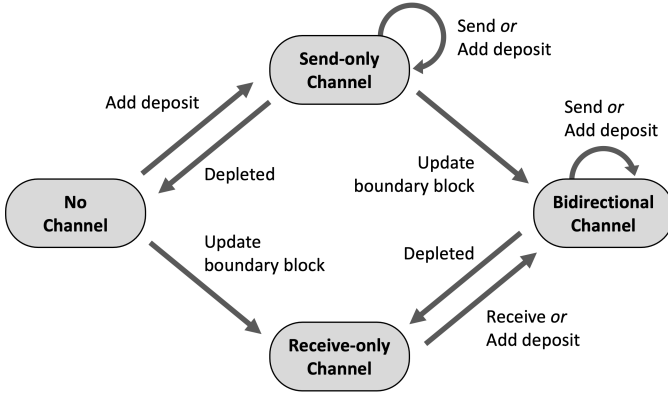


Fig. 3. RouTEE channel type transition diagram. Channels could be dynamically opened, closed, and changed to other types of channels.

blocks. Likewise, the host can choose any one of the recent blocks as a start block and hard-code it into the RouTEE’s source code. In this case, the verification process becomes incredibly simple. However, to filter out fake blocks with a too low difficulty, the start block should be more than 2,016 blocks far from the latest block in order to calculate Bitcoin’s block difficulty inside the SGX.

Lastly, to measure the average on-chain transaction fee per byte (i.e., fee_{avg}), the host inserts on-chain transactions included in some recent blocks (e.g., for 2,016 blocks, the same as the block difficulty change period). Bitcoin’s block headers contain the Merkle root of transactions, hence RouTEE can determine that these inserted transactions are actually involved in a certain block with its header. Based on this average transaction fee, RouTEE sets the amount to charge users for on-chain settlement transaction fees.

All block headers inserted since the initialization should be stored inside the SGX. However, we note that this block header chain occupies little storage because Bitcoin’s block header size is just 80 bytes. Even if there were 1,000,000 blocks in Bitcoin, for example, it would not exceed 80 MB.

B. Channel Creation

After the initialization, users are able to obtain the result of the header chain verification (e.g., the initial block to start verifying and the latest block inside SGX) and build secure sessions with RouTEE by remote attestation. If they believe the initialization was successful, they will participate in RouTEE. Due to the session, the content of messages between the user and RouTEE is not revealed to others, especially the host.

As we mentioned in Section III-A, there are three kinds of channels in RouTEE, and users need to create their channels first. In fact, channels in RouTEE are closer to the users’ state than channels that exist explicitly in other payment networks. Thus, a channel’s state changes dynamically as a user’s state (i.e., balance, boundary block) changes. Fig. 3 shows channel state transitions through several user operations. Users who have no channel at first can only execute *add_deposit* or *up-*

date_boundary_block operations to open initial unidirectional channels.

1) *Add user*: Before users create channels, they need to enroll in RouTEE first via *add_user*. The user provides its public key, *user address*, and *settle address* to RouTEE. The user’s public key is utilized to verify the user’s signature when the user executes user operations such as multi-hop payments. A user address acts as the user’s ID in RouTEE. When the user settles its balance later, on-chain assets are shifted to its settle address. The user should inform its settle address in advance since the user’s channel might be settled automatically without the user’s request (See section V-B, VII-C for more details). Then RouTEE stores this user information in a state, and this user is then ready to build unidirectional channels through *add_deposit* or *update_boundary_block*. We note that each user has a *nonce* field in a user state in RouTEE. User operations described below only accept messages with the value equal to the nonce, increasing the nonce by one and thus invalidating the messages.

2) *Add deposit*: *add_deposit* is a preregistration process to inform RouTEE of a user who is going to lock its deposit. The user sends a beneficiary address, namely its user address, to RouTEE. Then RouTEE generates a new random address, called a *manager address*, inside SGX to receive the user’s on-chain deposit and transmits a response message containing the manager address. RouTEE also saves these addresses to a *pending deposit list* to detect a user’s deposit transaction later. Lastly, the user sends on-chain coins via a *pay-to-public-key-hash* (P2PKH) transaction, one of the standard transactions in Bitcoin, from any sender’s addresses to the manager address.

The source block, which includes the deposit transaction, will be fed to RouTEE by *insert_block* operation, which deals with deposit transactions, updating users’ maximum source block numbers and their balances. But the increased amount of the balance is less than the deposit amount. Since this on-chain deposit, namely a UTXO in Bitcoin, will be used to make a settlement transaction, RouTEE collects a fare from the user’s balance in advance for the settlement transaction fee. Let D_{amount} be the amount of this deposit transaction and $B_{increase}$ be the balance increase amount. Then $B_{increase}$ is determined as:

$$B_{increase} = D_{amount} - 148 \cdot fee_{avg}$$

This is because an on-chain transaction fee is proportional to the size of the transaction and a size of one input of P2PKH transaction is 148 bytes (See section IV-D for more details).

As described in Fig. 3, users who have no channel gain send-only channels, and users with receive-only channels obtain bidirectional channels by *add_deposit*. If a deposit transaction has not appeared in a certain block time period (e.g., for 100 blocks), RouTEE concludes that the user does not broadcast the deposit transaction, and deletes information about the deposit from the pending deposit list.

3) *Update boundary block*: *update_boundary_block* increases a user’s boundary block number and is performed as follows. In order to get valid block headers, users can

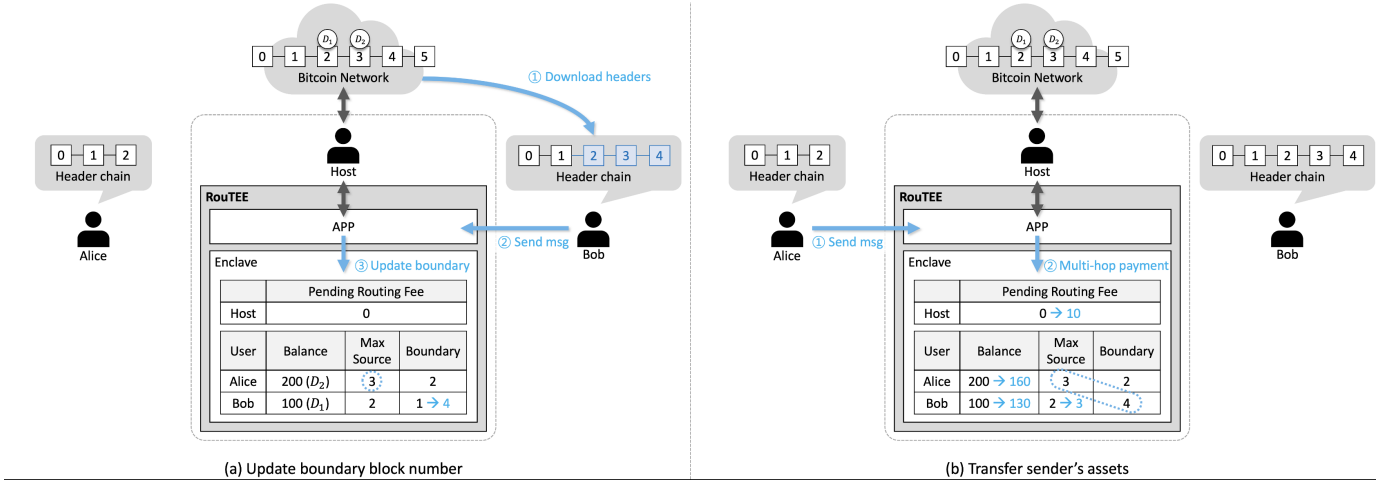


Fig. 4. An example of a multi-hop payment in RouTEE. Alice tries to send her balance derived from the deposit D_2 in block 3 to Bob, but his boundary block number is 1 which is less than 3. This means that Bob does not trust Alice's balance yet. (a) Alice informs Bob of her maximum source block number (in this case, 3). Bob downloads block headers from full nodes to verify whether block 3 is valid. He receives up to block 4 and sets his boundary block as it by *update_boundary_block*, in order to prepare to receive from Alice. (b) Alice executes a multi-hop payment to send 30 to Bob. RouTEE checks that Bob believes Alice's balance is valid (i.e., Alice's maximum source block number is less than or equal to Bob's boundary block number. In this case, check $3 \leq 4$). Then Alice's balance is moved to Bob, and the host gets a pending routing fee for this multi-hop payment from Alice. Now Bob has balances from different source blocks, so Bob's maximum source block number should be updated to 3.

download header chains from full node users by running light clients. If they trust the data source, it is sufficient to receive only one block and select it as a boundary block. Then the user sends its user address, signature, and boundary block's block number and hash value to RouTEE. The signature is required to authenticate the user. If the user's boundary block parameters match a header within RouTEE's header chain, and the new boundary block's number is greater than the user's previous one, RouTEE updates the user's boundary block number to it. As illustrated in Fig. 3, for users who have no channel with RouTEE, *update_boundary_block* permits them to receive valid balances from other users, opening receive-only channels. This operation also opens bidirectional channels for users with send-only channels.

C. Payment

There are three agents who participate in a multi-hop payment: a sender, a receiver, and RouTEE as a unique intermediary. We note that, in RouTEE, there is no direct channel between two users and every payment is always a 2-hop payment, which is the shortest and most efficient multi-hop payment. As described in Fig. 4, multi-hop payments proceed in two phases: a *ready phase* and a *transfer phase*. Suppose that the sender already executed *add_deposit* and has some balance to send. During the ready phase, the receiver should be ready to receive a balance from the sender. If the receiver's boundary block number is less than the sender's maximum source block number, the receiver cannot fully trust the sender's balance, because that means the sender might have invalid balances from source blocks that the receiver does not consider valid yet. In order to allow the sender to execute a multi-hop payment, the receiver should update its boundary block to the block that is newer than the sender's maximum

source block by executing the *update_boundary_block* operation. But if the receiver already has a proper boundary block number, this ready phase can be omitted.

After both users are ready, they proceed to the transfer phase. The sender transmits a message to RouTEE, including the sender's address, its signature, the receiver's address, a payment amount, and a routing fee amount. The sender's address and signature are used to authenticate the sender. Then inside SGX, RouTEE checks conditions: whether the sender has enough balance to afford this payment, and whether the sender's maximum source block number is equal to or greater than the receiver's boundary block number. Once the payment request is deemed to be acceptable, the receiver gets paid and RouTEE obtains a pending routing fee from the sender. However, RouTEE cannot settle this pending fee now. It will be confirmed after the settlement is successfully completed (See section IV-E for more details). Then the receiver's maximum source block number should be updated because the receiver may have the balance from a newer source block after this payment.

There are some notable observations in the payment process. A channel between users is not necessary, so they can save a lot of channel creation costs, which entail expensive on-chain transaction fees. Payments could succeed even with receivers being off-line, and only receivers may need to read block headers to update boundary blocks. Furthermore, when receivers should do so, they do not have to synchronize with the blockchain until it reaches the latest block. After updating boundary blocks, users have no need to care about events in the blockchain (i.e., not monitoring new transactions). Users can set their own k value higher than RouTEE's for the k -confirmation assumption, meaning that they can be as conservative as they want when judging a block's finality.

Also, senders do not have to consider the network state (e.g., the topology, balances in channels, and payment paths).

D. Settlement

Users can request to settle balances in their channels for any amount. As a settlement requires an on-chain transaction, users need to pay settlement fees for making an on-chain settlement transaction. Settlement protocol details are as follows. A user sends the balance amount to settle, the settlement fee to be paid, its address and its signature, to RouTEE. To prevent free rides on the settlement, a minimum amount of settlement fee has been put in place. Users need to pay more than $34 \cdot fee_{avg}$, because the size of one output of a P2PKH transaction is 34 bytes. Then RouTEE verifies that the user has enough balance for this settlement request, and inserts it into the settlement request queue in which requests are sorted by the amount of the settlement fee.

Now RouTEE tries to make a settlement transaction, which settles the largest number of users with sufficient settlement fees, using a greedy method. Let Tx_{inputs} be the number of transaction inputs, and $Tx_{outputs}$ be the number of transaction outputs. Then, the size of a P2PKH transaction (i.e., Tx_{size}) can be calculated as:

$$Tx_{size} = 148 \cdot Tx_{inputs} + 34 \cdot Tx_{outputs} + 10$$

10 bytes is for other extra fields of a transaction, excepting inputs and outputs. Then an on-chain transaction fee (i.e., Tx_{fee}) can be determined as:

$$Tx_{fee} = Tx_{size} \cdot fee_{avg}$$

The important thing is that a settlement transaction uses all deposits (i.e., UTXOs) that RouTEE owns, which is a *spend-all-settlement* method. Thus a settlement transaction's Tx_{inputs} is the number of deposits, and $Tx_{outputs}$ is the same as the number of users who requested a settlement plus one to make a *leftover deposit* with leftover balances after the settlement. Users who participate in the settlement have to split the fee for this leftover deposit. As mentioned in section IV-B, deposit senders have already paid for this settlement. Briefly, if the amount of collected fees from users (i.e., from deposit senders and settle requesters) is greater than Tx_{fee} , RouTEE creates a settlement transaction and broadcasts it. Otherwise, it just waits for other settle requests with sufficient fees. However, if fee_{avg} soars, settle request users should pay more fees because they need to pay for the settlement transaction's inputs also (as pre-collected fares become not enough).

After making a secure settlement transaction, some portion of the pending routing fee will be confirmed for the host. Let $RF_{pending}$ be the amount of total pending routing fees, S_{amount} be the amount of balances to be settled by this settlement transaction, and B_{total} be the amount of total user balances in RouTEE. Then the amount of pending routing fees to be confirmed (i.e., $RF_{confirmed}$) is determined as:

$$RF_{confirmed} = RF_{pending} \cdot \frac{S_{amount}}{B_{total}}$$

This means that the host can gain the whole routing fees only when the host settles all users' balances completely, that being when S_{amount} is equal to B_{total} . The settlement transaction and $RF_{confirmed}$ are stored and will be used in the *insert_block* operation to finally confirm routing fees and allow the host to withdraw that confirmed routing fee.

E. Block Insertion

A host needs to insert block headers and transactions into RouTEE to detect deposit transactions and settlement transactions. *insert_block* operates as follows. The host inserts a block header and its transactions into RouTEE. This operation can only be performed by the host as it requires the host's signature also. In order to verify block data, RouTEE verifies that the new block header satisfies the PoW conditions and that the Merkle root can be reproduced with these transactions. It then adds this header to the header chain, looks at transactions to update fee_{avg} , and searches for deposit transactions and settlement transactions. If a deposit transaction is found, it gets the corresponding user address with this transaction from the pending deposit list, increases the user's balance by $B_{increase}$, and updates the user's maximum source block number to this block number. If there is a settlement transaction, it similarly brings the matching $RF_{confirmed}$ value and increases the host's balance (i.e., confirmed routing fees) by that value.

V. SECURITY ANALYSIS

In this section, we show various kinds of attack strategies that hosts could choose and explain how the RouTEE protocol protects user balances from rational hosts. We emphasize that there is no case in which hosts gain unfair benefits while users lose their balances.

A. Fake Blockchain Data

All balances in RouTEE are derived from on-chain deposits locked in the blockchain. However, as described in Fig. 5, malicious hosts could generate fake blocks, which involve fake deposit transactions not actually broadcasted in the blockchain network intended to deceive RouTEE. Although this attack costs a lot of mining time to create blocks with adequate difficulty, it is possible because there is no time limit and adversaries can earn invalid balances in RouTEE without staking their on-chain assets.

After obtaining illegal balances, adversaries need to execute multi-hop payments or settlements to gain actual profits. For this reason, users, particularly payees, have to verify whether or not payers' balances are derived from valid source blocks. They can efficiently filter invalid balances out by setting boundary blocks since fake blocks have different hash values than users' blocks. However, there might be an *eclipse attack* [30] where adversaries who take all the network connections of a victim supply fake blockchain information to make them believe it is the correct main chain. To avoid eclipse attacks, users should fetch several header chains from various full nodes, compare them, and choose the reliable one according to the longest chain rule.

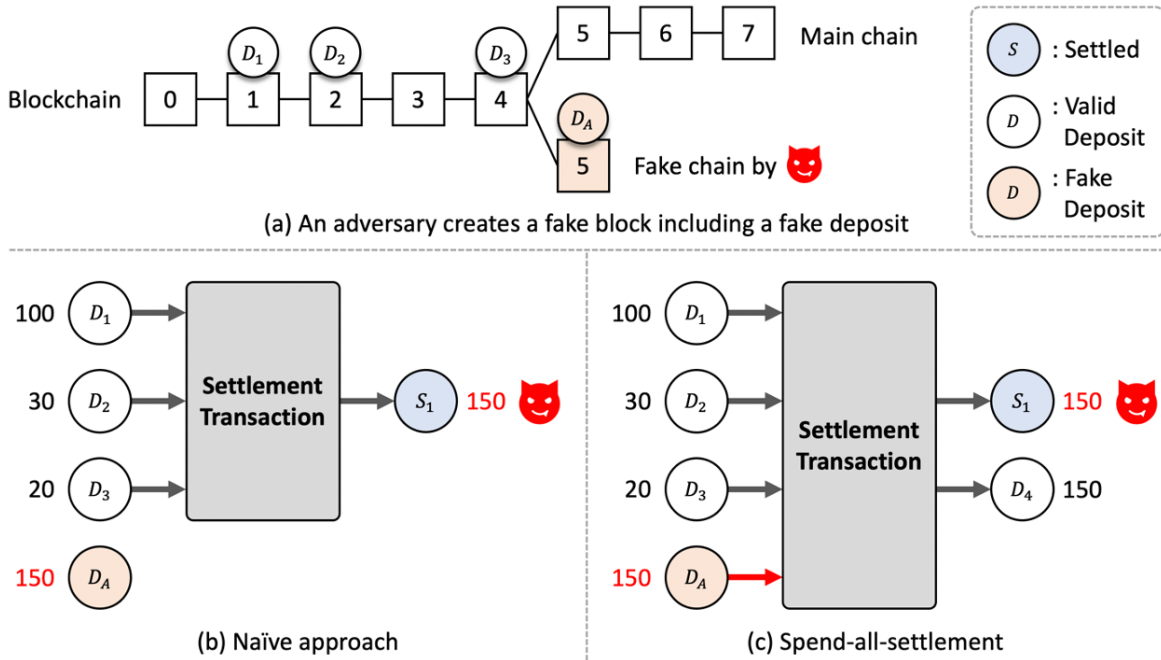


Fig. 5. An attack scenario using fake blockchain data. (a) White blocks are included in a valid main chain. A red block is generated by the malicious host. The host inserts the red block into RouTEE to add the fake deposit D_A and tries to settle its invalid balance. (b) If RouTEE picks old deposits to make a settlement transaction (naive approach), the deposit D_A is not spent, which means the transaction is valid in both chains. Thus, the host can broadcast the transaction to the Bitcoin network, stealing other valid deposits. (c) On the contrary, if the settlement transaction contains all deposits (spend-all-settlement), it is only valid in the fake chain as there is no D_A in the main chain. Thus the host is not able to gain illegal benefits.

Still, the host can try to settle its invalid balances to steal valid on-chain deposits from other users. Our key idea for the spend-all-settlement is to invalidate on-chain transactions that are made based on fake blockchain data. As illustrated in Fig. 5, we thought of a naive approach for settlements that spend not all deposits, but rather the least set of deposits starting from the oldest one to just meet a total settlement amount. Settlement transactions made in this way are valid transactions in the fake chain, of course, and even in the main chain because they only use valid deposits. Thus adversaries can broadcast them to the blockchain network and take on-chain assets, with users then unable to reclaim their assets.

If we apply the *spend-all-settlement* method, on the other hand, settlement transactions include every deposit. So if there is at least one fake deposit in the settlement transactions, they would be rejected from the main chain, as there is no such UTXO to spend. To conclude, employing boundary block verification and the *spend-all-settlement* prevents illegal attempts to gain invalid benefits through payments and settlements, respectively. Therefore, the best strategy for rational hosts is simply feeding correct blockchain data to confirm user balances and earn routing fees.

B. Abortion

Hosts can abort operating RouTEE in various ways. First, the host can simply shut down the machine running RouTEE. If SGX is turned off, all private data in SGX such as user balance states and the block header chain disappear because the secure memory space in SGX is volatile. However, a

rational host would not forcibly shut down RouTEE, as the host does not want to abandon the pending routing fees in it. To terminate RouTEE normally, the host should go through a termination process that settles all users' balances to the settle addresses obtained in advance by broadcasting settlement transactions. The host could then take all confirmed routing fees.

Next, the host could stop feeding blockchain data into RouTEE. If there are deposit transactions not yet inserted into SGX, then they are bound in the blockchain forever, and there is no way to retrieve it. Similarly, the host could generate a fake blockchain that does not contain the user's deposit, but the host has no financial motivation to ignore deposits, losing all routing fees and maybe spending an expensive mining cost.

We might fundamentally prevent these attacks by utilizing *time-locked transactions*, which make deposits retrievable after a certain amount of time. However, their transaction fees are more expensive than P2PKH transactions, and RouTEE must broadcast additional transactions that bring time-locked assets to RouTEE within the time limit (users may need to pay for this transaction also). To conclude, P2PKH transactions are appropriate for RouTEE because we are only concerned about rational adversaries and time-locked transactions are inefficient and unnecessarily expensive.

Lastly, the host might not broadcast settlement transactions. Pending routing fees induce the host to fulfill its duty since abandoning settlement transactions means that the host gives up confirming its pending routing fees. What's more, settle-

TABLE I
ROUTEE USER OPERATION PERFORMANCE

Operation	Throughput (op/sec)	Latency (ms)	
		Idle [99 th %]	Busy [99 th %]
<i>add user</i>	1,222	42.7 [45.9]	47.1 [50.6]
<i>add deposit</i>	1,092	44.8 [59.3]	48.1 [55.1]
<i>update boundary block</i>	19,998	38.8 [41.7]	37.4 [40.1]
<i>multi-hop payment</i>	18,677	38.6 [48.4]	43.2 [56.0]
<i>settlement</i>	19,607	38.9 [43.5]	36.5 [39.6]

ment transactions are sequentially included in the blockchain, as they spend the leftover deposit from the previous one. This means that if the host drops a settlement transaction, the host cannot broadcast any settlement transactions and cannot obtain any confirmed routing fees from that moment. Thus hosts are motivated to broadcast settlement transactions honestly.

C. Message Abusing

As messages between users and RouTEE are encrypted, hosts cannot find out their contents or manipulate them. They are not able to link a user address and the user’s on-chain deposit, nor do they know the details of multi-hop payments and settlements.

Hosts may try to drop or delay messages, but it does not give them any financial benefit since operations in RouTEE bring and confirm routing fees. Even if they are going to censor a particular user’s messages or requests for a specific operation in the face of a potential financial loss, secure sessions between them would make it impossible.

Since the minimum amount of the routing fee is determined by the hosts, they can execute a *message reordering attack*, in which they extort a payer’s balance by delivering the user’s multi-hop payment message to RouTEE after changing the minimum routing fee amount. Users can easily prevent this attack by specifying their routing fee amounts in their messages. In case of a *replay attack*, where adversaries insert the same message several times into RouTEE (e.g., execute the same multi-hop payments to get routing fees), a user’s nonce field will efficiently block it.

VI. EVALUATION

We implement a RouTEE prototype for Bitcoin using SGX. We use the Intel SGX SDK for Linux [31] and Bitcoin core libraries [32]. To generate Bitcoin addresses (i.e., manager addresses), we utilize secp256k1 which is available from the Bitcoin core. From SGX SDK, we choose AES128-GCM to encrypt/decrypt messages between users and RouTEE and RSA digital signatures created by 3072-bit keys. Bitcoin employs ECDSA signatures, but, being unsuitable for RouTEE, it takes a lot of time to verify signatures.

A. Performance of User Operations

For experiments, we implement user clients that send encrypted and signed messages to RouTEE using Python. We run a RouTEE program on a machine with an Intel i9-9900K CPU and 64GB of RAM running Ubuntu 16.04 LTS. It has 8-core/16-thread but we only use four threads.

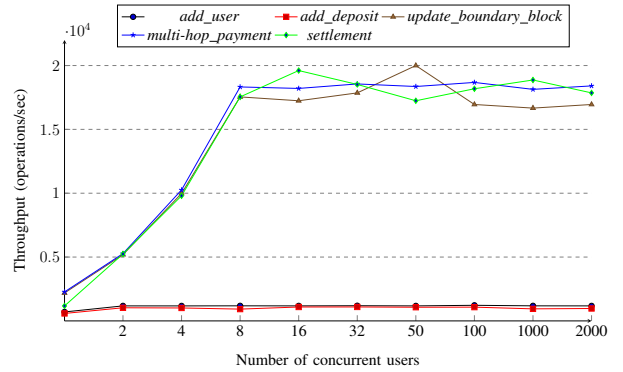


Fig. 6. User operation throughput

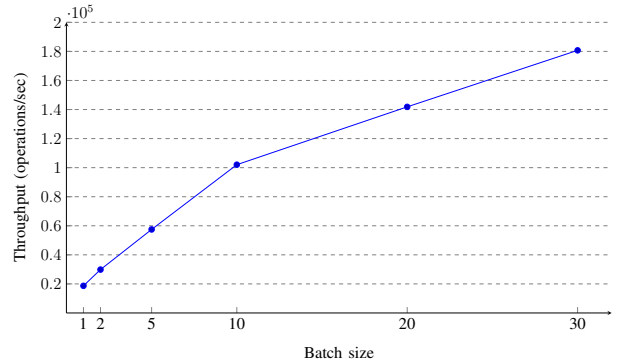


Fig. 7. Batched payments throughput

Throughput. We define the throughput as the number of operations RouTEE can process in one second. In order to measure RouTEE’s throughput for each user operation, we make user clients continuously send the same type of user operation messages with random parameters (e.g., user addresses, payment amounts). Supplying a sufficient amount of messages to find a maximum throughput value, we execute multiple client programs on a machine with an AMD Ryzen Threadripper 2990WX CPU, 128GB of RAM, and 32-core/64-thread running Ubuntu 18.04 LTS, connected with the RouTEE server over the local network. We also repeat this experiment with various numbers of concurrent users. As suggested in Fig. 6, RouTEE deals with thousands of concurrent users, maintaining its maximum transaction throughput.

As it executes dynamic allocation to register users in the user state, the *add_user* operation shows a maximum throughput of about 1,200 op/sec, which is relatively low compared to other operations. Similarly, the *add_deposit* operation has a maximum throughput of about 1,100 op/sec, also quite low relatively speaking, as it generates random private keys to make random manager addresses. However, these operations are more scalable than other payment networks when we consider that one Bitcoin block cannot include more than 3,000 transactions, meaning that they cannot open more than 3,000 channels every 10 minutes.

The *update_boundary_block* and settlement operations show similar throughput of about 18,000 op/sec. Given the

TABLE II
LIGHT CLIENT EXECUTION TIME FOR DOWNLOADING AND VERIFYING
2,016 VALID BLOCK HEADERS

	Time (sec)				
	Min	Med	Avg	99 th %	Max
Network latency	0.161	0.315	0.499	3.744	16.679
Verification time	0.054	0.088	0.089	0.126	0.297
Total	0.216	0.404	0.588	3.871	16.977

nature of payment networks, *update_boundary_block* would occur more frequently than settlements. Even if every 100,000 users update their boundary blocks when a new block is created, it will only take 6 seconds. We note that there are about 15,000 public users in Lightning Network, opening about 36,000 channels as of November 2020 [33].

The multi-hop payment’s throughput is about 18,000 op/sec. This is similar to existing payment systems such as VISA, which is known to be capable of dealing with about 24,000 payments [34]. Furthermore, we can boost the throughput by batching multi-hop payments. Senders can make batched multi-hop payments that contain multiple receivers and payment amounts. We measure the throughput of batched payments and the result is shown in Fig. 7. The throughput increases as the size of the batch increases, and users can choose various batch sizes. For example, RouTEE achieves an approximately 30,000 op/sec throughput when the size is two, and about 180,000 op/sec when the size is 30. The graph is not completely linear because as the size increases, the message size also grows, costing more network overhead. We note that the throughput can be enhanced by employing additional hub nodes or multi threads. The results of the throughputs are listed in Table I.

Latency. We define the latency as the time between sending a message to RouTEE and receiving a response message from RouTEE. To measure the RouTEE’s latency for each user operation in a realistic network environment, we run the client program on a t2.micro Amazon EC2 instance (RTT: 39.9 ms, Bandwidth: 31.4 Mbits). The client sends a message to RouTEE, waits for the response, and repeats this continuously. Table I shows the result. Idle means that there is no concurrent user in RouTEE, and Busy means that RouTEE is executing operations at its maximum throughput speed. Most operations are returned within about 50 ms, which is a very small latency, and there is almost no difference in the latency between idle and busy states.

B. Performance of Host Operations

To provide blockchain data, we run the Bitcoin core program as a full node on the same machine as RouTEE. Since we do not need to evaluate Bitcoin network interactions, we simply establish the Bitcoin local private network, called *regnet*, and create 100 blocks which contain 2,500 randomly generated transactions each (for almost any period, the average number of transactions per block does not exceed 2,500 [35]). The RouTEE program can obtain blockchain data such as

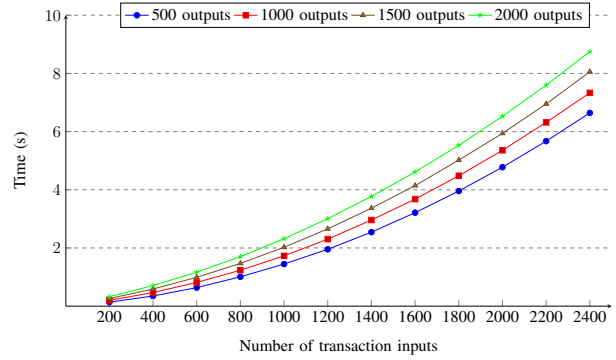


Fig. 8. Settlement transaction generation time

headers and transactions through RPC requests to the Bitcoin core.

Block Verification. We first insert 100 block headers to measure the block header verification time (i.e., a light client’s verification inside SGX). Then we insert 100 headers with their transactions to measure the *insert_block* operation execution time. The result is that it takes 18.56 ms to verify a single header and 45.16 ms for an *insert_block* operation.

Initialization. To initialize RouTEE, a host needs to feed a header chain to RouTEE. Based on the result above, it takes about 3.37 hours to start verifying headers from the genesis block to the latest block (about 655,000 blocks). If we start from the 295,000th block, which is the latest checkpoint block in the Bitcoin core [36], it takes 1.85 hours. Furthermore, it only takes 37.41 seconds to verify 2,016 blocks by hard-coding the block that is 2,016 blocks away from the latest block. It takes an additional one minute to verify 2,016 full blocks to finish initialization, calculating fee_{avg} . We note that the initialization needs to be performed only once.

Make Settlement Transaction. To measure the settlement transaction generation time, we repeatedly create settlement transactions with various numbers of inputs and outputs within RouTEE’s SGX. Fig. 8 shows the result. The execution time increases as the number of transaction inputs increases, and it takes a little more time to deal with more outputs. This is because most of the time is spent making signatures for inputs and hashing data several times, and the hashing process time grows quadratically as the inputs increase.

The number of inputs equals the number of deposits, and the number of outputs equals the number of settle request users (plus one for a leftover deposit). If RouTEE owns 2,000 deposits and 2,000 users request settlement, it only takes 6.52 seconds to generate the settlement transaction, with the transaction’s size being roughly 364 KB. As Bitcoin’s block size limit is about 1 MB, the transaction is sufficiently small to be included in the block, meaning that RouTEE can settle thousands of users simultaneously within a few seconds.

C. Performance of Light Client

We also measure how long it takes to download header chains through a light client. We choose Electrum [37] which

is one of the most popular light clients for Bitcoin. Electrum batches 2,016 block headers and sends header requests to various full nodes. We run Electrum on a t2.micro Amazon EC2 instance and note that it has very limited network resources. Electrum downloads a header chain from the genesis block to the latest block, and we repeat this process 100 times.

Table II shows the result. When Electrum requests to the close full node, it only takes 0.161 seconds to download 2,016 headers. Verifying these headers can be finished within 0.054 seconds, meaning that users need only 1.16 minutes to get the whole verified header chain in the best case. Most of the execution time is spent waiting for the download to be finished, and it varies depending on which full node is connected. When Electrum connects to the node far away from it, it takes 16.977 seconds in total, indicating that it requires at most 1.53 hours to download all headers. However, we notice that after downloading it once, users can easily catch up to the latest block within about 17 seconds every two weeks (i.e., 2,016 blocks). We also stress that users only need to download headers, meaning that they only require about 55 MB to store all headers.

D. Comparison of Multi-hop Payment Performances

We compare RouTEE to Lightning Network, which is the state-of-the-art payment network and has an open-sourced implementation called Lightning Network Daemon (LND) [38]. However, it is difficult to compare them objectively since they have quite different network topologies and protocols. In Lightning Network, the multi-hop payment results would vary greatly depending on the network state, routing users, and payment amounts. Unfortunately, there is no public data on network topologies or payment history in payment networks, so it is hard to measure the realistic performance of Lightning Network. By contrast, RouTEE's multi-hop payments never fail if participants are rational. Thus, if we can measure their performance in realistic environments, RouTEE will outperform Lightning Network.

For this reason, we measure the performance of the 1-hop payment in Lightning Network because it is a lot similar to the multi-hop payment in RouTEE without updating the receiver's boundary block (i.e., the sender simply sends the multi-hop payment message). In Lightning Network, receivers initiate payments, forwarding their invoices to senders. Then senders decode invoices and send response messages to receivers, which is the end of 1-hop payments. To make the equivalent environment, the sender client runs on the machine that executed the RouTEE program, and the receiver client runs on the machine that executed the client program, measuring the maximum invoice decoding throughput in a similar manner. As a result, it takes 28.033 seconds to process 10,000 invoices, which means that its throughput is about 360 payments/sec with a latency of 165 ms. Considering RouTEE's payment throughput (i.e., 18,677 payments/sec with a latency of about 40 ms from Table I), Lightning Network nodes are not suitable for acting as a hub node.

VII. DISCUSSION

In this section, we consider security issues in more detail and describe possible extensions to protect RouTEE even from various kinds of irrational adversaries to enhance its security.

A. Obtaining Valid Blockchain Data

It is hard to determine whether this blockchain data is absolutely valid or not. We simply find a chain that looks relatively more valid than the other ones (e.g., by length of chain and block difficulty). Judging the chain's validity in the SGX is even more difficult as the host may restrict input data into SGX. Also, users might receive invalid chain data, if they are victims of eclipse attacks or carelessly download only from a single malicious node. In short, there might be the case that a malicious host profits from fake data.

To protect these kinds of naive users, we can leverage statistical characteristics of block mining [39]. In Bitcoin, a block is mined every 10 minutes on average, and the block interval time between two blocks is a random variable that has an exponential distribution. Adversaries with less mining power than the main chain's miners will have difficulty keeping up with this mining speed. Thus it is possible to detect weird blocks inserted too slowly, by measuring block interval times with a trusted relative timer inside SGX. However, this may yield false-positive results due to innocent but unlucky blocks, which result in deferring block acceptance for a certain amount of time (e.g., for 120 blocks).

There is another approach [40] which brings authenticated data into SGX from trustworthy web sites. Nowadays, many web sites [41], [42] provide blockchain-related data in real-time. RouTEE can therefore obtain block headers and transactions without hosts. In order to reduce the risk of website hacking, RouTEE should interact with various websites and decide which data is correct by a majority vote. This could be combined with the above method to mitigate false-positive results.

Breaking this system is quite impractical since the adversary should have mining power equivalent to all the Bitcoin miners and be able to manipulate several websites at once. Thus, users who trust data obtained in this manner no longer need to run light clients and can set their boundary block as the latest block inside SGX. This makes RouTEE more convenient for users.

B. Crash Fault Tolerance

There might be unintentional software or hardware faults so we need to save internal data in SGX periodically to restart RouTEE with the same state. SGX supports a sealing/unsealing feature [16] that allows data to be securely stored and loaded within SGX only. Before utilizing this feature, we must prevent *roll-back attacks*, where an adversary gives previous data and loads it as if this were the latest one, thereby breaking the integrity of SGX. To do so, SGX enables implementing hardware monotonic counters with the non-volatile memory in SGX [31] in order to track and trace the current state.

However, in the case of RouTEE, the state could be changed very frequently. Thus, the monotonic counter in SGX might not be suitable for RouTEE, because it has a limited performance and would wear out quickly after about 1 million writes [43]. To overcome this limitation, RouTEE could batch operations to reduce the number of writes to the counter. In addition, there is other research [43], [44] that could enhance the data integrity for SGX.

C. Network Failure

Considering unexpected long-term network failures between users and RouTEE, we could design RouTEE to be able to settle users' channels, when their balances have not changed for a certain period of time (e.g., for 1,000 blocks) in order to guarantee their assets. In a different approach, we could leverage the underlying blockchain to deliver messages into RouTEE against the network failure and even indiscriminate censorship, similar to another framework using a TEE [45]. In Bitcoin, users are able to write arbitrary data to the blockchain through the OP_RETURN instructions of the script language. This means that RouTEE could receive encrypted messages through on-chain transactions. It is possible to block even these messages by generating a fake chain that does not contain them, but it causes far more losses (i.e., block mining costs, abandoning routing fees) than benefits.

VIII. RELATED WORK

Payment networks. There has been a lot of research done on payment networks. Duplex Micropayment Channels [46] utilize time-locks to ensure that the latest transaction can be broadcast first. Lightning Network [4] and Raiden [5] are payment networks based on Bitcoin and Ethereum respectively. Flare [47] proposes an optimized routing path searching algorithm for Lightning Network. Revive [48] rebalances payment channels to re-fund depleted channels without broadcasting on-chain transactions. Perun [49] offers a new method called *virtual payment channel*, which makes the routing more efficient but requires Turing-complete smart contracts. Intermediate users can be virtual payment hubs, but they would need a lot of channels that have sufficient balances to create virtual channels, and their balances would be locked until virtual channels are closed. Sprites [50] reduces collateral locking time during multi-hop payments and can partially add and withdraw balances. Pisa [51] allows users to go off-line for a longer period of time by employing a third party called a *custodian* who monitors blockchains on their behalf. Fulgor and Rayo [52] are new protocols that reinforce privacy and concurrency for payment networks. Anonymous multi-hop locks (AMHLs) [13] enable privacy-preserving multi-hop payments.

Trusted execution environments with blockchains. Recent studies have solved various problems in blockchains through TEEs. TownCrier [40] feeds authenticated data to smart contracts, overcoming the oracle problem. Obscuro [53] is a secure mixer platform for Bitcoin to enhance the anonymity of transactions. FastKitten [45] allows smart

contracts to be executed over blockchains without complex languages. Also, users can efficiently interact with smart contracts by off-chain transactions. Tesseract [39] is a secure cryptocurrency exchange that enables assets from different blockchains to be swapped. Ekiden [54] makes smart contracts more confidential and efficient. Bite [55] protects the privacy of light clients from other full nodes. Teechan [56] and Teechain [57] establish secure payment channels. To the best of our knowledge, Teechain is the most relevant to our work. However, it requires every user to have a TEE. Users also have to check that their counterparty's deposit transaction is included in the blockchain when they want to allocate their deposits to their channels. In addition, Teechain supports multi-hop payments but their mechanism is fundamentally the same as the existing payment network's method, implying that it has the same limitations such as a high probability of failure and struggling to track the network state to find payment paths.

IX. CONCLUSION

Payment networks allow more payment transactions to be handled on behalf of blockchains with low scalability, but they are still inefficient and have many limitations. Users have a hard time managing their channels and multi-hop payments are difficult to achieve. We present RouTEE, a new payment network with a secure routing hub that fully leverages a star network topology and a TEE. Offering payment channels with high liquidity, it supports multi-hop payments that do not fail in normal situations. We consider adversaries that have control of the RouTEE platform and analyze the security of our system, proving that following the protocol honestly is the best strategy for rational hosts. We also evaluate RouTEE and demonstrate that RouTEE provides a practical and scalable payment network.

REFERENCES

- [1] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: <https://www.bitcoin.org/bitcoin.pdf>
- [2] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [3] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.
- [4] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.
- [5] Raiden network. [Online]. Available: <https://raiden.network/>
- [6] Z. Avarikioti, L. Heimbach, Y. Wang, and R. Wattenhofer, "Ride the lightning: The game theory of payment channels," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 264–283.
- [7] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *Hasp@ isca*, vol. 10, no. 1, 2013.
- [8] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.
- [9] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Annual International Cryptology Conference*. Springer, 1992, pp. 139–147.
- [10] I. A. Seres, L. Gulyás, D. A. Nagy, and P. Burcsi, "Topological analysis of bitcoin's lightning network," in *Mathematical Research for Blockchain Economy*. Springer, 2020, pp. 1–12.

- [11] J. Herrera-Joancomartí, G. Navarro-Arribas, A. Ranchal-Pedrosa, C. Pérez-Solà, and J. Garcia-Alfaro, "On the difficulty of hiding the balance of lightning network channels," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 602–612.
- [12] G. Kappos, H. Yousaf, A. Piotrowska, S. Kanjalkar, S. Delgado-Segura, A. Miller, and S. Meiklejohn, "An empirical analysis of privacy in the lightning network," *arXiv preprint arXiv:2003.12470*, 2020.
- [13] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *NDSS*, 2019.
- [14] C. Pérez-Solà, A. Ranchal-Pedrosa, J. Herrera-Joancomartí, G. Navarro-Arribas, and J. Garcia-Alfaro, "Lockdown: Balance availability attack against lightning network channels," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 245–263.
- [15] E. Rohrer, J. Malliaris, and F. Tschorsch, "Discharged payment channels: Quantifying the lightning network's resilience to topology-based attacks," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019, pp. 347–356.
- [16] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. Citeseer, 2013, p. 7.
- [17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostianen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: {SGX} cache attacks are practical," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.
- [18] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 523–539.
- [19] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foresadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.
- [20] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against intel {SGX}," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1213–1227.
- [21] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.
- [22] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs," in *NDSS*, 2017.
- [23] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," in *NDSS*, 2017.
- [24] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjà vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 7–18.
- [25] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting {SGX} enclaves from practical side-channel attacks," in *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 227–240.
- [26] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A commodity obfuscation engine on intel sgx," in *NDSS*, 2019.
- [27] T. Alves, "Trustzone: Integrated hardware and software security," *White paper*, 2004.
- [28] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, "Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 211–227.
- [29] R. Shostak, M. Pease, and L. Lamport, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.
- [30] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 129–144.
- [31] Intel Software Guard Extensions SDK Developer Reference for Linux OS. [Online]. Available: https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel_SGX_Developer_Reference_Linux_2.1.3_Open_Source.pdf
- [32] The Bitcoin Community. Bitcoin Core version 0.13.1 released. [Online]. Available: <https://bitcoin.org/bin/bitcoin-core-0.13.1/>
- [33] IML. IML: Lightning Network Search and Analysis Engine. [Online]. Available: <https://1ml.com/>
- [34] VISA. Visa's transactions per second. [Online]. Available: <https://usa.visa.com/run-your-business/small-business-tools/retail.html>
- [35] Blockchain.com. Average transactions per block. [Online]. Available: <https://www.blockchain.com/charts/n-transactions-per-block>
- [36] Bitcoin Core. The latest checkpoint block. [Online]. Available: <https://github.com/bitcoin/blob/master/src/chainparams.cpp#L160>
- [37] Electrum. Electrum: Bitcoin Wallet. [Online]. Available: <https://electrum.org/#home>
- [38] Lightning Network Community. Lightning Network Daemon. [Online]. Available: <https://github.com/lightningnetwork/lnd>
- [39] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-time cryptocurrency exchange using trusted hardware," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1521–1538.
- [40] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 270–282.
- [41] Blockchain.com: Blockchain Explorer. [Online]. Available: <https://www.blockchain.com/explorer>
- [42] BTC.com: Bitcoin Explorer. [Online]. Available: <https://btc.com/>
- [43] S. Matetic, M. Ahmed, K. Kostianen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "{ROTE}: Rollback protection for trusted execution," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1289–1306.
- [44] R. Strackx and F. Piessens, "Ariadne: A minimal approach to state continuity," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 875–892.
- [45] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "Fastkitten: Practical smart contracts on bitcoin," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 801–818.
- [46] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*. Springer, 2015, pp. 3–18.
- [47] P. Prihodko, S. Zhigulin, M. Sahnó, A. Ostrovskiy, and O. Osuntokun, "Flare: An approach to routing in lightning network," *White Paper*, 2016.
- [48] R. Khalil and A. Gervais, "Revive: Rebalancing off-blockchain payment networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 439–453.
- [49] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 106–123.
- [50] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 508–526.
- [51] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller, "Pisa: Arbitration outsourcing for state channels," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 16–30.
- [52] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and privacy with payment-channel networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 455–471.
- [53] M. Tran, L. Luu, M. S. Kang, I. Bentov, and P. Saxena, "Obscuro: A bitcoin mixer using trusted execution environments," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 692–701.
- [54] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.
- [55] S. Matetic, K. Wüst, M. Schneider, K. Kostianen, G. Karame, and S. Capkun, "{BITE}: Bitcoin lightweight client privacy using trusted

execution,” in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 783–800.

[56] J. Lind, I. Eyal, P. Pietzuch, and E. G. Sirer, “Teechan: Payment channels using trusted execution environments,” *arXiv preprint arXiv:1612.07766*, 2016.

[57] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch, “Teechain: a secure payment network with asynchronous blockchain access,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 63–79.