BANKLAVES: Concept for a Trustworthy Decentralized Payment Service for Bitcoin

Matthias Grundmann, Marc Leinweber, Hannes Hartenstein
firstname.lastname@kit.edu
Institute of Telematics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

***Abstract*—We explore challenges of and present a concept for a decentralized payment service which is based on trusted execution environments. The system guarantees that users can always cash out their funds without depending on the cooperation of other network members, hence minimizing the trust required in other network members. We present an overview of the system, motivate key components for a secure architecture and provide a communication protocol. We prove that the payment service users can cash out their funds at any time without any dependence on other network members.**

I. INTRODUCTION

In the last years, several second layer technologies have emerged that are built on top of cryptocurrencies such as Bitcoin [1]. Whether those are sidechains, smart contracts, or payment channel networks – a common use case is a payment service that allows a subgroup of the cryptocurrency's users to manage their funds. Money can be transferred into the second layer, moved between participating users and cashed back out to the cryptocurrency. An exchange or an online wallet provider can also be seen as such a service on the second layer.

All those approaches come with their specific shortcomings. A central provider needs to be trusted, in a sidechain the majority of the chain's miners need to be trusted, smart contracts make all transactions public, and payment channel networks require a valid route to exist between the sender and the receiver of a transaction. In this work we explore how trusted execution environments (TEEs) can be used to build a payment service that does not require trust into other users, hides the users' transaction history, and enables users to transfer their funds freely inside the network. TEEs empower users to validate via remote attestation that the correct code is executed and their data is secured even from a malicious operating system. To keep the scope focused, we choose Bitcoin as the underlying cryptocurrency of the payment service.

We explore challenges of and present an example for a **decentralized trustworthy payment service** for Bitcoin based on a *network of TEEs* called BANKLAVES. The payment service offers the following functions and can thus be compared to a decentralized "bank" for bitcoins: (1) account creation and closing, (2) deposit of bitcoins, (3) internal transactions to other accounts and (4) external transactions, including withdrawals, to Bitcoin addresses. In contrast to a centralized solution, we require for our concept that each user can **cash out their balance at any time** without dependence on a third party.

To perform any of the operations above, a user has to participate in the decentralized network and execute the payment service client locally in a TEE. Since it is unrealistic that all users are willing or able to run the client continuously, it is required that the functions above work **without all users being online** and that the **current local state can be backed up**. State extraction, however, makes it possible to *replay old states*. Now, the challenge is to allow a local client being in a particular state to perform a cash out, but to prevent this cash out if the client was reverted to that state.

In sum, the developed protocol (1) enables every user to cash out their funds at any time, (2) is secure against attackers replaying old states and (3) offers its functionality without all users being online.

The remainder of this work is structured as follows. In Section II we give an overview of TEEs and TEE-based approaches for Bitcoin services. In Section III a concept overview and a simplistic approach are presented from which we derive the challenges related to the security objectives. Those challenges are addressed in Section IV where we present the key components of our concept. In Section V the BANKLAVES protocol based on the key components is explained. We provide a proof to show that the proposed protocol allows a user to cash out at any time (Section VI). We discuss potentials for improvement and future work in Section VII before drawing a conclusion.

II. RELATED WORK

In this section, we outline trusted execution environments and our requirements on concrete TEE implementations. Subsequently, we describe related work from the field of Bitcoin combined with TEEs. In the past, research has shown that specific TEE implementations, due to implementation flaws, are vulnerable to attacks [2]. In this work, we assume "bug-free" implementations and explore how such an "ideal" TEE can be used to build a payment service.

A. Trusted Execution Environments (TEEs)

A trusted execution environment is established by trustworthy features in hard- and software that assist, protect or monitor user and system processes and that are, to a certain degree, demonstrably compliant to their stated functionality. Prominent examples for TEE implementations are Intel's

Security Guard Extensions (SGX) [3, 4] or AMD's *Memory Encryption Technology* [5]. Those technologies differ in the features they provide and in the use cases and attacker models they are designed for. Due to its feature set, current research and proposed architectures are typically based on SGX or at least concepts borrowed from SGX. SGX is designed to separate user space applications from the entire system. All components of the system except the CPU are considered as potentially malicious. Hence, the root of trust is the Intel CPU and the required trust is therefore reduced to Intel as the hardware manufacturer.

The features required for our work are *isolated execution*, *protected memory*, and *remote attestation*. Thereby, code execution cannot be interfered with by any third party and runtime data is protected with regard to confidentiality, integrity and freshness. The code becomes identifiable and its identity can be proven to a remote party. The secure container separating the code and data from the host operating system is called *enclave*. To run code inside an enclave it needs to be initialized by an untrusted wrapper application that starts the enclave and performs its system calls.

B. Use of TEEs in Bitcoin environments

Teechain [6] uses TEEs to implement a payment channel network. In a payment channel network, users have to deposit their funds into payment channels between them and other users. This limits the transferable credit to the capacity of the channels and it is not guaranteed that a route to another user exists. Furthermore, this implies that transactions have to be routed via other network members, which causes a transaction delay and communication overhead. In our approach, all funds can be moved freely inside the network. To defend against replay attacks, Teechain relies on hardware monotonic counters. As hardware monotonic counters are limited in their performance [7], Teechain does not use hardware monotonic counters for setups that require higher performance, but then enclaves are required to always be running in order to prevent replaying old states. To backup those running enclaves, their state can be replicated to other running enclaves (chain replication). When transactions are performed, all chain members have to acknowledge the state transition. As replication chains induce performance overhead and limit the cash-out possibility, we developed an approach which is secure against replay attacks without the need of hardware monotonic counters while still allowing backups of an enclave's state. Concerning the implemented service, Teechain is most likely the closest approach to our work because it can also be seen as an implementation of a decentralized payment service. By presenting BANKLAVES, which differs largely from Teechain's design, we show by example that the design space of blockchain technology and TEEs is not yet fully explored and comprises different approaches to solve similar challenges.

Tesseract [8] implements a cryptocurrency exchange where users can deposit coins and trade them against other coins. To ensure that users can still access their funds in case the services becomes unavailable, Tesseract implements a time limit for each deposit transaction. A deposit transaction can be spent on two conditions: either with the secret the Tesseract server knows or two weeks after publication with a secret owned by the user. So, the deposit is valid only for a limited time of two weeks. Obscuro [9] uses a similar mechanism. In our approach deposits are valid for an unlimited time and it is guaranteed that users are always able to reclaim their deposits (cf. Section VI).

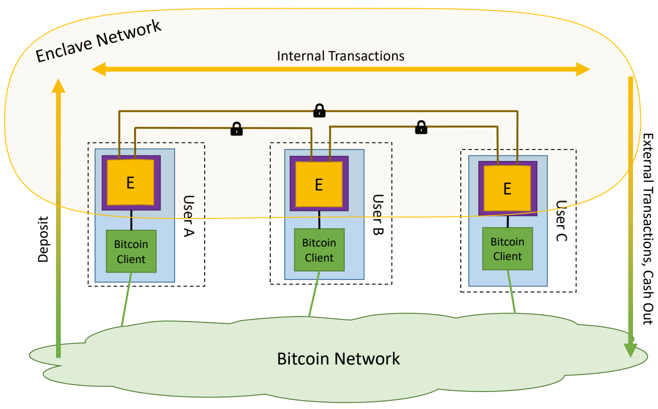


Fig. 1. Overview of an enclave network with three users. The dashed boxes show which machines are controlled by each user. Each machine (blue box) runs the same application (purple box) and a Bitcoin client, which is used by the application to request new blocks and transactions. The main part of an application is the enclave (orange box) with the trusted execution environment. All enclaves are connected to each other over the Internet.

III. CONCEPT OVERVIEW AND CHALLENGES

Our proposed approach is built on top of a decentralized network of enclaves (Fig. 1). To be a member of the BANKLAVES enclave network, each user executes the same client code, which is an open source software that can be publicly validated. The client consists of a trusted and an untrusted part. The trusted part is the code that is run inside the enclave and is therefore protected against tampering and reading of data. The untrusted part provides the user interface and network communication as well as secondary storage accesses. Each user also runs a client for Bitcoin, which is used by the enclave to retrieve blocks and transactions. The enclaves run by the users connect to each other and form the network of enclaves.

Each enclave runs the same code and offers its owner the following actions: depositing bitcoins, performing internal transactions to another network member, performing external transactions to Bitcoin addresses, and cashing out the current balance. Each enclave E_i stores a list containing the balances c_m of each network member m in its state S_{E_i}. After an enclave performed a transaction, it distributes the updated balances in the network. To give a better overview of the system's functions, we present a simplified and insecure approach before we explain the key components of our full approach that are needed for a secure protocol.

A. Simplistic approach

To create a new network, Alice starts her client which creates the first enclave E_{Alice} and Bob joins the network by letting his enclave E_{Bob} perform a remote attestation with E_{Alice}. To make a deposit, Alice asks her enclave to generate a deposit address for her. Then, Alice creates a Bitcoin transaction to this address, publishes it on the blockchain and gives the transaction to her enclave. E_{Alice} checks that the transaction is part of a confirmed block on the blockchain and increments Alice's balance in the state. The unspent output of the transaction used for the deposit is also stored in the state.

If Alice wants to make an internal transaction to Bob, she calls the respective function in her enclave and passes the amount and the receiver of the transaction. E_{Alice} verifies that Alice has enough balance available for that transaction and then updates the balances and sends them to E_{Bob}. At any time Alice and Bob can do a cash out without requiring interaction with the other party. If Alice wants to cash out, her enclave creates a Bitcoin transaction that she can publish on the blockchain. This transaction spends c_{Alice} bitcoins from the unspent transaction outputs managed by the enclave network to an address provided by Alice.

B. Assumptions and attack model

The users do not trust each other. This implies that the local trusted parts do not trust the local and remote untrusted parts which include the Bitcoin clients and, thus, the blockchain I/O. Users trust TEEs, both their own and those of other users, as well as their own computer including the operating system and the Bitcoin client. We assume that the network members generally aim to provide a working payment service.

We assume that an attacker can roll back their local enclave to an arbitrary older state. Furthermore, an attacker can prevent communication between the overlay network members without any restrictions. However, we do not consider an attacker that could attack arbitrary connections of the underlying Internet architecture because this would break the security of the Bitcoin ecosystem and, thus, the foundation of our approach. This means that we assume users to always be able to read and write to the Bitcoin blockchain. We do not consider problems specific to concrete trusted execution environment implementations and assume a bug free implementation of the proposed architecture.

C. Security objectives and challenges

A payment service is subject to several objectives that are demanded by its users. The balances of users and the transactions should stay *confidential*. In a distributed setup, the network members need to have a consistent view of the system. Hence, the *integrity* of the overall system is another objective. Lastly, the users of a payment service want to control their deposited coins, as far as possible, without restrictions. Thus, the service must be *available* for its users. Being based on TEEs, the simplistic approach can easily fulfill the confidentiality objective. Availability and integrity cannot be fulfilled trivially which is shown by the following challenges:

C1 *No system boundary:* The simplistic approach lacks a binding between the enclaves to the payment service they form. It has to be ensured that an enclave only operates in its enclave network to prevent inconsistencies.

C2 *Wrong blockchain:* Alice's enclave E_{Alice} depends on Alice to provide her with the correct blockchain. Alice might create a deposit transaction and include it in a forked blockchain that is mined by her, but not publish it on the main chain. If the enclave only sees Alice's fork, E_{Alice} accepts the deposit and allows an internal transaction to Bob.

C3 *Rollback attack:* Alice could perform a double spend by running a rollback attack. Storing the state of her enclave as S_0, she can transfer some bitcoins internally to Bob, which leads to state S_1. After doing a rollback of her enclave to state S_0, she can (1) perform an internal transaction to Charlie, who does not know of state S_1, or (2) perform a cash out and get her original balance cashed out.

C4 *Conflicting cash out transactions:* If Alice and Bob both cash out, it is possible that their cash out transactions both spend the same transaction output if they share a deposit, as a consequence of an internal transaction.

IV. Key Components

The aforementioned challenges lead to several key components of the proposed concept which are elaborated in this section. Table I relates the key components to the challenges.

A. Shared secrets as relationship binding

Each enclave has to know to which enclave network it belongs to. Firstly, the enclaves perform mutual remote attestations to ensure that they communicate with a code-identical counterpart. Secondly, the first enclave of a new payment service network draws a random byte string K_{System} that serves as an identifier of the new network. The enclaves compare their K_{System} at communication establishment.

Furthermore, an enclave has to be able to identify the user on behalf of which it performs payment service operations. Thus, we introduce shared secrets K_u between each user u and the user's local enclave. A user has to use this secret to authenticate against the enclave if the enclave is (re-)started.

B. Block identifier comparison as blockchain verification

To defend against an attacker providing a wrong blockchain, we introduce the identifier b_{E_i} of the most current confirmed block that is stored by each enclave E_i and compared when performing transactions. Each enclave receives all new headers of the Bitcoin blockchain and verifies them starting from the genesis block or, for better performance, from a hardcoded newer checkpoint. During an internal transaction to Bob, the system compares the latest confirmed block identifiers of Alice and Bob and aborts if they do not match. If Bob gives the correct blockchain to his enclave E_{Bob}, but Alice provides her enclave with a forked chain, the block identifiers will not match and the transaction will not succeed. Because the code run inside the enclaves is trusted, it is enough to verify that the enclaves see the same blockchain. The enclave that accepted a deposit can be trusted to have verified that the deposit is part of the given blockchain.

As an optimization, each enclave can validate that new blocks adhere to the difficulty rule of Bitcoin and that blocks arrive in intervals according to the expected distribution, as proposed in [8]. This makes a fork attack more difficult. However, to perform time-based validations, a trusted clock [10] is required.

C. Leader as rollback protection for internal transactions

Say, Alice has a balance of 5 bitcoins. The simplistic approach allows her to do the following attack: She stores her current state S_0 and then transfers 3 bitcoins to Bob, which decrements her balance to 2 in her and Bob's enclave. Then she restores state S_0 and transfers 3 bitcoins to Charlie. Charlies enclave does not know about her previous transaction to Bob and so it accepts her transaction. Now Alice successfully executed a double spend because there are two views in the network about who received Alice's coins. To mitigate this attack we could prevent rollbacks by using hardware monotonic counters in the TEE, but those limit the performance, bind the TEE to specific hardware and render the counter unusable as soon as the counter reaches its maximum [7]. Instead, we handle rollbacks by introducing the role of a leader who is involved in all state updates. With each state update, a (non-hardware) state counter is incremented. The enclave with the highest state counter becomes leader and verifies and executes all state updates. The leader itself cannot be rolled back, because when the leader is stopped or fails, it loses its status as leader and the enclave with the most current state becomes leader (cf. Section V-C4). Introducing a leader to a decentralized concept might sound contradictory. We therefore emphasize that our concept is indeed decentralized, because each enclave is able to provide a cash out operation self-sufficiently, because it has the most current state that is relevant for its owner.

D. Emergency cash outs as rollback protection for cash outs

In the presented simplistic approach it is possible for a user Alice to rollback to an old state and cash out the balance at that time. As long as Alice's enclave E_{Alice} does not communicate with other enclaves, it cannot distinguish whether its current state was replayed or it has the correct most current state. Therefore, E_{Alice} will always allow to create a cash out transaction and we use the Bitcoin blockchain to resolve possible conflicts. We construct a cash out transaction that locks the output for a given time span and makes the output also spendable for the other enclaves. Now, Bob's enclave has to watch the blockchain and when it sees a cash out transaction, it verifies that the transaction gives only so many bitcoins to Alice as E_{Bob} has stored as c_{Alice}. If Alice tried to cheat using a rollback attack and the amount in the transaction is too high, E_{Bob} creates a breach remedy transaction which moves the bitcoins back to the enclave. This mechanism is

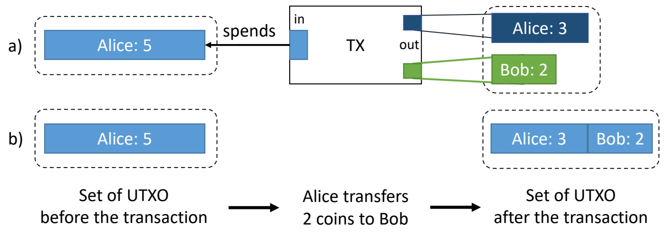

a)
Alice: 5
spends
in
TX
out
Alice: 3
Bob: 2
b)
Alice: 5
Alice: 3
Bob: 2
Set of UTXO before the transaction
Alice transfers 2 coins to Bob
Set of UTXO after the transaction

Fig. 2. Effect of a transaction of 2 coins from Alice to Bob using a) the Bitcoin blockchain and b) Satoshi Bookkeeping for internal transactions. Alice's UTXO is a) spent and two new UTXO are created (transaction fees are ignored) or b) internally marked as being shared by Alice and Bob.

known from payment channel networks [11]. However, it has the drawback that Alice has to wait for her cash out to be usable for her, while the other enclaves have the opportunity to verify the cash out. The advantage, however, is that Alice can always create a cash out transaction without communication with others. As this feature is only needed in the emergency case when another party behaves incorrectly, we call this type of cash out an *emergency cash out* (ECO). We also introduce the *withdrawal* as another type of cash out, which works like an external transaction and needs cooperation of other network members, but makes the output directly spendable by Alice.

E. Satoshi bookkeeping to remove the reliance on the leader

Bob's enclave E_{Bob} must not assume misbehavior by Alice unless it is ensured that Alice did in fact misbehave. If E_{Bob} has an outdated version of c_{Alice}, it might wrongly create a breach remedy transaction. To make sure that the breach remedy transaction is only created from the most current state, we could limit the ability to create it to the leader, but this would allow the user running the leader enclave to prevent the breach remedy transaction from being published to the blockchain. Instead, users should be able to publish breach remedy transactions independent of the leader. To allow that, we developed a way that allows users to create correct breach remedy transactions even if they do not have the most current global state.

The idea is that each enclave manages its user's bitcoins. Therefore we internally emulate the effect of blockchain transactions on the set of unspent transaction outputs (UTXO). As Fig. 2 shows, on the blockchain a transaction spending a UTXO would create two UTXOs: a change UTXO and a UTXO spendable by the receiver. In the enclave network we simply store the information that the UTXO is now partially owned by the receiver, to avoid the creation of blockchain transactions. We implement this by storing not only the deposited UTXO, but also the allocation of the UTXO to their respective owners.

An ECO transaction contains the current state counter and for each input two numbers that specify which part of the respective UTXO is spent[1]. If an ECO transaction is spotted on the blockchain that does not conflict with the UTXO

[1]80 bytes are available for additional data using one OP_RETURN in a transaction. [9]

TABLE I
OVERVIEW OF THE COMPONENTS

Challenge	Component
(C1) No system boundary	K_{System} Identifier
(C2) Wrong blockchain	Block identifier
(C3) (1) Rollback attack (int. transaction)	Leader
(C3) (2) Rollback attack (cash out)	Emergency cash out (ECO)
(C4) Conflicting cash out transactions	Satoshi bookkeeping

belonging to the enclave's owner, an enclave does not need to create a breach remedy transaction. However, if the published transaction conflicts with a UTXO belonging to the enclave's owner and the published transaction includes a lower state counter than the enclave's state counter, then a breach remedy transaction should be published. As the leader always knows the most current state, the leader can create breach remedy transactions as well. This means in case of the publication of an outdated ECO transaction, the leader and the users who are cheated by that transaction can create a breach remedy transaction.

Note that, with this construction, ECOs may conflict in non-malicious scenarios, too. When Alice and Bob own shares of the same UTXO and both create an ECO for the most current state, the Bitcoin network will not accept both transactions since both spend the same UTXO. To allow Bob to get his money, Bob can show Alice's transaction to his enclave and E_{Bob} will create a new transaction, which spends an output of Alice's transaction (so Bob needs to make sure Alice's transaction gets confirmed on the blockchain instead of trying to win the race against Alice and publish his two transactions).

V. BANKLAVES PROTOCOL

The protocol is designed to offer the functions listed in Section I: network join, deposit, internal and external transactions including withdrawals and network leave. To allow those functions to be performed, we first need a function to create a new payment service enclave network and a function to elect a leader. Furthermore, we provide a function that implements the emergency cash out component and a function to recover the local client with global state information. Network members who wish to perform an operation establish a secure and reliable communication channel to the parties involved in the operation and perform a mutual remote attestation. The leader only accepts one channel per member at a time.

A. Network state and state identification

The state of the network S_{E_u} as stored by enclave E_u at user u is defined by the set of current network members M, their identifiers K_M and account balances c_M, the state counter value σ^c_M at which state their balances changed last, the current deposit indices δ_M, private and public base keys sk^{B}_M and pk^{B}_M for deposit key derivation (cf. Section V-B2), emergency keys pk^{E}_{M}, the UTXO set D, the set of ownerships Ω, and a state counter $\sigma_{E_{\text{u}}}$:

$$S_{E_u} = (M, K_M, c_M, \sigma^c_M, \delta_M, pk^{\text{B}}_M, sk^{\text{B}}_M, pk^{\text{E}}_{\text{M}}, D, \Omega, \sigma_{E_{\text{u}}}).$$

Here, K_M is used for $\{K_j \mid j \in M\}$. The set of current network members M consists of all users who performed "network join" but did not perform an ECO or a "network leave". D is a set of spendable deposits (the UTXOs). Each entry $d \in D$ is a tuple of the transaction id of the deposit transaction, the deposited amount, and the information needed to derive the private key to spend the deposit (deposit index and depositing user). Ω is a set of ownerships that assign an owner to a part of a UTXO. Each ownership $\omega \in \Omega$ is a tuple of an owner, the deposit d, and the start position (ω.startpos) and the size (ω.amount) of the part that is assigned to the owner. As an example we give the ownerships according to the new set of UTXO in Fig. 2 b): $\Omega = \{(\text{“Alice”}, d_1, 0, 3), (\text{“Bob”}, d_1, 3, 2)\}$. We denote by $\Omega_u \subseteq \Omega$ all ownerships ω of user u. In Section V-B3 we explain how an internal transaction affects Ω.

The state counter σ_{E_u} is a unique identifier for the current state. The leader increments σ_{E_u} for network joins, deposits, external and internal transactions, network leaves and ECOs. To ensure that the state counter increases monotonically and identifies a state unambiguously, other enclaves do not change the state counter.

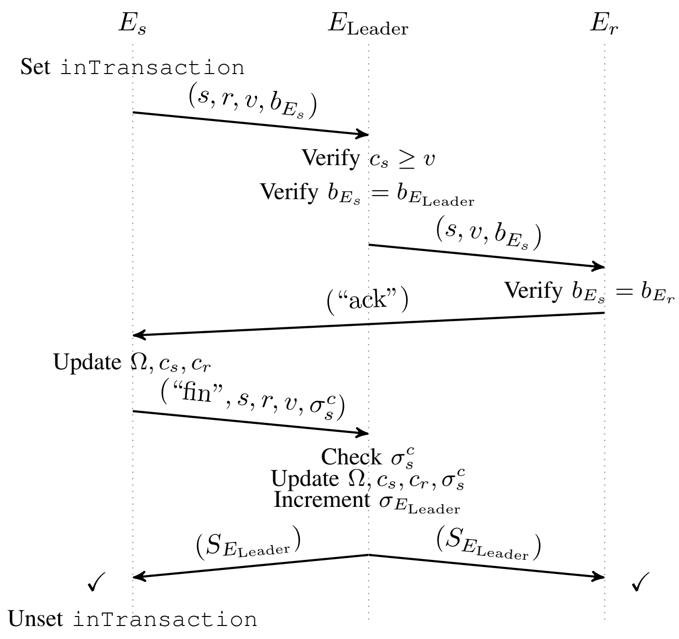
B. Payment service operations

1) Network join: Bob wants to join an enclave network. He starts his client, which initializes E_{Bob} and starts the join procedure parameterized with the IP address[2] of the current leader client, which runs the E_{Leader} enclave. Using this function, Bob's enclave and E_{Leader} perform a mutual remote attestation, which verifies that both run the correct code. E_{Bob} marks that it is owned by Bob, generates K_{Bob} and a secp256k1 [12] elliptic curve private key $sk^{\text{B}}_{\text{Bob}}$ with the public key $pk^{\text{B}}_{\text{Bob}}$. Furthermore, E_{Bob} asks Bob for his emergency cash out address $pk^{\text{E}}_{\text{Bob}}$. The generated keys and $pk^{\text{E}}_{\text{Bob}}$ are sent to E_{Leader}, which adds them to its state. E_{Leader} in return sends the current state $S_{E_{\text{Leader}}}$ to E_{Bob}. The generated keys are owned and managed by the decentralized enclave network. A public key is associated with a specific network member to identify the corresponding deposits. The TEE ensures that the implemented access control mechanisms on the private keys are always enforced.

2) Deposit: To get bitcoins into the payment service, Bob can make a deposit. Bob starts the deposit function offered by his enclave E_{Bob}, which creates a new Bitcoin deposit key pair $(pk^{\text{D}}_{\text{Bob},\delta_{\text{Bob}}}, sk^{\text{D}}_{\text{Bob},\delta_{\text{Bob}}})$. This deposit key pair is generated from Bob's secp256k1 base key pair ($sk^{\text{B}}_{\text{Bob}}$ and $pk^{\text{B}}_{\text{Bob}}$), δ_{Bob} and K_{System} using the key derivation algorithm used in the Lightning Network [13]. After the key generation, E_{Bob} gives $pk^{\text{D}}_{\text{Bob},\delta_{\text{Bob}}}$ to Bob. Bob uses his Bitcoin client to create a transaction t_{Deposit} transferring some bitcoins to $pk^{\text{D}}_{\text{Bob},\delta_{\text{Bob}}}$ and publishes it on the Bitcoin blockchain.

Bob's enclave E_{Bob} monitors the blockchain using Bob's Bitcoin client and waits until a block containing t_{Deposit} is confirmed. E_{Bob} sends t_{Deposit}, Bob's name, and the most current confirmed block id $b_{E_{\text{Bob}}}$ to E_{Leader}. E_{Leader} then

[2]In practice it might be necessary to be invited to a network or that the current network members need to accept the new member.



E_s
E_{Leader}
E_r
Set inTransaction
(s, r, v, b_{E_s})
Verify $c_s \geq v$
Verify $b_{E_s} = b_{E_{\text{Leader}}}$
(s, v, b_{E_s})
Verify $b_{E_s} = b_{E_r}$
("ack")
Update Ω, c_s, c_r
("fin", s, r, v, σ_s^c)
Check σ_s^c
Update $\Omega, c_s, c_r, \sigma_s^c$
Increment $\sigma_{E_{\text{Leader}}}$
$(S_{E_{\text{Leader}}})$
$(S_{E_{\text{Leader}}})$
✓
✓
Unset inTransaction

Fig. 3. Sequence diagram of an internal transaction from user s to user r.

verifies that $b_{E_{\text{Bob}}}$ matches $b_{E_{\text{Leader}}}$ to make sure that E_{Bob} verified the confirmation of the deposit transaction on the same blockchain that the leader knows[3]. After a successful validation, E_{Leader} adds a new deposit to the UTXO set D and creates an ownership ω in Ω for it, in which the whole amount of the deposit is marked as being owned by Bob. E_{Leader} finally increments Bob's current credit c_{Bob} by the amount transferred to $pk^{\text{D}}_{\text{Bob},\delta_{\text{Bob}}}$ and the deposit index δ_{Bob} by 1, updates the state counter $\sigma_{E_{\text{Leader}}} \leftarrow \sigma_{E_{\text{Leader}}} + 1$, sets $\sigma^{\text{c}}_{\text{Bob}}$ to the new state counter value and sends the updated state to all connected enclaves.

3) Internal transaction: For Alice to make a transaction to another member Bob of the same payment service, Alice calls the function INTERNALTRANSACTION("Alice", "Bob", v), illustrated in Fig. 3, on E_{Alice}. The parameter v is the amount of bitcoins that Alice wants to transfer. Alice's enclave sets the `inTransaction` flag to make sure that no other transaction is started in parallel. Then, E_{Alice} sends a tuple containing the transaction parameters and her last confirmed block id to E_{Leader}. E_{Leader} verifies that Alice has enough credit to cover the transaction and that Alice's block id and its own block id match. If they match, E_{Leader} sends the data to E_{Bob}, which also verifies that the received block id matches the local one and sends an "ack" to E_{Alice} on success.

After receiving the "ack", E_{Alice} updates the ownerships Ω (see next paragraph), updates c_{Alice} and c_{Bob} to the values after the transaction and sends a "fin" back to E_{Leader}. E_{Leader} verifies that Alice included in her "fin" the same value of σ^c_{Alice} as E_{Leader} has stored for σ^c_{Alice}. Then, E_{Leader} also updates Ω, c_{Alice} and c_{Bob}, increments $\sigma_{E_{\text{Leader}}}$, sets σ^c_{Alice} to the new $\sigma_{E_{\text{Leader}}}$ and sends the new state to all connected enclaves. When E_{Alice} and E_{Bob} receive the new state, they accept the transaction as successfully executed and E_{Alice} removes the `inTransaction` flag.

We shortly explain a simplistic algorithm for changing Ω during an internal transaction of v from sender s to receiver r, which, however, is not optimized for reducing fragmentation: Copy v to v_t. We loop over each ownership ω in Ω_s and compare v_t to ω.amount. If $\omega.\text{amount} \leq v_t$, we change the owner of ω to r and reduce v_t by ω.amount. If $\omega.\text{amount} > v_t$, we reduce ω.amount by v_t, put a new ownership entry $(r, \omega.d, \omega.\text{startpos} + \omega.\text{amount}, v_t)$ in Ω, and stop.

In case Alice does not receive the state update from the leader, she does not know whether the leader processed the transaction or not. To prevent an invalid ECO in the first case, Alice can only cash out her balance available after the transaction, although in the later case Bob has not received the transaction. The problem that Alice and the leader need to find consensus on the new state without knowing whether the other party received a message is analogous to the *Two Generals Problem* [14, 15] and is thus proven unsolvable. In such a situation, Alice can only send the "fin" again to conclude the transaction. The verification of σ^c_{Alice} prevents the transaction from being executed twice. Note that, while Alice needs to keep state during the internal transaction, the leader and Bob do not need to keep state. So, the leader might even change during the process of the transaction.

4) Emergency cash out: It might happen that Alice's enclave E_{Alice} cannot reach the leader. In that case, Alice cannot perform or receive any transactions anymore, but she can still cash out. To that end, E_{Alice} generates a new Bitcoin transaction with a time-locked output to the emergency address stored for Alice, a change output to an address owned by the enclave network, and, as additional data, the value of the state counter and for each input amounts and the start positions for the parts that are owned by Alice. As inputs to the transaction, E_{Alice} chooses all deposits from Ω_{Alice}.

After the time lock Alice is free to spend her coins. During the time lock all other enclaves E_u have to monitor the blockchain and check the transactions (this means each enclave has to be online at least once during the time lock). If they find Alice's cash out transaction, they verify that the ownerships of Alice encoded in the transaction do not conflict with Ω_u. Two ownerships ω_1 and ω_2 conflict if they reference the same deposit d and their intervals defined by $[\omega.\text{startpos}, \omega.\text{startpos} + \omega.\text{amount}]$ intersect. Two sets Ω_1 and Ω_2 conflict if they contain conflicting ownerships.

If Alice's transaction conflicts and it has a lower state counter, they create a breach remedy transaction that transfers the bitcoins back to the enclave. The time span for the time lock is a trade-off between Alice being able to spend her coins quickly and giving the other users enough time to verify her ECO. We suggest a lock time of 24 hours.

If Alice's transaction does not conflict but spends the same UTXO as referenced by Ω_u of a user u, u has to update Ω and D. To be able to perform internal transactions further on, the leader has to perform and announce a state update when an ECO is observed.

[3]It is enough to verify that $b_{E_{\text{Bob}}}$ is a parent of $b_{E_{\text{Leader}}}$.

If the ECO transaction is not accepted by the Bitcoin network because a double spending transaction t_{conflict} has been published by another user, Alice can show t_{conflict} to E_{Alice}, which will create another ECO transaction that spends the change output of t_{conflict}.

5) External transaction and withdrawal: Making an external transaction is similar to an ECO but the leader creates and publishes the transaction with no time lock. Additionally, the transaction is sent to Alice's enclave to publish it from her computer as well in case the leader's host does not publish it. The leader updates the state (c_{Alice}, Ω_{Alice}, $\sigma^{c}_{\text{Alice}}$, $\sigma_{E_{\text{Leader}}}$) and sends updates to all connected enclaves. Note that, if Alice shares UTXOs with other network members u, their Ω_u change, too. If the affected users are not online while Alice performs the external transaction, they receive the changed Ω from the leader as soon as they are online. A withdrawal is an external transaction to Alice herself.

6) Network leave: To leave the network, Alice performs a withdrawal of all her deposits and tells the leader to remove her from the state. The leader removes Alice's identifier and her keys from the state and increments the state counter.

C. Management operations

1) Network creation: To create a new payment service network, Alice starts her client, which creates an enclave E_{Alice}. E_{Alice} creates a system identifier K_{System} that is used to identify the payment service internally, but never leaves the network of enclaves. At this point E_{Alice} is the only enclave in the network and becomes the networks leader: $E_{\text{Leader}} \leftarrow E_{\text{Alice}}$. Then, the network join procedure is performed locally.

2) Enclave recovery: In case Bob lost his enclave (e.g., because his computer crashed), he starts the local client, which creates E_{Bob} and connects to the current E_{Leader}. After performing a mutual remote attestation, Bob authenticates himself by providing his identifier K_{Bob} to E_{Bob} which sends K_{Bob} to E_{Leader}. E_{Leader} then sends the current state to E_{Bob}. Now E_{Bob} is ready to be used.

3) Graceful leader handover: When the user running E_{Leader} shuts down the client, E_{Leader} will start a graceful leader handover. E_{Leader} chooses one of the enclaves it is connected to, say E_{Alice}. Then E_{Leader} sends the current state $S_{E_{\text{Leader}}}$ to the chosen enclave and stops being leader. E_{Alice} updates its state, marks itself as leader and propagates the information about the new leader in the network.

4) Leader election on loss of connectivity: If an enclave cannot reach the current leader anymore, it starts a leader election. A leader election has a timeout (e.g., 24 hours) during which all members $u \in M$ have to get online once and participate. Each enclave connects to all other enclaves and sends its own state counter. For each received state counter it compares the received state counter to its own. For the first received state counter that is higher than the own state counter it sends a commit to the sender enclave and ignores all other received state counters. If the state counters are equal, both enclaves draw a random value additionally and compare that instead of the state counter. For each received commit an enclave either forwards the commit to the enclave it committed to, or, if it did not commit to another enclave, it counts the number of received commits. If all enclaves are online, an enclave will eventually have received commits (directly or indirectly) from all other enclaves. This enclave becomes the new leader and publishes the current state to all other enclaves. In this process, only *one* enclave with the most current state (viz. the highest state counter) can become the new leader. It is not possible to reelect the old leader enclave without additional checks since it might have been rolled back. If the process is not successful, a user can use the ECO function or wait.

D. Asymptotic analysis

The space consumption of the state lies in $O(|M| + |D| + |\Omega|)$, with the size of the set of ownerships $|\Omega|$ being in $O(|D| + n)$, where n is the number of performed internal transactions. The size of exchanged messages is maximal for state updates, which means the message size is in $O(|M| + |D| + |\Omega|)$. The complexity of the leader election lies in $O(|M|^2)$, the complexity of the other operations in $O(1)$. Please note that the algorithms leave space for optimization with regard to needed space and complexity.

VI. Proof of Cash Out Feasibility

In this section, we provide a proof for our availability and integrity objectives. Intuitively spoken, we prove that "As long as I am honest, I always get my money, no matter what funny things the others do."

Theorem 1. *At any time, except during the process of doing an internal transaction, a non-rolled back enclave can perform a valid cash out without the need for communication with other enclaves.*

Proof. We prove the theorem by showing that

(1) in an enclave network in a non-conflicting state S_n a cash out is always possible,

(2) when no rollbacks happen in an enclave network, a system in a non-conflicting state S_n can only change to a non-conflicting state S_{n+1}, and

(3) rollback-induced conflicts can be resolved in favor of a non-rolled back enclave.

An enclave network is in state S_n if all enclaves are in state S_n. A state S_n is non-conflicting if all $\omega \in \Omega$ of that state are non-conflicting (see V-B4).

(1) *Cash out with no conflicts:* The system is in a non-conflicting state S_n. As long as no ECO transaction has been published, each user u can create a valid ECO transaction. Say, Alice has already published an ECO transaction. If another user Bob, whose Ω_{Bob} do not spend common UTXO with Ω_{Alice}, creates an ECO transaction, the transaction is accepted by the Bitcoin network. However, if Ω_{Bob} do spend common UTXO with Ω_{Alice}, for the Bitcoin network Bob's ECO transaction will be a double spend for Alice's published ECO transaction. But because all $\omega \in \Omega$ are non-conflicting, E_{Bob} can verify that Alice's ECO transaction spends different parts of the UTXO and so E_{Bob} will create an ECO transaction for Bob depending on Alice's published ECO transaction. Cases in which more than one ECO transaction are already published are handled similarly.

(2) *State transition from non-conflicting state leads to non-conflicting state:* Not considering rollbacks, due to the assumptions of TEEs there are the following operations that lead to a new state:

Network join: Ω is not changed, so conflicts cannot occur.

Deposit: Deposits only change the Ω_u of the user u performing it; no new conflicts are created.

Internal transactions: W.l.o.g. Alice transfers internally c bitcoins to Bob. During the transaction the owner of parts with the sum of c bitcoins is changed from Alice to Bob. The Ω_u of the other users $u \in M \backslash \{\text{Alice}, \text{Bob}\}$ stay the same. Therefore, the new set Ω does not conflict. We except the process of internal transactions, because if Alice updates her state before the leader does, she has a different view on Ω_{Alice} and Ω_{Bob} than the rest of the network until the leader updates to the new state. Thus, if E_{Alice} is separated from the network during this time frame, Alice can only cash out her balance after the transaction although the transaction has not yet completed.

External transactions, ECOs, network leaves: These operations reduce Ω_u of the user u performing it and might update affected ownerships but do not change their amounts; no new conflicts are created.

(3) *Consideration of rollbacks:* We pick an arbitrary but firm non-rolled back enclave E_α. We call the state of E_α S_a with $a \leq n$ where n is the index of the most current state. The other enclaves are in states $S_i, i \leq n$ and some might be rolled back to earlier states. We prove that, in this situation, a valid cash out for E_α is still possible. For all enclaves E_u in a state $S_j, j \leq n$ two cases are to be distinguished: Either they are current or they have been rolled back. In the first case, their local Ω_u are the same in S_n as in S_j and, thus, non-conflicting to S_a. In the second case, for each enclave E_ρ rolled back to state $S_r, r < j$ there exist two possibilities:

$S_r, r \geq a$: If the Ω_α of E_α would be different in S_r than in S_a, then E_α would not be in its most current state. Thus, transactions created by E_ρ and E_α might use the same UTXO but they will not have conflicting Ω_ρ and Ω_α.

$S_r, r < a$: In this case Ω_ρ and Ω_α may conflict. However, conflicts can be resolved in favor of α by comparing the state counters. E_ρ creates an ECO with state counter r. E_α will notice the conflicting Ω_ρ and Ω_α and because $r < a$ it will create a breach remedy transaction. If E_α creates an ECO with state counter a, E_ρ will notice the conflicting Ω_ρ and Ω_α. It will not be able to create a breach remedy transaction, because its own state counter r is smaller than a.

With steps (1) - (3) we proved that starting from a non-conflicting state, in each following state, even conflicting ones, a non-rolled back enclave can create a valid cash out transactions that cannot be attacked. As each enclave network starts with an empty D and Ω, which is trivially non-conflicting, this proves the statement. □

VII. Discussion and Future Work

Privacy of the transaction history: The security objectives contain confidentiality goals for the user balances and their performed transactions. The TEEs hide their local state from curious attackers. To prevent information on the transaction history from being leaked by the additional data in external transactions, the system identifier K_{System} can be used to derive a symmetric key to encrypt this data.

Power of the leader's host, incentives and fees: The system running the leader enclave is in a position where it is possible to prevent the enclave network from performing its actual task. It can actively refuse the network's services for dedicated network members. Additionally, it is able to sabotage the complete network by going offline permanently and refuse participation in a leader election process. However, because of the assumptions of TEEs, the leader enclave itself cannot be malicious and, thus, the leader's host cannot manipulate the network's state in its favor. An idea to incentivize correct behavior and availability by the leader is to introduce a fee, which is paid to the leader. Additionally, fees can be used as a reserve to pay for blockchain transactions issued by the enclave network itself. We consider such a system as future work. Unwarranted ECOs should be punished by the payment system to discourage users from trying to cheat the system.

Usability: We require users to be online to receive transactions to prevent rollback attacks against the receiver of a transaction. If users want to receive transactions without their active participation and are willing to take the risk, they could inform the leader to accept transactions in their absence.

Extended use cases for future work: The payment service could be used to participate in a *payment channel network* with the whole system instead of each user individually. As a consequence, more and better funded channels could be opened. The payment service could also be used in a *credit network* as a trustworthy gateway. It would accept deposits in Bitcoin and open credit links to the depositor in return, so the depositor could use the credit in the network to make payments to other parties.

VIII. Conclusion

We presented a concept for a payment service that is implemented as a decentralized network of TEEs in which users do not need to trust each other. Having TEEs as an interface between a user and the payment service limits each user to perform only valid actions by protecting the integrity of the code and the confidentiality of necessary secrets. Invalid actions caused by rollbacks of TEEs are prevented by our protocol instead of using hardware monotonic counters. However, as our approach heavily relies on the security of TEEs, which have been shown to be compromisable, we plan to address challenges coming with stronger adversary models in future work. Additionally, we plan to conduct a performance analysis using Intel SGX.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018.

[3] 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," in *Proc. of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013.

[4] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013.

[5] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," 2016. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[6] J. Lind, I. Eyal, F. Kelbert, O. Naor, P. R. Pietzuch, and E. G. Sirer, "Teechain: Scalable Blockchain Payments using Trusted Execution Environments," 2017. [Online]. Available: http://arxiv.org/abs/1707.05454

[7] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback Protection for Trusted Execution," *IACR Cryptology ePrint Archive*, vol. 2017, p. 48, 2017.

[8] I. Bentov, Y. Ji, F. Zhang, Y. Li, X. Zhao, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware," *IACR Cryptology ePrint Archive*, vol. 2017, p. 1153, 2017.

[9] M. Tran, L. Luu, M. S. Kang, I. Bentov, and P. Saxena, "Obscuro: A Bitcoin Mixer using Trusted Execution Environments," *IACR Cryptology ePrint Archive*, vol. 2017, p. 974, 2017.

[10] H. Liang, M. Li, Q. Zhang, Y. Yu, L. Jiang, and Y. Chen, "Aurora: Providing Trusted System Services for Enclaves On an Untrusted System," 2018. [Online]. Available: http://arxiv.org/abs/1802.03530

[11] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016. [Online]. Available: https://lightning.network/lightning-network-paper.pdf

[12] Certicom, "SEC 2: Recommended Elliptic Curve Domain Parameters," in *Standards for Efficient Cryptography 2 (SEC 2)*. Certicom Research, 2000. [Online]. Available: http://www.secg.org/sec2-v2.pdf

[13] *BOLT 3: Bitcoin Transaction and Script Formats*, Lightning Network In-Progress Specifications, 2018. [Online]. Available: https://github.com/lightningnetwork/lightning-rfc/blob/914ebab9080ccccb0ff176cb16b7a6ba21e2/03-transactions.md

[14] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber, "Some Constraints and Tradeoffs in the Design of Network Communications," in *Proc. of the Fifth ACM Symposium on Operating Systems Principles*, ser. SOSP '75. New York, NY, USA: ACM, 1975.

[15] J. N. Gray, "Notes on data base operating systems," in *Operating Systems*. Springer, 1978, pp. 393–481.