Teechain: A Secure Payment Network with Asynchronous Blockchain Access

Joshua Lind
Imperial College London

Oded Naor
Technion - Israel Institute of Technology and IC3

Ittay Eyal
Technion - Israel Institute of Technology and IC3

Florian Kelbert
Imperial College London

Emin Gün Sirer
Cornell University and IC3

Peter Pietzuch
Imperial College London

This is an extended version of the SOSP 2019 paper [55].

Abstract

Blockchains such as Bitcoin and Ethereum execute payment transactions securely, but their performance is limited by the need for global consensus. Payment networks overcome this limitation through *off-chain* transactions. Instead of writing to the blockchain for each transaction, they only settle the final payment balances with the underlying blockchain. When executing off-chain transactions in current payment networks, parties must access the blockchain within bounded time to detect misbehaving parties that deviate from the protocol. This opens a window for attacks in which a malicious party can steal funds by deliberately delaying other parties' blockchain access and prevents parties from using payment networks when disconnected from the blockchain.

We present *Teechain*, the first layer-two payment network that executes off-chain transactions *asynchronously* with respect to the underlying blockchain. To prevent parties from misbehaving, Teechain uses *treasuries*, protected by hardware trusted execution environments (TEEs), to establish off-chain payment channels between parties. Treasuries maintain collateral funds and can exchange transactions efficiently and securely, without interacting with the underlying blockchain. To mitigate against treasury failures and to avoid having to trust all TEEs, Teechain replicates the state of treasuries using *committee chains*, a new variant of chain replication with threshold secret sharing. Teechain achieves at least a 33× higher transaction throughput than the state-of-the-art Lightning payment network. A 30-machine Teechain deployment can handle over 1 million Bitcoin transactions per second.

1 Introduction

Cryptocurrencies, such as Bitcoin [65] and Ethereum [97], offer secure payments between distrusting parties using blockchains. Existing blockchains have limited performance due to their need for consensus across all transactions: global throughput is capped at a handful of transactions per second; transactions take minutes to hours to be processed and parties must maintain a history of every transaction executed.

Payment networks, such as Lightning [73] and Raiden [91], have been proposed as a more performant second layer on top of a blockchain. They allow parties to move fund deposits from the blockchain into point-to-point *payment channels*. Parties then exchange payment transactions directly *off-chain* via these channels, without having to involve the blockchain. Before a channel is terminated, it is *settled* by writing its final balance as a transaction back to the blockchain. Payment networks can therefore operate with higher transaction throughput and lower latency than blockchains [22].

Protocols for payment channels must ensure that parties cannot steal funds. In particular, only the most recent channel balance must be settled on the blockchain; otherwise a malicious party can settle a channel at a previous balance. Existing protocols thus require parties to *monitor* the underlying blockchain [73]: if a party observes that a stale balance is settled on the blockchain, they have a bounded *reaction time* Δ to invalidate the settlement. This requirement for *synchronous blockchain access*, i.e., parties must read blockchain transactions and write them within Δ, has drawbacks: (i) it makes payment networks vulnerable to attacks in which an adversary deliberately delays writes to [9, 24, 41–43, 76, 83] or reads from the blockchain [57] beyond Δ to steal funds; (ii) it prevents parties from using payment networks without connectivity to the blockchain; and (iii) it complicates the cryptographic protocols and the number of messages exchanged because parties must provide each other with means to cancel stale settlements [73].

Our key idea is that, rather than requiring parties to rely on the underlying blockchain to detect misbehaviour during off-chain transactions, we explore a design for a payment network in which parties use *trusted execution environments* (TEEs) [19, 67] as a root-of-trust to enforce faithful protocol execution. TEEs are a hardware security feature in modern CPUs [5, 37] that ensures the confidentiality and integrity of code and data. At the same time, we want our design to be resilient against TEE failures and attacks that compromise a subset of the TEEs [12, 64, 66, 94].

We describe *Teechain*, a new payment network that supports secure and performant payments on existing blockchains. Teechain only requires *asynchronous blockchain access*, i.e., parties are not assumed to read and write transactions on the blockchain within bounded time. Teechain uses trusted *treasuries*, which are protected by TEEs, to maintain fund deposits for off-chain payment channels. By relying on TEEs, treasuries can employ a new efficient off-chain payment protocol that simplifies both payment and settlement. To mitigate against TEE failures or compromises, treasuries replicate their state among a *committee* of treasuries. Within each committee, a treasury must have approval from a subset of other committee treasuries to make an off-chain transaction or settle a payment channel. TEEs therefore improve the efficiency of payment channels but the security of Teechain does not depend on that of individual TEEs.

Overall the design of Teechain makes three contributions:

(C1) Dynamic deposits with treasuries. Due to their binding with a blockchain, existing payment networks only support a fixed assignment of deposits to channels: parties cannot add or remove deposits after a payment channel is established. Instead, Teechain separates the ownership of fund deposits and channel assignment using treasuries. It only requires blockchain interaction during the initial creation of a fund deposit, whereby a treasury exclusively owns each deposit by storing the private keys for that deposit in a TEE. Parties can assign deposits to channels upon establishment using the treasuries, and move them in and out of channels at runtime. Since deposit assignment does not require blockchain access, new payment channels are established within seconds.

(C2) Payments with asynchronous blockchain access. After associating a fund deposit with a channel, a party makes a payment through a single integrity-protected message exchange. A payment message decrements the channel balance of its treasury and increments the balance of the recipient's treasury. This is done by duplicating the pair of balances across both treasuries, and updating them atomically. To settle the channel, a party requests a settlement transaction from the treasury, which is a blockchain transaction with the final balance. Settlement transactions can be written to the blockchain in unbounded time because the treasuries ensure that only a single transaction can be generated for a channel.

(C3) Committee chains. As private keys maintained by treasuries to spend fund deposits are stored inside TEEs, accidental TEE failures or malicious TEE compromises could result in fund loss or theft. Teechain therefore uses *committee chains*, which are committees of treasuries responsible for managing deposits. To replicate deposit balances in a committee chain, Teechain employs a new *force-freeze replication* protocol that prevents roll-back attacks. If a treasury in the chain fails to update its balance after a payment or tries to roll-back to a stale balance, the state of all treasuries is frozen, and they can only settle their balances safely. To mitigate against compromised TEEs [94], the committee chain uses the *multi-signature* support [88] of the underlying blockchain: a threshold number of signatures by treasuries from the committee chain are necessary to settle a payment channel.

We implement Teechain using Intel's SGX TEEs [36] and deploy it on Bitcoin.[1] Teechain achieves substantially higher throughput due to its more efficient off-chain payment protocol between treasuries: compared to the Lightning Network [52], Teechain handles 33×–145× more payment transactions depending on the size of committee chains. Channel establishment takes seconds, as opposed to minutes or hours [20, 73]. Teechain also reduces the number of transactions stored on the blockchain by at least 25% compared to the Lightning Network.

2 Secure Payment Networks for Blockchains

2.1 Blockchain protocols

In cryptocurrencies such as Bitcoin [65], Ethereum [89] and Zerocash [74], a set of nodes connect over a peer-to-peer network to operate as a replicated state machine. This state machine maintains an append-only *ledger* that contains the history of all transactions in the system. Each transaction is a payment from one system user, a *party*, to another, secured cryptographically. The ledger is a chain of *blocks*, or *blockchain*, such that each block contains a list of transactions.

Each transaction is a list of instructions that update the state of the blockchain. Different cryptocurrencies implement transactions that move funds differently: Bitcoin [65] follows an *unspent-transaction-output* (UTXO) model in which transactions consume, or use as *input*, a set of previously unspent transactions, where the *output* of those transactions are owned by the sender. A payment transaction therefore consumes unspent input transactions and generates new output transactions that recipients can spend; Ethereum [89] uses an *account model* in which a user's account balance is represented as an integer stored on the blockchain and updated by transactions.

Users are represented by cryptographic public keys. A user's transaction is validated with a cryptographic signature produced by the matching private key. To prevent users from *double-spending*, i.e., signing multiple transactions that spend the same funds, blockchains enforce that funds can only be spent once by making double-spending transactions *conflict*: only one transaction in a set of conflicting transactions can be written to the blockchain. Transactions may also support more elaborate conditions such as *m*-out-of-*n multi-signatures* that require signing by multiple users: such transactions must be signed by any *m* keys from a set of *n* keys.

[1] An initial release of Teechain can be found at: https://teechain.network.

In blockchains, nodes must agree on the order of transactions, i.e., they must reach consensus. The details of blockchain consensus are immaterial to this work—we treat consensus as a black box. Consensus, however, limits transaction throughput [96] and incurs high storage costs. In Bitcoin, global throughput is limited to 7 transactions per second [65], and the total size of the blockchain is 100s of GBs [20]. Due to consensus, transactions may also take arbitrarily long to be written to the blockchain—minutes or even days [9].

2.2 Payment networks and channels

Payment networks [56], such as Lightning [73] and Raiden [91], try to overcome the performance limitations of blockchains by allowing parties to exchange funds directly, *off-chain*. To execute a transaction, they establish a point-to-point *payment channel* [20, 54, 62, 72, 73]. A payment channel is a protocol between two parties, A and B, that updates their balances directly through message exchange. When a payment channel is closed, the payment network *settles* the channel by writing the final balances of A and B back to the blockchain using a *settlement* transaction. Since payment networks do not write to the blockchain for each transaction, their transaction throughput is higher and latencies lower compared to on-chain payments [73]. Payment networks also reduce the number of transactions stored on the blockchain because only final balances are recorded [73, 91].

To establish a payment channel c, as shown in Fig. 1, one or both of A and B write *fund deposit* transactions to the blockchain. These place funds into a 2-out-of-2 *multi-signature* account [88] owned by both parties, and requires both A and B to cryptographically sign any transaction in order to spend the funds. A creates a fund deposit d of \$1000 for c (step ❶). Using the fund deposits, A and B can then execute payment transactions: a new payment transaction is generated and signed by both parties, spending from the channel deposits and reflecting the new balances. For example, A pays B \$100 using tx_1 signed by both A and B (step ❷), and B, whose balance is now \$100, sends A \$50 using tx_2, also signed by both parties (step ❸). Note that tx_1 and tx_2 do not require interaction with the blockchain and that each payment takes into account all previous payments and updates the current state of the payment channel. At any time, either A or B may close the channel by writing the most recent payment transaction to the blockchain: B settles the channel by writing tx_2 to the blockchain with their final balances (step ❹).

Payment networks also support *multi-hop* payments [56, 62, 73] in which multiple payment channels, c_1 to c_n, are concatenated to form a payment path. This allows for payments between parties that do not have a direct payment channel. This makes payment networks useful in practice, because it allows payments between parties without long-lived financial relationships, e.g., e-commerce buyers and sellers who conduct transactions via intermediaries [1] such as Amazon [2] and eBay [27]. Similar to a single payment channel, any party along the path can unilaterally settle its channels. The added guarantee is atomicity: either all channels c_1 to c_n are settled at the state after the multi-hop payment, or all settle before it.

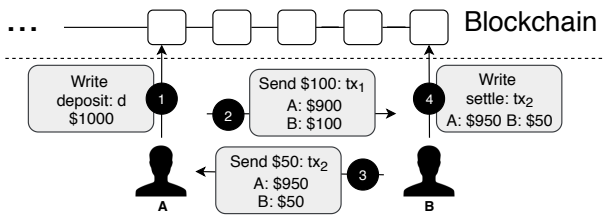

Figure 1. Payment channel in operation

2.3 Limitations of payment networks

To avoid fund theft or loss, payment networks must only settle channels with the most recent payment transaction; otherwise a malicious party can launch a *roll-back* attack in which they settle the channel at a previous payment transaction with a stale balance. For example, in Fig. 1, step ❹, if B settled c using tx_1 instead of tx_2 it would allow B to steal \$50 from A.

Existing payment networks [20, 72, 73] overcome this problem by requiring parties to detect misbehaviour using information available on the blockchain: when using a payment channel, each party monitors the blockchain for a settlement transaction written by its counterparty to settle the channel. If an old settlement transaction is written, the party negates its effect by writing the most up-to-date settlement transaction to the blockchain within a bounded *reaction time* Δ.

For this mechanism to work, the payment network must assume that parties can read and write transactions on the blockchain within the fixed upper bound Δ. We refer to this assumption as *synchronous blockchain access*.

In practice, it is not always possible to ensure synchronous blockchain access during payment channel operation. The load on the blockchain may result in long queues to write transactions [9]. Moreover, an attacker may delay transaction writes deliberately, such as by controlling the order in which transactions are written [41, 42, 83], or censoring transactions [24, 43, 76]. Attackers may also partition victims from the network [57], preventing them from accessing the blockchain at all. Current payment networks therefore face a trade-off when selecting the reaction time Δ: a short Δ allows for quick settlement but facilitates the above attacks.

The requirement for synchronous blockchain access also prevents parties from using payment channels when they are *disconnected* from the blockchain. This negates one of the benefits of payment networks: parties can no longer exchange payments directly with only point-to-point network connections. For example, it becomes impossible to use a payment channel between two devices that are directly connected, but do not have connectivity to the Internet and thus the rest of the blockchain [23].

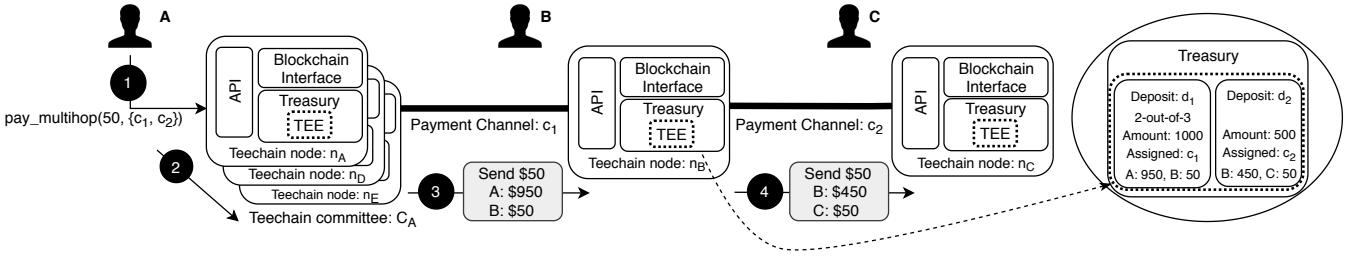

Figure 2. Teechain overview (Teechain nodes operate *treasuries* to store and manage funds. Users construct *payment channels* between nodes to exchange funds directly, and execute *multi-hop* payments along concatenated payment channels. *Committee chains* with multiple treasuries replicate and protect state.)

3 Teechain Design

Next we introduce how Teechain uses trusted execution (§3.1), state the threat model (§3.2), give an overview of its design (§3.3), describe treasuries (§3.4) and committees (§3.5), and analyse how the design handles different threats (§3.6).

3.1 Trusted execution as a root-of-trust

The requirement for synchronous blockchain access in existing payment networks comes from the fact that their protocols use the blockchain as a root-of-trust: parties executing the payment protocol monitor the blockchain to discover when other parties deviate from the protocol, and react appropriately.

We explore a design that introduces a separate root-of-trust that, independently of the blockchain, ensures the faithful execution of a payment protocol. Our idea is for the payment network to use *trusted execution environments* (TEEs) [37, 44] during the execution of a payment protocol. TEEs are encrypted and integrity-protected memory regions, which are isolated by the CPU hardware from the rest of the software stack, including higher privileged system software. Multiple TEE implementations are commercially available, including Intel SGX [37], ARM TrustZone [5] and AMD SEV [44], with several others currently under way, such as KeyStone Enclave [45], Multizone [34] and OP-TEE [53]. Intel CPUs from the Skylake generation onwards support *Software Guard Extensions* (SGX) [35], a set of new instructions that permit applications to create TEEs called SGX enclaves.

By using TEEs as an independent root-of-trust, we want our design to only require *asynchronous blockchain access*, i.e., the payment protocol must not assume that transaction reads and writes to the blockchain complete within a fixed upper bound, but only complete *eventually*. To achieve asynchronous blockchain access, a payment network must protect the security of funds, regardless of blockchain access times.

We define the security of funds in terms of *balance security*: at any time during the payment protocol execution, each party should be able to perform a finite set of actions that eventually results in them receiving their *perceived balance* on the underlying blockchain. A party's perceived balance is their initial balance on the blockchain plus any payments received in the payment network, minus any payments made. Our design must ensure balance security regardless of how long transaction reads and writes may take.

3.2 Threat model

We assume that mutually distrusting parties use a blockchain to exchange funds and that their machines have TEEs. Parties trust their own machines, including the hardware and software, but distrust the machines of others [31]. We assume that TEEs on machines are normally trustworthy, but a subset of TEEs may suffer arbitrary integrity and confidentiality compromises. They may be compromised by other parties or external attackers who want to violate balance security (§3.1).

Parties are rational, selfish and potentially malicious, i.e., they may attempt to steal funds and deviate from the payment protocol, if it benefits them. We also assume that parties may collude with one another. Parties are connected via a network, with some behind firewalls or network address translation (NAT). Parties may drop, modify and replay messages. An attacker may delay or prevent others from accessing the blockchain for an unbounded amount of time, but we assume this cannot occur indefinitely.

3.3 Design overview

Fig. 2 shows the design of Teechain. Teechain constructs a peer-to-peer payment network in which parties operate Teechain *nodes*, e.g., node n_A is operated by party A. Each node comprises: (i) an API for parties to interact with the payment network; (ii) an interface through which to read and write blockchain transactions; and (iii) a TEE-protected *treasury* that securely holds and manages parties' funds.

Treasuries ensure the faithful execution of the payment protocol. They are external to the blockchain and manage payment channels, execute payment transactions and control the access to funds. To avoid blindly trusting treasuries to behave honestly, Teechain uses TEEs to ensure the confidentiality and integrity of treasuries. By using TEEs, Teechain achieves asynchronous blockchain access because treasuries operate correctly, autonomously and protect against misbehaviour by parties without having to interact with the blockchain.

As TEE implementations may suffer from confidentiality, integrity and availability failures [12, 64, 94], Teechain avoids

Table 1. Teechain API

Teechain API	Inputs	Outputs	API Description
Deposits (§4.1)			
new_deposit	tx, $\mathsf{pub}_1 \ldots \mathsf{pub}_n$	d_id	Creates a new fund deposit (d_id) using a transaction (tx) and a set of treasury public keys
release_deposit	d_id	tx	Refunds an unassociated fund deposit (d_id) by generating and returning a transaction (tx)
approve_deposit	d_id, pub	$\top\|\bot$	Requests approval for a deposit (d_id) from a specific treasury (pub)
Payment channels (§4.2)			
new_pay_channel	pub	c_id	Creates a new payment channel (c_id) with a given treasury (pub)
associate_deposit	d_id, c_id	$\top\|\bot$	Associates an approved fund deposit (d_id) with an existing payment channel (c_id)
dissociate_deposit	d_id, c_id	$\top\|\bot$	Dissociates a previously associated fund deposit (d_id) from a payment channel (c_id)
Payments (§4.2)			
pay_channel	val, c_id	$\top\|\bot$	Pays an amount (val) to the other user in a payment channel (c_id)
pay_multihop	val, $\mathsf{c\_id}_1 \ldots \mathsf{c\_id}_n$	$\top\|\bot$	Executes a multi-hop payment of an amount (val) along a given path of payment channels
Settlement (§4.2)			
settle_channel	c_id	tx	Settles a payment channel (c_id) by generating a settlement transaction (tx)
eject_multihop	c_id	tx	Settles a payment channel (c_id) during the execution of a multi-hop payment
eject_multihop popt	c_id, popt	tx	Settles a payment channel (c_id) using a *PoPT* (popt) during a multi-hop payment
Chain replication (§5)			
assign_comm_chain	pub	$\top\|\bot$	Assigns this treasury to a committee chain by joining the last treasury (pub) in the chain

trusting individual TEEs for security. Instead, Teechain operates *committees* of treasuries: these are groups of treasuries that manage a single collection of funds together. Fig. 2 shows a committee C_A that constitutes of the treasures at nodes n_A, n_D and n_E. Within each committee, a treasury must have approval from a subset of other treasuries to make an off-chain transaction or settle a payment channel.

Tab. 1 shows the API that Teechain provides to parties. It supports (i) creating deposits (§4.1), (ii) operating payment channels (§4.2) and (iii) constructing committees (§5). Teechain generates unique identifiers for each deposit and channel, e.g., when a deposit is created (new_deposit), a unique identifier is returned as a handle to be used in subsequent API calls. Treasuries are identified through unique public keys.

To execute payments, Teechain forms *payment channels* between nodes with network connectivity. Treasuries communicate via these channels to update payment balances. Fig. 2 shows channel c_1 between A and B; and c_2 between B and C.

Multi-hop payments can be executed along payment channel paths. In Fig. 2, a payment from A to C is executed: A invokes the API to execute a multi-hop payment of \$50 along path c_1–c_2 to C (step ❶); node n_A notifies the treasuries of its committee of the upcoming balance update (step ❷); the treasuries for nodes n_A and n_B update the balances of A and B in c_1 (step ❸); and the treasuries for nodes n_B and n_C update the balances of B and C in c_2 (step ❹). The final state is that A's balance has been deducted \$50 in c_1, B's balance incremented by \$50 in c_1 and decremented by \$50 in c_2, and C's balance incremented by \$50 in c_2.

3.4 Treasuries

Treasuries generate public/private key pairs for *treasury addresses*, which are cryptocurrency addresses owned exclusively by a treasury. They are generated securely inside each TEE, and their private keys are stored in TEE memory.

Parties can send funds to these addresses in the form of fund *deposits*. A call to new_deposit from Tab. 1 creates a deposit. It requires a deposit transaction tx, which sends funds to a set of treasury addresses, identified by the treasury public keys, $\mathsf{pub}_1 \ldots \mathsf{pub}_n$. In Fig. 2, deposit d_2 sends \$500 to the treasury at node n_B. Deposits can be associated by a treasury with a payment channel, thus incrementing the balance of that party in the channel. Fig. 2 shows two deposits registered with the treasury of node n_B: d_1 of \$1000 assigned to channel c_1; d_2 of \$500 assigned to channel c_2.

Parties must verify the integrity of treasuries before trusting them with funds; Teechain uses the remote attestation support of TEEs for verification [38, 40]. A TEE (i) measures the treasury code; (ii) cryptographically signs the measurement and the treasury's public key; and (iii) provides the signed measurement and public key to the remote party. The remote party then verifies the attestation, i.e., the remote party ensures that the attestation is correctly signed by the TEE hardware and that the measurement corresponds to a known treasury implementation. Parties can thus verify that a specific treasury, identified by its public key, is operating genuine TEE hardware.

Users without a TEE-enabled node of their own can use a remote node to manage their funds through *treasury outsourcing*. For this, the party attests a remote treasury and provisions it with a secret key, giving it the same abilities as a local party. To avoid having to trust a single remote treasury, Teechain constructs committees with multiple remote treasuries (§3.5).

3.5 Committee chains

Committees are groups of treasuries that jointly manage fund deposits. For each deposit owned by a committee, a minimum number of committee members are required to sign transactions before that deposit can be spent, thus tolerating a fixed number of TEE failures. For this, Teechain uses the multi-signature support of the blockchain [88]: each fund deposit is paid into an *m-out-of-n* treasury address, where *m* treasury

signatures are required to spend the deposit. The n committee members correspond to the n public keys provided to new_deposit in Tab. 1, when the deposit is created.

All committee members must agree on the proportion of each deposit owned by the parties in a payment channel. To achieve agreement, Teechain uses a variant of *chain replication* [95], which offers strong consistency without requiring all nodes to communicate directly. This is beneficial because parties may not have direct connectivity due to network address translation (NAT) and firewalls.

With chain replication, Teechain must prevent *roll-back* and state *forking* attacks [11] in which an attacker partitions the committee members into disjoint subgroups that can settle a channel at different balances using different deposit states. Forking a committee chain in this way would allow attackers to roll-back to old payment states to steal funds.

Teechain achieves this with a new variant of chain replication called *force-freeze replication*: if any committee member fails or refuses to update to the latest agreed upon state, the replication chain is broken, freezing all nodes at the current state. This prevents future state updates and requires that all channels are settled and unused deposits released. We describe force-freeze replication in more detail in §5.

3.6 Threat analysis

Malicious parties. Teechain assumes parties are rational and selfish, i.e., parties behave in their best financial interest (§3.2). We consider two possible cases: (i) A is a malicious local party; and (ii) B is a malicious remote party. In the case of a local malicious party A, Teechain requires parties to encrypt and sign all API calls made to a local (or outsourced) treasury (Tab. 1). A only has access to their own funds but cannot affect other funds, as enforced by the local treasury.

In the case of a malicious remote party B who wishes to steal A's funds, B must either interact with the Teechain API to force a protocol deviation or drop/replay/modify messages on the network. Teechain secures funds with treasuries and uses TEEs to ensure faithful protocol execution. Treasuries use encrypted, authenticated and freshness-protected messages.

Compromised treasuries. Current TEE implementations are vulnerable to attacks, e.g., through side-channels [12, 64, 94], and Teechain assumes that treasury compromises are possible (§3.2). We consider two cases of a compromised treasury T, which wishes to attack A: (i) T_L is a local treasury that A interacts with directly (i.e., the treasury at node n_A in Fig. 2); and (ii) T_R is a remote treasury on another node in the Teechain network.

A compromised local treasury T_L cannot steal A's funds due to Teechain's m-out-of-n committees for deposits. To steal a deposit, T_L would need to compromise $m-1$ treasuries in the committee. To prevent T_L from deceiving A when interacting with the Teechain API, Teechain requires the results of API calls to be signed by all n committee treasuries, except when an API call returns a blockchain transaction, which only requires m signatures. If T_L fails to coordinate correctly with the committee, A settles channels and returns deposits.

Mitigating a compromised remote treasury T_R is similar to the local case above: committees protect deposits, and thus T_R needs to compromise $m-1$ other treasuries. Note that, although the requirement for n signatures on API calls means that T_R can force channel settlements, it does not gain financially from this. Similar to prior work [13], Teechain assumes committee members are paid fees for participation.

Global TEE compromises. To mitigate global TEE compromises, in which many treasuries are compromised simultaneously, Teechain is designed to be TEE-agnostic, thus avoiding dependencies on a single TEE implementation. This allows parties in the network to protect deposits using committees of *heterogeneous* TEEs. Under global TEE compromises, e.g., when a specific TEE vendor leaks hardware private keys or a batch of TEEs are found to be faulty, parties can lower their risk by including sufficiently heterogeneous TEEs in their committee chains.

Compromises to the attestation mechanism of a particular TEE implementation, e.g., as done by the Foreshadow [94] attack against the Intel attestation service, do not affect funds held by committees. As described in §3.4, remote attestation ensures that a specific treasury, identified by its public key, operates genuine TEE hardware. Even if remote attestation has been compromised, an attacker can only create new malicious treasuries, but cannot spoof other treasuries or committee members in the network. To steal funds, an attacker would need to bias the selection of future committee members. We discuss committee member selection in §5.2.

4 Payment Protocol

This section describes Teechain's deposit allocation (§4.1), its payment channel protocol (§4.2), its multi-hop payment protocol (§4.3), and sketches their security proofs (§4.4).

4.1 Allocating dynamic deposits

Deposits can be created at any time and associated/dissociated with payment channels dynamically. Alg. 1 shows the protocol executed by treasuries for the API calls from Tab. 1.

To construct a new deposit d, parties invoke new_deposit (Alg. 1, line 1) and present a deposit transaction tx and the list of treasury public keys forming the committee that tx sends funds to. The treasury then verifies that tx sends funds to an m-out-of-n multi-signature address using the committee members' public keys, $\mathsf{pub}_1 \ldots \mathsf{pub}_n$, and notifies the committee of the new tx (see §5). The treasury then constructs a new deposit d, forwards tx to the blockchain, and returns d's unique identifier to the requester (line 6), signed by all committee members—we omit signature collection for brevity.

A payment channel c may contain zero or more deposits through deposit association. The sum of the deposits associated with c must be equal to the sum of the balances of A

Algorithm 1: Teechain payment protocol executed by the treasury at each node (Based on the API shown in Table 1. For brevity, we omit the collection of committee member signatures at the end of each API call (see §3.6).)

```
1  on new_deposit(tx,
     pub_1...pub_n):
2    verify_tx(tx, pub_1...pub_n)
3    d ← create_new_deposit(tx)
4    deposits[d.id] ← d /* store dep */
5    write_to_blockchain(tx)
6    return d.id /* return deposit id */
7  on release_deposit(d_id):
8    d ← deposits[d_id]
9    assert(d.chan = ∅) /* unassoc */
10   tx ← gen_deposit_refund(d)
11   deposits[d.id] ← ∅ /* clear dep */
12   write_to_blockchain(tx)
13   return tx /* return refund */
14 on approve_deposit(d_id, pub):
15   d ← deposits[d_id]
16   apprv ← ask_approve_remote(d,
       pub)
17   d.apprv[pub] ← apprv
18   return apprv /* return approval */
19 on new_pay_channel(pub):
20   c ← create_channel_with(pub)
21   (c.my_bal, c.rem_bal) ← (0, 0)
22   channels[c.id] ← c
23   return c.id /* return channel id */
24 on associate_deposit(d_id,
     c_id):
25   d ← deposits[d_id]
26   c ← channels[c_id]
27   assert(d.chan = ∅) /* unassoc */
28   assert(d.apprv[c.pub])
29   d.chan ← c /* add assoc */
30   c.my_bal ← c.my_bal + d.val
31   send_assoc_to_remote(d, c)
32 on dissociate_deposit(d_id,
     c_id):
33   d ← deposits[d_id]
34   c ← channels[c_id]
35   assert(d.chan = c)
36   send_dissoc_to_remote(d, c)
37   d.chan ← ∅ /* remove assoc */
38   c.my_bal ← c.my_bal – d.val
39 on pay_channel(val, c_id):
40   c ← channels[c_id]
41   assert(c.my_bal ≥ val)
42   c.my_bal ← c.my_bal – val
43   c.rem_bal ← c.rem_bal + val
44   send_pay_to_remote(c, val)
45 on settle_channel(c_id):
46   c ← channels[c_id]
47   if neutral_balance(c) then
48     /* terminate off-chain */
49     dissociate_all_deposits(c);
50     channels[c.id] ← ∅
51     return ∅
52   else
53     /* terminate on-chain */
54     tx ← get_settle_for_bals(c)
55     send_settle_to_remote(c, tx)
56     channels[c.id] ← ∅
57     write_to_blockchain(tx)
58     return tx
59 on pay_multihop(val,
     c_id_1...c_id_n):
60   c_1 ← channels[c_id_1]
61   . . .
62   c_n ← channels[c_id_n]
63   lock(val, c_1, . . . , c_n) /* Alg.2 */
64   wait_for_unlock()
65   return ⊤ /* payment done */
66 on eject_multihop(c_id):
67   c ← channels[c_id]
68   s ← c.state
69   if s = lock ∨ s = sign ∨
       s = postpayment ∨
       s = unlock then
70     return settle_channel (c_id)
71   return c.τ /* settle all */
72 on eject_multihop(c_id, popt):
73   s ← popt.state
74   if s = lock ∨ s = sign then
75     return settle_prepay(c_id)
76   if s = postpayment ∨
       s = unlock then
77     return settle_postpay(c_id)
```

and B in c, i.e., deposits are distributed to A and B. Before a deposit d can be associated with c, it must be approved by the remote party in c (e.g., B if A requests approval) using approve_deposit (line 14). Approval contacts the remote party via its treasury and queries if the deposit is eligible for association with c. Deposit approval therefore allows A and B to define which deposits can be associated with c. Due to our assumption of asynchronous blockchain access, this may take unbounded time. Deposits need to be approved only once.

Approved deposits can be *associated* with a single channel using associate_deposit, and *dissociated* using dissociate_deposit (lines 24 and 32). When deposit d is associated with c by A, the treasuries increase A's balance by the deposit amount (lines 30 and 31); dissociation decrements A's balance (lines 36 and 38). Disassociation can only be done if the participant's balance is greater than or equal to the deposit amount. *Unassociated* deposits are deposits not associated with any channel. They can be returned upon request through release_deposit (line 7): a new transaction tx is generated and signed by the appropriate committee treasuries, and written to the blockchain; d is then removed from the treasury.

4.2 Using payment channels

To create payment channels between treasuries without blockchain interaction, participants call new_pay_channel and provide the public key of the treasury with which to create the channel (Alg. 1, line 19). The two treasuries then establish a secure communication channel using authenticated Diffie-Hellman [49] for key provisioning and remote attestation. Using the secure channel, the treasuries assign a unique channel identifier to the channel c, initialize both participant's balances to 0, and return the channel identifier.

To execute a payment along a channel, the sender calls pay_channel (Alg. 6), which specifies the amount to send and the channel identifier. The sender's treasury first ensures that the sender has sufficient funds before decrementing the sender's balance and incrementing the recipient's balance locally (lines 42 and 43). It then forwards the payment to the recipient's treasury to update balances. If the payment is not received by the recipient, e.g., due to a network failure, the sender settles the channel and writes the balances to the blockchain to allow the remote party to see the final state of the channel. This prevents balance inconsistencies.

As deposits can only be associated with a single channel, participants may suffer from *deposit lock-in*: when a large deposit is added to a channel but only a small fraction is spent, it leaves the remaining locked-in until the channel is settled. In a channel c with deposit d_x of amount a_x, after payments of value p_x have been made, the locked-in funds f_x are $a_x - p_x$. If f_x is large, there is a high fund lock-in. To avoid this, participants can perform *deposit rebalancing*: they associate another deposit d_y of value v_y, where $v_x > v_y \geq p_x$, with c and dissociate d_x from c. This reduces the lock-in.

At any time, either party may settle the channel using settle_channel (line 45). If the balances of the parties are *neutral*, i.e., equivalent to their deposits as if no payments were made, the treasuries can terminate the channel off-chain by simply disassociating all deposits from the channel. Off-chain termination avoids writing a settlement transaction to the blockchain (see §6.4); otherwise, the local treasury generates a blockchain transaction tx using the deposits and balances in the channel, collects signatures from the committee members, and writes tx to the blockchain.

Algorithm 2: Teechain multi-hop payment protocol (For brevity, we omit the messages exchanged between treasuries after each step, i.e., the messages in Fig. 3. Payment channels in the path are denoted: $c_1 \dots c_n$. Treasuries in the path are numbered $1 \dots n+1$. pos denotes a treasuries' position.)

```
1  on ❶ lock(val, c_1, ..., c_n):
2  |  if pos ≤ n then
3  |  |  assert(c_pos.my_bal ≥ val)
4  |  |  lock_channel(c_pos, val)
5  |  if pos > 1 then
6  |  |  lock_channel(c_pos−1, val)
7  on lock_channel(c, val):
8  |  (c.state, c.val) ← (lock, val)
9  on ❷ sign(τ̄, c_1, ..., c_n):
10 |  if pos > 1 then
11 |  |  sign_channel(c_pos−1, τ̄)
12 |  if pos ≤ n then
13 |  |  sign_channel(c_pos, τ̄)
14 on sign_channel(c, τ̄):
15 |  τ̄ ← add_chan_settle_post(τ̄, c)
16 |  c.state ← sign
17 on ❸ pre(τ̄, c_1, ..., c_n):
18 |  if pos ≤ n then
19 |  |  pre_channel(c_pos, τ̄)
20 |  if i > 1 then
21 |  |  pre_channel(c_pos−1, τ̄)
22 on pre_channel(c, τ̄):
23 |  c.τ̄ ← τ̄ /* store τ̄ for if settle */
24 |  c.state ← prepayment
25 on ❹ inter(c_1, ..., c_n):
26 |  if pos > 1 then
27 |  |  increase_my_bal(c_pos−1)
28 |  if pos ≤ n then
29 |  |  decrease_my_bal(c_pos)
30 on increase_my_bal(c):
31 |  c.my_bal ← c.my_bal + c.val
32 |  c.rem_bal ← c.rem_bal − c.val
33 |  c.state ← inter
34 on decrease_my_bal(c):
35 |  c.my_bal ← c.my_bal − c.val
36 |  c.rem_bal ← c.rem_bal + c.val
37 |  c.state ← inter
38 on ❺ post(c_1, ..., c_n):
39 |  if pos ≤ n then
40 |  |  post_channel(c_pos)
41 |  if pos > 1 then
42 |  |  post_channel(c_pos−1)
43 on post_channel(c):
44 |  c.τ̄ ← ∅ /* not needed */
45 |  c.state ← postpayment
46 on ❻ unlock(c_1, ..., c_n):
47 |  if pos > 1 then
48 |  |  c_pos−1.state ← idle
49 |  if pos ≤ n then
50 |  |  c_pos.state ← idle
```

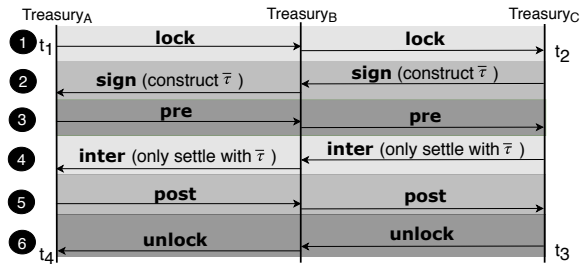

Figure 3. Protocol for multi-hop payments

4.3 Executing multi-hop payments

To do a multi-hop payment across multiple payment channels, parties invoke pay_multihop (Alg. 1, line 59) with the payment amount and the channel identifiers of the path.[2] All channels in the path must update their balances atomically otherwise intermediaries could lose funds. For example, B in Fig. 2 retains the same total funds post-payment, i.e., their balance is incremented by \$50 in c_1 and decremented by \$50 in c_2; if c_1 is not updated and only c_2 updates, B pays C personally.

One approach to ensure atomic channel updates is to freeze channels by preventing parties from settling them until the multi-hop payment completes. This has the problem that if a failure occurs along the path, channels are frozen indefinitely. To overcome this, Teechain allows parties to settle channels even if a multi-hop payment is being executed. Teechain achieves this using a *proof of premature termination* (*PoPT*). When a party prematurely settles a channel c during a multi-hop payment, the settlement transaction tx can be used by other parties in the path to determine the state s of settlement: c was either settled pre-payment (s = pre), i.e., before the payment has occurred, or post-payment (s = post), i.e., after the payment has occurred. The parties can present tx to their treasuries as a *PoPT* to settle all channels in the same state s.

Teechain enforces that settlement transactions in state pre will *conflict* (§2.1) with those in state post. If a channel in the path is terminated prematurely using eject_multihop (Alg. 1, line 66), the first settlement transaction tx to be written to the blockchain determines the state at which all channels are settled. If a channel in a different state tries to settle afterwards, the transaction is rejected by the blockchain. The party can present tx to its treasuries as *PoPT* through eject_multihop (Alg. 1, line 72), which generates a settlement transaction without conflict. Conflicts prevent Teechain from assuming how long settlements take to be written to the blockchain.

For blockchains with expressive transactions [89], smart contracts can be used to ensure conflicts between settlement transactions in different states. Channels in a multi-hop payment can simply transition from pre to post in a single step.

For other blockchains, e.g., Bitcoin, Teechain must enforce transaction conflicts manually. Teechain constructs an intermediate *path settlement transaction* $\overline{\tau}$ that settles all channels in state post using a single blockchain transaction. $\overline{\tau}$ conflicts with individual settlement transactions in pre and post because it spends the same deposits. Teechain uses $\overline{\tau}$ to transition channels from state pre to post by moving to an intermediate state inter between the transition first. Channels in state inter can only settle using $\overline{\tau}$. If a party decides to settle a channel while it is in state inter, they settle all channels in the path. Therefore, during the transition from pre to inter, either the first channel settlement transaction tx written to the blockchain is in pre, in which case $\overline{\tau}$ cannot be written to the blockchain and all channels settle at pre by presenting tx as *PoPT*; or tx is $\overline{\tau}$, in which case all channels are settled in post using $\overline{\tau}$. The transition from inter to post is analogous.

Payment execution. Fig. 3 shows the messages exchanged by the treasuries when A executes a multi-hop payment to C via B. Alg. 2 shows the corresponding protocol steps.

Teechain requires three network round trips to complete the payment: step ❶ locks the channel and ensures sufficient balances (Alg. 2, line 1); step ❷ constructs $\overline{\tau}$ with all treasuries writing their channel balances and signatures (line 9); Teechain then updates the channel balances from pre (❸,

[2] We assume that participants determine paths in Teechain out-of-band.

line 17) to inter (❹, line 25) to post (❺, line 38) payment state; and finally, step ❻ unlocks the channels (line 46).

As multi-hop payments lock channels, this prevents concurrent payments. Teechain therefore dynamically constructs new channels for concurrent payments using unassociated deposits, as needed. This is feasible because Teechain can create channels and assign deposits with low latency. Teechain coalesces no longer needed payment channels by: (i) executing multi-hop payments in a cycle along the channels until they are at a neutral balance; and (ii) terminating the channels off-chain through deposit dissociation (see §4.2). We evaluate dynamic channel construction in §6.3.

4.4 Payment protocol security

Teechain's payment protocol (Algs. 1 and 2) achieves balance security (§3.1) under asynchronous blockchain access, i.e., parties can always receive their funds on the blockchain, regardless of blockchain access times or other parties' actions. We sketch a proof below, and the full details are in Appendix A. We first show that Teechain achieves asynchronous blockchain access, and then prove balance security.

When Teechain writes to (new_deposit, release_deposit, settle_channel, eject_multihop) or reads from the blockchain (approve_deposit), the protocol makes no assumption about the duration of these operations. For example, when ejecting from a multi-hop payment prematurely (eject_multihop), Teechain uses the first settlement transaction written to the blockchain to determine the state at which all channels in a payment path are settled (§4.3). By considering all blockchain interactions on a case-by-case basis (Algs. 1 and 2), we can see Teechain operates with asynchronous blockchain access. **Payment channel security.** We now prove that Teechain achieves balance security using the Universal Composability (UC) framework [14]. Our definition of balance security (§3.1) under UC is similar to prior work [26, 56, 62]. We model committees as a single treasury executing the protocol.

Under UC, we consider a *real* world, in which parties run the Teechain protocol $\pi_{Teechain}$ (Alg. 1), and an *ideal* world, in which parties interact with an *ideal functionality*, $\mathcal{F}_{Teechain}$, a trusted third party that implements Teechain's API (Alg. 1). Adversarial behavior is introduced in the ideal world by a simulator $\mathcal{S}$ with appropriate adversarial abilities (§3.2).

To prove that Teechain achieves balance security, we show that (i) the real and ideal worlds are indistinguishable to an external observer $\mathcal{E}$. This implies that any attack violating balance security in the real world is also possible in the ideal one; and (ii) $\mathcal{F}_{Teechain}$ achieves balance security in the ideal world. This proves that $\pi_{Teechain}$ also achieves balance security.

We prove indistinguishability between the real and ideal worlds through a series of five *hybrid steps*, starting at the real world H_0, and ending in the ideal world H_5. In each step, a key element is changed and indistinguishability is proven. As commonly done [7, 92], in H_0, the desired behavior of TEEs and the blockchain are replaced by two ideal functionalities, $\mathcal{F}_{TEE}$ and $\mathcal{F}_B$, respectively (defined in [68, 71]). In H_1 and H_2, $\mathcal{S}$ simulates $\mathcal{F}_{TEE}$ and $\mathcal{F}_B$, respectively, and in H_3 and H_4, incorrectly signed messages to $\mathcal{F}_{TEE}$ and $\mathcal{F}_B$, are dropped, to tolerate attacks on the signing schemes. Finally, in H_5, we prove equivalence between $\pi_{Teechain}$ and $\mathcal{F}_{Teechain}$ to $\mathcal{E}$. Next, we prove that $\mathcal{F}_{Teechain}$ achieves balance security by showing that a party can always eventually place transactions on the blockchain that grant it an amount equal to its perceived balance. This is done by ordering $\mathcal{F}_{Teechain}$ to create transactions that close all open channels, remove all unassociated deposits, and place them on the blockchain. Since Teechain does not make blockchain timing assumptions, denial-of-service attacks [33, 57]), do not violate balance security.

Multi-hop payment security. We show that the multi-hop protocol also maintains balance security. As shown in Fig. 3, consider a payment from A to C via B of amount val at the following times: A begins step lock of the protocol at t_1; at $t_2 > t_1$, C begins step lock; at $t_3 > t_2$: C completes step unlock; and, at $t_4 > t_3$, A completes the protocol with unlock.

For A, the perceived balance for the channel is: before t_1 as if val was not paid; after t_4, as if val was paid; between t_1 and t_4 either is acceptable. For C, the same as A but t_1 replaced with t_2, and t_4 with t_3. The perceived balance of the intermediate B is not affected. A considers the payment complete *iff* C considers it complete; funds are not lost or created.

We show that A and C can unilaterally reclaim their perceived balance. Note that single channel payments do not interfere with multi-hop payments, because all channels are locked (§4.3). At any point, A and C can settle the channels in either the pre- or post-payment states, either with single settlement transactions or using $\overline{\tau}$ (see Alg. 2). For example, if a node is in state lock, the others are either in unlock or lock or in lock or sign. In all cases, if a node settles, the rest of the nodes can only settle in the same state (pre- or post-payment), in accordance with balance security.

5 Committee Chains

This section describes force-freeze replication in committee chains (§5.1), committee configurations (§5.2), and persistent storage for committee members (§5.3).

5.1 Force-freeze replication

To maintain consensus among committee members, Teechain uses *force-freeze* replication, a new variant of chain replication [95]. The nodes form a chain, with the primary at the head, and the last backup at the tail. On an update, the primary propagates the update down the chain. Each node forwards the update to its backup, and waits for an acknowledgement before executing the update. When the primary receives an acknowledgement, the entire chain has updated. This provides strong consistency among the nodes.

Traditional chain replication [95] continues to execute state updates even after nodes have failed to update. Applying this naively to treasuries in a committee, would make Teechain

Algorithm 3: Force-freeze chain replication (For brevity, we omit message encryption, authentication and freshness.)

```
1  on assign_comm_chain(pub):
2      assert(pred = ∅) /* no chain */
3      attest_and_auth_DH(pub)
4      pred ← pub /* set chain pred */
5      send(addTail) to (pub)
6      wait_for(update, s) from (pub)
7      return ⊤
8  on receive(addTail) from (pub):
9      assert(succ = ∅) /* current tail */
10     attest_and_auth_DH(pub)
11     succ ← pub
12     send(update, curr_state) to
         (pub)
13 on receive(update, s) from
     (pub):
14     assert(pred = pub)
15     if succ = ∅ then
16         update_state_to(s)
17         ack ← create_signed_ack()
18     else
19         ack ← send(update, s) to
             (succ)
20         if fail_or_invalid(ack) then
21             freeze() /* can't update */
22         else
23             update_state_to(s)
24             ack ← sign_ack(ack)
25     send(ack) to pub
```

vulnerable to roll-back and state forking attacks (§3.5). Instead, in *force-freeze replication* (Alg. 3), if a node receives an update request (line 13) and it or its successor fails to update, the chain is frozen at its current state (line 21). All channels must now settle and release unused deposits.

Parties construct force-freeze replication chains using assign_comm_chain (line 1), which assigns a treasury to the end of the chain: a party provides the public key of the node at the tail of the chain. To secure state updates along the chain, nodes construct secure communication channels (lines 3 and 10).

To prevent malicious treasuries from executing denial-of-service attacks by freezing committee chains through forced failures, Teechain employs incentives for committee members: parties are assumed to be financially rational (§3.6), and committee members are paid fees for participation. If a committee member forces a freeze, it loses any participation fees that it has accumulated in that committee.

Unlike other replication protocols, e.g., Paxos [50] and PBFT [16], force-freeze replication uses a chain communication topology and therefore does not not require full network connectivity, which is impractical in peer-to-peer networks. Other consensus protocols may enhance liveness, but this comes at the cost of increased network communication. It also increases protocol complexity—a benefit of force-freeze replication is that it is simple to implement and reason about.

5.2 Committee chain configurations

To ensure balance security (§3.1) despite compromised treasuries, Teechain uses committees chains of size n for each deposit, and requires at least m treasuries in a committee to sign a blockchain transaction before that deposit can be spent. To violate balance security, an attacker must compromise at least m treasuries in a committee, or cause $(n - m) + 1$ treasuries to fail, e.g., crash or stop responding.

The values of m and n affect security: (i) 1-out-of-1 deposits provide no fault tolerance against crash failures or compromises; (ii) 1-out-of-n committee chains provide crash fault tolerance for treasuries but do not tolerate their compromises; and (iii) in the general case, as m increases, more signatures are appended to each transaction, impacting their size. We explore this trade-off in §6.4.

As deposits must be approved before association with payment channels (§3.4), parties can choose the values of n and m for their deposits and channels. For small deposits, a 1-out-of-1 committee chain may be sufficient as there is little loss if a failure occurs; for medium deposits, 1-out-of-n may be desirable to tolerate crash failures; and for large deposits, e.g., 2-out-of-3 committee chains are required to tolerate attacks. Larger committees, e.g., with more than five members, may only be required for high-value deposits.

To prevent an attacker from biasing the selection of committee members, parties select the committee treasuries themselves on deposit creation (new_deposit, Tab. 1). Selection criteria may include treasury reputation, trusted TEE vendors and implementations, blacklisted treasuries, and TEE heterogeneity. To avoid Sybil attacks [21], Teechain can leverage several techniques, such as requiring treasuries to provide a proof-of-stake [8], operate in a permissioned setting [4], or use a reputation system.

Payment channels may contain multiple deposits, each with a separate committee chain. These chains do not have to be updated atomically: for payments that span multiple deposits, the committee chains must be identical, and thus the state updates can be batched. If a large payment spans deposits of multiple committee chains, the payment is broken down into smaller payments, only affecting one deposit at a time. Having many deposits, each with distinct committee members, affects performance (see §6.1).

5.3 Committee chains with secure persistent storage

In addition to committee chains, Teechain also supports the optional use of secure persistent storage for crash fault tolerance. After a failure, a treasury can reload its state, settle channels and return deposits. To overcome roll-back attacks, state freshness must be guaranteed by the TEE hardware [3], e.g., through hardware monotonic counters [39].

Current Intel SGX implementations throttle accesses to hardware monotonic counters to tens of increments per second [58, 82], which limits performance. As a mitigation, Teechain batches transactions at the client side, similar to other payment networks [73] that merge payments from the same sender/recipient pairs. Current SGX implementations also limit the number of writes for hardware counters to 1 million [58]. For the majority of parties in Teechain, this should be high enough. When the limit is reached, Teechain forces treasuries to settle channels and return deposits.

6 Evaluation

We explore the performance of payment channels (§6.1), multi-hop payments (§6.2), payment networks (§6.3), and blockchain storage costs (§6.4).

Table 2. Channel performance

Payment channel	Throughput (tx/sec)	Latency (ms)	[99th %]
Lightning Network	1,000	387	[420]
Teechain			
n = 1	130,311	86	[93]
n = 2 (IL)	34,115	292	[301]
n = 3 (IL, UK)	33,180	415	[432]
n = 4 (IL, US, UK)	33,178	672	[691]
n = 1 (batching)	150,311	191	[196]
n = 3 (batching)	135,331	516	[530]
n = 3 (outsourced)	33,178	483	[494]
Persistent storage			
n = 1	10	288	[294]
n = 1 (batching)	145,786	401	[408]

Table 3. API performance

API operation	Latency (ms) [99th %] Local		Outsourced	
Lightning Network				
new_pay_channel	60 min.	[N/A]		
Teechain				
new_pay_channel	2,810	[4,205]	4,322	[5,201]
assign_comm_chain	2,765	[3,910]	2,852	[3,993]
associate_deposit				
n = 1	101	[110]		
n = 2 (IL)	289	[297]		
n = 3 (IL & UK)	422	[429]	489	[514]
n = 4 (IL, US & UK)	677	[681]		
Persistent storage	302	[309]		

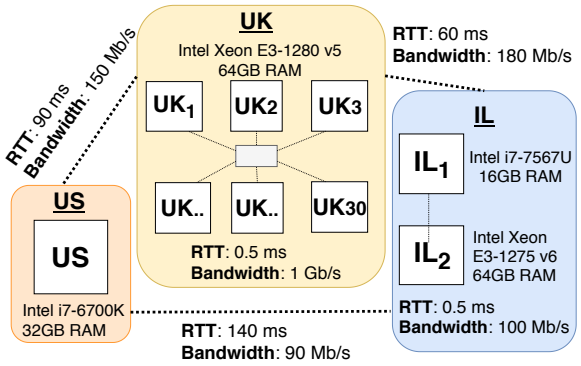

Figure 4. Evaluation setup

We implement Teechain using Intel SGX for the Bitcoin blockchain. We use the Linux Intel SGX SDK version 2.1 [36] and a subset of Bitcoin core [87]. A release of our implementation is available at: https://teechain.network. Teechain consists of 20,000 lines of C/C++ code inside the TEE, and 65,000 lines of untrusted code. As the Linux SGX SDK does not support monotonic counters on all hardware [36], we emulate them with a delay of 100 ms [58, 82].

Our implementation is hardened against side-channel attacks. Although TEE compromises are mitigated by committee chains (§5), Teechain uses timing and memory-access side-channel resistant libraries for sensitive data: (i) secp256k1, a constant time and memory library for elliptic curve operations [86]; (ii) a side-channel resistant implementation of Elliptic-Curve Diffie-Hellman [90]; and (iii) AES-GCM using AES-NI [90], immune to software side channels [36].

To measure performance, we define throughput as the number of transactions sent per second, and latency as the time from when a payment is issued until an acknowledgement is received. At the time of writing, the only payment network with a public implementation is the Lightning Network (LN) [73]. We compare Teechain against the Lightning Network Daemon (LND) [52]. Both Teechain and LN can optionally batch transactions at the client side, merging multiple payments into a single payment with increased latency.

6.1 Performance of payment channels

We want to answer three questions: (i) what is the throughput of a payment channel? (ii) how do committee chains affect performance? (iii) what is the benefit of transaction batching?

We deploy Teechain on 33 SGX-capable machines in the UK, the US and Israel. Fig. 4 shows the network topology and hardware set-up. We construct a payment channel between *US* and UK_1. To evaluate treasury outsourcing, IL_1 acts as a non-SGX client using *US* as a remote treasury.

In all experiments, committee chains have the same length, as the performance is bound by the slowest party. We vary n for m-out-of-n committee chains. Note that m does not affect channel throughput because all n committee members must replicate the state regardless. When batching transactions, we batch for 100 ms before sending a transaction. Teechain requires one round-trip for a payment, while LN requires two [73]. Teechain can pipeline payments but LN only supports sequential transactions and must batch by default.

Tab. 2 shows the observed throughput and latency. LN achieves a maximum throughput of 1,000 tx/sec with a latency of 387 ms (99th percentile at 420 ms). With a committee chain of n=1, Teechain has two orders of magnitude higher throughput with a latency of 86 ms (no batching). With n=2 (i.e., an extra committee member in Israel), the throughput of Teechain is 34× compared to LN, with similar latencies. Adding more members to each party's committee chain only increases latency, and throughput remains unchanged. Using persistent storage, performance is capped by the TEE hardware counters, resulting in 10 tx/sec, which can be overcome by transaction batching. Teechain achieves between 135–150× better performance than LN when batching.

Tab. 3 shows the performance of different API calls. LN channel creation takes approx. 60 mins, as a transaction must be placed onto the blockchain and confirmation takes 6 Bitcoin blocks. Since Teechain decouples channels and deposits, channel creation takes only 2.8 secs; we assume deposits are already on the blockchain. Creation of an outsourced payment channel takes 4.3 secs, as the client (IL_1) must also verify the

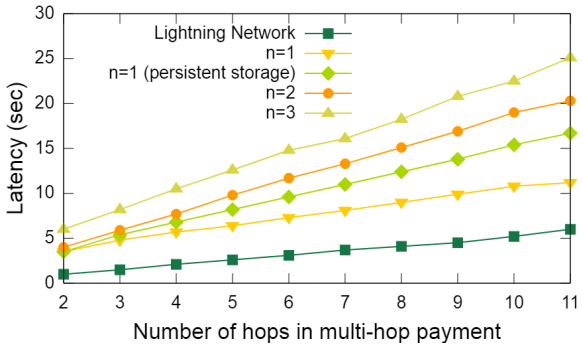

Figure 5. Multi-hop performance

integrity of the outsourced treasury (*US*). Adding new members to a committee chain incurs similar latencies as channel creation. The latency for associating deposits depends on the committee length n, and dissociation is similar.

In summary, channel throughput is affected by committee chains: (i) without batching, committee chains with n=1 achieve the best performance, and persistent storage performs worst due to hardware counters; (ii) with batching, Teechain achieves higher throughput for committee chains and persistent storage hides the delay for counters. The latency depends on the network, committee length and batching delay.

6.2 Performance of multi-hop payments

Next we evaluate the performance of multi-hop payments and investigate: (i) how does latency increase with the number of hops in a payment path? and (ii) how do committee chains affect multi-hop performance?

For our experiments, we limit the maximum number of hops in a payment path to 11, as longer payment paths are unlikely to be seen in practice. Recent work [77] studying LN shows that the average number of hops between two parties is approximately 3.

We construct the 11 payment channels, all of which are transatlantic in the topology from Fig. 4. We send transactions along the path $UK \rightarrow US \rightarrow IL \rightarrow UK$. For *UK* and *IL*, we split the payment channels equally between the machines to spread load. All experiments use the same payment channels and committee chains of the same length. Committee members are deployed in different failure domains.

We measure the latency of multi-hop payments. We vary the number of hops and the number of committee members per committee chain for each node. Fig. 5 shows that LN scales linearly with chain length, taking 1 sec to complete a payment across 2 hops (2 channels) and 6 secs for 11 hops. Teechain also scales linearly but with a different slope: with n=1, the latency is about 2× that of LN; using n=3, payments across 2 hops take 5 secs; payments across 11 hops take 26 secs. The 3–4× overhead compared to LN is a due to the extra network round trips for multi-hop payments.

To update all channels in a multi-hop payment consistently, both Teechain and LN do not pipeline payments. Therefore, throughput is 1/latency. Teechain and LN batch transactions: throughput becomes the batch size divided by the latency to complete the payment. We compare the throughput for Teechain and LN, with each Teechain node using committee chains of n=3. Teechain batches 135,000 tx/sec; LN batches 1,000 tx/sec (see §6.1). With this, the throughput of Teechain for 2 hops is 14,062 tx/sec, and for 11 hops is 3,649 tx/sec. For LN, throughput for 2 hops is 862 tx/sec, and 139 tx/sec for 11 hops. Teechain thus outperforms LN by 16–26×.

In summary, Teechain requires three network round trips to complete a payment, while LN requires only 1.5. Teechain must synchronize nodes off-chain with extra messages to support asynchronous blockchain access. In addition, Teechain is network-bound: chain replication increases latency.

6.3 Performance of payment networks

We evaluate the performance of a complete Teechain payment network and investigate how its throughput is affected by (i) the network topology and (ii) the committee chains.

We use 30 machines located in the UK (see Fig. 4). As there exist no public micro-payment datasets, we use the transactions from the Bitcoin blockchain. We filter out transactions that we cannot replay, such as those that spend to/from multi-signature addresses. For transactions with multiple inputs and outputs, we choose only one. The resulting dataset has over 150 million payments from a source to a recipient address.

We construct two payment network topologies: (i) a *complete* graph, in which all node pairs have direct payment channels; and (ii) a *hub-and-spoke* topology (see Fig. 6), in which the nodes are connected with 3 tiers of connectivity: tier 1 nodes have the highest connectivity and tier 3 nodes the lowest. We emulate wide-area network links by adding 100 ms latency between machines.

To execute payments, we assign Bitcoin addresses to the machines. For the complete graph, we randomly and evenly assign Bitcoin addresses; for the hub-and-spoke graph, we distribute the addresses in a skewed fashion, with larger nodes being assigned more addresses than smaller nodes (50% of addresses to tier 1 nodes, 35% to tier 2, and 15% to tier 3). For each graph deployment, we compare the throughput with differently-sized committee chains, for n=1 to n=3 committee members per deposit. We vary the number of nodes in the deployment from 5 to 30 machines.

Fig. 7 shows the throughput for the complete graph topology. For all committee chain lengths, throughput scales linearly with the node number. Committee chains of length $n = 1$ perform best (2.2 million tx/sec with 30 machines); committee chains with $n > 1$ limit throughput (1 million tx/sec). There is little difference (9%) between $n = 2$ committee members and $n = 3$; throughput is bottlenecked by the time to replicate state.

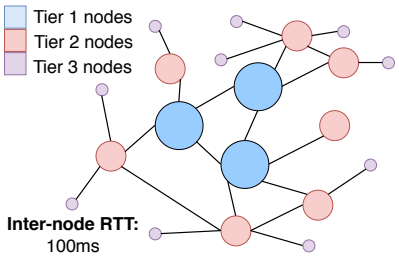

Figure 6. Hub-and-spoke network topology

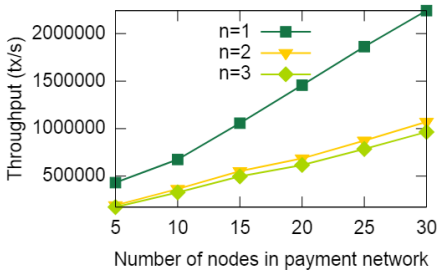

Figure 7. Throughput for complete topology

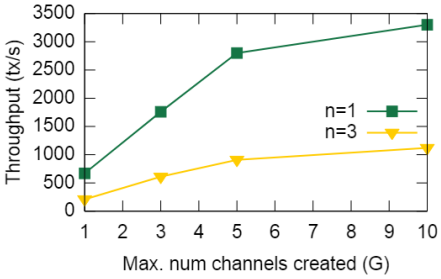

Figure 8. Throughput for hub-and-spoke topology

Next we consider the hub-and-spoke graph topology. Multi-hop payments use the shortest path—if there are multiple paths, only one is chosen. As multi-hop payments lock channels during execution, payments compete with one another. To overcome this, Teechain uses dynamic channel creation to allow concurrent payments between endpoints (see §4.3).

Fig. 8 shows how the throughput increases as intermediate nodes (i.e., tier 1 and 2) are permitted to create more dynamic channels. Without dynamic channels, i.e., $G = 1$, with $n = 3$ committee chains, the network achieves around 210 tx/sec, with an average latency of 720 ms. With $G > 1$, throughput scales almost linearly with the number of channels, for both $n = 1$ and $n = 3$. We obtain diminishing returns as G increases further because tier 3 nodes become congested.

In summary, payment throughput is lower in the hub-and-spoke topology compared to the complete topology by several orders of magnitude. This is a result of locking channels for multi-hop payments: while dynamic channel creation alleviates contention, best performance requires high connectivity.

Given that Teechain and LN exhibit different performance for single and multi-hop payments, any in-depth comparison requires careful treatment of many aspects, including the employed payment routing algorithm, the choice of transaction batching interval in LN, the number of dynamic channels created in Teechain, and the used contention avoidance algorithm [56]. We defer more experiments to future work.

6.4 Blockchain cost

We evaluate and compare: (i) the number of transactions placed on the blockchain; and (ii) the blockchain cost. We define the *blockchain cost* as the amount of data placed on the blockchain to open and close a payment channel. Unlike existing solutions, Teechain can assign multiple deposits to a single channel. For a fair comparison, we assume at most 2 deposits per channel, and we abstract from particular blockchains by counting the pairs of public keys and signatures [13]. We compare with the Lightning Network (LN) [73], Duplex Micropayment Channels (DMC) [20] and Scalable Funding of Micropayment Channels (SFMC) [13].

Tab. 4 shows the number of transactions and the blockchain cost. For all solutions but LN, the cost is higher if one party unilaterally closes the channel. For DMC, the number of transactions required ranges from 2 to $3 + d$, where $d \geq 1$ defines the DMC transaction chain length. In LN, 4 transactions must be placed onto the blockchain, which result in a cost of 6 across bilateral and unilateral termination. For SFMC, the number of transactions ranges from $2/n$ to $(1 + i)/n + (3 + d)$, where n is the total number of constructed payment channels and i and d define the funding and transaction chain's lengths. respectively. Since each SFMC transaction requires p signatures and is shared between the n payment channels, the blockchain cost is $2p/n$ if all parties collaboratively close payment channels; $(1 + i)(p/n) + 2(3 + d)$ if closed unilaterally.

Table 4. Number of transactions and blockchain costs

Payment channel	Bilateral		Unilateral	
	No. txs	Cost	No. txs	Cost
LN [73]	4	6	4	6
DMC [20]	2	4	$3 + d$	$2(3 + d)$
SFMC [13]	$2/n$	$2p/n$	$(1 + i)/n$ $+(3 + d)$	$(1 + i)(p/n)$ $+2(3 + d)$
Teechain	1	$1 + (n/2)$	3	$2 + (n_1/2) + (n_2/2)$ $+m_1 + m_2$

Teechain constructs funding deposits using m-out-of-n transactions. If the channel has a single deposit and is settled off-chain, only one transaction is required, with a cost of $1 + (n/2)$, i.e., the cost of a signature and public key to spend funds into the treasury address, and n public keys for committee members; otherwise, with 2 deposits assigned to a channel, Teechain requires 3 transactions, with the cost including the two funding deposits and the settlement transaction.

We observe that, with a 2-out-of-3 multi-signature for each funding deposit, Teechain places 25%–75% fewer transactions on the blockchain than LN, and is up to 58% more efficient for bilateral termination. For DMC and bilateral closures, Teechain places 50% fewer transactions and 37% less data on the blockchain than DMC. While Teechain outperforms SMFC when closing channels unilaterally, SMFC uses fewer transactions under bilateral closure if $n = 1$ and $p/n > 1$. SFMC amortises transactions across multiple parties

and channels at the cost of having to trust all involved parties. Teechain does not make this assumption.

7 Related Work

Payment channels and networks. Unilateral payment channels were first discussed in [32]. Duplex Micropayment Channels [20] use time-locked transactions to prevent old channel states from being published. Lightning Network (LN) [73] supports multi-hop payments but requires users to monitor the blockchain. Pisa [60] builds on LN and allows third parties to monitor the blockchain on behalf of other users. REVIVE [46] rebalances payment channels, but locks the funds during the rebalancing process. Sprites [62] can add and remove funds to channels dynamically, but requires smart-contracts [84]. State channels [26, 61] is a generalization of payment networks, but also requires smart-contracts. Fulgor and Rayo [56] attempt to add concurrency and privacy to existing payment networks.

All of these proposals assume synchronous blockchain access. To the best of our knowledge, Teechain is the first system to avoid this assumption.

Blockchain layer scaling. Prior work addresses the scalability and performance limitations of blockchains by departing from chain structures [51, 80, 89], changing block generation [28, 69, 72], operating in a permissioned setting [4, 30] and using classical consensus [16, 59, 63]. Other approaches operate global committees [29, 70, 85] or shard transactions to concurrent blockchains [47, 48] in order to scale. Unlike these, Teechain executes payments without the blockchain, and users can choose whether or not to use Teechain in conjunction with a blockchain.

Fundamentally, on-chain protocols must reach consensus (global or per shard) [96] for each transaction and thus cannot achieve the performance of Teechain: by operating multiple concurrent and independent committees, Teechain can scale throughput with the number of users and committees in the network. As with any second-layer solutions, Teechain places deposit and settlement transactions on the blockchain and thus benefits from improved blockchain performance.

Trusted hardware and blockchains. Prior work proposes electronic payment systems [81] based on secure coprocessors [25], smart cards [18], and trusted hardware modules [10]. They utilize dedicated hardware to enforce double-spending protection. However, these solutions do not integrate asynchronously with a blockchain and make weaker security assumptions, such as assuming no hardware compromises.

Microsoft's Confidential Consortium Framework (CCF) [78] operates a permissioned blockchain using TEEs to enable high throughput and confidentiality for transactions. Unlike Teechain, CCF does not operate on top of an existing permissionless blockchain, but instead assumes a permissioned setting in which the identities of all members of the CCF consortium are known.

TEEChan [54] uses TEEs to realize single-hop payment channels with limited lifetimes. It provides limited fault tolerance, requires synchronous blockchain access, does not support multi-hop payments, and cannot create payment channels instantly or dynamically assign deposits. TownCrier [98] enables a secure data-feed for blockchain contracts; Tesseract [7] is a secure multi-blockchain cryptocurrency exchange; Ekiden [17] offers a platform for privacy-preserving smart contracts; Obscuro [93] constructs a Bitcoin privacy mechanism; and Paralysis Proofs [99] allows consensus reconfiguration with a blockchain. Apart from the different goals, Teechain uses a more refined security model: clients use a remote TEE to prevent fraud and a local TEE for availability.

8 Conclusion

Teechain is the first payment network to operate with asynchronous blockchain access and offer dynamic deposits. Teechain mitigates against TEE compromises through a novel combination of force-freeze replication and m-out-of-n signatures to construct committee chains. We evaluate Teechain using Intel SGX on Bitcoin; our results show orders of magnitude performance gains compared to the state of the art.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd, David Andersen, for their feedback and suggestions. This project received funding from the European Union Horizon 2020 research and innovation programme under the SecureCloud project (690111); the Israel Science Foundation; the US-Israel Binational Science Foundation (BSF); the US National Science Foundation (NSF); the Israel Cyber Bureau; Engima MPC Inc; and a Mel Berlin Cyber-Security Scholarship. We also thank Intel for their donation of SGX servers.

References

[1] Syed Taha Ali, Dylan Clarke, and Patrick McCorry. 2017. The Nuts and Bolts of Micropayments: a Survey. *Preprint arXiv:1710.02964*.

[2] Amazon. 2019. https://www.amazon.com/.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, Vol. 13.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *EuroSys*.

[5] ARM Ltd. 2017. TrustZone. https://www.arm.com/products/security-on-arm/trustzone. Accessed May 2017.

[6] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. 2016. Foundations of hardware-based attested computation and application to SGX. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 245–260.

[7] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xueyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2017. Tesseract: Real-Time

Cryptocurrency Exchange using Trusted Hardware. *IACR Cryptology ePrint Archive* 2017, 1153.

[8] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. 2014. Proof of Activity: Extending BitcoinâĂŹs Proof of Work via Proof of Stake. ePrint Archive, Report 2014/452. http://eprint.iacr.org/2014/452.

[9] blockchain.info. 2018. Average Confirmation Time. https://blockchain.info/charts/avg-confirmation-time?timespan=all&daysAverageString=7. Accessed May 2018.

[10] Jean-Paul Boly, Antoon Bosselaers, Ronald Cramer, Rolf Michelsen, Stig Mjølsnes, Frank Muller, Torben Pedersen, Birgit Pfitzmann, Peter De Rooij, Berry Schoenmakers, et al. 1994. The ESPRIT project CAFEâĂŤHigh security digital payment systems. In *European Symposium on Research in Computer Security*. Springer, 217–230.

[11] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 157–168.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. *arXiv:1702.07521*, 33.

[13] Conrad Burchert, Christian Decker, and Roger Wattenhofer. 2017. Scalable Funding of Bitcoin Micropayment Channel Networks. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 361–377.

[14] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*. IEEE, 136–145.

[15] Ran Canetti. 2004. Universally composable signature, certification, and authentication. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE, 219–233.

[16] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine Fault Tolerance. In *OSDI*, Vol. 99. 173–186.

[17] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2018. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *Preprint arXiv:1804.05141*.

[18] Eric K Clemons, David C Croson, and Bruce W Weber. 1996. Reengineering money: the Mondex stored value card and beyond. *International Journal of Electronic Commerce* 1, 2, 5–31.

[19] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[20] Christian Decker and Roger Wattenhofer. 2015. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium*. https://doi.org/10.1007/978-3-319-21741-3_1

[21] John R Douceur. 2002. The sybil attack. In *International workshop on peer-to-peer systems*. Springer, 251–260.

[22] Tadge Dryja. 2015. Scalability of lightning with different bips and some back-of-the-envelope calculations. http://diyhpl.us/wiki/transcripts/scalingbitcoin/hong-kong/overview-of-bips-necessary-for-lightning/.

[23] Thaddeus Dryja. 2016. Unlinkable outsourced channel monitoring. https://youtu.be/Gzg_u9gHc5Q?t=2875.

[24] DwarfPool. 2016. Why DwarfPool mines mostly empty blocks and only few ones with transactions. https://www.reddit.com/r/ethereum/comments/57c1yn/why_dwarfpool_mines_mostly_empty_blocks_and_only/. Accessed Feb 2018.

[25] Joan G Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert Van Doorn, and Sean W Smith. 2001. Building the IBM 4758 secure coprocessor. *Computer* 34, 10, 57–66.

[26] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General state channel networks. In *Proceedings of 2018 SIGSAC Conference on Computer and Communications Security*. ACM, 949–966.

[27] Ebay. 2019. https://www.ebay.com/.

[28] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2016)*.

[29] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 51–68.

[30] Gideon Greenspan. 2015. MultiChain private blockchainâĂŤWhite paper. http://www.multichain.com/download/MultiChain-White-Paper.pdf.

[31] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. 2019. SoK: Off The Chain Transactions. ePrint Archive, Report 2019/360. https://eprint.iacr.org/2019/360.

[32] Mike Hearn and Jeremy Spilman. 2015. Rapidly-adjusted micropayments to a pre-determined party. https://en.bitcoin.it/wiki/Contract.

[33] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. 2015. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 129–144. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman

[34] Hex-Five Security. 2018. Multizone: The first Trusted Execution Environment for RISC-V. https://hex-five.com/.

[35] Intel. 2015. Product Change Notification. https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf. Accessed May 2018.

[36] Intel. 2017. Intel SGX SDK for Linux. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf. Accessed May 2017.

[37] Intel Corp. 2014. Software Guard Extensions Programming Reference, Ref. 329298-002US. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf

[38] Intel Inc. 2016. Intel Software Guard Extensions Remote Attestation End-to-End Example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example. Accessed May 2017.

[39] Intel Inc. 2017. sgx_create_monotonic_counter. https://software.intel.com/en-us/node/709160. Accessed May 2017.

[40] Johnson, Simon et al. 2016. Intel® Software Guard Extensions: EPID Provisioning and Attestation Services. https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services.

[41] Jordan Pearson. 2015. WikiLeaks Is Now a Target In the Massive Spam Attack on Bitcoin. https://motherboard.vice.com/en_us/article/ezvw7z/wikileaks-is-now-a-target-in-the-massive-spam-attack-on-bitcoin. Accessed Feb 2018.

[42] Joseph Young. 2017. Analyst: Suspicious Bitcoin Mempool Activity, Transaction Fees Spike to 16. https://cointelegraph.com/news/analyst-suspicious-bitcoin-mempool-activity-transaction-fees-spike-to-16. Accessed Feb 2018.

[43] JP Buntinx. 2017. F2Pool Allegedly Prevented Users From Investing in Status ICO. https://themerkle.com/f2pool-allegedly-prevented-users-from-investing-in-status-ico/. Accessed Feb 2018.

[44] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption. *White paper*.

[45] Keystone Project. 2018. Keystone: Open-source Secure Hardware Enclave. https://keystone-enclave.org/.

[46] Rami Khalil and Arthur Gervais. 2017. Revive: Rebalancing Off-Blockchain Payment Networks. *Gas* 200, 400.

[47] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing.

In *25th USENIX Security Symposium (USENIX Security 16)*.

[48] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 583–598.

[49] Hugo Krawczyk. 2003. SIGMA: The âĂŸSIGn-and-MAcâĂŹapproach to authenticated Diffie-Hellman and its use in the IKE protocols. In *Annual International Cryptology Conference*. Springer, 400–425.

[50] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News. Dec 2001* 32, 4, 18–25.

[51] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. 2015. Inclusive Block Chain Protocols. In *Financial Cryptography*. Puerto Rico.

[52] Lightning Network community. 2017. Lightning Network Daemon. https://github.com/lightningnetwork/lnd. Accessed May 2017.

[53] Linaro. 2014. Open Portable Trusted Execution Environment. https://www.op-tee.org/.

[54] Joshua Lind, Ittay Eyal, Peter Pietzuch, and Emin Gün Sirer. 2016. Teechan: Payment channels using trusted execution environments. *Preprint arXiv:1612.07766*.

[55] Joshua Lind, Oded Naor, Florian Kelbert, Ittay Eyal, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Proceedings of the 27th Symposium on Operating Systems Principles*.

[56] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. 2017. Concurrency and privacy with payment-channel networks.

[57] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. 2018. Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network. *IACR Cryptology ePrint Archive* 2018, 236.

[58] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. Cryptology ePrint Archive, Report 2017/048. http://eprint.iacr.org/2017/048.

[59] David Mazieres. 2015. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus. https://www.stellar.org/papers/stellar-consensus-protocol.pdf.

[60] Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. 2018. Pisa: Arbitration Outsourcing for State Channels. *IACR Cryptology ePrint Archive* 2018, 582.

[61] Patrick McCorry, Chris Buckland, Surya Bakshi, Karl Wüst, and Andrew Miller. 2018. You sank my battleship! A case study to evaluate state channels as a scaling solution for cryptocurrencies.

[62] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. 2017. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812*.

[63] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[64] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 69–90.

[65] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. http://www.bitcoin.org/bitcoin.pdf.

[66] Dan OâĂŹKeeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. 2018. Spectre attack against SGX enclave.

[67] Open Enclave SDK Community. 2018. Open Enclave SDK. https://openenclave.io/sdk/.

[68] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 643–673.

[69] Rafael Pass and Elaine Shi. 2016. Hybrid Consensus: Efficient Consensus in the Permissionless Model. ePrint Archive, Report 2016/917.

[70] Rafael Pass and Elaine Shi. 2018. Thunderella: Blockchains with optimistic instant confirmation. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–33.

[71] Rafael Pass, Elaine Shi, and Florian Tramer. 2017. Formal abstractions for attested execution secure processors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 260–289.

[72] Joseph Poon and Vitalik Buterin. 2017. Plasma: Scalable autonomous smart contracts.

[73] Joseph Poon and Thaddeus Dryja. 2016. The Bitcoin Lightning Network: Scalable off-chain instant payments. Technical Report (draft 0.5.9.1). https://lightning.network. Accessed May 2017.

[74] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 459–474.

[75] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 38–54.

[76] SECBIT. 2018. How the winner got Fomo3D prize âĂŤ A Detailed Explanation. https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f. Accessed Sep 2018.

[77] István András Seres, László Gulyás, Dániel A Nagy, and Péter Burcsi. 2019. Topological Analysis of Bitcoin's Lightning Network. *Preprint arXiv:1901.04972*.

[78] Alex Shamis, Amaury Chamayou, Christine Avanessians, Christoph M. Wintersteiger, Edward Ashton, Felix Schuster, CÃľdric Fournet, Julien Maffre, Kartik Nayak, Mark Russinovich, Matthew Kerner, Miguel Castro, Thomas Moscibroda, Olga Vrousgou, Roy Schwartz, Sid Krishna, Sylvan Clebsch, and Olya Ohrimenko. 2019. *CCF: A Framework for Building Confidential Verifiable Replicated Services*. Technical Report MSR-TR-2019-16. Microsoft. https://www.microsoft.com/en-us/research/publication/ccf-a-framework-for-building-confidential-verifiable-replicated-services/

[79] Elaine Shi, Fan Zhang, Rafael Pass, Srini Devadas, Dawn Song, and Chang Liu. 2015. Trusted hardware: Life, the composable universe, and everything. *manuscript* (2015).

[80] Yonatan Sompolinsky and Aviv Zohar. 2015. Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains. In *Financial Cryptography*. Puerto Rico.

[81] Susan Stepney, David Cooper, and Jim Woodcock. 2000. An electronic purse: Specification, refinement and proof. Oxford University.

[82] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 875–892. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx

[83] superfreek. 2017. BTC Spam attack. 200,000 unconfirmed transactions halts bitcoin. https://steemit.com/cryptocurrency/@superfreek/btc-spam-attack-200-000-unconfirmed-transactions-halts-bitcoin. Accessed Feb 2018.

[84] Nick Szabo. 1997. The idea of smart contracts. *Nick SzaboâĂŹs Papers and Concise Tutorials* 6.

[85] Team Rocket. 2018. Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5-o2FYopNPVYwrRVGV.

[86] The Bitcoin Community. 2013. libsecp256k1. https://github.com/bitcoin-core/secp256k1.

[87] The Bitcoin Community. 2016. Bitcoin Core version 0.13.1 released. https://bitcoin.org/en/release/v0.13.1. Accessed May 2017.

[88] The Bitcoin community. 2017. M-of-N Multisig, Multisig Output. https://bitcoin.org/en/glossary/multisig. Accessed May 2017.
[89] The Ethereum community. 2017. Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper. Accessed May 2017.
[90] The Linux-SGX community. 2016. Intel(R) Software Guard Extensions for Linux OS. https://github.com/intel/linux-sgx.
[91] The Raiden Network community. 2017. The Raiden Network. https://raiden.network/. Accessed October 2017.
[92] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2016. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. Cryptology ePrint Archive, Report 2016/635. http://eprint.iacr.org/2016/635.
[93] Muoi Tran, Loi Luu, Min Suk Kang, Iddo Bentov, and Prateek Saxena. 2017. Obscuro: A Bitcoin Mixer using Trusted Execution Environments. *IACR Cryptology ePrint Archive* 2017, 974.
[94] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*.
[95] Robbert Van Renesse and Fred B Schneider. 2004. Chain Replication for Supporting High Throughput and Availability.. In *6th Symposium on Operating Systems Design and Implementation*, Vol. 4. 91–104.
[96] Marko Vukolić. 2015. The quest for scalable blockchain fabric: Proof-of-Work vs. BFT replication. In *International Workshop on Open Problems in Network Security*. Springer, 112–125.
[97] Gavin Wood. 2016. Ethereum: A Secure Decentralised Generalised Transaction Ledger (EIP-150). http://gavwood.com/Paper.pdf.
[98] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 270–282.
[99] Fan Zhang, Philip Daian, Gabriel Kaptchuk, Iddo Bentov, Ian Miers, and Ari Juels. 2017. Paralysis Proofs: Secure Dynamic Access Structures for Cryptocurrencies and More.

A Teechain Protocol Correctness

We first intuitively define the security guarantees a payment network should provide (Appendix A.1) and describe the framework we use to construct our proofs (Appendix A.2 and Appendix A.3). We then formally prove that Teechain achieves the desired security properties for both channels (Appendix A.4) and multi-hop payments (Appendix A.5).

A.1 Security Guarantees

Teechain protects the funds of all participants in the network; despite what others may do, funds cannot be stolen or double spent. We define *balance security* to express intuitively: at any point during an execution, any party can unilaterally reclaim the channels' balances and unassociated deposits on the underlying blockchain, correctly reflecting all payments. Participants must be able to do so despite other's actions.

To formalize balance security, we first define an execution trace σ as a time-ordered series of events, where each event represents an operation and its return value; σ_t denotes the prefix of σ until time t. For execution trace σ, user u, and time t, denote by $L_t(u)$ the balance of u in σ at time t on the blockchain, i.e., the sum of all the funds u has access to on the blockchain. In particular, $L_0(u)$ is the initial balance of u in σ. Denote by $paid_t(u)$ the accumulated sum of all payments made by u in σ_t using Teechain channels; and by $rcvd_t(u)$ the accumulated sum of payments received by u in σ_t. We thus define the *perceived balance* of u in σ at time t as $perceivedBal_t(u) = L_0(u) + rcvd_t(u) - paid_t(u)$. We therefore define *balance security* as:

Definition A.1 (Balance security)**.** *A protocol satisfies* balance security *if for any prefix σ_t, any well-behaved user u can unilaterally perform a series of operations, possibly interleaved with operations of other users, that will complete at finite time $t' \geq t$, after which at any time $t'' \geq t'$: $L_{t''}(u) \geq perceivedBal_t(u)$.*

A well-behaved user, is one that faithfully follows the Teechain protocol, stores private keys used to communicate with the ledger securly, and does not leak them to other users in the system.

A.2 Ideal Functionalities and Simulation Based Security

Simulation Based Proofs in the Universal-Composability Framework. Our formal proof for Teechain's channel protocol is based on the simulation based security framework Universal Composability (UC) [14], which is a general purpose framework for modeling and constructing secure protocols. The model is based on a system of interactive Turing machines (ITMs), which are described based upon how they behave when receiving messages from other ITMs.

The UC framework includes several ITMs: an environment $\mathcal{E}$, which represents the external world. The environment chooses the inputs given to each party in the system, and observes the outputs. The framework also includes honest parties which follow the protocol, and a byzantine adversary $\mathcal{A}$, that can corrupt users at will. Our model deals with an adaptive adversary, i.e., once a user is corrupted by $\mathcal{A}$, it cannot be uncorrupted again until the end of the execution. We only define the security guarantees of honest users who follow the Teechain protocol.

The model also includes ideal functionalities, which act as idealized third parties, and implement some target specifications. We describe the behavior of such ideal functionalities based on an exposed API. This API exhibits the desired properties of the protocol. Ideal functionalities are also used in the real-world in order to represent network primitives and setup assumptions, and we use them to also model TEEs and the blockchain.

The proof that some protocol captures a specific property in the UC framework consists of the following stages:

1. Showing that any real-world execution is indistinguishable to the external environment $\mathcal{E}$ from an equivalent ideal-world execution. The proof is based on describing a simulator $\mathcal{S}$ in the ideal-world, which translates every adversary $\mathcal{A}$ in the real-world into a simulated attacker, which is indistinguishable to the environment. We do so in hybrid steps, and in each step we prove indistinguishability to the environment from the previous hybrid step.
2. Proving that the desired property of the protocol is maintained by the ideal functionality in the ideal-world.

Since the real-world and ideal-world are indistinguishable, if an attacker breaks a security guarantee in the real-world, then it will also break in the ideal-one. Thus, to prove a security guarantee holds in the real-world, it is sufficient to show it holds in the ideal one.

Real-World Execution. The real-world Teechain channel protocol is identical to the one described in §4, except that we model the TEE and the ledger as ideal functionalities in the UC framework. This model is based on previous works that formalized the model, such as [6, 56, 75, 79, 98].

Ideal functionality $\mathcal{F}_{TEE}$. $\mathcal{F}_{TEE}$ is an ideal functionality that models a TEE. This model is based on a version of the ideal functionality of Shi et al. [79]. User $\mathcal{U}$ is a Teechain user equipped with a TEE; *prog* is some program to run in an enclave; *inp*, *outp*, *mem* are the *prog* input, output and memory tape respectively. We further let *sid* as the session identifier and *id* is the enclave identifier. λ is a security parameter, and Σ, KGen are a signature scheme and its key generation algorithm respectively. Lastly, let PK_{TEE}, SK_{TEE} be a TEE's public-key and secret-key generated on the TEE's initialization.

An enclave is an isolated software container loaded with some program, in our case the Teechain protocol. $\mathcal{F}_{TEE}$ abstracts an enclave as a third party trusted for execution, confidentiality and authenticity, with respect to any user that is part of the system, and in particular $\mathcal{U}$ that owns the TEE.

We now describe the API of $\mathcal{F}_{TEE}$. When initialized, $\mathcal{F}_{TEE}$ generates a public secret key pair, and publicizes the public key, i.e., other users or TEEs can verify messages signed by other $\mathcal{F}_{TEE}$. This corresponds to an attestation service provided in the real-world execution. In addition, $\mathcal{F}_{TEE}$ includes two calls, one for the installation of a new program *prog*, and the other one is a resume call for *prog* with some input.

The full API of $\mathcal{F}_{TEE}$ is:

Algorithm 4: $\mathcal{F}_{TEE}$'s API

Input: $(PK_{TEE}, SK_{TEE}) \leftarrow KGen\left(1^{\lambda}\right)$

on receive $(sid, idx, \mathsf{install}, inp)$ from $\mathcal{U}_i$:
  if $(idx, \mathcal{U}_i, \_, \_)$ *is not stored* **then**
    store $(idx, \mathcal{U}_i, prog, inp)$
  else
    return
  end
on receive $(sid, idx, \mathsf{resume}, inp)$ from $\mathcal{U}_i$:
  if $(idx, \mathcal{U}_i, prog, mem)$ *is stored* **then**
    $outp, \overline{mem} \leftarrow prog\,(inp, mem)$
    store $(idx, \mathcal{U}_i, prog, \overline{mem})$
    $\overline{\sigma} \leftarrow \Sigma.Sign\,(SK_{TEE}, (prog, outp))$
    return $(sid, idx, outp, \overline{\sigma})$ to $\mathcal{U}_i$
  else
    return $\perp$
  end

$\mathcal{F}_{TEE}$ is a *setup assumption* [14] that models the functionality offered by real-world TEEs, and in particular Intel's SGX. Due to this, $\mathcal{F}_{TEE}$ uses a "real" signature scheme Σ, rather than an ideal version of it [15]. We assume that the signature scheme Σ used by $\mathcal{F}_{TEE}$ is unforgeable under chosen attacks, and that all parties $\mathcal{U}_i$ know PK_{TEE} of all the other TEEs at the start of the execution. We note that in order to deal with real-world SGX vulnerabilities, we use different methods to mitigate such attacks (see §4.4).

Each user $\mathcal{U}$ is identified by unique id (simply denoted by $\mathcal{U}$, or Alice and Bob), and a session id *sid*, obtained from the environment $\mathcal{E}$ [15]. Parties send messages to each other via authenticated channels, and the adversary $\mathcal{A}$ observes all messages sent over the network. We use the standard "delayed messages" terminology [15]: when a message *msg* is sent between users, *msg* is first sent to $\mathcal{A}$ (the simulator $\mathcal{S}$), and forwarded to the intended user $\mathcal{U}$, after the acknowledgment by $\mathcal{S}$. Furthermore, $\mathcal{S}$ can delay messages between parties (i.e., asynchronous network), but eventually delivers them.

Note that $\mathcal{F}_{TEE}$ is a local ideal functionality: a user $\mathcal{U}$ can talk to its $\mathcal{F}_{TEE}$ without the messages being leaked to $\mathcal{A}$. However, when $\mathcal{A}$ corrupts a user it gets full access to the user's software and hardware. $\mathcal{A}$ can fully observe all the calls made to $\mathcal{F}_{TEE}$, but cannot tamper with the hardware's confidentiality, integrity and authenticity guarantees.

Ideal functionality $\mathcal{F}_B$. An ideal functionality that represents the ledger, i.e., the underlying blockchain. It maintains a single list, B, that represents the balances of all the users in the system. Each entry has a public key PK and any instruction to edit an entry requires the message to be signed by the corresponding private key, SK. Entries in B are in the form of (PK, val), where $\mathsf{val}$is the current balance associated with the public key.

At the beginning of the execution, i.e., at $t = 0$, B contains initial entries with balances for the different public keys in the system. Also, some of the users also have the corresponding secret keys matching to the above public keys. $\mathcal{F}_B$ exposes two types of API calls, one is designed to transfer money from one entry in B to another, and one is designed to query $\mathcal{F}_B$ on existing entries.

Algorithm 5: $\mathcal{F}_B$'s API

Input: The initial balances of all the public keys are stored in B

on receive $(\mathsf{transfer}, PK_u, \mathsf{val}, \overline{\sigma}, PK_v)$ from u:
  if $\Sigma.verify(PK_u, \overline{\sigma}) \wedge \exists bal : B(PK_u) = bal \wedge bal \geq val$ **then**
    $B(PK_v) \leftarrow B(PK_v) + \mathsf{val}$
    $B(PK_u) \leftarrow B(PK_u) - \mathsf{val}$
  else
    return $\perp$
  end
on receive $(\mathsf{getLedgerBalance}, PK_u)$
  if $\exists bal : B(PK_u) = bal$ **then**
    return bal
  else
    return $\perp$
  end

We model the real-world execution of Teechain in the $(\mathcal{F}_{TEE}, \mathcal{F}_B)$ hybrid-model.

We are now ready to formally define balance security and perceived balance of any user using Teechain.

Ideal-World Model. We define the ideal functionality $\mathcal{F}_{Teechain}$ which captures Teechain's protocol. $\mathcal{F}_{Teechain}$ is defined by an internal state of its internal variables, and by an API, which users can invoke. An invocation of an API call *call* by user u with arguments $(arg1, arg2, \dots)$ is denoted by: $call_u(arg1, arg2, \dots)$.

The internal variables $\mathcal{F}_{Teechain}$ maintains are described in Fig. 9, a description of each call is described in Fig. 10, and the complete algorithm is described in Alg. 6.

A.3 Indistinguishability of the real-world and ideal-world

We now move to show that the real-world execution is indistinguishable to the external environment from the ideal-world one.

$\mathcal{F}_{Teechain}$ maintains the following internal variables:

L Set of funds for the users using $\mathcal{F}_{Teechain}$. *L* captures the same functionality as $\mathcal{F}_B$ in the real-world.
Entries are in the form of (u, val), where $\mathsf{val}$is u's amount on the ledger.

C Set of all open channels. We denote as u and v as two users who have an open channel between them, *amountU* and *amountV* as u's and v's balances on the channel, c_{id} as a unique channel identifier agreed between u and v prior to the opening, and *isSymmetric* as a boolean variable indicating the two different states a channel between two users can be in, i.e., whether the channel is open for both users or just the channel opening initiator.
Entries in *C* are in the form of $(c_{id}, u, v, amountU, amountV, isSymmetric)$.

D Set of deposits associated with an open channel. We denote *deposit_id* as a unique identifier given to each new deposit by $\mathcal{F}_{Teechain}$, valas the amount of the deposit, u as the user who added the deposit, c_{id} as the channel identifier the deposit is associated to (or $\perp$ if the deposit is not associated with any channel), and *isSymmetric* as a boolean variable indicating if the deposit is associated to a channel, and both users are aware of this association.
Entries in *D* are in the form of $(deposit\_id, \mathsf{val}, u, c_{id}, isSymmetric)$.

PendingDeposits This is a set of deposits in the process of dissociation from a channel, i.e., a user u who initiated a deposit dissociation and the dissociation is not approved yet by the other party of the channel.
Entries in *PendingDeposits*: are in the format of $(deposit\_id, u, isSymmetric)$.

PendingPayments Set of pending payments, i.e., if user u sent a payment to v, and v did not accept the payment yet.
Entries in *PendingPayments*: are in the form of $(c_{id}, v, \mathsf{val})$. c_{id} is the channel id where the payment with amount $\mathsf{val}$is pending.

PendingChannels Set of channels in the process of being settled. In order to fully settle a channel both of its users need to settle it separately. If only one of the users settled the channel and other user did not, then this set will reflect it.
Entries in *PendingChannels*: are in the form of (c_{id}, v), were c_{id} is the channel identifier and v is the party that did not settle the channel yet.

PendingLedgerPayments Set of pending payments waiting to be added to the ledger *L*.
Entries in *PendingLedgerPayments* are in the form of $(ledger\_payment\_id, u, \mathsf{val})$, where *ledger_payment_id* is a unique id of the payment given by $\mathcal{F}_{Teechain}$, u is the recipient of the payment of amount val.

Figure 9. $\mathcal{F}_{Teechain}$'s Internal Variables

Lemma 1. *The Teechain channel protocol in the* $(\mathcal{F}_{TEE}, \mathcal{F}_B)$ *hybrid model UC-realizes the ideal functionality* $\mathcal{F}_{Teechain}$.

Proof. Let $\mathcal{A}$ be an adversary against the Teechain protocol. We construct an ideal-world adversary (the simulator) $\mathcal{S}$, such that any environment $\mathcal{E}$ cannot distinguish between interacting with the adversary $\mathcal{A}$ and the Teechain protocol or with the simulator $\mathcal{S}$ and the ideal functionality $\mathcal{F}_{Teechain}$.

We do so by a series of hybrid steps starting at the real-world execution of the protocol in the $(\mathcal{F}_{TEE}, \mathcal{F}_B)$ hybrid - H_0, and eventually reaching the ideal world. At each step we prove indistinguishability.

- **Hybrid** H_0 is the real-world execution in the $(\mathcal{F}_{TEE}, \mathcal{F}_B)$ hybrid model.
- **Hybrid** H_1 behaves the same as H_0 except that $\mathcal{S}$ generates a secret-public key pair (SK, PK) for the signing scheme Σ and publishes the public key PK. When $\mathcal{A}$ wants to communicate with its $\mathcal{F}_{TEE}$, $\mathcal{S}$ faithfully emulates $\mathcal{F}_{TEE}$'s behavior, and records $\mathcal{A}$'s messages.
  As $\mathcal{S}$'s simulation of the real-world protocol is done perfectly, the environment $\mathcal{E}$ cannot distinguish between H_0 (the real-world execution) and H_1.
- **Hybrid** H_2 behaves the same as H_1 except for the following difference: whenever $\mathcal{A}$ wants to communicate with $\mathcal{F}_B$, $\mathcal{S}$ faithfully emulates $\mathcal{F}_B$'s behavior for $\mathcal{A}$. As $\mathcal{A}$'s view in H_2 are perfectly emulated for him when interacting with the ledger, no environment $\mathcal{E}$ can distinguish between H_2 and H_1.
- **Hybrid** H_3 behaves the same as H_2 except for the following difference: If $\mathcal{A}$ invoked its $\mathcal{F}_{TEE}$ with a correct $\mathsf{install}$ message with program *prog*, then for every correct resume message $\mathcal{S}$ records the tuple (msg,σ) from $\mathcal{F}_{TEE}$, where *msg* is the output of running *prog* in $\mathcal{F}_{TEE}$, and σ is the signature generated inside the $\mathcal{F}_{TEE}$, using the SK generated in H_1.
  Let Ω denote all such possible tuples. If $(msg, \sigma) \notin \Omega$ then $\mathcal{S}$ aborts, otherwise, $\mathcal{S}$ delivers the message to Bob (the party in the protocol that is not controlled by $\mathcal{A}$).
  We can argue that H_2 is indistinguishable from H_1 by reducing the problem to the EU-CMA of the signing scheme. If $\mathcal{A}$ does not send one of the correct tuples to the other party, then the other party's attestation (verification mechanism) will fail (but with negligible probability). Otherwise, $\mathcal{E}$ and $\mathcal{A}$ can be leveraged to construct an adversary that succeeds in a signature forgery.
- **Hybrid** H_4 is the same as H_3 except that for the following difference: Whenever $\mathcal{A}$ delivers a signed message (by himself or from his $\mathcal{F}_{TEE}$) to other users in the system signed by a secret key of Teechain's protocol, $\mathcal{S}$ behaves as in H_3, i.e., $\mathcal{S}$ records the tuple (msg, σ), and

The following calls are for user interaction with the ledger:

getLedgerBalance Upon receiving $\mathsf{getLedgerBalance}_v(u)$ from any user v in the system, $\mathcal{F}_{Teechain}$ returns the current entry in L indicating u's balance on the ledger.

acceptLedgerPayment Upon receiving this call from a user u in the system, $\mathcal{F}_{Teechain}$ checks if u has a pending payment waiting to be reflected on the ledger L. If so, $\mathcal{F}_{Teechain}$ changes L to reflect the payment.

The following calls capture the unilateral deposit handling of users:

addDeposit When receiving this call $\mathcal{F}_{Teechain}$ adds a new deposit for user u.

removeDeposit User u can call this function to remove an unassociated deposit from the system and add the deposit's amount back on the ledger.

The last set of calls captures the logic of user interaction with other users and handle channels in the system:

openChannel u invokes this call in order to initiate a channel opening between her and another user in the system.

acceptChannelOpen v can invoke this call to complete the channel opening process. This call can be invoked by the v after openChannel is invoked.

associateDeposit u can invoke this call in order to start the process of associating a deposit with a specific channel.

acceptAssociateDeposit v invokes this call to complete the process of associating the deposit *deposit_id*. In the real-world execution v needs to make sure at this stage that the deposit is on the ledger.

dissociateDeposit Invoked by u to start dissociating deposit *deposit_id* from a channel it is associated to.

acceptDissociate The second phase of dissociating a deposit from a channel, indicating that the other party in the channel accepted the dissociation.

ackDissociate Completes the dissociation of a deposit from a channel.
After the call ends successfully the deposit can be removed or associated again with another channel by u.

pay User u invokes this function to pay val on channel c_{id} to user v. When the call ends, $\mathcal{F}_{Teechain}$ returns to u a payment id *pending_payment_id*.

receivePayment v invokes this function to accept a payment with payment id *pending_payment_id* made by another user on a channel. After the call ends, v's balance on the channel is increased by val.

settleChannel u invokes this call to settle channel c_{id}, i.e., receive his current balance from the channel on the ledger. When invoked, $\mathcal{F}_{Teechain}$ generates a pending payment *pending_payment_id* which reflects u's balance on the channel and returns it to u.

Figure 10. $\mathcal{F}_{Teechain}$'s Calls

as in H_4, $\mathcal{S}$ aborts if *msg* is not signed correctly by the corresponding secret key.
H_4 is indistinguishable from H_3 for the same reasons H_3 is indistinguishable from H_2, i.e., otherwise $\mathcal{E}$ and $\mathcal{A}$ will be able to construct an adversary that can succeed in signature forgery.

- **Hybrid** H_5 is the ideal world execution, i.e., we map the calls in the simulated real-world to calls of $\mathcal{S}$ to $\mathcal{F}_{Teechain}$.
  - **new_pay_channel** Whenever a user sends new_pay_channel to $\mathcal{F}_{TEE}$ to create a new channel, $\mathcal{S}$ sends a openChannel message to $\mathcal{F}_{Teechain}$ in the ideal world.
  - ***newChannelAck*** Whenever *newChannelAck* is delivered to the recipient in the real-world, $\mathcal{S}$ invokes acceptChannelOpen to $\mathcal{F}_{Teechain}$. *newChannelAck* is delivered when new_pay_channel arrives to the counterpart's $\mathcal{F}_{TEE}$.
  - **new_deposit** Whenever a user invokes his $\mathcal{F}_{TEE}$ with new_deposit after correctly transferring money on the ledger by invoking transfer to $\mathcal{F}_B$, $\mathcal{S}$ invokes $\mathcal{F}_{Teechain}$ with addDeposit, and records the returned *deposit_id*.
  - **release_deposit** Whenever a user invokes release_deposit to his $\mathcal{F}_{TEE}$, $\mathcal{S}$ calls removeDeposit to $\mathcal{F}_{Teechain}$ with the corresponding *pending_payment_id*, and when the user calls transfer to $\mathcal{F}_B$ in order to place the tx on the ledger, $\mathcal{S}$ calls acceptLedgerPayment with the corresponding *pending_payment_id*.
  - **associate_deposit** Whenever a user sends associate_deposit to another user in the system, $\mathcal{S}$ sends associateDeposit to $\mathcal{F}_{Teechain}$ on that user's behalf.
  - **approve_deposit** Whenever a user invokes $\mathcal{F}_{TEE}$ with approve_deposit, the simulator $\mathcal{S}$ calls acceptAssociateDeposit with the corresponding deposit id *deposit_id*.
  - **dissociate_deposit** When a user correctly invokes his $\mathcal{F}_{TEE}$ with dissociate_deposit, $\mathcal{S}$ calls dissociateDeposit.
  - **dissociatedDeposit** When a user passes to his $\mathcal{F}_{TEE}$ dissociatedDeposit message, $\mathcal{S}$ invokes dissociateDeposit to $\mathcal{F}_{Teechain}$. The dissociatedDeposit is the message send to $\mathcal{F}_{TEE}$ by the counterpart upon receiving the dissociate_deposit.
  - **dissociatedDepositAck** When a user invokes his $\mathcal{F}_{TEE}$ with the message dissociatedDepositAck, $\mathcal{S}$ invokes ackDissociate in the ideal world. dissociatedDepositAck is delivered to $\mathcal{F}_{TEE}$ upon receiving the ack from dissociatedDeposit.

Algorithm 6: $\mathcal{F}_{Teechain}$'s API

$\forall u : L(u) \leftarrow \bot$
$\forall c_{id} : C(c_{id}) \leftarrow \bot$
$\forall deposit\_id : D(deposit\_id) \leftarrow \bot$
$\forall deposit\_id : PendingDeposits(deposit\_id) \leftarrow \bot$
$\forall pending\_payment\_id :$ $PendingPayments(pending\_payment\_id) \leftarrow \bot$
$\forall c_{id} : PendingChannels(c_{id}) \leftarrow \bot$
$\forall ledger\_payment\_id :$ $PendingLedgerPayments(ledger\_payment\_id) \leftarrow \bot$
on getLedgerBalance$_v(u)$ /* The Ledger is public, any user can get the balance of any other user in the system */
  if $L(u) \neq \bot$ **then**
    return $(success, L(u))$
  else
    return $(fail)$
on acceptLedgerPayment$_u(ledger_payment_id)$
  if $\exists val :$ $PendingLedgerPayments(ledger\_payment\_id) = (u, amount)$ **then**
    $L(u) \leftarrow L(u) + \mathsf{val}$
    $PendingLedgerPayments(ledger\_payment\_id) \leftarrow \bot$
    return $(success)$
  else
    return $(fail)$
on addDeposit$_u(amount)$
  if $L(u) \geq amount$ /* If u has enough money on the ledger to create the deposit */ **then**
    $deposit\_id \leftarrow textitdeposit\_counter$
    $deposit\_counter \leftarrow deposit\_counter + 1$
    $D(deposit\_id) \leftarrow (amount, u, \bot, \bot)$
    $L(u) \leftarrow L(u) - \mathsf{val}$
    return $(success, deposit\_id)$
  else
    return $(fail)$
on removeDeposit$_u(deposit_id)$
  if $\exists \mathsf{val}, isSymmetric : D(deposit\_id) = (amount, u, \bot, isSymmetric)$ /* deposit entry is not associated with any channel */ **then**
    $PendingLedgerPayments(ledger\_payment\_id) \leftarrow (u,amount)$
    $ledger\_payment\_id \leftarrow ledger\_payment\_id + 1$
    $D(deposit\_id) \leftarrow \bot$
    return $(success, ledger\_payment\_id)$
  else
    return $(fail)$
on openChannel$_u(c_{id}, v)$
  if $C(c_{id}) = \bot$ /* there is not a channel entry with c_{id} */ **then**
    $C(c_{id}) \leftarrow (u, v, 0, 0, \bot)$ /* initialize an entry for the channel with capacity 0 for both sides */
    return $(success)$
  else
    return $(fail)$
on acceptChannelOpen$_v(c_{id})$
  if $\exists amountU, amountV : C(c_{id}) = (u, v, amountU, amountV, \bot)$ **then**
    $C(c_{id}) \leftarrow (u, v, amountU, amountV, \top)$
    return $(success)$
  else
    return $(fail)$
on associateDeposit$_u(c_{id}, deposit_id)$
  if $\exists val : D(deposit\_id) = (amount, u, \bot, \bot) \wedge \exists x, y, amountX, amountY, isSymmetric : C(c_{id}) = (x, y, amountU, amountV, isSymmetric) \wedge (x = u \vee y = u)$ **then**
    if $y = u \wedge isSymmetric = \bot$ /* the channel is not accepted by both parties */ **then**
      return $(fail)$
    $D(deposit\_id) \leftarrow (amount, u, c_{id}, \bot)$ /* assign the deposit to channel c_{id} */ **if** $x = u$ **then**
      $C(c_{id}) \leftarrow (x, y, amountX + amount, amountY, isSymmetric)$
    else
      $C(c_{id}) \leftarrow (x, y, amountX, amountY + amount, isSymmetric)$
    return $(success)$
  else
    return $(fail)$
on acceptAssociateDeposit$_v(c_{id}, deposit_id)$
  if $\exists val : D(deposit\_id) = (amount, u, c_{id}, \bot)$ **then**
    $D(deposit\_id) \leftarrow (amount, u, c_{id}, \top)$
    return $(success)$
  else
    return $(fail)$
on dissociateDeposit$_u(deposit_id)$
  if $\exists val, c_{id}, isSymmetric : D(deposit\_id) = (amount, u, c_{id}, isSymmetric) \wedge \exists v, amountU, amountV, isSymmetric : C(c_{id}) = (c_{id}, u, v, amountU, amountV, isSymmetric) \wedge val \leq amountU$ /* The deposit is accepted by both sides, and u has enough capacity in the channel to dissociate */ **then**
    $C(c_{id}) \leftarrow (c_{id}, u, v, amountU - \mathsf{val}, amountV, isSymmetric)$ /* deduct the deposit's amount from u's capacity in the channel */
    $PendingDeposits(deposit\_id) \leftarrow (u, \bot)$ /* entry indicating $deposit\_id$ is about to be dissociated */
    return $(success)$
  else
    return $(fail)$
on acceptDissociate$_v(deposit_id)$
  if $\exists u$ s.t. $PendingDeposits(deposit\_id) = (u, \bot)$ **then**
    $PendingDeposits(deposit\_id) \leftarrow (u, \top)$
    return $(success)$
  else
    return $(fail)$
on ackDissociate$_u(deposit_id)$
  if $PendingDeposits(deposit\_id) = (u, \top)$ **then**
    $PendingDeposits(deposit\_id) \leftarrow \bot$
    $D(deposit\_id) \leftarrow (deposit\ id, amount, u, \bot, \bot)$ //change third argument of the entry to indicate the dissociation is completed
    return $(success)$
  else
    return $(fail)$
on pay$_u(c_{id}, amount)$
  if $\exists x,y, amountX, amountY, isSymmetric : C(c_{id}) = (x, y, amountX, amountY, isSymmetric) \wedge ((isSymmetric = \top) \vee (isSymmetric = \bot \wedge x = u))$ **then**
    if $x = u \wedge val \leq amountX$ **then**
      $C(c_{id}) \leftarrow (x, y, amountX - amount, amountY, isSymmetric)$
      $PendingPayments(pending\_payment\_id) \leftarrow (y, c_{id}, amount)$
    if $y = u \wedge val \leq amountY$ **then**
      $C(c_{id}) \leftarrow (x, y, amountX, amountY - amount, isSymmetric)$
      $PendingPayments(pending\_payment\_id) \leftarrow (x, c_{id}, amount)$
    $pending\_payment\_id \leftarrow pending\_payment\_id + 1$
    return $(success, pending\_payment\_id)$
  else
    return $(fail)$
on receivePayment$_v(pending_payment_id)$
  if $\exists c_{id}, amount :$ $PendingPayments(pending\_payment\_id) = (v, c_{id}, val)$ **then**
    $C(c_{id}) \leftarrow (u, v, amountV + \mathsf{val}, isSymmetric)$
    return $(success)$
    $PendingPayments(pending\_payment\_id) \leftarrow \bot$
  else
    return $(fail)$
on settleChannel$_u(c_{id})$
  if $\exists v, amountU, amountV, isSymmetric : C(c_{id}) = (u, v, amountU, amountV, isSymmetric)$ **then**
    $pendingDeposits \leftarrow \{deposit\_id | PendingDeposits(deposit\_id) = (u, \bot) \vee (u, \top)\}$
    $pendingDepositsSum \leftarrow \sum_{x \in pendingDeposits} x$ /* Need to remove all the deposits that their dissociation is not completed */
    $PendingLedgerPayments(ledger\_payment\_id) \leftarrow (u, amountU + pendingDepositsSum)$
    $ledger\_payment\_id \leftarrow ledger\_payment\_id + 1$
    foreach $(deposit\_id \in pendingDeposits)$: $PendingDeposits(deposit\_id) = \bot$
    foreach $(deposit\_id \in pendingDeposits)$: $D(deposit\_id) \leftarrow \bot$
    $C(c_{id}) \leftarrow (u, v, 0, amountV, isSymmetric)$
    if $PendingChannels(c_{id}) = \bot$ /* If this is the first settle message */ **then**
      $PendingChannels(c_{id}) \leftarrow v$
    else
      $deposits \leftarrow \{deposit\_id | \exists amount, isSymmetric, x : D(deposit\_id) = (amount, x, c_{id}, isSymmetric)\}$
      foreach $(deposit\_id \in deposits)$: $D(deposit\_id) \leftarrow \bot$ /* remove all the deposits associated with the channel */
      $C(c_{id}) \leftarrow \bot$
      $PendingChannels(c_{id}) \leftarrow \bot$
    return $(success, ledger\_payment\_id)$
  else
    return $(fail)$

- **pay_channel** When a user invokes pay_channel to $\mathcal{F}_{TEE}$, $\mathcal{S}$ invokes pay to $\mathcal{F}_{Teechain}$.
- **paid** Whenever a user passes paid to his $\mathcal{F}_{TEE}$, then $\mathcal{S}$ calls receivePayment. This message is delivered

when the counterpart receives the pay_channel and passes it to his $\mathcal{F}_{TEE}$.

- **settle_channel** Whenever a user invokes settle_channel to his $\mathcal{F}_{TEE}$, $\mathcal{S}$ invokes settleChannel commands on behalf of the user of the channel, i.e., for a channel between u and v them $\mathcal{S}$ will invoke $\mathsf{settleChannel}_u$ and $\mathsf{settleChannel}_v$ with the channel id c_{id}.

In H_4, $\mathcal{S}$ can faithfully interact with $\mathcal{F}_{Teechain}$, while faithfully emulating $\mathcal{A}$'s view of the real-world. $\mathcal{S}$ can then output to $\mathcal{E}$ exactly $\mathcal{A}$'s output in the real-world. Thus, there does not exist any environment $\mathcal{E}$ that can distinguish between interaction with $\mathcal{A}$ and the Teechain channel protocol, from interaction with $\mathcal{S}$ and $\mathcal{F}_{Teechain}$.

□

Next, we define balance security in the ideal-world, and prove that $\mathcal{F}_{Teechain}$ captures it.

A.4 Balance security in the Ideal-World

We proved that for any environment $\mathcal{E}$ the ideal-world and the real-world executions are indistinguishable. Therefore, we can define the security interest of a payment channel in the ideal-world, and prove that the ideal-world execution achieves this desired property. Since both the ideal and the real world are indistinguishable to any environment, then both the definition of balance security, and the proof that it holds in the ideal-world will also hold in the real-world, thus concluding the proof.

Notations and Definitions. We define our security goal to be *balance security*. Intuitively, this means that at any point in time, an honest user u can choose to settle a channel c_{id} with another user v, even if v is corrupt or crashes arbitrarily, and receive her balance on the ledger without the ability of v or any other user w to affect the outcome.

To formally discuss *balance security* we first define a few notations and definitions:

An *execution* $\sigma = (ev_1, ev_2, \ldots, ev_n) = ((op_1, ret_1), (op_2, ret_2), \ldots, (op_n, ret_n))$ is a series of events ev, each consists of a call op to $\mathcal{F}_{Teechain}$ and its return value from $\mathcal{F}_{Teechain}$, ret. Each event might also result in the change of the state, i.e., the internal variables maintained by $\mathcal{F}_{Teechain}$.

$L_t(u)$ is the balance on the Ledger for user u at time t, i.e., if any user v calls $\mathsf{getLedgerBalance}_v(u)$ at time t, then $L_t(u)$ is the returned value.

The initial time 0 is a time in an execution in the ideal-world prior to any calls to $\mathcal{F}_{Teechain}$, with some initial value on the Ledger for the users in the run, e.g. users u, v, w with some initial values $L_0(u), L_0(v), L_0(w)$ respectively. σ_t is the prefix of the execution σ from time 0 to t.

We denote $payments_t(u)$ be the set of all the amounts of successful pay calls to $\mathcal{F}_{Teechain}$ from u that returned *success* during σ_t:

$$payments_t(u) = \{\mathsf{val} \mid \exists i, c_{id}, v, \mathsf{val}, pending\_payment\_id : ev_i \in \sigma_t, op_i = \mathsf{pay}_u(c_{id}, v, \mathsf{val}), ret_i = (success, pending\_payment\_id)\}$$

Let $paid_t(u)$ be the sum of all payments in $payments_t(u)$, i.e.,

$$paid_t(u) = \sum_{\mathsf{val} \in payments_t(u)} \mathsf{val}$$

We denote $receivedPayments_t(u)$ to be a set of all the amounts of successful receivePayment calls to $\mathcal{F}_{Teechain}$ from u that returned *success* during σ_t:

$$receivedPayments_t(u) = \{\mathsf{val} \mid \exists i, pending\_payment\_id, c_{id}, \mathsf{val} : ev_i \in \sigma_t, op_i = \mathsf{receivePayment}_u(pending\_payment\_id), \exists j < i, \mathsf{val} : ev_j \in \sigma_t, op_j = \mathsf{pay}_u(c_{id}, \mathsf{val}), ret_j = (success, pending\_payment\_id)\}$$

Let $receivedPayments_t(u)$ be the sum of all such received payments in $receivedPayments_t(u)$, i.e.

$$rcvd_t(u) = \sum_{\mathsf{val} \in receivedPayments_t(u)} \mathsf{val}$$

We are now ready to define the *perceived balance* in the ideal-world of user u.

Definition A.2 (Perceived Balance in the Ideal-World)**.** *The perceived balance of user u at time t is defined as*

$$perceivedBal_t(u) = L_0(u) + rcvd_t(u) - paid_t(u)$$

We are now ready to formally define balance security.

Definition A.3 (Balance Security)**.** *We say an algorithm satisfies* balance security *if for any prefix σ_t for $t \geq 0$, and for any well-behaved user u in the system, u can preform a series of operations, possibly interleaved with operations of other users, that will be completed in a finite time t', after which at any time $t'' \geq t'$: $L_{t''}(u) \geq perceivedBal_t(u)$.*

A user u is honest or well-behaved if she follows the series of operations of the algorithm. We note that definitions definition A.3 and definition A.1 are equal to the environment $\mathcal{E}$ as the ideal-world and the real-world are indistinguishable to it.

Theorem A.1. *[Balance Security Theorem] The ideal functionality $\mathcal{F}_{Teechain}$ achieves balance security.*

In order to show that $\mathcal{F}_{Teechain}$ achieves balance security we need to show an algorithm, i.e., a series of operations that an honest user u has to preform in order to receive her perceived balance $perceivedBal_t(u)$.

We define the following sets, based on the internal variables of $\mathcal{F}_{Teechain}$:

1. All the deposit ids in D that are not currently associated with any channel at time t:

$$innerDeposits_t(u) = \{deposit\_id | \exists deposit\_id, isSymmetric : D(deposit\_id) = (\mathsf{val}, u, \perp, isSymmetric)\}$$

2. All the channel ids that have not been settled yet until time t. i.e.:

$$innerChannels_t(u) = \{c_{id} | \exists c_{id}, v, amountV, isSymmetric : C(c_{id}) = (u, v, amountU, amountV, isSymmetric) \vee C(c_{id}) = (v, u, amountV, amountU, isSymmetric), PendingChannels(c_{id}) \neq u\}$$

3. All the ledger payment ids *ledger_payment_id* that are associated with u and have not been placed on the ledger yet:

$$innerPendingLedgerPayment_t(u) = \{ledger\_payment\_id | \exists ledger\_payment\_id : PendingLedgerPayments(ledger\_payment\_id) = (u, \mathsf{val})\}$$

In order for u to receive her perceived balance $perceivedBal_t(u)$, we describe an algorithm, i.e., a series of calls to $\mathcal{F}_{Teechain}$.

1. u places a set of remove operations for each $deposit\_id \in innerDeposits_t(u)$:

$$OPS_1 = \{\mathsf{removeDeposit}_u(deposit\_id) | deposit\_id \in innerDeposits_t(u)\}$$

Lemma 2. *If u places the calls to $\mathcal{F}_{Teechain}$ described in OPS_1, then the return value of each call is success with some ledger payment id value ledger_payment_id.*

Proof. The if statement in the removeDeposit algorithm (Alg. 6, Line 31) will be true for any deposit id $deposit\_id \in innerDeposits_t(u)$, which means that $\mathcal{F}_{Teechain}$ will create a new ledger payment id *ledger_payment_id* and return *success* with *ledger_payment_id* (Line 35). □

Lemma 3. *User u has all the deposit ids $deposit\_id \in innerDeposits_t(u)$, such that she has the ability to invoke all the calls in OPS_1.*

Proof. OPS_1 is defined on a set of deposit ids *deposit_id* such that there exists the entry $D(deposit\_id) = (\mathsf{val}, u, \perp, isSymmetric)$. The only option for such an entry to be generated, is if during σ_t $\mathsf{addDeposit}_u(\mathsf{val})$ was invoked by u. When the call returns successfully, $\mathcal{F}_{Teechain}$ returns $(success, ledger\_payment\_id)$ to u. Thus, at time t, u already has all the deposit ids she needs in order to invoke the removeDeposit calls in OPS_1. □

2. u places a set of settle operations for each channel id $c_{id} \in innerChannels_t(u)$:

$$OPS_2 = \{\mathsf{settle}_u(c_{id}) | c_{id} \in innerChannels_t(u)\}$$

Lemma 4. *If u places the calls to $\mathcal{F}_{Teechain}$ described in OPS_2, then the return value of each call is success with some ledger payment id value ledger_payment_id.*

Proof. The if statement of the settleChannel algorithm (Alg. 6, Line 108) will be true for any c_{id} in $innerChannels_t(u)$, thus $\mathcal{F}_{Teechain}$ will return *success* with a ledger id *ledger_payment_id* (Line 123). □

Lemma 5. *User u has all the channel ids $c_{id} \in innerChannels_t(u)$ such that she can invoke all the settleChannel calls in OPS_2.*

Proof. OPS_2 is defined as settleChannel calls for channel ids c_{id} such that $C(c_{id})$ exists with user u. The only option for such an entry to be generated is if u invoked $\mathsf{openChannel}_u(c_{id})$ during σ_t or invoked $\mathsf{acceptChannelOpen}_u(c_{id})$. Thus, at time t, u has all the channels ids she needs in order to invoke the settleChannel calls as defined in OPS_2. □

We proved that if u places the calls to $\mathcal{F}_{Teechain}$ described in OPS_1 and in OPS_2, then the return value of all these calls is $(success, ledger\_payment\_id)$.
We denote by t_1 the time in which all the calls in $OPS_1 \cup OPS_2$ return successfully.
Let us define a set of the return values for each operation in $OPS_1 \cup OPS_2$ at t_1:

$$RET = \{ledger\_payment\_id | \exists i : ev_i = (op_i, ret_i) \in \sigma_{t_1}, op_i \in OPS_1 \cup OPS_2, ret_i = (success, ledger\_payment\_id)\}$$

3. u places a set of acceptLedgerPayment operations for each $ledger\_payment\_id \in RET \cup innerPendingLedgerPayment_t(u)$:

$$OPS_3 = \{\mathsf{acceptLedgerPayment}_u(ledger\_payment\_id) | ledger\_payment\_id \in RET \cup innerPendingLedgerPayment_t(u)\}$$

Lemma 6. *If u places the calls described in OPS_3 then $\mathcal{F}_{Teechain}$ returns success for each of them.*

Proof. All the pending payment ids of the calls described in OPS_3 are in *PendingPayments* as proved in Lemma 2 and Lemma 4, thus when u places acceptLedgerPayment with those pending *pending_payment_id*, $\mathcal{F}_{Teechain}$ will return *success*. □

Definition A.4 (balance security algorithm)**.** *The suffix for u in order to receive $perceivedBal_t(u)$ in the prefix σ_t is:*

$$OPS_u \triangleq (OPS_1, OPS_2, OPS_3)$$

We now move to show that by invoking the calls in Definition A.4, any user can receive her perceived balance, thus proving Theorem A.1.

Let $innerChannelBalance_t(u)$ be the set of all the capacities of all the open channels user u has with other users in the system at a given time t:

$$\begin{aligned} innerChannelBalance_t(u) = \{amountU \mid \\ \exists c_{id}, v, amountV, isSymmetric : \\ (C(c_{id}) = (u, v, amountU, amountV, isSymmetric) \\ \vee\ C(c_{id}) = (v, u, amountV, amountU, isSymmetric)), \\ PendingChannels(c_{id}) \neq u\} \end{aligned}$$

Let $innerChannelBalanceSum_t(u)$ be the sum of all the capacities in $innerChannelBalance_t(u)$:

$$innerChannelBalanceSum_t(u) = \sum_{\mathsf{val} \in innerChannelBalance_t(u)} \mathsf{val}$$

Let $innerDepositBalance_t(u)$ be the set of the amounts of deposits that user u added and not removed, and are not associated with any channel at a given time t, i.e.:

$$\begin{aligned} innerDepositBalance_t(u) = \{\mathsf{val} \mid \\ \exists deposit\_id, isSymmetric : \\ D(deposit\_id) = (\mathsf{val}, u, \bot, isSymmetric)\} \end{aligned}$$

Let $innerDepositBalanceSum_t(u)$ be the sum of all amounts in $innerDepositBalance_t(u)$:

$$innerDepositBalanceSum_t(u) = \sum_{\mathsf{val} \in innerDepositBalance_t(u)} \mathsf{val}$$

Let $innerPendingLedgerOpsSum_t(u)$ be a set of all pending ledger operations from user u at a given time t, i.e.:

$$\begin{aligned} innerPendingLedgerOpsSum_t(u) = \\ = \{\mathsf{val} \mid \exists ledger\_payment\_id : \\ PendingLedgerPayments(ledger\_payment\_id) = (u, \mathsf{val})\} \end{aligned}$$

Let $innerPendingLedgerOpsSum_t(u)$ be the sum of all the amount of the ledger payment operations in $innerPendingLedgerOpsSum_t(u)$:

$$innerPendingLedgerOpsSum_t(u) = \sum_{\mathsf{val} \in innerPendingLedgerOpsSum_t(u)} \mathsf{val}$$

Let $innerPendingDeposits_t(u)$ be a set of all amounts of deposits in the process of being dissociated from a channel, i.e., all deposits in *PendingDeposits*:

$$\begin{aligned} innerPendingDeposits_t(u) = \\ = \{\mathsf{val} \mid \exists deposit\_id, c_{id}, isSymmetric : \\ PendingDeposits(deposit\_id) = (u, isSymmetric), \\ D(deposit\_id) = (\mathsf{val}, u, c_{id}, isSymmetric)\} \end{aligned}$$

Let $innerPendingDepositsSum_t(u)$ be the sum of all the amounts of pending deposits in $innerPendingDeposits_t(u)$:

$$innerPendingDepositsSum_t(u) = \sum_{\mathsf{val} \in innerPendingDepositsSum_t(u)} \mathsf{val}$$

Definition A.5. *Let $stateBalance_t(u)$ be the balance of u, as defined by the internal state of $\mathcal{F}_{Teechain}$ at a given time t, i.e.:*

$$\begin{aligned} stateBalance_t(u) = L_t(u) + innerChannelBalanceSum_t(u) + \\ + innerDepositBalanceSum_t(u) + \\ + innerPendingLedgerOpsSum_t(u) + \\ + innerPendingDepositsSum_t(u) \end{aligned}$$

We begin by showing, that at any given time t the state balance of u is the same as the perceived balance as defined in Definition A.2.

Proposition 1. *At any execution σ the inner balance of u and the perceived balances are equal, i.e.:*

$$perceivedBal_t(u) = stateBalance_t(u)$$

Proof. We will prove this by induction on the execution σ.

Inductive base. In the beginning of the execution σ at $t = 0$ all the internal variables of $\mathcal{F}_{Teechain}$ are $\bot$, i.e., for any internal variable f of $\mathcal{F}_{Teechain}$, and for any x: $f(x) = \bot$. Thus:

$$\begin{aligned} innerChannelBalance_0(u) &= \varnothing \\ innerDepositBalance_0(u) &= \varnothing \\ innerPendingLedgerOpsSum_0(u) &= \varnothing \\ innerPendingDepositsSum_0(u) &= \varnothing \end{aligned}$$

Which means that:

$$\begin{aligned} innerChannelBalanceSum_0(u) &= 0 \\ innerDepositBalanceSum_0(u) &= 0 \\ innerPendingLedgerOpsSum_0(u) &= 0 \\ innerPendingDepositsSum_0(u) &= 0 \end{aligned}$$

And: $stateBalance_0(u) = perceivedBal_0(u)$

Inductive step. Let us assume that in step $i < t$: $stateBalance_i(u) = perceivedBal_i(u)$. We show that after the next event at step $i + 1$: $stateBalance_{i+1}(u) = perceivedBal_{i+1}(u)$.

First we note that the inner balance as defined above does not change if $\mathcal{F}_{Teechain}$ returns *fail* for any call, as the if statement in the beginning of each of its calls will cause it to return *fail* and change nothing, and in particular, not change $L_i(u)$.

We go over all the event types, each of them a result of a call to $\mathcal{F}_{Teechain}$, and show that for each of them, under the induction assumption for step i, it also holds for step $i + 1$, i.e.:

$stateBalance_{i+1}(u) = perceivedBal_{i+1}(u)$

- **getLedgerBalance** This algorithm does not affect any variable of $\mathcal{F}_{Teechain}$.
- **acceptLedgerPayment** This algorithm takes valfrom *PendingLedgerPayments* at time

i and adds it to $L_{i+1}(u)$, then it removes $PendingLedgerPayments(ledger\_payment\_id)$, i.e.:

$$innerPendingLedgerOpsSum_{i+1}(u) = innerPendingLedgerOpsSum_i(u) - amount$$

$$L_{i+1}(u) = L_i(u) + \mathsf{val}$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

- **addDeposit** This algorithm deducts val from $L_i(u)$ and adds an entry $D(depositEntry)$ with val, i.e.:

$$innerDepositBalanceSum_{i+1}(u) = innerDepositBalanceSum_i(u) + \mathsf{val}$$

$$L_{i+1}(u) = L_i(u) - \mathsf{val}$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

- **removeDeposit** This algorithm removes $D(deposit\_id)$ and adds amount to $PendingLedgerPayments(ledger\_payment\_id)$, i.e.:

$$innerPendingLedgerOpsSum_{i+1}(u) = innerPendingLedgerOpsSum_i(u) + \mathsf{val}$$

$$innerDepositBalanceSum_{i+1}(u) = innerDepositBalanceSum_i(u) - \mathsf{val}$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

- **openChannel** This algorithm does not affect the inner balance and therefore at step $i+1$ the inner and perceived balances are the same as in step i.
- **acceptChannelOpen** This algorithm does not affect the inner balance and therefore at step $i+1$ the inner and perceived balances are the same as in step i.
- **associateDeposit** This algorithm changes the third argument of $D(deposit\_id)$ to c_{id}, which logically means that the deposit $deposit\_id$ is now associated with channel c_{id}.
  By doing so, it moves val from $innerDepositBalanceSum_i(u)$ to $innerChannelBalanceSum_{i+1}(u)$, i.e.:

$$innerDepositBalanceSum_{i+1}(u) = innerDepositBalanceSum_i(u) - \mathsf{val}$$

$$innerChannelBalanceSum_{i+1}(u) = innerChannelBalanceSum_i(u) + \mathsf{val}$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

- **acceptAssociateDeposit** This algorithm does not affect the inner balance and therefore at step $i+1$ the inner and perceived balances are the same as in step i.
- **dissociateDeposit** This algorithm deducts val from the balance of u at time i in channel c_{id} and adds it to $PendingDeposits(deposit\_id)$ at $i+1$, i.e.:

$$innerChannelBalanceSum_{i+1}(u) = innerChannelBalanceSum_i(u) - \mathsf{val}$$

$$innerPendingDepositsSum_{i+1}(u) = innerPendingDepositsSum_i(u) + \mathsf{val}$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

- **acceptDissociate** This algorithm does not affect the inner balance and therefore at step $i+1$ the inner and perceived balances are the same as in step i.
- **ackDissociate** This algorithm removes the entry $PendingDeposits(deposit\_id)$ and changes the third argument of $D(deposit\_id)$ to $\perp$, which logically means that the deposit is not associated with any channel:

$$innerPendingDepositsSum_{i+1}(u) = innerPendingDepositsSum_i(u) - \mathsf{val}$$

$$innerDepositBalanceSum_{i+1}(u) = innerDepositBalanceSum_i(u) + \mathsf{val}$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

- **pay** This algorithm deducts val from the balance of u in channel c_{id} at time i, thus changing $stateBalance_i(u)$, but it also deducts val from $perceivedBal_i(u)$, i.e.:

$$innerChannelBalanceSum_{i+1}(u) = innerChannelBalanceSum_i(u) - \mathsf{val}$$

$$paid_{i+1}(u) = paid_i(u) - \mathsf{val}$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

- **receivePayment** This algorithm adds val to the balance of u in channel c_{id} at time i, thus changing $stateBalance_i(u)$, but it also adds val to $perceivedBal_i(u)$, i.e.:

$$innerChannelBalanceSum_{i+1}(u) = innerChannelBalanceSum_i(u) + \mathsf{val}$$

$$rcvd_{i+1}(u) = rcvd_i(u) + \mathsf{val}$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

• **settleChannel** This algorithm takes the current balance of u in channel c_{id}, and the sum of the deposits in the process of dissociation and deducts the total amount from u ands it as a pending ledger operation, i.e.:

$$innerChannelBalance_{i+1}(u) = 0$$

$$innerPendingDepositsSum_{i+1}(u) = 0$$

$$innerPendingLedgerOpsSum_{i+1}(u) = \\ = innerPendingLedgerOpsSum_i(u) + \\ + innerChannelBalanceSum_i(u) + \\ + innerPendingDepositsSum_i(u)$$

$$stateBalance_i(u) = stateBalance_{i+1}(u) = \\ = perceivedBal_i(u) = perceivedBal_{i+1}(u)$$

This concludes the inductive step, we proved that $stateBalance_t(u) = perceivedBal_t(u)$ for any time t during the execution σ. □

Next we show that for any open channel c_{id} the sum of the balances on both sides of the channel is less or equal to the sum of the deposits associated with the channel.

We first denote by $deposits_t(u)$ all the deposits that at a given time t are associated with a given channel c_{id}:

$$deposits_t(u) = \{\mathsf{val} | \exists deposit\_id, isSymmetric, u : \\ D(deposit\_id) = (\mathsf{val}, u, c_{id}, isSymetric), \\ PendingDeposits(deposit\_id) = \bot\}$$

We denote by $depositsSum_t(c_{id})$ the sum of the amounts of the deposits that are associated at a given time t with channel c_{id}:

$$depositsSum_t(c_{id}) = \sum_{\mathsf{val} \in deposits_t(u)} \mathsf{val}$$

We denote by $channelCapacity_t(c_{id})$ the sum of the capacities of the two users u and v which have a channel between them with channel id c_{id} at time t, i.e., for channel entry $C(c_{id}) = (u, v, amountU, amountV, isSymmetric)$ we denote:

$$channelCapacity_t(c_{id}) = amountU + amountV$$

Proposition 2. *At any given time t during the execution σ the sum of the deposits associated with a given channel c_{id} is always greater or equal to the balances of both users of the channel u and v, i.e.,:*

$$channelCapacity_t(c_{id}) \leq depositsSum_t(c_{id}) \quad (1)$$

Proof. We will prove this by induction on the execution σ:

Inductive base. At the initial step 0: $\forall c_{id} : C(c_{id}) = \bot$. Therefore, there are no open channels in system at all.

Inductive step. We assume that at step i the induction assumption holds for all entries in C, i.e.,

$$channelCapacity_0(c_{id}) \leq depositsSum_0(c_{id}).$$

We will prove that at step $i + 1$ the proposition holds as well.

Let us look at all the operations in $\mathcal{F}_{Teechain}$. We note that the channel balance as defined above does not change if $\mathcal{F}_{Teechain}$ returns *fail* for any call, as the if statement in the beginning of each of its calls will cause it to return *fail* does not change any internal variable of $\mathcal{F}_{Teechain}$. Thus, we go over all operations that return *success*:

- **getLedgerBalance** This call does not affect the balance of any channel in C.
- **acceptLedgerPayment** This call does not affect the balance of any channel in C.
- **acceptLedgerPayment** This call does not affect the balance of any channel in C.
- **addDeposit** This call does not affect the balance of any channel in C.
- **removeDeposit** This call does not affect the balance of any channel in C.
- **openChannel** In this call a new channel entry in $C(c_{id})$ is generated in the form of $(u, v, 0, 0, \bot)$. Thus, no deposit is associated at step i with channel c_{id}, which means that $depositsSum_{i+1}(c_{id}) = channelCapacity_{i+1}(c_{id}) = 0$.
- **acceptChannelOpen** The only effect of this call on $C(c_{id})$ is that it changes the last argument of the entry $C(c_{id})$ from $\bot$ to $\top$. This means that if at step i Equation (1) holds then it also holds at step $i + 1$.
- **associateDeposit** In any successful call associateDeposit$(c_{id}, deposit_id)$, $\mathcal{F}_{Teechain}$ adds the deposit val to either $amountU$ or $amountV$, and changes the third argument of $D(deposit\_id)$ to c_{id}, i.e., at step $i + 1$: the val of $deposit\_id$ will be in $deposits_{i+1}(u)$, and $depositsSum_{i+1}(c_{id}) = depositsSum_i(c_{id}) + \mathsf{val}$ and $channelCapacity_{i+1}(c_{id}) = channelCapacity_i(c_{id}) + \mathsf{val} \Rightarrow channelCapacity_{i+1}(c_{id}) \leq depositsSum_{i+1}(c_{id})$.
- **acceptAssociateDeposit** This call does not affect the balance of any channel in C as it only changes the last argument of $D(deposit\_id)$ from $\bot$ to $\top$.
- **dissociateDeposit** In this case val is deducted from either $amountU$ or $amountV$, and $PendingDeposits(deposit\_id)$ entry is generated, i.e., $depositsSum_{i+1}(c_{id}) = depositsSum_i(c_{id}) - \mathsf{val}$ and $channelCapacity_{i+1}(c_{id}) = channelCapacity_i(c_{id}) - \mathsf{val} \Rightarrow channelCapacity_{i+1}(c_{id}) \leq depositsSum_{i+1}(c_{id})$.
- **acceptDissociate** This algorithm only changes $PendingDeposits(deposit\_id) = (u, \bot)$ to $(u, \top)$ and does not affect the balance of any channel in C.
- **ackDissociate** This algorithm removes $PendingDeposits(deposit\_id)$ and also changes the third argument of $D(deposit\_id)$ to $\bot$, thus the deposit was not in $deposits_i(u)$ at step i and is not in $deposits_{i+1}(u)$ at step $i + 1$.
- **pay** This call deducts val from either $amountU$ or $amountV$ in $C(c_{id})$. In addition, in this call $\mathcal{F}_{Teechain}$ adds a new entry to $PendingPayments$ with a new generated $pending\_payment\_id$, i.e.,

$PendingPayments(pending\_payment\_id) = (v, c_{id}, \mathsf{val}$. This means that $channelCapacity_{i+1}(c_{id}) = channelCapacity_i(c_{id}) - \mathsf{val}$, $deposits_{i+1}(u) = deposits_i(u) \Rightarrow channelCapacity_{i+1}(c_{id}) < deposits_{i+1}(u)$

- **receivePayment** In this call, $\mathcal{F}_{Teechain}$ adds the valin $PendingPayments(pending\_payment\_id)$ to $amountU$ or $amountV$ in $C(c_{id})$, and then removes $PendingPayments(pending\_payment\_id)$.
  In order for the call receivePayment to return *success*, there has to be at step i an entry $PendingPayments(pending\_payment\_id)$ in the form of $(v, c_{id}, \mathsf{val})$. The only call in which $\mathcal{F}_{Teechain}$ adds a new entry to *PendingPayments* is pay. This pay call needs to be called by either one of the parties in channel c_{id} and $\mathsf{val}$is deducted in that call from $amountU$ or $amountV$.
  In addition, no other call deducts valonly from the right hand side of Equation (1), which means that, $channelCapacity_i(c_{id}) + \mathsf{val} \leq depositsSum_i(c_{id})$, $channelCapacity_{i+1}(c_{id}) = channelCapacity_i(c_{id}) + \mathsf{val} \Rightarrow channelCapacity_{i+1}(c_{id}) \leq depositsSum_{i+1}(c_{id})$.
- **settleChannel** This calls settles the channel c_{id}, and can be called twice:
  - If this is the second time settleChannel is called, then $C(c_{id})$ is removed, thus $channelCapacity_{i+1}(c_{id})$ is undefined at $i + 1$.
  - If this is the first call to settleChannel, then after the calls ends at step $i + 1$, $C(c_{id})$ is updated s.t. $amountU = 0$.
    In addition, all the deposits s.t. $\exists deposit\_id, isSymmetric : PendingDeposits(deposit\_id) = (u, isSymmetric)$ are removed from D. The only deposits which have an entry in *PendingDeposits* are deposits with $deposit\_id$ s.t. $\mathsf{dissociateDeposit}_u(deposit\_id)$ was called during σ_i, i.e., $\mathsf{val}$was already deducted from $channelCapacity_i(c_{id})$.
    This means that $channelCapacity_{i+1}(c_{id}) = channelCapacity_i(c_{id}) - amountU = amountV$, $depositsSum_{i+1}(c_{id}) = depositsSum_i(c_{id})$, thus, $channelCapacity_{i+1}(c_{id}) \leq depositsSum_{i+1}(c_{id})$.

□

Finally, we prove that at any given time t, any user u has the ability to receive $stateBalance_t(u)$ by preforming the operations of the balance algorithm A.4.

Proposition 3. *If a user u preforms the operations described in balance security algorithm A.4 as a suffix to the prefix σ_t, interleaved with operation of other users, then for any time $t'' \geq t'$ such that $op_{t''} = \mathsf{getLedgerBalance}(u)$ and $ret_{t''} = (success, amount)$, then $\mathsf{val} = stateBalance_t(u)$.*

Proof. Let us look at the for sets of operations consisting the balance security algorithm (definition A.4): $OPS_u = (OPS_1, OPS_2, OPS_3)$.

- ***innerDepositBalance***$_t(u)$ Let us look at OPS_1. This set consists of $\mathsf{removeDeposit}_u$ calls to $\mathcal{F}_{Teechain}$ for each $deposit\_id \in innerDeposits_t(u)$. $innerDepositBalanceSum_t(u)$ is defined as the sum of all those deposits. We proved in Lemma 2 that $\mathcal{F}_{Teechain}$ returns $(success, ledger\_payment\_id)$ for all these calls, and for each call adds the deposit amount $\mathsf{val}$to $PendingLedgerPayments(deposit\_id)$. This means that at time t_1 when all calls in OPS_1 have been called then:

$$innerDepositBalanceSum_{t_1}(u) = 0$$
$$innerPendingLedgerOpsSum_{t_1}(u) =$$
$$= innerPendingLedgerOpsSum_t(u) +$$
$$+ innerDepositBalanceSum_t(u)$$

- ***innerChannelBalance***$_t(u)$***, innerPendingDeposits***$_t(u)$ Let us look at OPS_2. This set consists of $\mathsf{settleChannel}_u(c_{id})$ calls for each channel $c_{id} \in innerChannels_t(u)$. $innerChannelBalanceSum_t(u)$ is defined as the sum of all u's open channels' balances, and $innerPendingDeposits_t(u)$ is defined as all the deposits in the process of dissociation from a channel. We proved in Lemma 4 that $\mathcal{F}_{Teechain}$ returns $(success, ledger\_payment\_id)$ for all these calls, and for each call adds u's channel balance $amountU$ to $PendingPayments(deposit\_id)$.
  In addition OPS_2 does not change any existing entries in *PendingLedgerPayments*, therefore not changing $innerPendingLedgerOpsSum_{t_1}(u)$. Thus, at time t_2 when all calls in OPS_2 have been completed:

$$innerChannelBalanceSum_{t_2}(u) =$$
$$= innerPendingDepositsSum_{t_2}(u) = 0$$

$$innerPendingLedgerOpsSum_{t_2}(u) =$$
$$= innerPendingLedgerOpsSum_{t_1}(u) +$$
$$+ innerChannelBalanceSum_t(u) +$$
$$+ innerPendingDepositsSum_t(u)$$

- ***innerPendingLedgerOpsSum***$_t(u)$ This set consists of $\mathsf{acceptLedgerPayment}_u$ for each ledger payment ids $ledger\_payment\_id$ that were the return values of the operations in $OPS_1 \cup OPS_2$ in addition to all pending ledger operatios in $innerPendingLedgerPayment_t(u)$. This means that at a time t_3 when the calls in OPS_3

finish successfully (as proved in lemma 6):

$$innerPendingLedgerOpsSum_{t_3}(u) = \\ = innerPendingLedgerOpsSum_{t_2}(u) - \\ - innerChannelBalanceSum_t(u) - \\ - innerDepositBalanceSum_t(u) = 0$$

$$L_{t_3}(u) = L_t(u) + \\ + innerChannelBalanceSum_t(u) + \\ + innerDepositBalanceSum_t(u) + \\ + innerPendingDepositsSum_t(u) + \\ + innerPendingLedgerOpsSum_t(u)$$

We note that t_3 is the time after all the operations in OPS_u have been invoked and returned successfully, and that $L_{t_3}(u) = stateBalance_t(u)$. Therefore, if any user calls getLedgerBalance(u), $\mathcal{F}_{Teechain}$ will return $(success, \mathsf{val})$ such that $\mathsf{val} = stateBalance_t(u)$. □

We proved that at any point in time a user u can preform a series of calls to $\mathcal{F}_{Teechain}$ and receive her inner balance. Since the inner balance is always equal to u's perceived balance and we showed that the ability of u to preform these operations fo not affect the perceived balance of other users in the system then any user can choose to preform these operations and receive their balance.

We recall Theorem A.1:

[Balance Security Theorem] The ideal functionality $\mathcal{F}_{Teechain}$ achieves balance security.

This concludes our proof to the theorem, and since the ideal-world and the real-world are indistinguishable to any external environment $\mathcal{E}$ then Teechain achieves balance security.

A.5 Multi-hop payments

Here we show that multi-hop payments satisfy balance security. As long as a multi-hop payment is not completed, the perceived balance of a user might be either post-payment or pre-payment (see §4.3). Thus, we define the perceived balance of a user in a multi-hop payment between u and v, where u is routing a payment of valto v. For execution trace σ let: t_1 be the time in σ at which u enters stage lock of the protocol; $t_2 > t_1$ be the time in σ at which v enters stage lock; $t_3 > t_2$ the time in σ at which v enters stage idle; and $t_4 > t_3$ the time in σ at which u ends the protocol and enters stage idle. See Fig. 3.

The users' *perceived balances* are as follows: For u, the perceived balance is: before t_1 as if valwas not paid; after t_4 as if valwas paid; between t_1 and t_4 either option is acceptable. For v, the perceived balance is: before t_2 as if valwas not paid; after t_3 as if valwas paid; between t_2 and t_3 either option is acceptable. The perceived balance of any intermediate party in the multi-hop payment does not change.

We prove Theorem A.1 by showing that every node p in a multi-hop payment (including u and v) can unilaterally reclaim their perceived balance at any point in time. We further show that if p settles, then all the channels of the multi-hop payment will always consistently settle the in either the pre-payment or post-payment state. Note that single channel payments do not interfere with multi-hop payments, as all the channels in the multi-hop payment are locked (§4.3).

Stage: idle. If p is in stage idle, then all other nodes of the payment are either in stage idle or lock. Node p and all other nodes can only obtain the pre-payment settlement transaction and subsequently stop the protocol. In this stage, the perceived balance of both u and v reflects the pre-payment state, thus satisfying balance security.

Stage: lock. If p is in stage lock, all other nodes are either (i) in stage idle and in stage lock, or (ii) in stage lock and in stage sign. All nodes can only settle their channels at the pre-payment state.

Stage: sign. If p is in stage sign, all nodes are either (i) in stage lock and in stage sign, or (ii) in stage sign and in stage prepayment. Case (i): If any node ejects, it settles its channels in the pre-payment state. This prevents progress and no node will reach the prepayment stage. Node p can then similarly eject and settle its channels in the pre-payment state. Case (ii): Any node in the prepayment stage might eject with $\overline{\tau}$. In this case, all channels will be settled in post-payment state.

Stage: prepayment. If p is in stage prepayment, all nodes are either (i) in stage sign and in stage prepayment, or (ii) in stage prepayment and in stage update. Case (i): Any node in stage sign may eject and settle its channels in the pre-payment state. Node p can then present this settlement transaction to its TEE as *PoPT* and obtain pre-payment settlement transactions for its channels. Node p can also voluntarily eject and obtain $\overline{\tau}$. Placing $\overline{\tau}$ onto the blockchain fails if one of the channels was already settled, in which case p can obtain settlement transactions for its channels as above. Case (ii): Any node can eject and settle the multi-hop payment with $\overline{\tau}$. If nodes have reached update, then all nodes passed stage sign, therefore none can generate local settlements.

Stage: update. If p is in stage update, all nodes are either (i) in stage prepayment and in stage update, or (ii) in stage update and in stage postUpdate. Case (i): All nodes can eject and settle the multi-hop payment with $\overline{\tau}$. None can generate individual settlements. Case (ii): Nodes in postUpdate can only settle their channels individually at post-payment state. Node p can present its TEE with single-channel settling transactions as *PoPT* and obtain settlement transactions to terminate in post-update state.

Stage: postUpdate. If p is in stage postUpdate, all nodes are either (i) in stage update and in stage postUpdate, or (ii) in stage postUpdate and in stage idle. Case (i): Nodes in

the update stage can voluntarily settle the entire multi-hop payment with $\overline{\tau}$. Nodes in stage postUpdate can settle their local channels at post-payment, and provide node p with *PoPT* to do the same. Case (ii): All nodes can only settle their local channels at post-payment.

Stage: unlock. When p returns to the unlock stage, other nodes are either in unlock, or some are in stage update; all nodes can only settle their channels at the post-payment state.

With this, we conclude that Teechains multi-hop payment satisfy balance security: (i) if either u or v are in the initial idle stage, they are both only able to settle pre-payment; (ii) between stage lock and postUpdate, both u and v may settle their channels in either pre-payment or post-payment state. However, they will always consistently settle the same state; (iii) once reaching stage unlock, both u and v will settle the post-payment state. The balance for any intermediate nodes does not change during the course of the payment routing. Thus, all participants in the multi-hop payment protocol are always able to reclaim their perceived balance.

This concludes our proof. We proved theorem A.1 and showed that the Teechain protocol (both channel and multi-hop payments) guarantee balance security.