NOCUST – A Securely Scalable Commit-Chain

Rami Khalil
Imperial College London
Liquidity Network

Arthur Gervais
Imperial College London
Liquidity Network

Guillaume Felley
Liquidity Network

Abstract

To scale permissionless blockchains, one avenue is to enact transactions outside, or off- the blockchain, but to secure them *by* a parent-chain with collateral. Payment channels are one such instantiation, but suffer from expensive channel setups, fragmented deposit costs and further challenges.

We present NOCUST, a 2nd-layer commit-chain, secure against an adversary that attempts to double-spend assets while controlling a set of malicious users, the commit-chain operator, or both. NOCUST does not require an additional consensus mechanism and the smart contract of a parent-blockchain acts as a means of efficient dispute resolution, via either provable consistency through zkSNARKs or users auditing once in a time-window the commit-chain's integrity.

NOCUST (i) does not require a parent-chain transaction for peers to join, (ii) achieves delayed transaction finality without additional collateral, (iii) achieves instant finality with less collateral than payment channels and (iv) does not require recipients of transactions to be online. Users do not need to be constantly online to monitor their state, minimizing the need for watchtowers. NOCUST tempers the danger of channel closing timeouts of payment channels by enforcing fair mass exits. We show how NOCUST scales in practice to over one billion users on a PoW blockchain.

1 Introduction

Since the beginning of centralized banking in Mesopotamia [1], financial intermediaries evolved as middlemen between parties that have surplus capital and others that desire access to liquid funds. Such financial intermediaries traditionally operate as custodians, as they (temporarily) hold the transmitted funds, and therefore are entrusted with enacting secure transaction policies.

While the emergence of decentralized Proof-of-Work (PoW) ledgers have portrayed a mechanism of performing financial transactions without a centralized intermediary, low-throughput, volatility of transaction fees and privacy constraints are fundamentally hindering the practical use of such ledgers. To improve transaction utility, different classes of blockchain scaling solutions are being pursued. Alternative consensus mechanisms [2, 3] or sharding [4] typically introduce different trust assumptions.

In this paper we focus on upper layer scaling solutions such as payment channel networks (PCN) [5] which reduce the load on the blockchain ledger by performing operations securely off-chain. Numerous contributions address the performance characteristics of payment channel networks [6, 7, 8, 9, 10]. While PCN should improve transaction throughputs, they face multiple challenges: (i) channel establishment requires at least one expensive and slow parent-chain transaction, (ii) PCN rely on complex routing topologies which need to be setup and maintained and (iii) funds allocated to a payment channel are typically bound between two parties and can only be transferred further over fee-contingent routes. The likely biggest usability bottleneck, however, is the condition that a transaction recipient *needs to be online to receive an incoming transfer* - parting from the provisions of traditional blockchain transactions.

This work We propose a novel 2nd-layer construction NOCUST, which is non-custodial (i.e. it doesn't hold the user's assets), and by design can achieve the same transaction throughput as custodian banks or credit card processors. Contrary to side chains [11], NOCUST does not require an additional consensus mechanism and relies on a parent ledger. The NOCUST operator *commits* at regular intervals *checkpoints* of the 2nd-layer state to its parent-ledger (hence we term NOCUST a *commit-chain*) — or is forced to seize operation. Our security analysis shows how commit-chain users securely maintain custody of their funds, even in the absence of the operator's availability or under its adversarial behavior and collusion with all other commit-chain users.

A user can join a NOCUST commit-chain without interaction of the parent-chain, receive and forward commit-chain assets to any NOCUST user, and forego the need for a costly parent-chain initialization. Defining transaction finality as

the point at which a transfer is irreversible, we demonstrate how NOCUST achieves delayed finality after a time window, without additional collateral by the commit-chain operator. We also show how a NOCUST commit-chain achieves *instant* finality with collateral stake set up by the NOCUST operator. We show that this collateral requirement is reduced compared to PCN. A NOCUST instance can allocate its collateral to all its users using only one parent-chain transaction.

To demonstrate its practicality, we show how NOCUST scales to billions of users, while the dispute mediation and withdrawal costs remain below 0.5 USD. The operational parent-chain fees for a commit-chain operator remain below 1.5 USD per month, irrespective of the number of commit-chain transfers and users. We show how users can operate lightweight clients that only need to store 4 kb plus their transfers of e.g., the last 72 hours.

The main contributions of our work are as follows:

Merkleized Interval Tree: To the best of our knowledge, we're the first to specify the details of a scalable *commit-chain*. At the heart of NOCUST lies a novel multi-layered *Merkleized interval tree* which is carefully designed to provide an efficient *account-based* 2$^{\text{nd}}$-layer ledger (efficient deposit, withdraw, dispute, client requirements etc.).

Provable Commit-Chain Consistency: We design an additional NOCUST version, where the commit-chain integrity is enforced by a constant-sized zkSNARK proof on the parent-chain — foregoing the need for users to actively dispute consistency.

Active and Passive Delivery: We design two transaction deliverance models, one where the recipient acknowledges a transfer, and one where the receiver of a NOCUST transaction can securely remain offline during reception.

Instant Onboarding: We specify an elegant user-onboarding procedure, whereby users can join a NOCUST commit-chain without any parent-chain transaction, then receive and forward assets to other user within NOCUST.

Instant Transactions: NOCUST achieves delayed finality for users' transactions without additional collateral. *Instant finality* is achieved by staking the estimated transaction volume of a time period. This collateral is allocated efficiently with only one parent-chain transaction for all users.

Fair Mass Exit: In the event of a mass-exit, or operator misbehaviour, all user accounts are closed fairly.

The remainder of the paper is organized as follows. Section 2 provides the background and reviews related work, while Section 3 provides an overview of NOCUST. Section 4 presents the details of NOCUST and Section 4.5 increases NOCUST's security with the use of zkSNARKs. Section 5 provides the security analysis, and Section 6 evaluates NOCUST in practice, showing how to scale beyond one billion users. We conclude the paper in Section 7.

2 Background and Related Work

In this section, we provide the necessary background on permissionless blockchains and payment channel networks.

The majority of permissionless blockchains [12, 13] rely on *Proof-of-Work* (PoW) [14], a computationally expensive puzzle. They allow mutually mistrusting peers to interact, without relying on a centralized custodian, and any peer can join or leave the network at any time. The blockchain provides a coarse-grained time stamping service that can act as an electronic payment solution solving the double-spending problem — the problem of spending an electronic coin multiple times. The central costs associated to PoW blockchains is the requirement that all peers are need to be made aware of all transactions to not be vulnerable to double-spending. PoW blockchains support up to 10 transactions per second [15], for an in-depth background we refer to [16, 17].

2.1 Payment Channels

Payment channels establish direct P2P payment channels between two parties [18]. A channel is like a private two-party ledger, instantiated and closed with a respective parent-chain transaction. A channel is collateralized with a parent-chain security deposit, the counter-parties do not need to trust each other when accepting off-chain transactions. Transaction costs are reduced, because channel transactions avoid costly parent-chain transactions (besides channel establishment and closure). Payment channel constructions either rely on blockchain based time locks [18], or on punishment, i.e. if one party misbehaves, the other party can claim funds of the channel [5]. For a pair of users that are not directly connected via a payment channel, a payment can be routed along a set of payment channels, i.e. over a payment channel network. Linked payments are *atomically* executed or invalidated, intermediate hops are eligible to collect payment forwarding fees. We outline in the Appendix in Section A.1 the fundamental drawbacks of payment channel networks.

2.1.1 Improvements and Alternative Constructions

Existing PCN are still in early development and allow for several improvement proposals [6, 7, 19, 20, 8]. A multitude of works cover privacy enhanced PCN designs [21, 22, 23, 24, 25], while Malavolta *et al.* [26] study in detail the tradeoffs between concurrency and privacy in payment channel networks. Perun [9] reduces communication complexity through virtual channels. Teechan [27] and Teechain [28] trade the need for a blockchain clock with a trusted hardware assumption which enables very efficient off-chain payments. Plasma [29] proposes to connect an UTXO ledger with a parent account-based ledger. With the incorporation of a UTXO model comes all of the inefficiencies of transaction history validation and the inflexibility of transaction output expenditure. Costs for securely using accounts increase with a grow-

ing UTXO set. Plasma does not specify a mechanism for mitigating risk of reversal on yet to be finalized transactions, parting its guarantees from those of PCN. NOCUST features a more efficient account-based ledger that mitigates these issues, permitting users to more efficiently verify the integrity of their accounts, requiring less parent-chain data for dispute and enabling instant transaction finality. Several contributions focus on state channels that execute arbitrary code off-chain [30, 6, 31], while Pisa [32] allows to outsource channel maintenance if e.g. a peer crashes.

2.2 Zero Knowledge Proofs

Zero Knowledge Proofs of Knowledge [33] allow a prover to prove to a verifier that a certain statement is true, without the need to reveal any other information. zkSNARK [34] systems enhance ZKPs to be non-interactive, i.e. require no interactions between the prover and the verifier except transmission of the proof. zkSNARK are succinct, i.e. the proof length is reasonably small (a few hundred bytes), and proofs can be verified with few computational costs (and therefore e.g. within a smart contract). zkSNARK, however, require a trusted setup phase. To prove in a commit-chain that a particular state transition is valid, the chain rules must first be converted into a (pre-processed) zkSNARK.

Computation → Circuit → R1CS → QAP → zkSNARK

The first step is to translate the desired computation into the smallest possible circuits, i.e. a mathematical representation. Next, a Rank 1 Constraint System (R1CS) makes sure that the input and output values to each gate within the circuit remain sound. Instead of being pre-occupied to verify nearly one constraint per wire in a circuit, which is very costly, the circuit is represented as a Quadratic Arithmetic Program (QAP) [35]. Here, the constraints are represented between polynomials, instead of numbers. Verifying that the two polynomials of a gate match at one randomly chosen point provides a high probability that the proof is correct. The public reference string of a zkSNARK provides, in encrypted form, the point of the QAP that should be evaluated. Neither the verifier nor the prover should therefore know this point, and should therefore be incapable to construct fraudulent polynomials/proofs. From a QAP, a zkSNARK is created by the prover with random shifts of the original polynomials. In this work, we heavily utilize the contributions in [36, 37, 38], where preprocessed zkSNARKS are designed to be embeddable in one another in a technique called *recursive composition*. Nonetheless, we conjecture that the methods introduced in this work could be applicable in other ZKP systems, especially when converted from recursive computations to iterative ones.

3 NOCUST Overview

We provide an overview of NOCUST's architecture, operations and assumed attacker models. A NOCUST operator is a centralized operator that coordinates and ratifies the execution of payments in a pool of collateral deposited into a parent-chain smart contract. The access to this collateral pool is moderated by the smart contract, which expects to periodically receive a commitment to the state of the 2nd-layer ledger from the operator — hence the name commit-chain. This state contains each user's account, and the commitment to this large state is constructed in such a way that it is efficient to prove and verify that a user's account was updated correctly by the operator to the smart contract, such that withdrawals and deposits can be securely enacted.

Untrusted Centralized Intermediaries The NOCUST operator becomes only a single point of failure for availability, but not custody of funds or integrity of operation. The centralized model provides a significant advantage in terms of communication cost between its parties compared to decentralized ledger networks and PCN. The complete disappearance of a NOCUST commit-chain operator, or a malicious attempt by it to double-spend or seize user funds only leads to its halt, and does not affect the ability of users to exit using the smart contract with their latest confirmed balance. Despite adversarial or malfunctioning operator behaviour, users do not lose custody of their digital assets.

3.1 System Model

Our system model features the following entities:

Parent-Chain: We assume a parent-ledger, an integrity protected and immutable root of trust, that allows for the convenient deployment of tamper-proof smart contracts. The ledger contains a global view of accounts with balances and transactions, and extra associated data. Each account in the ledger is controlled by a private key.

User: A user owns at least one private key/account in the ledger and acts as recipient and payer of cryptocurrency assets on the parent-ledger and on the commit-chain.

Commit-Chain Smart Contract: The smart contract acts as supervisor of the NOCUST operator and verifies its correct operation, automatically verifies consistency (in the zkSNARK model), or accepts dispute initializations from users and halting the operator in case of misbehaviour.

Commit-Chain Operator: The operator server mediates the communication between users, and is required to commit at regular time intervals a constant-sized checkpoint of all commit-chain accounts to the smart contract.

We also assume an underlying communication network, where all the participants can communicate directly with the operator and (if needed) among each other.

3.2 High-Level Operation

In this section, we outline NOCUST's high-level operations. NOCUST supports *active* and *passive* asset transfers. Under active delivery, the recipient is assumed to be online to sign a transaction receipt (a common assumption in PCN). Under passive delivery, the receipt of a transaction is not sought, and the transaction is communicated to the recipient at a later stage, either by the operator or out-of-band by the payer.

Figure 1 outlines the high-level operations, for both, active and passive delivery of commit-chain transactions represented as IOU. The payer (1) deposits assets from the parent- to the commit-chain and (2) authorizes a debit towards the recipient. Under active delivery, the operator (3) notifies the recipient of the incoming transfer, the recipient (4) approves the state update to their account. The server (5) signs both updated states and (6) gives its respective signatures to both users, and (7) forwards the recipients'. The NOCUST server (8) submits each *eon* a **periodic constant-sized checkpoint** aggregating the user balances to the smart contract. The users (9) observe the checkpoint submissions and verify *once* every *eon* it's integrity. When utilizing zkSNARK, the smart contract verifies the complete correctness of the submitted checkpoint, users only verify its liveness.

3.2.1 Bootstrapping

We outline the following setup operations of NOCUST.

zkSNARK Setup: Generating a set of public parameters for any party to use to prove or verify a zkSNARK must be done such that the full set of initialization parameters, referred to as "toxic-waste", is unrecoverable [39, 40][1].

Operator Setup: The operator deploys a smart contract on the parent-chain and initializes a genesis checkpoint.

User Registration: Given its private key, a user signs a registration message and sends this message to the operator.

3.2.2 Cross-Chain Conversion

Assets are converted between their parent- and commit-chain representations with a parent-chain transaction. One unit of a parent-chain asset is equal to the mapped commit-chain asset, and enforced by the smart contract. Commit-chain assets cannot be minted without parent-chain backing.

Deposit: Parent-Chain $\rightarrow$ Commit-Chain: Parent-chain coins deposited in the smart contract mint NOCUST coins.

Withdraw: Parent-Chain $\leftarrow$ Commit-Chain: Commit-chain coins are withdrawn from the smart contract to the parent-chain. NOCUST supports non-collaborative full withdrawal, without required approval by the operator, and collaborative partial withdrawal. Withdrawals take up to two *eons* to be declared final by the smart contract. To improve this, the operator makes an instantly approved withdrawal by sending the requested amount from its own separate pool of funds and later reclaiming the requested amount when it is approved by the smart contract.

Commit-Chain Data Structure: Commit-chain assets are managed within the multi-layered Merkelized interval tree (cf. Figure 2), separating individual user account balances in exclusive, non-intersecting interval allotments. The constant-sized tree root plus the total commit-chain assets represent the checkpoint. Every internal Merkle tree node is annotated with the continuous interval its two children occupy. This allows for a straightforward verification that the sum of all accounts in the tree match the actual parent-chain balance of the smart contract. Double-spends become obvious and fractional reserve scenarios are impossible.

3.2.3 Transmission of Commit-Chain Assets

A payer spends commit-chain assets to a recipient as follows:

Debit Authorization: The payer approves a state update to their own account via a new IOU, containing the details of the transaction, a unique randomly generated nonce, the transferred amount, the intended recipient and if the approval of the recipient is mandatory (i.e. active delivery).

Transmission: The payer forwards the signed state update and IOU to the operator, which notifies the recipient.

Credit Authorization: For active delivery, the recipient approves a state update to their own account that includes this new IOU, and returns its approval to the NOCUST server.

Ratification: The operator, now in possession of both payer and recipient state update authorizations (active delivery), or just the payer's authorization (passive delivery), signs the updated states and reveals its signatures.

3.2.4 Parent-Chain Settlement

We define the time between two checkpoints as an *eon*. Once the periodic *eon* has passed, the operator collects all not-yet settled IOUs, reconstructs the Merkelized interval tree (cf. Figure 2), generates a proof that this interval tree represents a correct transition from the last state, and commits the constant-sized checkpoint to the parent-chain.

3.2.5 Disputes

NOCUST is a challenge-response protocol, where users report fraudulent operators, while the smart contract mediates.

Consistency Challenge (non-zkSNARK model): Each user is only preoccupied to verify their respective balance interval of a checkpoint. Verification requires the partial

[1] Recovery would allow the forgery of proofs that are acceptable to verifiers, but containing proving parameters that would be rejected if they were revealed. To deter this possibility, the multi-party computation, used to populate the parameters of Zcash, can be used to generate a set of parameters that are reusable by any NOCUST instance.

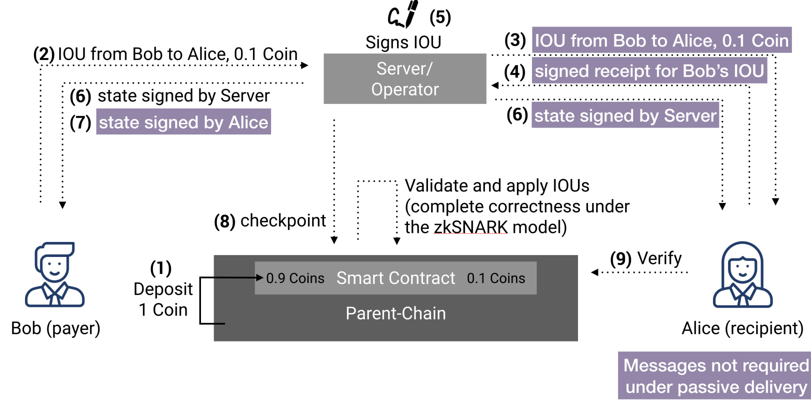

Figure 1: High-level flow of operation of NOCUST, for both, active and passive delivery of commit-chain transactions.

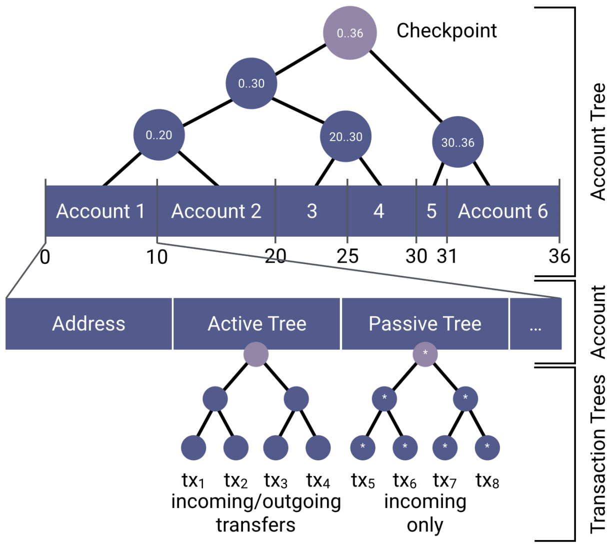

Figure 2: Merkleized interval tree representing the commit-chain assets. Here are 6 accounts with their respective commit-chain balance (36 total commit-chain coins). Note, not a simple Merkle sum tree, intermediate nodes are further annotated with the balance intervals (cf. Section 4.3.1).

Merkle tree from the operator and to compare the state to the locally stored ratified state. Only the most recent checkpoint can be challenged. In case of misbehaviour, users issue a challenge to NOCUST smart contract, which the server must promptly answer with valid information.

Liveness Challenge (zkSNARK model): The zkSNARK allows to provably verify the commit-chain consistency by the smart contract (cf. Section 4.5), obsoleting the consistency challenge. Users are still required to verify the that the state transitions of their accounts are the latest.

Data Availability Challenge: Users are assumed to come online once per *eon*, and request their partial Merkle tree needed to withdraw assets from the commit-chain. If the operator is non-responsive, users challenge the operator through the smart contract to provide the requested data. To prevent griefing attacks from the users towards the operator, those challenges can be restricted to be only initiated by the owners of the account, or a set of explicitly whitelisted third parties such as watchtowers [32].

3.3 Main Properties

NOCUST provides the following properties for the entities:

Ledger: NOCUST requires periodic checkpoint transactions, irrespective of the number of IOU's exchanged over the operator, increasing the ledgers' transaction throughput.

User: Users convert assets between parent- and commit-chain and transmit 2$^{\text{nd}}$-layer funds via an IOU. Payer receive non-repudiable receipts, while recipients can enforce IOUs on the smart contract. Concerning security, users remain custodians of their assets, even in the presence of a malicious operator or a collusion with all other users. Users can initialize disputes with the operator. Even if the operator disappears, users are able to withdraw their funds. Regarding privacy, IOUs are visible to the payer, operator and recipient, but can be hidden from the remaining commit- and parent-chain users. A recipient can instantly and trustlessly accept an IOU as payment if the operator allocated a sufficient amount of collateral towards the recipient.

Operator: The non-custodial operator is occupied with accounting the respective user account balances, submitting checkpoints on time and managing its collateral attributions. An operator cannot perform successful double-spending attacks, assuming users would challenge the operator's behaviour on-time within an *eon*. The operator cannot create funds without being halted by the smart contract.

Collateral Requirements for Instant Finality If the NOCUST operator chooses to disappear, or malfunctions prior to a checkpoint, not-yet included commit-chain transfers are lost. We define instant IOU finality such that at the moment of the IOU ratification reception, a recipient is certain to be

able to retrieve the amount of the incoming assets (and no adversary can steal or e.g. double-spend the IOU). The operator allocates an amount of collateral to each user using the Merkleized interval tree to claim in case of failure (cf. Section 5.2). The amount required is equivalent to the incoming transaction volume of the user within two *eons*.

Liveness Requirements For trustless operation of the operator, users are required to be online at least once within an *eon*. Otherwise, a malicious operator can alter the balance interval allocation within the Merklelized interval tree when zkSNARKS are not utilized. If users publish their commit-chain data, others can issue challenges on their behalves, similar to PCN watchtowers [32].

3.4 Attacker Models

We consider the adversary to control all network communication between users, and between the user and the operator (Dolev-Yao [41]). Users, however, can always read and write to the parent-chain which is considered non-malicious[2], but might be congested[3]. We assume that a user is online once within an *eon* (unless outsourced to a watchtower [32]).

Malicious payer: A malicious user might attempt to sign two conflicting debit authorizations, i.e. to spend more than the user would be eligible to. As the operator is benign and required to counter-sign each transfer, the user cannot misbehave without the operators' collusion.

Malicious payer colluding with the operator: A malicious operator can collude with a user to try to double-spend assets: (1) by creating more commit-chain assets, i.e. by extending the Merkleized interval tree, or by (2) shrinking an allocated interval of a user.
The smart contract rejects a checkpoint that accounts for more commit-chain assets than deposited in the contract. If the NOCUST operator incorrectly manipulates a user's account balance, the user will be able to detect and dispute this using the smart contract. In our zkSNARK model, the contract would immediately reject an inconsistent state.

Malicious user attacking the operator: A malicious user can open many accounts on the commit-chain and flood the operator with data availability challenges. To minimize the impact of this attack, the operator can (i) only allow a limited number of users to be able to dispute a particular account (e.g. a watchtower), (ii) require human verification on account opening, and (iii) require the user to subsidize the response costs of a data availability challenge.

[2]e.g. there is no ongoing 51% attack or active double-spending/selfish-mining attacks against the NOCUST smart contract.

[3]For example, if a checkpoint submission would cost 1k USD (current costs are about 0.22 USD), then the operational costs under a 36 hour *eon* are 20k USD per month.

4 NOCUST Details

This section outlines the details of the commit-chain $\not\subset$.

4.1 Operational Requirements

To disqualify $\not\subset$ from being a trusted custodian of its user's assets and the payments it facilitates, it would have to provide the following guarantees:

1. Funds may not be transacted without user authorization.
2. Transactions must always be executed correctly.
3. Users must always be able to independently withdraw their commit-chain balances to a parent-chain account.

These guarantees need to be provided by the system to an honest participant $\mathbb{P}_i$ regardless of the behavior of $\not\subset$ or of any other participant $\mathbb{P}_j$ $(i \neq j)$ (cf. proofs in Section 5). NOCUST utilizes the following components:

- A specialized account based commit-chain ledger $\mathbb{B}$.
- A smart contract $\mathbb{V}_{\not\subset}$ to manage the global $\mathbb{B}^G$.
- An operator $\mathbb{O}_{\not\subset}$ to manage the local $\mathbb{B}^L$.
- A recursive zkSNARK procedure for $\mathbb{V}_{\not\subset}$ to verify $\mathbb{B}^L$.
- A protocol for interaction between $\mathbb{P}$, $\mathbb{O}_{\not\subset}$, and $\mathbb{V}_{\not\subset}$.

4.2 The Commit-Chain Data Structure $\mathbb{B}$

In this section we present a scheme for managing local and global balance as well as transaction information in $\mathbb{B}$.

Separation of the Commit-Chain Ledger The commit-chain $\mathbb{B}$ is separated into the locally stored commit-chain $\mathbb{B}^L$, containing information related to balances and transfers performed through $\mathbb{O}_{\not\subset}$, and the globally stored parent-chain $\mathbb{B}^G$, which is comprised of information on balances and operations through $\mathbb{V}_{\not\subset}$. It is important to note that different parties may have different views of the contents of $\mathbb{B}^L$, but contents of $\mathbb{B}^G$ are assumed to be globally consistent.

Time Progression The information in $\mathbb{B}^L$ is committed to on $\mathbb{B}^G$ to make the transfers through $\mathbb{O}_{\not\subset}$ enforceable by $\mathbb{V}_{\not\subset}$ (i.e. allow a $\mathbb{P}_i$ to withdraw funds received through $\mathbb{O}_{\not\subset}$ from a $\mathbb{P}_j$ via $\mathbb{V}_{\not\subset}$). This constant-sized commitment is sent once periodically from $\mathbb{O}_{\not\subset}$ to $\mathbb{V}_{\not\subset}$, within a so-called *eon*. An *eon*, within the context of our scheme, is further divided into a fixed number of *eras* (for brevity equivalent to a block confirmation within the context of a blockchain). Consequently, an *eon* represents the amount of time for a fixed number of blocks to be generated. We also use the term *epoch* to denote a quarter of an *eon*. We use $\mathbb{B}(e)$ to refer to the state of the commit-chain ledger at *eon* number *e as of all eras passed*, and similarly for $\mathbb{B}^L(e)$ and $\mathbb{B}^G(e)$.

4.2.1 Stored Information

In the following we differentiate between locally and globally stored information for the commit-chain ledger $\mathbb{B}$.

Local Information For every *eon* e, for every $\mathbb{P}_i$, $\mathbb{B}^L(e)$ can store the following entries:

$A_i(e)$**:** Initially allotted balance of $\mathbb{P}_i$ for e.
$R_i^a(e-1)$**:** Total actively received by $\mathbb{P}_i$ in $e-1$.
$R_i^p(e-1)$**:** Total passively delivered to $\mathbb{P}_i$.
$S_i^a(e-1)$**:** Total sent on the commit-chain by $\mathbb{P}_i$.
$S_i^p(e-1)$**:** Passive sent transfer delivery offset for $\mathbb{P}_i$.
$T_i^a(e-1)$**:** Commit-chain transactions involving $\mathbb{P}_i$.
$T_i^p(e-1)$**:** Passively delivered transactions to $\mathbb{P}_i$.
$C_i(e+2)$**:** The amount exclusively allocated for $\mathbb{P}_i$ to claim in case the $\not\subset$ instance is halted during $e+2$.

Global Information For every *eon* e, for every $\mathbb{P}_i$, $\mathbb{B}^G(e)$ can store the following entries:

$\mathbb{D}_i(e)$**:** Total amount deposited by $\mathbb{P}_i$ during e.
$\mathbb{W}_i(e)$**:** Total withdraw amount requested by $\mathbb{P}_i$ during e.
$\mathbb{E}_i(e)$**:** Has an exit been initiated by $\mathbb{P}_i$ during e.
$\mathbb{X}_i^b(e)$**:** A challenge of the integrity of $\mathbb{P}_i$'s balance in $\not\subset$.
$\mathbb{X}_i^d(e)$**:** A challenge of the delivery of a transfer in $\not\subset$.
$\mathbb{U}_i(e)$**:** The **additional** amount exclusively allocated to $\mathbb{P}_i$ in case the $\not\subset$ instance is halted during *eons* e or $e+1$.

Through combining the locally stored and globally stored information, $A_i(e)$ is to be calculated as follows[4] in $\mathbb{B}$:

$$A_i(e) = A_i(e-1) + \mathbb{D}_i(e-1) + R_i^{a\cup p}(e-1) - \mathbb{W}_i(e-1) - S_i(e-1) \quad (1)$$

Additionally, For every *eon* e, $\mathbb{B}^G(e)$ can store a commitment (cf. Section 4.3.1) by $\mathbb{O}_{\not\subset}$ to the contents of $\mathbb{B}^L(e)$.

4.3 $\mathbb{T}^A_{\not\subset}$ Periodic Commitments

We now describe the data-structures and message formats that enable the efficient provable integrity of a NOCUST *eon*.

4.3.1 Merkleized Interval Tree-Structure

To provably account for the allotted balances $A_i(e)$ of each $\mathbb{P}_i$ at the beginning of an *eon* e, we design a novel *Merklelized interval tree* $\mathbb{T}^A_{\not\subset}(e)$. The Merkleized interval tree is similar to the augmented Merkle tree proposed by Luu *et al.* [42], yet built and utilized in completely different means. The nodes in this Merkle tree [43] are augmented to store the user balances in an efficient manner that allows $\mathbb{V}_{\not\subset}$ to securely verify the correct and exclusive allotment of funds by $\mathbb{O}_{\not\subset}$. A node $t_n(e)$ of $\mathbb{T}^A_{\not\subset}$ is structured as defined in Equation 2.

$$t_n(e) = \langle \text{offset}_n(e), \text{information}_n(e), \text{allotment}_n(e) \rangle \quad (2)$$

offset and allotment are both numeric values, while information is a cryptographic commitment to the information contained within this node. A leaf $t_i(e)$ is used to represent the commit-chain account of a $\mathbb{P}_i$ at *eon* e, whereby $\text{allotment}_i(e)$ is equal to $A_i(e)$ (cf. Section 4.2.1), and $\text{offset}_i(e)$ corresponds to the sum of the allotted balances of all participants ordered before $\mathbb{P}_i$ (cf. Equation 4).

$$\text{allotment}_i(e) = A_i(e) \quad (3)$$

$$\text{offset}_i(e) = \sum_{j<i} \text{allotment}_j(e) \quad (4)$$

$\text{information}_i(e)$ is the cryptographic hash of the parent-chain address of $\mathbb{P}_i$ and the commitment of the last balance update agreed to by $\mathbb{P}_i$ in the previous *eon*. More precisely,

$$\text{information}_i(e) = \{\text{address}_i, \text{update}_i^p(e-1), \text{update}_i^a(e-1)\} \quad (5)$$

where $\text{update}_i(e)$ represents the last state update of the commit-chain account of $\mathbb{P}_i$ at *eon* e (cf. Section 4.3.3). An internal node $t_u(e)$, with a left child $t_p(e)$ and a right child $t_q(e)$, is constructed per Equation 6 and Equation 7:

$$\text{allotment}_u(e) = \text{allotment}_p(e) + \text{allotment}_q(e) \quad (6)$$

$$\text{offset}_u(e) = \text{offset}_p(e) \quad (7)$$

$\text{information}_u(e)$ is a cryptographic commitment similar to that of an internal node of a Merkle tree but with the addition of $\text{offset}_q(e)$ as a third middle value.

$$\text{information}_u(e) = \{t_p(e), \text{offset}_q(e), t_q(e)\} \quad (8)$$

It's important to note that the middle value of $\text{offset}_q(e)$ is interchangeable with that of $\text{offset}_p(e)$ + $\text{allotment}_p(e)$ as they must be equal in correct instances of this structure.

4.3.2 Proof of Exclusive Allotment

For each $\mathbb{P}_i$ included in $\mathbb{T}^A_{\not\subset}(e)$, a proof of exclusive allotment $\tau_i^A(e)$ can be constructed. The main goal of this construct is to prove that $\mathbb{P}_i$ *exclusively* owns an allotment of size $A_i(e)$ within the pool of user funds covered by $\mathbb{T}^A_{\not\subset}(e)$. $\tau_i^A(e)$ is constructed similar to a regular Merkle tree membership proof, whereby the nodes adjacent to the path from the root to the leaf constitute the membership proof hash chain. However, in addition to the hashes of the nodes in the membership proof, a boundary value Ω is required for each node:

$$\Omega(t_i(e), t_n(e)) = \begin{cases} \text{offset}_n(e) & t_n(e) \text{ left child} \\ \text{offset}_n(e) + \text{allotment}_n(e) & t_n(e) \text{ right child} \end{cases} \quad (9)$$

[4] Equation 1 naturally reads as: The initially allotted balance for $\mathbb{P}_i$ for the current *eon* is equal to the sum of $\mathbb{P}_i$'s initially allotted balance, the parent-chain amount deposited in its favor, and the amount it received on the commit-chain in the previous *eon* minus the amount $\mathbb{P}_i$ requested for withdrawal and the amount it transferred out on the commit-chain in the previous *eon*.

The procedure of verifying $\tau_i^A(e)$ is similar to that of verifying set membership in a Merkle tree but the node reconstruction is done so according to the definitions of the $\mathbb{T}_{\not\subset}^A$ structure in Section 4.3.1 in conjunction with the Ω values. This bounds the size of a $\tau_i^A(e)$ to $\mathbb{O}(\log|\mathbb{P}|)$.

4.3.3 Monotonic $\mathbb{P}$-State Structure

The information in $\text{update}_i(e)$ contained in $\mathbb{T}_{\not\subset}^A(e)$ divided into an authorized portion, $\text{update}_i^a(e)$, and a passive portion, $\text{update}_i^p(e)$. The authorized portion must be accompanied by a signature from $\mathbb{P}_i$, the passive portion can be set by $\mathbb{O}_{\not\subset}$. The authorized portion $\text{update}_i^a(e)$ is structured as follows:

$$\text{update}_i^a(e) = \{T_i^a(e), S_i^a(e), R_i^a(e)\} \tag{10}$$

$T_i^a(e)$ is committed to using a Merkle tree where the leaves are the individual commit-chain transfers authorized by $\mathbb{P}_i$ during *eon* e. The Merkle tree used to create the commitment for $T_i^a(e)$ is not augmented. A commit-chain transfer $\mathfrak{T}$ is a tuple of the following information:

$$\mathfrak{T} := \langle eon, payer, recipient, nonce, amount, offset \rangle$$

The passive portion $\text{update}_i^p(e)$ is structured as follows:

$$\text{update}_i^p(e) = \{T_i^p(e), S_i^p(e), R_i^p(e)\} \tag{11}$$

The first notable difference is that $S_i^p(e)$ does not specify an amount, but rather an offset that is to be specified by $\mathbb{O}_{\not\subset}$ to allow $\mathbb{V}_{\not\subset}$ to secure passively delivered transfers. The second difference is how the commitment for $T_i^p(e)$ is constructed; $T_i^p(e)$ may only contain incoming passive transfers for $\mathbb{P}_i$ during e, and the commitment is constructed using the annotated Merkle tree structure from Section 4.3.1 where the following node definitions are used:

$$\text{allotment}_{\mathfrak{T}} = \mathfrak{T}.amount \tag{12}$$

$$\text{offset}_{\mathfrak{T}} = \sum_{T_i^p(e) \ni \mathfrak{T}' < \mathfrak{T}} \mathfrak{T}'.amount \tag{13}$$

$$\text{information}_{\mathfrak{T}} = \{\mathfrak{T}\} \tag{14}$$

In both cases, we refer to the proof of membership that a transfer $\mathfrak{T}$ belongs to a transaction set $T_i(e)$ as $\lambda(\mathfrak{T} \in T_i(e))$. A proof of membership in $T_i^p(e)$ would be accompanied by the respective Ω values, similar to a $\tau_i^A(e)$, and the root node allotment is the value of $R_i^p(e)$.

4.3.4 Proof of Exclusive Insurance Collateral

To guarantee instant transaction finality, $\mathbb{O}_{\not\subset}$ stakes collateral that is to be claimed by the recipients in case of failing to finalize transactions within two *eons*. To efficiently manage the allocations of the staked collateral, $\mathbb{O}_{\not\subset}$ commits each *eon* e to a simplified version of the $\mathbb{T}_{\not\subset}^A$ structure from Section 4.3.1, referred to as $\mathbb{T}_{\not\subset}^C$, that *exclusively* divides this collateral pool among members of $\mathbb{P}$. All funds in the pool covered by $\mathbb{T}_{\not\subset}^C$ are separated from those covered by $\mathbb{T}_{\not\subset}$.

In instances of $\mathbb{T}_{\not\subset}^C$, $\text{information}_i(e)$ is only comprised of address_i, and $A_i(e)$ is replaced by $C_i(e+2)$. We refer to constructs of the proof that $\mathbb{P}_i$ will be exclusively assigned a collateral portion of size $C_i(e+2)$ two *eons* after e as $\tau_i^C(e)$.

The funds in the $\mathbb{T}_{\not\subset}^C$ pool are only to be touched by a member of $\mathbb{P}$ in case the $\not\subset$ instance is halted. An exclusive collateral allotment proof $\tau_i^C(e)$ for $\mathbb{P}_i$ can be trivially constructed and secured using the same methods for constructing $\mathbb{T}_{\not\subset}^A$ and τ_i^A. Using this specification, $\mathbb{O}_{\not\subset}$ can increase individual collateral allotments on the parent-chain of members of $\mathbb{P}$ during the current *eon e*, and submit a complete re-assignment of all collateral that takes effect starting from $e+2$ using a *single constant-sized* commitment. The *eon* delay between the submission of a $\mathbb{T}_{\not\subset}^C$ and its enforcement ensures that a $\mathbb{P}_i$ does not accept a transfer during an *eon e* without certainty of whether the collateral allocated to itself insures instant finality or not for this transfer. When utilizing this, $\mathbb{O}_{\not\subset}$ is required to commit to a valid instance of $\mathbb{T}_{\not\subset}^C$ alongside each $\mathbb{T}_{\not\subset}^A(e)$ commitment and provide the respective proofs to each $\mathbb{P}_i$. If $\mathbb{P}_i$ does not learn $\tau_i^C(e-1)$ by the start of *eon e*, it should assume that $C_i(e+1)$ is zero, and reason about the affected transfers as if they are not guaranteed finality in case of the $\not\subset$ instance being halted.

4.3.5 Exit allocations

$\mathbb{O}_{\not\subset}$ is required to post another simplified version of the $\mathbb{T}_{\not\subset}^A$ structure from Section 4.3.1 that enables a $\mathbb{P}_i$ to exit from the $\not\subset$ instance with all of its funds. This structure is denoted by $\mathbb{T}_{\not\subset}^E$, and $\text{information}_i(e)$ is again only comprised of address_i. The purpose of this structure is to exclusively allocate the finalized balances of users who utilized $\mathbb{E}_i$ in $\mathbb{B}^G$. Once a $\mathbb{P}_i$ initiates an exit in *eon* $e-1$, $A_i(e)$ is to be set to zero in $\mathbb{T}_{\not\subset}^A(e)$, and its value is instead put in $\mathbb{T}_{\not\subset}^E(e)$, whose funds are also isolated. $\mathbb{V}_{\not\subset}$ makes sure that $\mathbb{O}_{\not\subset}$ correctly carries out this non-optional task and that every $\mathbb{P}_i$ learns their respective $\tau_i^E(e)$ so that the third requirement from Section 4.1 is fulfilled or the $\not\subset$ instance is halted. $\mathbb{P}_i$ then submits $\tau_i^E(e)$ to $\mathbb{V}_{\not\subset}$ in *eon* $e+1$ to finalize their exit with their balance.

4.4 Involved Participants

$\mathbb{V}_{\not\subset}$ **Parent-chain Verifier** The parent-chain component $\mathbb{V}_{\not\subset}$ acts as the moderator of the bridge between $\mathbb{B}^L$ and $\mathbb{B}^G$. We assume that its procedures are executed honestly by $\mathbb{BC}$, and it supports the following operations.

The $\mathbb{O}_{\not\subset}$ can *(1) commit once per* eon *e a* $\mathbb{T}_{\not\subset}^A(e)$, $\mathbb{T}_{\not\subset}^C(e)$ *and* $\mathbb{T}_{\not\subset}^E(e)$ for which the $\mathbb{V}_{\not\subset}$ can then *(2) verify a respective* $\tau_i^A(e)$, $\tau_i^C(e)$ *or* $\tau_i^E(e)$. A $\mathbb{P}_i$ can *(3) make a deposit into* $\not\subset$, by sending assets to $\mathbb{V}_{\not\subset}$. A $\mathbb{P}_i$ can send a request to $\mathbb{V}_{\not\subset}$

to *(4) initiate a withdrawal from* $\not\subset$ by providing a signed withdrawal authorization $\mathfrak{W}_i$ from $\mathbb{O}_{\not\subset}$, or *(5) request an exit from* $\not\subset$ without providing any data. *(6) A withdrawal can be confirmed* and the $\mathbb{V}_{\not\subset}$ issues a transfer from the balance pool it manages in favor of $\mathbb{P}_i$ with the requested amount after two eons. The operator can *(7) proxy a withdrawal* by transferring the requested amount from its own balance pool and rerouting the pending withdrawal destination to itself once it is confirmed. To moderate the validity of the committed $\mathbb{T}^A_{\not\subset}(e)$, $\mathbb{V}_{\not\subset}$ can *(8) open a balance update challenge* $\mathbb{X}^b_i(e)$ given a $\tau^A_i(e-1)$ and an $\text{update}_i(e-1)$ signed by $\mathbb{O}_{\not\subset}$ as inputs from a $\mathbb{P}_i$, or *(9) open a transfer delivery challenge* $\mathbb{X}^d_i(e)$ given an $\text{update}^a_j(e-1)$ signed by $\mathbb{O}_{\not\subset}$, and a transfer $\mathfrak{T}^j_i(e-1) \in T_j(e-1)$ as inputs from $\mathbb{P}_i$ or $\mathbb{P}_j$. Both challenge types can be closed once $\mathbb{O}_{\not\subset}$ presents the corresponding proofs of valid operation to $\mathbb{V}_{\not\subset}$ as input. In case the commit-chain is halted at *eon e*, i.e $\mathbb{T}^A_{\not\subset}(e)$, $\mathbb{T}^C_{\not\subset}(e)$ and $\mathbb{T}^E_{\not\subset}(e)$ are deemed invalid, $\mathbb{V}_{\not\subset}$ can *(10) transfer* $A_i(e-1) + C_i(e-2)$ *to* $\mathbb{P}_i$ given valid $\tau^C_i(e)$, $\tau^C_i(e-2)$ as input. $\mathbb{V}_{\not\subset}$ can *(11) transfer* $A_i(e-1)$ *to* $\mathbb{P}_i$ if $\mathbb{P}_i$ had requested an exit in *eon* $e-2$, and provides a valid $\tau^E_i(e-1)$, or *(12) transfer* $\mathbb{W}_i(e-2)$ *to* $\mathbb{P}_i$ if no $\mathbb{T}_{\not\subset}(e-1)$ was contested. A more detailed specification is presented in Appendix D.3.

$\mathbb{O}_{\not\subset}$ Commit-chain Operator The commit-chain component $\mathbb{O}_{\not\subset}$ acts as the facilitator of transfers between members of $\mathbb{P}$, and is not assumed as exempt from behaving dishonestly, but is designed to support the following operations.

As the operator of the $\not\subset$ instance, $\mathbb{O}_{\not\subset}$ can *(1) admit new members to* $\mathbb{P}$ by providing them with initially empty accounts. At the end of *eon* $e-1$ it *(2) creates* $\mathbb{T}^{A,C,E}_{\not\subset}(e)$ using all of the confirmed information from $e-1$, and *(3) commits the roots of* $\mathbb{T}^{A,C,E}_{\not\subset}(e)$ to $\mathbb{V}_{\not\subset}$. To prove to a $\mathbb{P}_i$ that its account in $\mathbb{B}^L(e)$ is being managed properly, and to guarantee insurance collateral in case of a halt, $\mathbb{O}_{\not\subset}$ must *(4) provide* $\tau^{A,C,E}_i(e)$ to a $\mathbb{P}_i$ upon request or risk a challenge being opened through $\mathbb{V}_{\not\subset}$. At its discretion, $\mathbb{O}_{\not\subset}$ is in charge of *(5) delivering transfers* $\mathfrak{T}^i_j(e)$ between any two $\mathbb{P}_i$, $\mathbb{P}_j$ provided the payer holds sufficient balance. For any $\mathbb{P}_i$, $\mathbb{O}_{\not\subset}$ must *(6) credit deposits* $\mathbb{D}_i(e)$*, debit withdrawals* $\mathbb{W}_i(e)$ *and facilitate exits* $\mathbb{E}_i(e)$ or risk facing challenges by any ill-affected $\mathbb{P}_i$ in $e+1$. $\mathbb{O}_{\not\subset}$ can *(7) facilitate partial withdrawals* $\mathbb{W}_i(e)$ through providing authorization messages $\mathfrak{W}_i(e)$. Selectively, the operator can *(8) proxy withdrawals* by transferring their requested amounts to the recipients for instant use and accept to be reimbursed by the later occurring confirmation of the original withdrawal. In such cases where challenges are issued, $\mathbb{O}_{\not\subset}$ must *(9) close challenges* $\mathbb{X}^b_i(e)$, $\mathbb{X}^d_i(e)$ within a timely manner, or face being halted by $\mathbb{V}_{\not\subset}$. A more detailed specification is presented in Appendix D.4.

$\mathbb{P}$ Users Members of $\mathbb{P}$ are the main parties interested in transferring funds to each other using the $\not\subset$ instance, and are designed to support the following operations.

First an interested $\mathbb{P}_i$ can *(1) join* $\not\subset$ by submitting a request to $\mathbb{O}_{\not\subset}$ for an initially empty account. After receiving this approval it can *(2) send and receive transfers* $\mathfrak{T}^i_j(e)$ to/from any other $\mathbb{P}_j$ in the same $\not\subset$ instance, *(3) deposit* $\mathbb{D}_i(e)$ funds into its $\not\subset$ account, or *(4) partially withdraw* $\mathbb{W}_i(e)$ funds from it by requesting a $\mathfrak{W}_i(e)$ from $\mathbb{O}_{\not\subset}$. As the custodian of its account, $\mathbb{P}_i$ must *(5) audit* $\tau^{A,C,E}_i(e)$ for each *eon e*, and *(6) issue challenges* $\mathbb{X}^b_i(e)$ *and* $\mathbb{X}^d_i(e)$ using $\mathbb{V}_{\not\subset}$ in case of any discrepancies or missing data. In case of the $\not\subset$ instance being halted, a $\mathbb{P}_i$ should *(7) recover funds* by providing the necessary proofs on the parent-chain. Lastly clients can *(8) request a full exit* using $\mathbb{V}_{\not\subset}$. A more detailed specification is presented in Appendix D.5.

4.5 Provably Consistent Checkpoints

In this section we present verification procedures for a non-interactive ZKP environment, allowing $\mathbb{V}_{\not\subset}$ to efficiently verify the consistency of every submitted $\mathbb{T}^A_{\not\subset}$ and prevent $\mathbb{O}_{\not\subset}$ from being able to submit a semantically invalid checkpoint.

We explicitly refer to two types of input and one type of output. *Moreover, there exists an implicit constant-sized prover output for every procedure which we refer to as* π. The verifier input and output are public values that are to be provided both when generating a proof for a procedure and when verifying it. The prover input is private to the prover and is to be used only when generating the prover output, π. Verifiers are required to obtain π if they wish to verify that a verifer input and output are accepted by the target procedure.

Algorithm 1: combiner

Verifier Input : $\mathbb{V}_L, \mathbb{V}_R, \mathbb{I}_L, \mathbb{I}_R, \mathbb{F}$
Prover Input : π_L, π_R
Verifier Output: $\mathbb{F}(\mathbb{O}_L, \mathbb{O}_R)$
$\mathbb{O}_L \leftarrow \mathbb{V}^{\pi_L}_L(\mathbb{I}_L)$
$\mathbb{O}_R \leftarrow \mathbb{V}^{\pi_R}_R(\mathbb{I}_R)$
return $\mathbb{F}(\mathbb{O}_L, \mathbb{O}_R)$

Algorithm 1 combines the results from two verification sub-procedures into one. Its verifier inputs are the two verifiers whose outputs are to be combined, the public inputs to those verifiers and the combination method of the outputs. The procedure's prover inputs are the two prover outputs of the to-be-combined subprocedures. *The size of the prover output* π *from the combiner remains constant even though it encapsulates* π_L *and* π_R. We concretize this abstract combination procedure to merge various sub-procedures into more useful ones through *nesting* zkSNARKS inside one another [36, 37, 38] without burdening the verifier with additional costs.

Transfer Delivery Consistency In the following we describe how to guarantee that every transfer $\mathfrak{T}^i_j$ debited in $\mathbb{T}^A_{\not\subset}$ from $\mathbb{P}_i$ was correctly credited in $\mathbb{T}^A_{\not\subset}$. Algorithm 4 (cf. Appendix) ensures that a single transfer delivery for a $\mathfrak{T}^i_j(e-1)$ is enforced in $\mathbb{T}^A_{\not\subset}(e)$. For brevity, we omit the differences between passive and active delivery. To verify that a larger set of transfers is entirely enforced, we compose a series of verifiers by parameterizing the combiner procedure described in Algorithm 1 as follows:

$\mathbb{F} = \text{hash}(\mathbb{O}_L \| \mathbb{O}_R)$, $\mathbb{I}_L = \mathbb{I}_R = t_{root}(e)$
$\mathbb{V}^0_L = \mathbb{V}^0_R = \text{verifyTxDelivery}$
$\mathbb{V}^N_L = \mathbb{V}^N_R = \text{combiner}(\mathbb{V}^{N-1}_L, \mathbb{V}^{N-1}_R, \mathbb{I}_L, \mathbb{I}_R, \mathbb{F})$

The output of the Nth toplevel combiner is the Merkle root of the set of transfers T^a_i whose enforcement is being proven. The definition leaves no room for sets whose size is not a perfect power of two, but we forgo discussing obvious remedies to this nuisance, such as using filler values, a different output combination method $\mathbb{F}$ or a different flavor of Algorithm 4.

State Update Consistency Algorithm 5 (cf. Appendix) is a nestable procedure that enforces the correct balance update per Equation 1, and the validation of Algorithm 4 on $T^a_i(e-1)$, for a $\mathbb{P}_i$. Algorithm 5 requires the pre-establishment of four verification procedures. $\mathbb{V}_D$, $\mathbb{V}_W$, and $\mathbb{V}_E$, are plug-in subroutines which return the values of $\mathbb{D}_i(e-1)$, $\mathbb{W}_i(e-1)$ and $\mathbb{E}_i(e-1)$ in $\mathbb{B}^G$. These procedures must be implemented to verifiably return $\mathbb{B}^G$ storage values from the parent chain $\mathbb{BC}$. We present an extended specification of the NOCUST ledger $\mathbb{B}$ in Appendix D.2 that permits the creation of such verifiers without the implementation of any $\mathbb{BC}$-specific secure storage introspection procedures. Lastly, $\mathbb{V}_T$ is defined to be the top-level instance of the transfer delivery proof combiner defined in Section 4.5. Enabling the verification then of an entire $\mathbb{T}^A_{\not\subset}$ commitment is accomplished through parameterizing the combiner from Algorithm 1 to output the reconstructed parent node of two verified neighbors, along with its offset and allotment such that the output from the root combiner would then be $0, t_{root}(e), \text{allotment}_{root}$ when the correct data is provided.

Non-collaborative Exit Consistency Algorithm 6 (cf. Appendix) can be combined and parameterized similar to Algorithm 5 to verify the integrity of a $\mathbb{T}^E_{\not\subset}$ commitment.

5 Security Analysis

In this section we analyze NOCUST's security.

5.1 Threat Model

We assume that the underlying $\mathbb{BC}$ allows to settle disputes on the integrity of $\not\subset$, and charges transaction fees. We consider these expenses as external to the balance of a $\mathbb{P}_i$ in $\not\subset$. We have designed NOCUST to minimize these expenses (cf. Section 4.3) and show their practicality in Section 6.

NOCUST is designed to prevent any honest member of $\mathbb{P}$ from losing any funds despite a strong set of adversarial capabilities. Further formalizing the attacker model from Section 3.4, we assume *(1)* an operator $\mathbb{O}_{\not\subset}$ and *(2)* $\mathbb{P}$ participants which can both receive incoming and send outgoing transfers. We assume the existence of an *irrational adversary* willing to sustain financial losses to cause honest parties to lose some or all of their funds in $\not\subset$. This irrational adversary may seize control of $\mathbb{O}_{\not\subset}$, some or all but one of $\mathbb{P}$, or a combination thereof, to attack an honest $\mathbb{P}_i$ not under its control. The adversary has full control of the identities associated with the compromised parties and may authorize any messages on their behalf or front-run any user input, but cannot violate the integrity of the honest users' identities. Moreover, an adversary may launch denial of service attacks that degrade the off-chain communication between $\mathbb{O}_{\not\subset}$ and members of $\mathbb{P}$, but may not compromise an honest $\mathbb{P}_i$'s communication with $\mathbb{BC}$, respectively $\mathbb{V}_{\not\subset}$. Moreover, we assume that the adversary is incapable of causing the underlying ledger layer $\mathbb{BC}$ to malfunction or misbehave. In the following discussion, we define malicious behavior as that which aims to cause an honest $\mathbb{P}_i$ to lose control of some or all of its funds in $\not\subset$ or cause an honest $\mathbb{O}_{\not\subset}$ to be forcibly shut down by $\mathbb{V}_{\not\subset}$.

5.2 Guarantees

In this section we explain how under the stated threat model, an honest $\mathbb{P}_i$ can securely maintain custody of its funds and ensure that its enacted transfers are correctly delivered in $\not\subset$, but will not be able to forcibly enact any new transfer $\mathfrak{T}^i_j(e)$ in the system without facilitation by $\mathbb{O}_{\not\subset}$. We also demonstrate how an honest $\mathbb{O}_{\not\subset}$ can sustain service under the malice of a subset of $\mathbb{P}$. We prove the security guarantees of NOCUST through proving that an honest $\mathbb{P}_i$ or honest $\mathbb{O}_{\not\subset}$ following the prescribed protocol may not end in a state where they cannot utilize $\mathbb{V}_{\not\subset}$ to enforce the integrity of $\not\subset$. We refrain from building a single comprehensive model of the system due to the many interactions present, and instead break down the system into components and prove their security properties. We argue that under the stated system model in Section 3.1 it suffices to prove the sanity of agent behavior in NOCUST due to the presence of $\mathbb{V}_{\not\subset}$ and the feasibility of deploying its functionality to operate honestly on $\mathbb{BC}$.

Exclusive Allotment NOCUST depends on a valid τ_i to guarantee to a $\mathbb{P}_i$ the exclusive allotment of a portion of size A_i in the funds managed by $\mathbb{V}_{\not\subset}$. We proceed to prove in Appendix C.1 that no valid instance of $\mathbb{T}^A_{\not\subset}$, $\mathbb{T}^C_{\not\subset}$, $\mathbb{T}^E_{\not\subset}$ or T^p_i may contain two overlapping allotments, and therefore that $\mathbb{V}_{\not\subset}$ cannot accept an invalid τ^A_i, τ^C_i, τ^E_i or $\lambda(\mathfrak{T}^i_j \in T^p_i)$.

Balance Custody An honest participant $\mathbb{P}_i$ of $\not\subset$ maintains custody of its balance because it may always resort to $\mathbb{V}_{\not\subset}$ in case $\mathbb{O}_{\not\subset}$ does not provide a valid $\tau_i^A(e)$ or $\tau_i^E(e)$ during the current *eon e*. $\mathbb{P}_i$ must maintain knowledge about its state to be able to utilize $\mathbb{V}_{\not\subset}$ for dispute regardless of how other $\mathbb{P}_j$ and $\mathbb{O}_{\not\subset}$ behave. By keeping track of every authorized $\text{update}_i^a(e)$ and every $\tau_i(e)$ from $\mathbb{O}_{\not\subset}$ for each *eon e* that passes after $\mathbb{P}_i$ entered $\not\subset$, $\mathbb{P}_i$ may always open $\mathbb{X}_i^b(e)$ in case $\mathbb{O}_{\not\subset}$ fails to provide a $\tau_i^A(e)$, or $\tau_i^E(e)$ in case of exit, with a correct exclusive $A_i(e)$. This guarantees that $\mathbb{P}_i$ is *always* able to enforce a provably correct update by $\mathbb{O}_{\not\subset}$ to its state, or halt $\not\subset$. As an exit may only be initiated and finalized through $\mathbb{V}_{\not\subset}$, no other $\mathbb{P}_j$ may attempt to lay a claim to any portion of A_i. A $\tau_i^E(e)$ may not be utilized to finalize any withdrawals until *eon* $e+1$ commences with no open challenges against the integrity of $\mathbb{T}_{\not\subset}^A(e)$ and $\mathbb{T}_{\not\subset}^E(e)$. This guarantees that only uncontested exclusive balance allotments in correct $\mathbb{T}_{\not\subset}^{A,C,E}(e)$ instances may be used to enact withdrawals and exits.

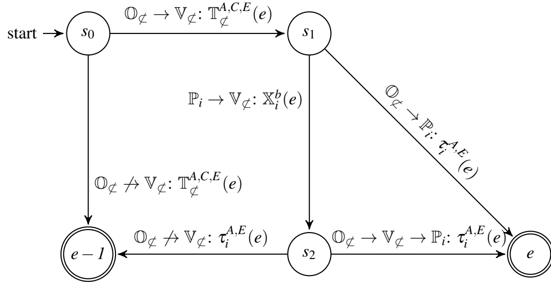

Figure 3: Finite state automaton capturing the custodian state of an honest $\mathbb{P}_i$ during an *eon e*. Given a $\mathbb{O}_{\not\subset}$'s commitment to $\mathbb{T}_{\not\subset}$, an honest $\mathbb{P}_i$ always knows a valid $\tau_i^A(e)$ either cooperatively with $\mathbb{O}_{\not\subset}$ or through $\mathbb{V}_{\not\subset}$. Terminal states denote which eon's balance $\mathbb{P}_i$ is given custody of.

We refer the reader to Appendix C.2 where we prove how a $\mathbb{P}_i$ in NOCUST can protect its funds to always reach a custodian state. For simplicity we omit states and transitions whereby $\mathbb{P}_i$ does not receive a valid $\tau_i(e)$ from $\mathbb{O}_{\not\subset}$ and chooses to not resort to $\mathbb{V}_{\not\subset}$ to demand its broadcast within the next *epoch*, as this behavior does not describe an honest $\mathbb{P}_i$. It is important to recall what $\mathbb{V}_{\not\subset}$ accepts as a valid response from $\mathbb{O}_{\not\subset}$ to a $\mathbb{X}_i^b(e)$ as stated in Section D.3. The $\text{update}_i^a(e-1)$ in the commitment must be as recent as that submitted in $\mathbb{X}_i^b(e)$ by $\mathbb{P}_i$, and must bear $\mathbb{P}_i$'s signature. This prevents $\mathbb{O}_{\not\subset}$ from attempting to commit an outdated or forged state, and provides $\mathbb{P}_i$ sufficient knowledge to enact any future delivery challenges.

Double-Spend Futility In the following discussion we refer to a double-spend as the adversary controlling a $\mathbb{P}_i$, with or without control of $\mathbb{O}_{\not\subset}$, to attempt any of the following:

1. Double-spend $\mathbb{P}_i$'s balance in $\mathbb{B}^L$
2. Spend $\mathbb{P}_i$'s balance in $\mathbb{B}^L$ and withdraw it from $\mathbb{B}^G$

In case the adversary lacks control of $\mathbb{O}_{\not\subset}$, attempting to double-spend only in $\mathbb{B}^L$ will be trivially prevented by an honest $\mathbb{O}_{\not\subset}$, and attempted withdrawals in *eon e* from $\mathbb{B}^G$ of funds spent in $e-1$ will also be cancelled by an honest $\mathbb{O}_{\not\subset}$ through $\mathbb{V}_{\not\subset}$. Even with control of $\mathbb{O}_{\not\subset}$, an adversary may not double-spend in the current *eon e* and able to construct valid $\mathbb{T}_{\not\subset}^{A,C,E}$ in the next *eon* $e+1$ correctly satisfying every member of $\mathbb{P}$, as valid instances of $\mathbb{T}_{\not\subset}^{A,C,E}$ guarantee exclusive allotments, the sizes of which must correspond to the confirmed balances expected by each honest member of $\mathbb{P}$. We refer the interested reader to Appendix C.3 for our proof.

Operational Integrity An adversary in control of some participants in $\mathbb{P}$ may maliciously open a set of challenges against $\mathbb{O}_{\not\subset}$ using $\mathbb{V}_{\not\subset}$. In NOCUST, there is no way for $\mathbb{V}_{\not\subset}$ to verify whether a participant is opening a challenge maliciously or not, and therefore $\mathbb{O}_{\not\subset}$ must answer every challenge opened. However, it is guaranteed that an honest $\mathbb{O}_{\not\subset}$ is able to close any challenge opened in $\mathbb{V}_{\not\subset}$, and a dishonest $\mathbb{O}_{\not\subset}$ that misconstructs $\mathbb{T}_{\not\subset}^{A,C,E}(e)$ will not be able to answer correct challenges in *e*.

The information required by an honest $\mathbb{O}_{\not\subset}$ to close a $\mathbb{X}_i^b(e)$ is $\tau_i^A(e)$, which is constructed by $\mathbb{O}_{\not\subset}$, $\tau_i^E(e)$ in case $\mathbb{E}_i(e-1)$ is set, and the $\text{update}_i(e-1)$ used in the construction. $\mathbb{O}_{\not\subset}$ always possesses sufficient knowledge to close any open challenges, and successfully does so if the committed $\mathbb{T}_{\not\subset}^{A,C,E}(e)$ correspond to the latest ratified contents of $\mathbb{B}(e-1)$, given the honesty of $\mathbb{V}_{\not\subset}$ in managing the pool of funds in the $\not\subset$ instance on $\mathbb{BC}$ (cf. Appendix C.4 for proof).

Instant Transaction Finality $\mathbb{V}_{\not\subset}$ is in charge of maintaining the insurance collateral pool balance $\mathbb{C}$ such that a valid $\tau_i^C(e)$ can always be used at most once in case the $\not\subset$ instance enters recovery to claim the amount it covers. A $\mathbb{P}_i$ with a ratified incoming transfer $\mathfrak{T}_i^j(e)$ can then either withdraw this transfer from $\mathbb{V}_{\not\subset}$ starting from *eon* $e+2$ if the $\not\subset$ instance remains operational, or can withdraw the staked collateral covering its amount from $\mathbb{V}_{\not\subset}$ if the instance enters recovery during *eons* e or $e+1$. For a $\mathbb{P}_i$ to guarantee itself instant finality on its ratified received amounts, it must only approve incoming transfers if they do not lead to a state update that violates either of the following two constraints:

$$R_i(e-1)+R_i(e) \leq C_i(e)+\mathbb{T}_i(e-1)+\mathbb{T}_i(e) \tag{15}$$

$$R_i(e) \leq C_i(e+1)+\mathbb{T}_i(e) \tag{16}$$

We refer the reader to Appendix C.5 for our proof.

Challenge Justification Decidability A $\mathbb{P}_i$ in possession of $\tau_i^{A,C,E}(e)$ can decide whether it needs to challenge the consistency of the contents of the current $\mathbb{T}_{\not\subset}^{A,C,E}(e)$, and which

inconsistency it particularly needs to challenge. We prove this guarantee through simple case analysis of the following three possible inconsistency scenarios:

Exclusive Balance Allotment: Discussed in Section 5.2.

Active Transfer Delivery: As a recipient, upon being shown a valid $\lambda(\mathfrak{T}_i^j(e-1) \in T_j^a(e-1))$ by a payer $\mathbb{P}_j$ for a $\mathfrak{T}_i^j(e-1)$ that required recipient approval, $\mathbb{P}_i$ can check if $\mathfrak{T}_i^j(e-1) \notin T_i^a(e-1)$, and infer that a $\mathbb{X}_i^d(e)$ on $\mathfrak{T}_i^j(e-1)$ should be executed as it will not be closeable by $\mathbb{O}_{\not\subset}$.

Passive Transfer Delivery: A recipient $\mathbb{P}_i$ of passive $\mathfrak{T}_i^j(e-1)$ can immediately check whether it knows of a different transfer $\mathfrak{T}_i^k(e-1)$ where the intervals $[\text{offset}_{\mathfrak{T}_i^j}, \text{offset}_{\mathfrak{T}_i^j} + \mathfrak{T}_i^j.amount)$ and $[\text{offset}_{\mathfrak{T}_i^k}, \text{offset}_{\mathfrak{T}_i^k} + \mathfrak{T}_i^k.amount)$ intersect, or if $\text{offset}_{\mathfrak{T}_i^j} + \mathfrak{T}_i^j.amount > R_i^p(e-1)$. All it requires is $\lambda(\mathfrak{T}_i^j(e-1) \in T_j^a(e-1))$, along with the corresponding $\text{offset}_{\mathfrak{T}_i^j}$ value, either in the form of $S_j^p(e)$ or $\mathfrak{T}_i^j(e-1).\text{offset}$. If there is an inconsistency, then $\mathbb{P}_i$ can infer that a $\mathbb{X}_i^d(e)$ on $\mathfrak{T}_i^j(e-1)$ or $\mathfrak{T}_i^k(e-1)$ should be executed as it will not be closeable by $\mathbb{O}_{\not\subset}$. This is because $T_i^p(e-1)$ is constructed with its transfers reserving exclusive allotments for their amounts (cf. Section 5.2).■

A $\mathbb{P}_i$'s reasoning here assumes that members of $\mathbb{P}$ are able to communicate directly. Without this assumption, and assuming that $\mathbb{O}_{\not\subset}$ is withholding data, the ability of a $\mathbb{P}_i$ to reason in advance whether a transfer delivery challenge is justifiable, before incurring the costs of issuing that challenge on $\mathbb{BC}$, is hindered. Trivially, a cooperative operator can produce verifiable proofs of correct transfer delivery to both payers and recipients after committing to $\mathbb{T}_{\not\subset}^{A,C,E}$.

Secure Instant Commit-Chain Registration Because off-chain channel establishment is not common in 2nd-layer solutions, we elaborate why NOCUST can securely provide this feature. The registration procedure allows a registering $\mathbb{P}_i$ to learn the operator's signature on an initially empty account. This signature allows to instantiate challenges in $\mathbb{V}_{\not\subset}$ in case no transfers were performed in favor of $\mathbb{P}_i$, but only parent-chain deposits. This allows the operator to securely federate who can and cannot deposit funds into it, and guarantees that $\mathbb{P}_i$ would be able to initiate a $\mathbb{X}_i^b(e+1)$ in case its deposits were not credited as an allotment in the next *eon* $e+1$. The full specification for opening a challenge is presented in Appendix D.3.9, where the information in $\mathbb{B}^G$ combined with $\mathbb{P}_i$'s knowledge of $\mathbb{O}_{\not\subset}$'s signature on the registration suffices to ensure custody as described in Section 5.2. A new $\mathbb{P}_i$ who joins during the current *eon* does, however, does not have any stake allocated to it upon registration, and thus has no instant finality guarantees until reserve-collateral is allocated to it. This is done through one parent-chain transaction to individually increase all $\mathbb{U}_i(e)$ in $\mathbb{V}_{\not\subset}$.

6 Evaluation

In this section, we evaluate the NOCUST in terms of real-world practicality and light-client usability. We deployed NOCUST on the Ethereum mainnet[5], with a Solidity smart contract (1894 LOC), costing 3.9M gas (11.14 USD[6]). Our implementation follows a 36 hour *eon* interval[7]. The operator's rver code is implemented in Python (6937 LOC) and operates with 2 cores, 4 GB of RAM and SSD drives. The server requires a reliable blockchain source to view and respond to challenges, we deployed a full *geth* and *parity* node. For users, we developed a JavaScript wallet library (9281 LOC) capable to issue challenges towards the operator. Simple user-onboarding is shown with a free crypto faucet[8].

Parent-Chain Costs We registered 147 815 addresses with $\mathbb{O}_{\not\subset}$ in exponential steps (cf. Figure 4). For each step, we randomly choose 10 users making 20 transactions towards random users, and then commit a checkpoint. Costs increase with each *additional height* of the ledger's Merkle tree and remain below 0.5 USD, even if one billion registrations.

Checkpoint (Operator): Checkpoint submission costs are constant 96 073 gas (0.072 USD) on the Ethereum mainet.

Deposit Costs (User): Creation of commit-chain from parent-chain amount to 64 720 gas (0.048 USD).

Withdrawal (User): A non-collaborative withdraw requires two parent-chain transactions separated by a dispute period, one to initialize a withdrawal, base cost of 169 238 gas (0.126 USD) and one to confirm (31 500 gas, 0.023 USD), if the initialization was not voided through a dispute.

Initiate State Update Costs (User): Ensures data availability and integrity of an account, base cost 281 686 gas (0.211 USD) growing $\log(n)$ cf. Figure 4.

Answer State Update Challenge (Operator): Base transaction cost of 80 769 gas (0.061 USD).

Initiate Delivery Challenge (User): Ensures transfer inclusion in checkpoint, cost 225 642 gas (0.169 USD).

Answer Delivery Challenge (Operator): Base cost of 68 152 gas (0.051 USD). Growing $\mathcal{O}(\log n + \log v)$ with n users and v transfers executed in the previous *eon*.

Parent-Chain Storage: If a user does not perform any withdrawal, deposit or challenge, no data is stored on the parent-chain for this user (besides the contribution to the constant-size checkpoint). A deposit adds 160 bytes, a withdrawal 192 bytes. A Merkle root amounts to 68 bytes, a hash of all parent-chain operations to 32 bytes.

[5] cf. contract 0xac8c3D5242b425DE1b86b17E407D8E949D994010

[6] Assuming a gas price of 5 Gwei and Ether price of 150 USD.

[7] Which allows for sufficient time to manually intervene if the operator fails to commit a checkpoint. Faster interval times reduce the amount of required collateral for instant finality, but require users to be more frequently online (i.e. once per *eon*) for trustless operation (and vice versa).

[8] The faucet allows to receive 100 Wei (1 Wei = 10^{-18} ETH, 0.00 USD).

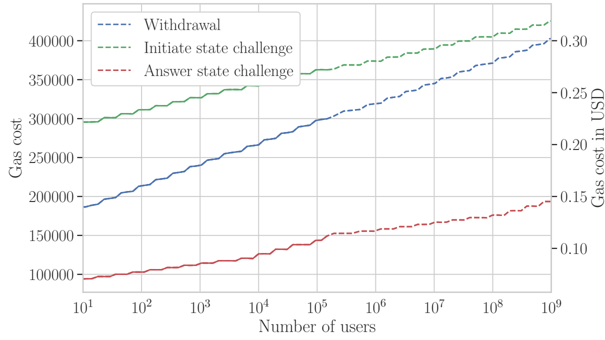

Figure 4: Cost evolution for challenges. The continuous line is empirical, the dotted line estimated. State challenge costs remain less than \$0.5, *at one billion users* registered to a NOCUST $\not\subset$, increase with the height ledger's Merkle tree. We assume that each user has at most one account/address.

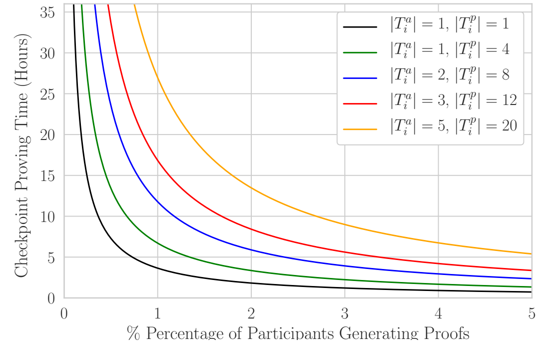

Figure 5: Time required to create a checkpoint consistency proof with a fraction of $\mathbb{P}$ to generate proofs in parallel. We assume one $\mathbb{P}_i$ corresponds to one CPU (cf. Table 2).

User Storage: Users store at least the data for the current and previous *eon*, including all commit-chain transfers with their signatures (312 bytes per transfer). After each checkpoint, the user queries the operator to retrieve the Merkle proof for his account. The proof grows logarithmically, 1280 bytes for 1M, 1920 bytes 1B user[9]. These user storage requirements are *ideal for lightweight clients*.

Operator Storage: All users account states, transfers of the current and previous round, and Merkle proofs for all users.

Transaction Throughput: About 20 tps on a single core between two accounts without network latency. Scales out with the accounts and based on non-optimized code.

zkSNARK Evaluation Table 2 (cf. Appendix E.2), shows the complexities (e.g. constraints) and computing times[10] (e.g. generation, proving) of our libsnark implementation, built to verify commitments containing up to 4 billion users and up to 4 billion transactions per user. Verification times are always $\leq 0.03s$, practical for smart contract execution.

$\mathbb{O}_{\not\subset}$ can distribute the task of creating a checkpoint consistency proof (constant-size 2690 bits). Figure 5 shows the time to create a proof, depending on the fraction of $\mathbb{P}$ collaborating (we ignore network latencies) with $\mathbb{O}_{\not\subset}$. We assume one CPU per $\mathbb{P}_i$. Given e.g. one billion users, one active and one passively delivered transfer per *eon* per user, generating a provably consistent checkpoint would require less than 4 hours with 1% of $\mathbb{P}$'s combined computational power.

Instant Finality Collateral Costs No stake is required from $\mathbb{O}_{\not\subset}$ to provide transaction finality within two *eons*. For *instant* finality (cf. Section 4.3.4), only the incoming transaction volume of a user within two *eons* is to be insured. To enable instant finality for $n = 1$M users and a value 100 USD per user over a timespan of 20 *eons*, $\mathbb{O}_{\not\subset}$ needs to stake 200M USD. This stake is "re-usable" each *eon*[11].

If $\mathbb{O}_{\not\subset}$ maintains a payment channel for each member of $\mathbb{P}$ [9] (i.e. a payment channel hub, PCH), collateral would be isolated in each channel, and cannot achieve finality without collateral. A PCH with $n = 1$M users, with each user expecting to receive 100 USD each *eon* within 20 *eons*, would require a $\mathbb{O}_{\not\subset}$ lockup of 2B USD (vs. 200M in NOCUST). When channel collateral is exhausted, the PCH is required to retrieve and consolidate its funds from other channels, either directly through parent-chain withdrawal (expensive) or coordinated rebalancing operations. Collateral fragmentation is expected to increase with a growing user-base.

7 Conclusion

In this work we presented a non-custodial commit-chain that can securely facilitate payments between participants in its 2nd-layer network, without reliance on a consensus mechanism as in side-chains, but rather on a practical challenge-response protocol that leverages a purpose-designed datastructure, and the consistency of which is further strengthened via zkSNARKs.

NOCUST not only raises the throughput of blockchains by several orders of magnitude, but (re)enables micropayments while economically supporting higher 2nd-layer value transfers. Contrary to payment channels, a recipient is not required to be online for a payment — securely solving a critical usability burden — while offering concise security proofs, coupled with an evaluation towards and beyond billions of user.

[9]The proof is used to verify the correctness of the account update, and published on the parent-chain to open a dispute if necessary.

[10]Measured on an Intel i7-7700K CPU at 4.20GHz.

[11]Note the *eon* time-window could be reduced to cut down the necessary collateral at the cost of increasing the availability requirements of users.

References

[1] Jimuta Naik. Beginning of the early banking industry in mesopotamia civilization from 8th century bce. 2014.

[2] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296. USENIX Association, 2016.

[3] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model, 2016.

[4] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[5] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2015.

[6] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *arXiv preprint arXiv:1702.05812*, 2017.

[7] Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. Flare: An approach to routing in lightning network. 2016.

[8] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[9] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment channels over cryptographic currencies. In *Security and Privacy (SP), 2019 IEEE Symposium on*. IEEE, 2019.

[10] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. Technical report, Cryptology ePrint Archive, Report 2016/701, 2016.

[11] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.

[12] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[13] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[14] Cynthia Dwork and Moni Naor. *Pricing via Processing or Combatting Junk Mail*, pages 139–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[15] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16. ACM, 2016.

[16] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 104–121. IEEE, 2015.

[17] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.

[18] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

[19] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable funding of bitcoin micropayment channel networks. *Royal Society open science*, 5(8):180089, 2018.

[20] Christian Decker, Rusty Russell, and Olaoluwa Osuntokun. eltoo: A simple layer2 protocol for bitcoin. *White paper: https://blockstream. com/eltoo. pdf*, 2018.

[21] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 473–489. ACM, 2017.

[22] The inevitability of privacy in lightning networks. https://www.kristovatlas.com/the-inevitability-of-privacy-in-lightning-networks/.

[23] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling payments fast and private: Efficient decentralized routing for path-based transactions. *arXiv preprint arXiv:1709.05748*, 2017.

[24] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. *Proceedings of NDSS 2017*, 2017.

[25] Ethan Heilman, Sebastien Lipmann, and Sharon Goldberg. The arwen trading protocols.

[26] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 455–471. ACM, 2017.

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

[28] Joshua Lind, Ittay Eyal, Florian Kelbert, Oded Naor, Peter Pietzuch, and Emin Gun Sirer. Teechain: Scalable blockchain payments using trusted execution environments. *arXiv preprint arXiv:1707.05454*, 2017.

[29] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, 2017.

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

[31] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *url: https://people. cs. uchicago. edu/teutsch/papers/truebit pdf*, 2017.

[32] Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. Pisa: Arbitration outsourcing for state channels. *IACR Cryptology ePrint Archive*, 2018:582, 2018.

[33] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of cryptology*, 1(2):77–94, 1988.

[34] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349. ACM, 2012.

[35] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[36] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 111–120. ACM, 2013.

[37] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017.

[38] Izaak Meckler and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. 2018.

[39] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multiparty computation for zk-snark parameters in the random beacon model. Technical report, Cryptology ePrint Archive, Report 2017/1050, 2017.

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

[41] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[42] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smart pool: Practical decentralized pooled mining. *IACR Cryptology ePrint Archive*, 2017:19, 2017.

[43] Ralph C. Merkle. *A Digital Signature Based on a Conventional Encryption Function*, pages 369–378. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[44] Merkle mountain ranges, 2019. https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md.

Appendices

A Addendum

A.1 Fundamental drawbacks of Payment Channel Networks

In the following we elaborate on the fundamental drawbacks of existing payment channel designs.

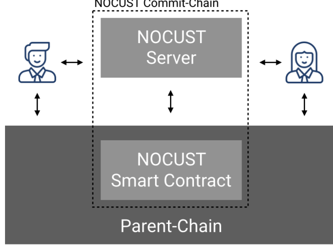

Figure 6: Commit-chain architecture. Under normal operation, a user communicates with the server to perform transactions towards other commit-chain users. Under adversarial conditions, the user resorts to the NOCUST smart contract on the parent-chain to perform a dispute process.

Online Requirement to Receive Transaction: PCN require the recipient of a transaction to actively accept an incoming transfer — parting from the usability feature offered by on-blockchain transactions.

Locked and Fragmented Collateral: PCN require collateral to be locked up for every channel. For example, given a PCN node with 1M channels, each channel sending on average 10k USD of transaction volume, would require the node to lock up a total of 10B USD. Collateral could be rebalanced by dynamically adjusting channels, which however would require costly parent-chain transactions.

Expensive Channel Setup: A PCN node with 1M users would be required to setup 1M parent-chain transactions for each channel setup. This in itself represents a substantial investment (beyond 100k USD on Ethereum).

Costly Routing: Route finding, has shown to be a significant difficulty in the realization of PCN [26]. Peers might become unresponsive, which requires payments to be diverted over newer, possibly more costly routes.

Online Watchdog Requirements: The two parties of a payment channel should remain online to observe their channel state continuously. That is, because if one party were to transition offline, the other party could attempt to close the channel with an outdated state and effectively double-spend a disputable amount. Users could outsource their channel to watchdogs, a third party, which however introduces new trust assumptions and costs.

Reduced Privacy: Because off-chain transactions are no longer recorded in the readable blockchain, one would argue that PCN offer better privacy guarantees than parent-chain transactions. Related work however has argued otherwise and proposed privacy enhanced payment channel designs [26, 21, 22, 23].

Double-Spending Attacks on Blockchain Congestion: Under a congested blockchain, channel termination could result in a bidding war among PCN nodes — and some channel might close unfairly. Incorrect channel termination can in particular be aggravated in the event of a *mass-exit*, where many channels are attempted to be closed. Note that the dispute resolution mechanism might in some cases not be worth considering if the disputed value is insignificant.

A.2 Architecture Overview

We provide an overview of our system model in Figure 6.

B Further Discussions

In this section, we discuss the privacy provisions of NOCUST and the implications of underlying $\mathbb{BC}$ congestion.

B.1 Privacy

In the course of running the protocol, participants acquire proofs of correct operation from and concede state updates to $\mathbb{O}_{\not\subset}$, while broadcasting some of that information to $\mathbb{V}_{\not\subset}$ on $\mathbb{BC}$. As the security of the protocol depends on non-repudiation and the forced revelation of information, we explore what each party in the protocol maintains knowledge of and can learn throughout NOCUST.

$\mathbb{O}_{\not\subset}$ **Knowledge:** $\mathbb{O}_{\not\subset}$ maintains knowledge of all transfers and balances in $\not\subset$. This is a requirement in NOCUST to enable $\mathbb{O}_{\not\subset}$ perform transfers and synchronize between $\mathbb{B}^G$ and $\mathbb{B}^L$ every *eon*, while retaining provable integrity. As such, an adversary in control of $\mathbb{O}_{\not\subset}$ has complete knowledge of all commit-chain information. Interestingly, at *eon* e, $\mathbb{O}_{\not\subset}$ need only maintain knowledge of e and $e-1$ to be able to construct $\mathbb{T}^{A,C,E}_{\not\subset}(e)$ and close any challenge $\mathbb{X}^b_i(e)$, $\mathbb{X}^d_i(e)$ by a $\mathbb{P}_i$. A $\mathbb{O}_{\not\subset}$ can erase $e-2$ at the end of *eon* $e-1$ to maintain a form of forward secrecy on the contents of $\mathbb{B}^L$ in $e-2$ without losing operational efficacy, which would retain privacy on all transfers enacted on the commit-chain prior to $e-1$ if $\mathbb{O}_{\not\subset}$ were compromised in e.

$\mathbb{P}$ **Knowledge:** Throughout its participation in a $\not\subset$, a $\mathbb{P}_i$ obtains $\tau^{A,C,E}_i(e)$ for every *eon* e, and constructs various $\mathfrak{T}^i_j(e)$ messages for different $\mathbb{P}_j$. A $\tau^A_i(e)$ reveals the allotment intervals at each height of $\mathbb{T}^A_{\not\subset}(e)$, but does not reveal individual account addresses or any transfer details. Therefore a $\mathbb{P}_i$ can learn that it has some neighbor account with a certain balance, and learn how the allotted intervals are designated at each level of $\mathbb{T}^A_{\not\subset}(e)$, without learning the identities of which members of $\mathbb{P}$ these allotments are made to. To enact a transfer $\mathfrak{T}^i_j(e)$ where the recipient's approval is required, $\mathbb{P}_i$ needs to sign a new $\text{update}^a_i(e)$ and send it to $\mathbb{O}_{\not\subset}$, and $\mathbb{P}_j$ needs to sign a new $\text{update}^a_j(e)$ and also send it to $\mathbb{O}_{\not\subset}$. $\mathbb{P}_i$ and $\mathbb{P}_j$ need not learn any information about the balance or transfer history of each other to construct these messages, but need to know the full details of the transfer $\mathfrak{T}^i_j(e)$ to ratify it in the state update authorizations they concede. When enacting a transfer $\mathfrak{T}^i_j$ that does not require the recipient's approval, however, $\mathbb{P}_i$ learns the $\text{offset}_{\mathfrak{T}}$ value from $\mathbb{O}_{\not\subset}$ after ratification as S^p_i, which reveals to $\mathbb{P}_i$ that

$\mathbb{P}_j$ has received at least S_i^p units passively prior to $\mathfrak{T}_j^i$ in the current round. Moreover, to enact a delivery challenge on a transfer $\mathfrak{T}_j^i(e-1)$, $\mathbb{P}_i$ and $\mathbb{P}_j$ need to share $\tau_i^A(e)$ and $\tau_j^A(e)$ to validate that the transfer was misenforced by $\mathbb{O}_{\not\subset}$.

$\mathbb{V}_{\not\subset}$ **Knowledge:** The privacy of the deposits, challenges, exits and withdrawals conducted through $\mathbb{V}_{\not\subset}$ is scoped by the underlying $\mathbb{BC}$ layer. In NOCUST we do not rely on any privacy of $\mathbb{BC}$ operations. Closing a balance update challenge $\mathbb{X}_i^b(e)$ requires that a $\mathbb{P}_i$ learn the $\tau_i(e)$ used in the closure to maintain custody of its account, which reveals $A_i(e)$. Therefore deposits may be used to guess a portion of a $\mathbb{P}_i$'s balance in $\mathbb{B}^L$ without interaction with $\mathbb{O}_{\not\subset}$, while closing a $\mathbb{X}_i^b(e)$ reveals $\mathbb{P}_i$'s balance in e, and initiating a $\mathbb{W}_i(e)$ reveals $\mathbb{P}_i$'s balance in $e-1$. It's noteworthy that a $\mathbb{P}_i$ wishing to mask its balance may assume multiple identities on $\mathbb{BC}$ (and consequently in $\mathbb{O}_{\not\subset}$) and fragment its deposits and withdrawals over them. However, we leave an extensive analysis for future work.

Comparison to Two-Party Payment Channels The guarantees provided in NOCUST are not trivial to compare with those of two-party channel networks. A party's maximum total balance may be inferred from the amount committed to a channel, as the commitment may only be in favor of one party or the other, and the exact amount can be learned from a parent-chain withdrawal which necessitates the broadcast of the latest off-chain state. The leakage of the off-chain balance during attempted withdrawals is similar in NOCUST and two-party channels, but inferring the maximum commit-chain balance prior to broadcast is not as simple in NOCUST due to the increased number of participants without leakage from $\mathbb{O}_{\not\subset}$. As we have not described the enactment of payments across multiple instances of $\not\subset$, we leave an extensive comparison to two-party channel networks for future work.

B.2 Underlying Ledger Congestion

The processing capacity of a $\mathbb{BC}$ plays a large role in practice in the security of 2nd-layer scaling solutions due to their use of limited time windows where off-chain operations can be disputed on the parent-chain.

Checkpoint Period: The maximum number of challenges that can be issued per *eon* is effectively dependent on the chosen *eon* duration in terms of number of *eras*, whereby with more *eras*, more challenge initiation transactions can be received by $\mathbb{BC}$, and more challenge responses, as the *epoch* duration would also increase.

Mass Dispute: In scenarios where a malicious hub attempts to seize its users' assets, an influx of dispute transactions will be submitted to $\mathbb{BC}$, and depending on the dispute timeout duration and $\mathbb{BC}$ congestion, some participants may not be able to commit their dispute on time. If the hub is instantiated across individual two-party payment channels, some channels may be disputed correctly, while others may be closed unfairly. In a $\not\subset$ commit-chain instance, however, *a single successful dispute is sufficient to protect all participants' assets from potential misbehavior by* $\mathbb{O}_{\not\subset}$. As $\mathbb{O}_{\not\subset}$ has to respond to all challenges, it must also succeed in committing its responses to $\mathbb{BC}$, and not only rely on the failure of participants to commit challenge initiations under congestion.

C Proofs

C.1 Exclusive Allotment

We proceed to prove that no valid instance of $\mathbb{T}_{\not\subset}$ may be used to construct a τ_i that permits a non-exclusive allotment by contradiction. Assume a valid instance of $\mathbb{T}_{\not\subset}$, and without loss of generality let t_x and t_y be two successive nodes $(y > x)$ within $\mathbb{T}_{\not\subset}$ that have overlapping allotments, where $\text{offset}_y < \text{offset}_x + \text{allotment}_x$.

Let t_u be their least common ancestor with t_p and t_q as its direct children such that t_p is an ancestor of t_x, and t_q of t_y. Without loss of generality, assume t_p and t_q are correctly reconstructible from τ_x and τ_y respectively.

Given τ_x, constructing t_u on the path up to t_{root} will be performed with knowledge of offset_p and allotment_p (from reconstructing t_p) and the boundary value and commitment of t_q supplied in τ_x.

$$\Omega(t_u, t_q) = \text{offset}_q + \text{allotment}_q \tag{17}$$

Recall the definition in Section 4.3.1. As offset_q is interchangeable with $\text{offset}_p + \text{allotment}_p$, reconstructing t_u will need to be performed as follows due to the lack of presence of offset_q in τ_x by substitution in equation 8 as follows:

$$\text{information}_u = \{t_p, \text{offset}_p + \text{allotment}_p, t_q\} \tag{18}$$

Given the correctness of the sub-tree of t_p in isolation, it follows that $\text{offset}_p + \text{allotment}_p = \text{offset}_x + \text{allotment}_x$, and therefore, assuming offset_q was used in the original commitment to the considered instance of $\mathbb{T}_{\not\subset}$, the reconstructed t_u will not match, and the remaining trail of reconstructed nodes in τ_x[12] will lead to a $t'_{root} \neq t_{root}$, violating the assumption that the instance under consideration is a valid $\mathbb{T}_{\not\subset}$ and that τ_x is acceptable ■

C.2 Balance Custody

We proceed to prove how an honest $\mathbb{P}_i$ in NOCUST can protect its funds through modelling the state of a $\mathbb{P}_i$'s custody as a finite state machine whereby $\mathbb{P}_i$ may always reach a custodian state. A $\mathbb{P}_i$ is considered a non-custodian in *eon*

[12] A symmetric argument can be made for τ_y

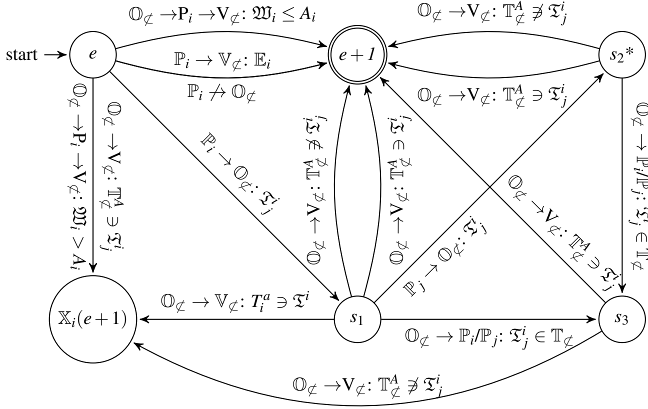

Figure 7: A finite state automaton capturing the provable integrity of $\mathbb{O}_{\not\subset}$. An honest $\mathbb{O}_{\not\subset}$ may not find itself in a state whereby it cannot prove its integrity in *eon* $e+1$ after committing to its e operations, while a dishonest $\mathbb{O}_{\not\subset}$ that attempts to reverse transfers or incorrectly enforce them will find itself unable to do so. *There exists a transition from s_2 to $\mathbb{X}_i(e+1)$ on ($\mathbb{O}_{\not\subset} \rightarrow \mathbb{V}_{\not\subset}$: $\mathbb{T}^A_{\not\subset} \ni \mathfrak{T}^i$) omitted for clarity.

e if $e-1$ had passed successfully ($\not\subset$ did not enter recovery) without $\mathbb{P}_i$ learning valid $\tau_i^{A,E}(e-1)$ that exclusively account for its confirmed balance, assuming $\mathbb{P}_i$ joined $\not\subset$ prior to $e-1$.

It's straightforward to infer from the automaton in Figure 3 that a $\mathbb{P}_i$ may always reach a state of custody from s_1 given that $\mathbb{O}_{\not\subset}$ commits to $\mathbb{T}^{A,C,E}_{\not\subset}(e)$ within the first epoch. Recall that if $\mathbb{O}_{\not\subset}$ does not commit to $\mathbb{T}^{A,C,E}_{\not\subset}(e)$ within the first *epoch*, the $\not\subset$ instance is halted, and therefore $\mathbb{P}_i$ retains custody of the previous allotment $A_i(e-1)$ which it may claim through $\mathbb{V}_{\not\subset}$'s recovery. This may also happen if $\mathbb{O}_{\not\subset}$ ignores $\mathbb{P}_i$'s challenge.■

C.3 Double-Spend Futility

Proof Let $\mathbb{P}_i$ and $\mathbb{O}_{\not\subset}$ be under the control of the adversary $\mathbb{A}$ such that the running balance of $\mathbb{P}_i$ during *eon* e is double-spent towards a subset of $\mathbb{P}$ whereby $A_i(e+1) < 0$ holds by the end of e. $\mathbb{A}$ must construct a valid $\mathbb{T}^A_{\not\subset}$ for $e+1$ to commit the transfers in e and successfully double-spend, while avoiding the halt of $\not\subset$ by an honest $\mathbb{P}_j$.

However, using Equation 23, validated by $\mathbb{V}_{\not\subset}$:

$$\begin{aligned}\sum_j A_j(e+1) &= \sum_j A_j(e) + R_j^{a\cup p}(e) + \mathbb{D}_j(e) - \mathbb{W}_j(e) - S_j^a(e) \\ &= \sum_j A_j(e) + \sum_j \mathbb{D}_j(e) - \mathbb{W}_j(e) + \sum_j R_j^{a\cup p}(e) - S_j^a(e) \\ &= \text{allotment}_{root}(e+1) + \sum_j R_j^{a\cup p}(e) - S_j^a(e)\end{aligned} \tag{19}$$

With Equation 19 in mind, if $\mathbb{A}$ were double-spending in $\mathbb{B}^L$ by not updating $S_i^a(e)$, then $\sum_j R_j^{a\cup p}(e) - S_j^a(e) > 0$ would follow, and $\text{allotment}_{root}(e+1) < \sum_j A_j(e+1)$ would lead to a challenge in $e+1$ by the affected honest $\mathbb{P}_j$ whose allotment is incorrect, foiling as well any concurrent double-spend in $\mathbb{B}^G$.

Moreover, if $\mathbb{A}$ were double-spending in $\mathbb{B}^L$ and updating $S_i^a(e)$ such that $\sum_j R_j^{a\cup p}(e) - S_j^a(e) = 0$, and/or double-spending through $\mathbb{W}_i(e)$ in $\mathbb{B}^G$, then an $\text{allotment}_{root}(e+1)$ would be rejected by $\mathbb{V}_{\not\subset}$ in violation of Equation 23.■

C.4 Operational Integrity

We proceed to prove how an honest $\mathbb{O}_{\not\subset}$ in NOCUST can maintain functionality under a subset of malicious users in $\mathbb{P}$, and how a dishonest $\mathbb{O}_{\not\subset}$ that attempts to compromise transfers will lead to the $\not\subset$ instance being stopped through a proof by case analysis, where we model the provability of a $\mathbb{O}_{\not\subset}$'s integrity as a finite state machine whereby transfers are facilitated by $\mathbb{O}_{\not\subset}$ in e and committed during $e+1$ in . A server is defined as maintaining provable integrity during *eon* e so long as it is able to close any challenge $\mathbb{X}_i(e)$ using $\mathbb{V}_{\not\subset}$.

The automaton presented in Figure 7 specifies how an honest $\mathbb{O}_{\not\subset}$ may always behave in such a way that allows it to retain provable integrity in $e+1$ regardless of the behavior of members of $\mathbb{P}$.

- Given no interactions between $\mathbb{O}_{\not\subset}$ and $\mathbb{P}_i$ during e, an honest $\mathbb{O}_{\not\subset}$ may create $\mathbb{T}^{A,C,E}_{\not\subset}(e+1)$ with $A_i(e+1)$ equal to $A_i(e)$ and no $\text{update}_i^a(e)$ applied.
- Once a $\mathbb{P}_i$ requests an exit $\mathbb{E}_i$, an honest $\mathbb{O}_{\not\subset}$ can construct $\mathbb{T}^E_{\not\subset}$ and retain all information necessary to close any $\mathbb{X}_i^b(e+1)$.
- $\mathbb{O}_{\not\subset}$ can justifiably authorize a $\mathbb{P}_i$ to initiate a partial withdrawal using $\mathbb{V}_{\not\subset}$, and still guarantee being able to satisfy the allotment constraint defined in Equation 23 in $e+1$, as long as it does not allow $\mathbb{P}_i$ to request to overdraw its funds.
- Given only an $\text{update}_i^a(e)$ signed by $\mathbb{P}_i$ where $\mathbb{P}_j$'s approval is mandatory, but no $\text{update}_j^a(e)$ signed by $\mathbb{P}_j$, an honest $\mathbb{O}_{\not\subset}$ may wait for $\mathbb{P}_j$ or discard $\mathfrak{T}^i_j$. No $\mathbb{X}_i^d(e+1)$ may be opened as $\mathbb{P}_i$ and $\mathbb{P}_j$ would not possess an $\text{update}_i^a(e)$ signed by $\mathbb{O}_{\not\subset}$ containing $\mathfrak{T}^i_j(e)$. A $\mathbb{X}_i^b(e+1)$ may be closed with the submission of a $\tau_i^{A,E}(e)$ reflecting the correct $A_i(e+1)$.
- Given an $\text{update}_i^a(e)$ signed by $\mathbb{P}_i$ where $\mathbb{P}_j$'s approval is not mandatory, an honest $\mathbb{O}_{\not\subset}$ may discard or synchronize $\mathfrak{T}^i_j(e)$, or commit to its delivery by sending a countersigned $\text{update}_i^a(e)$ to $\mathbb{P}_i$ and/or $\mathbb{P}_j$ and then must synchronize its delivery in $\mathbb{T}^A_{\not\subset}(e+1)$. The operator retains sufficient information to close any $\mathbb{X}_i^b(e+1)$ or $\mathbb{X}_i^d(e+1)$ in $\mathbb{V}_{\not\subset}$.
- Given an $\text{update}_i^a(e)$ signed by $\mathbb{P}_i$ and an $\text{update}_j^a(e)$ signed by $\mathbb{P}_j$, an honest $\mathbb{O}_{\not\subset}$ may discard or synchronize

$\mathfrak{T}_j^i(e)$, or commit to its delivery by sending a countersigned update^a to $\mathbb{P}_i$ and/or $\mathbb{P}_j$ and then must synchronize its delivery in $\mathbb{T}^A_{\not\subset}(e+1)$. The operator retains sufficient information to close any $\mathbb{X}_i^b(e+1)$ or $\mathbb{X}_i^d(e+1)$ in $\mathbb{V}_{\not\subset}$.

Moreover, a dishonest server which tries to debit a $\mathbb{P}_i$ without authorization, or without crediting the corresponding $\mathbb{P}_j$ in case of a $\mathfrak{T}_j^i$, or one that provides partial withdrawal authorizations $\mathfrak{W}_i$ that overspend a $\mathbb{P}_i$ balance, may not find itself in a state of provable integrity in $e+1$.

- Given no interactions between $\mathbb{O}_{\not\subset}$ and $\mathbb{P}_i$ during e, the operator cannot construct a valid $\mathbb{T}^A_{\not\subset}(e+1)$ containing an $\text{update}_i^a(e)$ signed by $\mathbb{P}_i$. As $\mathbb{O}_{\not\subset}$ cannot forge $\mathbb{P}_i$'s signature, it cannot close a $\mathbb{X}_i^b(e+1)$.
- If $\mathbb{O}_{\not\subset}$ authorizes a $\mathbb{P}_j$ to initiate a partial withdrawal using $\mathbb{V}_{\not\subset}$ that overdraws its funds, it will not be able to satisfy the allotment constraint defined in Equation 23 in $e+1$.
- Given only an $\text{update}_i^a(e)$ signed by $\mathbb{P}_i$ where $\mathbb{P}_j$'s approval is mandatory, the operator cannot construct a valid $\mathbb{T}^A_{\not\subset}(e+1)$ containing an $\text{update}_j^a(e)$ signed by $\mathbb{P}_j$. A $\mathbb{X}_i^d(e+1)$ on $\mathfrak{T}_j^i(e)$ by a custodian $\mathbb{P}_i$ will not be closeable by $\mathbb{O}_{\not\subset}$.
- Given an $\text{update}_i^a(e)$ signed by $\mathbb{P}_i$ where $\mathbb{P}_j$'s approval is not mandatory, the operator cannot construct a valid $\mathbb{T}^A_{\not\subset}(e+1)$ where $\mathfrak{T}_j^i(e)$ does not reserve an exclusive allotment in . A $\mathbb{X}_i^d(e+1)$ on $\mathfrak{T}_j^i(e)$ by a custodian $\mathbb{P}_i$ will not be closeable by $\mathbb{O}_{\not\subset}$.
- Once the operator delivers a countersigned $\text{update}_i^a(e)$ and/or $\text{update}_j^a(e)$ $(s_2 \to s_3)$ to either $\mathbb{P}_i$ or $\mathbb{P}_j$ respectively, it may not back out of enforcing $\mathfrak{T}_j^i(e)$, as $\mathbb{O}_{\not\subset}$ will not be able to close a $\mathbb{X}_i^b(e+1)$, and/or $\mathbb{X}_j^b(e+1)$, if it commits an outdated state in $\mathbb{T}^A_{\not\subset}(e+1)$.■

C.5 Instant Transaction Finality

We proceed to prove prove how a $\mathbb{P}_i$ enacting transfers in a $\not\subset$ instance is guaranteed to be able to finalize receipt of incoming amounts up to a known total amount, regardless of the adversary's behavior while controlling $\mathbb{O}_{\not\subset}$ and/or all other members of $\mathbb{P}$.

Proof The proof that a $\mathbb{P}_i$ observing the constraints in Equation 15 and Equation 16 has guaranteed finality of receipt on ratified incoming transfers is straightforward.

- If the $\not\subset$ instance fails during *eon* e, then $\mathbb{P}_i$ can recover an amount equal to the R.H.S of Equation 15.
- If the $\not\subset$ instance fails during *eon* $e+1$, then $\mathbb{P}_i$ can recover an amount equal to the R.H.S of Equation 16.
- Otherwise, $\mathfrak{T}_i^j(e)$ has been in included in $\mathbb{T}^A_{\not\subset}(e+1)$, and its amount can be withdrawn in $e+2$.

$\mathbb{U}_i(e-1)$ and $\mathbb{U}_i(e)$ are accounted for by $\mathbb{V}_{\not\subset}$, while $C_i(e)$ and $C_i(e+1)$ are committed to by and learned by $\mathbb{P}_i$ before *eon* e commences, or assumed to be zero. The exclusivity of the amounts $C_i(e)$ and $C_i(e+1)$ are guaranteed through validation of $\tau_i^C(e-2)$ and $\tau_i^C(e-1)$ respectively (ref. C.1)

Moreover, $\mathbb{O}_{\not\subset}$ cannot withdraw staked collateral such the total amount, $\mathbb{C}(e)$, promised in $\mathbb{T}^C_{\not\subset}(e-2)$, is unavailable in $\mathbb{V}_{\not\subset}$ for recovery, which means the recoverable amount from a $\tau_i^C(e-2)$ is always available in *eon* e. ■

D Extended Specifications

D.1 Sequential Overview of NOCUST

In Figure 8 we present a sequential view of NOCUST's participants and their actions.

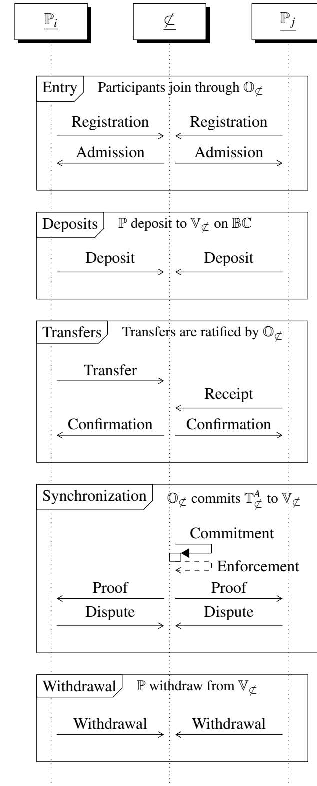

Figure 8: A sequential view of a NOCUST instance lifecycle. In practicality, transfers, deposits, and withdrawals may interleave post entry. Receipt of an commit-chain transfer is possible after admission and does not require a prior deposit.

D.2 $\mathbb{B}$ Bi-Modal Ledger For zkSNARKS

The motivation for this extended specification is to provide specifications for the three $\mathbb{B}^G$ verifiers: $\mathbb{V}_D$, $\mathbb{V}_W$, and

Algorithm 2: verifyDepositOperationInclusion

Verifier Input : $t_{root}^{\mathfrak{A}^D}(e), t_{root}^{\mathfrak{S}^D}(e)$
Prover Input : $\lambda(op \in \mathfrak{A}(e)), \lambda(op \in \mathfrak{S}_i^D(e)),$
$\lambda(\mathfrak{S}_i^D(e) \in \mathfrak{S}_D(e))$, op
Verifier Output: offset $_{op}$, information$_{op}$, allotment$_{op}$
verify $\lambda(op \in \mathfrak{A}(e))$ leads to $t_{root}^{\mathfrak{A}^D}(e)$
verify $\lambda(op \in \mathfrak{S}_i^D(e))$ and $\lambda(\mathfrak{S}_i^D(e) \in \mathfrak{S}_D(e))$ lead to $t_{root}^{\mathfrak{S}^D}(e)$
return offset $_{op}$, information$_{op}$, allotment$_{op}$

$\mathbb{V}_E$, that are required in Algorithms 5 and 6 without worrying about the storage introspection details of the underlying parent-chain $\mathbb{BC}$. For every *eon e*, $\mathbb{B}^G(e)$ is amended with the following:

$\mathfrak{A}(e)$**:** Accumulator of $\mathbb{V}_{\not\subset}$ operations performed in e.

The accumulator $\mathfrak{A}(e)$ is structured as a set of Merkle Mountain Ranges [44] that are built up as deposits $\mathbb{D}_i(e)$, withdrawals $\mathbb{W}_i(e)$ and exits $\mathbb{E}_i(e)$ are performed using $\mathbb{V}_{\not\subset}$ during the current *eon e*. To enable this structure to be usable, $\mathbb{V}_{\not\subset}$ will need to keep track of the current set of roots for $\mathfrak{A}(e)$, and append a new element for every deposit made to $\mathfrak{A}^D(e)$, withdrawal initialized to $\mathfrak{A}^W(e)$ and exit requested to $\mathfrak{A}^E(e)$. Each element should be a commitment of the details of the operation made:

$$\{operation, address, amount\}$$

$\mathbb{O}_{\not\subset}$ then re-organizes two of these three accumulators into Merkle tree structures for deposits $\mathfrak{S}^D(e)$ and withdrawals $\mathfrak{S}^W(e)$. Designed to consolidate the amounts of operations in $\mathfrak{A}^D(e)$ and $\mathfrak{A}^W(e)$ according to their $\mathbb{P}_i$, these trees are built up as Merkleized Interval Trees akin to the $\mathbb{T}_{\not\subset}^A$ structure from Section 4.3.1.

The following definitions are used for the leaves of $\mathfrak{S}^D(e)$ and symmetrically for $\mathfrak{S}^W(e)$, along with that of Equation 4:

$$\text{allotment}_i(e) = \mathbb{D}_i(e) \tag{20}$$

$$\text{information}_i(e) = \{\text{address}_i, \mathfrak{S}_i^D(e)\} \tag{21}$$

The subscripted $\mathfrak{S}_i^D(e)$ is defined as yet another annotated Merkle tree which consolidates the deposits in $\mathfrak{A}(e)$ that are only specific to one $\mathbb{P}_i$. Meaning that each leaf in $\mathfrak{S}^D(e)$ contains the subtree $\mathfrak{S}_i^D(e)$, and the allotment size of the leaf is the allotment size of the root of the subtree. The allotment size then of the leaves of the subtree are the individual amounts of the operations, and the information within them are commitments to their respective operation's details. This symmetrically applies to $\mathfrak{S}^W(e)$ and $\mathfrak{S}_i^W(e)$.

$\mathbb{B}^G(e)$ is further amended to then store commitments to the roots of $\mathfrak{S}^D(e)$ and $\mathfrak{S}^W(e)$. $\mathbb{V}_{\not\subset}$ is then extended to accept the submission of these commitments only when they are correctly constructed, the allotment of the root of $\mathfrak{S}^D(e)$ is equal to $\mathbb{D}(e)$, that of the root of $\mathfrak{S}^W(e)$ equals $\mathbb{W}(e)$ and that underlying data leaves obey the following constraints:

$$\forall p_i, p_j \in \mathfrak{S}^{\mathfrak{x}} : i < j \rightarrow \text{address}_i < \text{address}_j$$

$$op \in \mathfrak{A}^{op.type}(e) \leftrightarrow \exists i : op \in \mathfrak{S}_i^{op.type}(e)$$

The conditions on the allotment of the two roots are easy to validate in $\mathbb{V}_{\not\subset}$, but verifying the correct construction of the underlying data and the validity of the above two constraints will require the usage of the same zkSNARK combination scheme previously described in Section 4.5.

The combiner for the procedure defined in Algorithm 2, denoted as $\mathbb{V}_{DI}$, is left as an exercise to the reader, noting that it will have to be instantiated twice to prove both sides of the bi-implication in the second constraint. The verifiers and their combiners for proving the consistency of $\mathfrak{S}_i^W(e)$ with respect to $\mathfrak{A}^W(e)$ will be symmetric to those of $\mathfrak{S}_i^D(e)$.

Algorithm 3: verifyDepositAllotment

Verifier Input : $t_{root}^{\mathfrak{A}^D}(e), t_{root}^{\mathfrak{S}^D}(e)$
Prover Input : $\lambda(\mathfrak{S}_i^D(e) \in \mathfrak{S}_D(e)), \pi_{DI}$, op
Verifier Output: offset $_i(e)$, information$_i(e)$, allotment$_i(e)$
operations $\leftarrow \mathbb{V}_{DI}^{\pi_{DI}}(t_{root}^{\mathfrak{A}^D}(e), t_{root}^{\mathfrak{S}^D}(e))$
verify $\lambda(\mathfrak{S}_i^D(e) \in \mathfrak{S}_D(e))$ leads to $t_{root}^{\mathfrak{S}^D}(e)$
assert allotment$_i(e)$ is equal to deposits.allotment
return offset $_i(e)$, information$_i(e)$, allotment$_i(e)$

The combiner for the procedure defined in Algorithm 3 is left as an exercise to the reader, noting that the combiner should enforce the ordering constraint on leaf addresses. Our sought after $\mathbb{V}_D$ then becomes a procedure for retrieving the allotment of $\mathbb{P}_i$ from $\mathfrak{S}^D(e)$, another exercise for the reader. The verifiers and their combiners for proving the consistency of $\mathfrak{S}^W(e)$ with respect to $\mathfrak{A}^W(e)$ will be symmetric to those of $\mathfrak{S}^D(e)$, along with $\mathbb{V}_W$.

The full specification of $\mathbb{V}_E$ then becomes an exercise for the reader, where $\mathfrak{S}^E(e)$ contains no subtrees and is easily verifiable and $\mathbb{E}_i(e)$ is easily retrievable.

Algorithm 4: verifyTxDelivery

Verifier Input : $t_{root}(e)$
Prover Input : $\tau_j^A(e)$, update$_j(e)$,
$\lambda(\mathfrak{T}_j^i(e-1) \in T_j(e-1)), \mathfrak{T}_j^i(e-1)$
Verifier Output: hash$(\mathfrak{T}_j^i(e-1))$
verify $\tau_j^A(e)$ leads to $t_{root}(e)$
verify $\tau_j^A(e)$ applies update$_j(e)$
verify $\lambda(\mathfrak{T}_j^i(e-1) \in T_j(e-1))$
return *hash*$(\mathfrak{T}_j^i(e-1))$

Algorithm 5: verifyAccountIntegrity

Verifier Input : $t_{root}(e-1)$, $t_{root}(e)$
Prover Input : $\tau_i^A(e-1)$, $\tau_i^A(e)$, $\text{update}_i(e-1)$, π_D, π_W, π_E, π_T
Verifier Output: $\text{offset}_i(e)$, $\text{information}_i(e)$, $\text{allotment}_i(e)$
verify $\tau_i^A(e-1)$ leads to $t_{root}(e-1)$
verify $\tau_i^A(e)$ leads to $t_{root}(e)$
verify $\tau_i^A(e)$ applies $\text{update}_i(e-1)$
verify that $\text{update}_i(e-1)$ is signed by $\mathbb{P}_i$ if not empty
$\mathbb{D}_i(e-1) \leftarrow \mathbb{V}_D^{\pi_D}(\mathbb{B}^G(e-1), \mathbb{P}_i)$
$\mathbb{W}_i(e-1) \leftarrow \mathbb{V}_W^{\pi_W}(\mathbb{B}^G(e-1), \mathbb{P}_i)$
$\mathbb{E}_i(e-1) \leftarrow \mathbb{V}_E^{\pi_E}(\mathbb{B}^G(e-1), \mathbb{P}_i)$
verify $A_i(e)$ according to Equation 1 and $\mathbb{E}_i(e-1)$
verify result of $\mathbb{V}_T^{\pi_T}(t_{root}(e))$ is equal to $T_i^a(e-1)$
return $\text{offset}_i(e)$, $\text{information}_i(e)$, $\text{allotment}_i(e)$

Algorithm 6: verifyExitIntegrity

Verifier Input : $t_{root}^A(e-1)$, $t_{root}^E(e)$
Prover Input : $\tau_i^A(e-1)$, $\tau_i^E(e)$, $\text{update}_i(e-1)$, π_D, π_W, π_E, π_T
Verifier Output: $\text{offset}_i(e)$, $\text{information}_i(e)$, $\text{allotment}_i(e)$
$\mathbb{E}_i(e-1) \leftarrow \mathbb{V}_E^{\pi_E}(\mathbb{B}^G(e-1), \mathbb{P}_i)$
assert $\mathbb{E}_i(e-1)$ is true
verify $\tau_i^A(e-1)$ leads to $t_{root}^A(e-1)$
verify $\tau_i^E(e)$ leads to $t_{root}^E(e)$
verify that $\text{update}_i(e-1)$ is signed by $\mathbb{P}_i$ if not empty
$\mathbb{D}_i(e-1) \leftarrow \mathbb{V}_D^{\pi_D}(\mathbb{B}^G(e-1), \mathbb{P}_i)$
$\mathbb{W}_i(e-1) \leftarrow \mathbb{V}_W^{\pi_W}(\mathbb{B}^G(e-1), \mathbb{P}_i)$
verify $A_i(e)$ according to Equation 1
return $\text{offset}_i(e)$, $\text{information}_i(e)$, $\text{allotment}_i(e)$

D.3 $\mathbb{V}_{\not\subset}$ Parent-chain Verifier

The parent-chain component $\mathbb{V}_{\not\subset}$ acts as the bridge between the $\mathbb{O}_{\not\subset}$ ledger $\mathbb{B}^L$ and the $\mathbb{BC}$ ledger $\mathbb{B}^G$. Its procedures are assumed to be executed honestly by $\mathbb{BC}$, and it supports the following operations:

D.3.1 Commit $\mathbb{T}_{\not\subset}^{A,C,E}(e)$

Committing to $\mathbb{T}_{\not\subset}^{A,C,E}(e)$ may only be done once per *eon* e during its first *epoch* by $\mathbb{O}_{\not\subset}$. The commitment requires only submission of the root nodes of $\mathbb{T}_{\not\subset}^{A,C,E}(e)$ and not their full contents.

The commitment procedure involves no validation on information if no consistency proofs are required, but the following requirements always exist on the offset and allotment of the root nodes:

$$\text{offset}^A(e) = \text{offset}^C(e) = \text{offset}^E(e) = 0 \tag{22}$$

$$\begin{aligned}\text{allotment}^A(e) + \text{allotment}^E(e) = &\text{allotment}^A(e-1) \\ &+ \mathbb{D}(e-1) \\ &- \mathbb{W}(e-1)\end{aligned} \tag{23}$$

$$\text{allotment}^C(e) \leq \mathbb{C} \tag{24}$$

If the $\not\subset$ instance is setup to accept checkpoints that are accompanied by a proof of consistency, as described in Section 4.5, then $\mathbb{V}_{\not\subset}$ validates this proof on the parent-chain prior to accepting the commitment.

After running its validations, $\mathbb{V}_{\not\subset}$ stores the root $\mathbb{T}_{\not\subset}^{A,C,E}(e)$ nodes, making them available to any $\mathbb{P}$ or any other $\mathbb{V}_{\not\subset}$ procedure.

Preconditions:

- $\mathbb{O}_{\not\subset}$ must not have committed to $\mathbb{T}_{\not\subset}^{A,C,E}(e)$
- $\not\subset$ must not have entered recovery

Input: $t_{root}^A(e)$, $t_{root}^C(e)$, $t_{root}^E(e)$, π^A, π^E

1. Verify conditions of Equations 22, 23 and 24
2. Verify $\mathbb{V}_{\mathbb{T}^A}^{\pi^A}(t_{root}^A(e))$ and $\mathbb{V}_{\mathbb{T}^E}^{\pi^E}(t_{root}^E(e))$ if snarks setup
3. Store $t_{root}^A(e)$, $t_{root}^C(e)$ and $t_{root}^E(e)$

D.3.2 Verify $\tau_i^{A,C,E}(e)$

This verification procedure enables $\mathbb{V}_{\not\subset}$ to verify a $\tau_i(e)$ for any e in which $\mathbb{O}_{\not\subset}$ had committed to a $\mathbb{T}_{\not\subset}$. This validation acts as a foundation for the security of NOCUST.

Preconditions:

- $\mathbb{O}_{\not\subset}$ must have committed to $\mathbb{T}_{\not\subset}^{A,C,E}(e)$

Input: $\tau_i(e)$

1. Reconstruct $t'_{root}(e)$ from $\tau_i(e)$
2. output true iff $t'_{root}(e) = t_{root}(e)$

D.3.3 Receive Deposit $\mathbb{D}_i(e)$

For a $\mathbb{P}_i$ to make a deposit into $\not\subset$, it would simply send a transfer in the $\mathbb{BC}$ ledger with $\mathbb{V}_{\not\subset}$ as the recipient. The only requirement on $\mathbb{V}_{\not\subset}$ is then that it adds the value of the transfer to $\mathbb{D}_i(e)$, where e is the current *eon*.

Preconditions:

- $\not\subset$ must not have entered recovery

Input: $\mathbb{BC}$ transfer T from $\mathbb{P}_i$ to $\mathbb{V}_{\not\subset}$

1. Set $\mathbb{D}_i(e)$ to $\mathbb{D}_i(e) + \mathrm{T}.\text{amount}$

D.3.4 Initiate Exit $\mathbb{E}_i(e)$

A $\mathbb{P}_i$ can initiate a complete exit from $\not\subset$ by submitting a request to $\mathbb{V}_{\not\subset}$. $\mathbb{V}_{\not\subset}$ is simply required to set $\mathbb{E}_i(e)$ to true in response such that this exit can be finalized in *eon* $e+2$.

Preconditions:

- $\not\subset$ must not have entered recovery
- $\mathbb{P}_i$ must not be already pending exit

Input: -

1. Assert $\mathbb{E}_i(e)$ is false.
2. Set $\mathbb{E}_i(e)$ to true

D.3.5 Finalize Exit $\mathbb{E}_i(e)$

In *eon* e, an exit request can be finalized if it was scheduled in *eon* $e-2$ and the first *epoch* has passed, or if it had been scheduled in an *eon* $\leq e-3$. Upon confirmation of the exit, $\mathbb{V}_{\not\subset}$ issues a transfer from the balance pool it manages in favor of $\mathbb{P}_i$ with the requested amount.

Preconditions:

- $\mathbb{E}_i(s) > 0$ for some $s \leq e-2$

Input: $\tau_i^E(e-1)$

1. Reject if $s = e-2$ and the first *epoch* of e has not passed
2. Transfer $A_i(s)$ to $\mathbb{P}_i$ on $\mathbb{BC}$

D.3.6 Initiate Withdrawal $\mathbb{W}_i(e)$

A $\mathbb{P}_i$ can initiate a withdrawal from $\not\subset$ by submitting a request to $\mathbb{V}_{\not\subset}$. This request consists of the amount to be withdrawn once the request is confirmed in $e+1$, and of an authorization $\mathfrak{W}_i(e)$ that is signed by $\mathbb{O}_{\not\subset}$. After validation, $\mathbb{V}_{\not\subset}$ is required to set $\mathbb{W}_i(e)$ to the requested amount, while upon validation failure $\mathbb{V}_{\not\subset}$ should reject the request.

Preconditions:

- $\not\subset$ must not have entered recovery

Input: $\mathfrak{W}_i(e)$

1. Validate $\mathfrak{W}_i(e)$
2. Set $\mathbb{W}_i(e)$ to $\mathfrak{W}_i(e)$.amount

D.3.7 Proxy Withdrawal $\mathbb{W}_i(e)$

$\mathbb{O}_{\not\subset}$ can opt to act as a proxy for a withdrawal request, relieving the requesting $\mathbb{P}_i$ of the waiting period of 2 *eons* for confirmation, and instead transferring to it the requested amount on the parent-chain in exchange for being the recipient of the final confirmation of the withdrawal after the waiting period.

Preconditions:

- $\not\subset$ must not have entered recovery
- $W_i(s) > 0$ for some $s \leq e-2$

Input: $\mathbb{P}_i$

1. Forward $W_i(s)$ from $\mathbb{O}_{\not\subset}$ to $\mathbb{P}_i$ on $\mathbb{BC}$
2. Set recipient of $W_i(s)$ to $\mathbb{O}_{\not\subset}$

D.3.8 Confirm Withdrawal $\mathbb{W}_i(e)$

In *eon* e, a withdrawal request can be confirmed if it was scheduled in *eon* $e-2$ and the first *epoch* has passed, or if it had been scheduled in an *eon* $\leq e-3$. Upon confirmation of a withdrawal, $\mathbb{V}_{\not\subset}$ issues a transfer from the balance pool it manages in favor of $\mathbb{P}_i$ with the requested amount.

Preconditions:

- $W_i(s) > 0$ for some $s \leq e-2$

Input: none

1. Reject if $s = e-2$ and the first *epoch* of e has not passed
2. Transfer $W_i(s)$ to its recipient on $\mathbb{BC}$
3. Set $W_i(s)$ to 0

D.3.9 Open Balance Update Challenge $\mathbb{X}_i^b(e)$

Given a $\tau_i^A(e-1)$ and an $\mathtt{update}_i(e-1)$ signed by $\mathbb{O}_{\not\subset}$ as inputs from a $\mathbb{P}_i$, the $\mathbb{V}_{\not\subset}$ challenge procedure requires that the operator provides a satisfying $\tau_i^{A,E}(e)$ $\mathbb{V}_{\not\subset}$ before an *epoch* passes. Otherwise, $\not\subset$ is shut down, and all transactions since the beginning of $e-1$ are reverted.

Preconditions:

- $\not\subset$ must not have entered recovery

Input: At least one of $\tau_i^A(e-1)$ and $\mathtt{update}_i(e-1)$

- Verify $\tau_i^A(e-1)$, or $A_i(e-1) = 0$
- Verify $Sig_O(\mathtt{update}_i(e-1))$, or $R_i^a(e-1) = S_i^a(e-1) = 0$
- Store expected $A_i(e)$ in $\mathbb{X}_i^b(e)$

D.3.10 Close Balance Update Challenge $\mathbb{X}_i^b(e)$

Given a valid $\tau_i^{A,E}(e)$ as input from $\mathbb{O}_{\not\subset}$, $\mathbb{V}_{\not\subset}$ marks $\mathbb{X}_i^b(e)$ as closed if it were open within the last *epoch*.

Preconditions:

- $\not\subset$ must not have entered recovery
- $\exists\, \mathbb{X}_i^b(e)$ not older than an *epoch*

Input: $\tau_i^{A,E}(e)$, $\mathtt{update}_i(e-1)$

1. Verify $\tau_i^A(e)$
2. Verify $Sig_i(\mathtt{update}_i(e-1))$
3. Verify $Sig_O(\mathtt{update}_i(e-1))$
4. Verify $\mathtt{update}_i(e-1)$ is at least as recent as in $\mathbb{X}_i^b(e)$
5. Validate that $\tau_i^A(e)$ ratifies $\mathtt{update}_i(e-1)$
6. If $\mathbb{E}_i(e-1)$ is true, verify $\tau_i^E(e)$
7. Mark $\mathbb{X}_i^b(e)$ closed

D.3.11 Open Transfer Delivery Challenge $\mathbb{X}_i^d(e)$

Given an $\text{update}_j^a(e-1)$ signed by $\mathbb{O}_{\not\subset}$, and a transfer $\mathfrak{T}_i^j(e-1) \in T_j^a(e-1)$ as inputs from $\mathbb{P}_i$ or $\mathbb{P}_j$, the $\mathbb{V}_{\not\subset}$ delivery challenge procedure requires that the operator provide a satisfying $\tau_i^A(e)$ and $\lambda(\mathfrak{T}_i^j(e-1) \in T_i(e-1))$ to $\mathbb{V}_{\not\subset}$ before an *epoch* passes. Otherwise, $\not\subset$ is shut down, and all transactions since the beginning of $e-1$ are reverted.

Preconditions:

- $\not\subset$ must not have entered recovery

Input: $\text{update}_j^a(e-1), \mathfrak{T}_j^i(e-1), \lambda(\mathfrak{T}_i^j(e-1) \in T_j^a(e-1))$

- Verify $Sig_O(\text{update}_j^a(e-1))$
- Verify $\lambda(\mathfrak{T}_i^j(e-1) \in T_j^a(e-1))$
- Verify $\text{offset}_{\mathfrak{T}} \in \mathfrak{T}_i^j(e-1)$ or $\mathfrak{T}_i^j(e-1)$ required authorization from recipient.
- Store $\mathfrak{T}_i^j(e-1)$ in $\mathbb{X}_i^d(e)$

D.3.12 Open Offset Transfer Delivery Challenge $\mathbb{X}_i^d(e)$

Given an $\text{update}_j(e-1)$ signed by $\mathbb{O}_{\not\subset}$, the last outgoing passive delivery transfer $\mathfrak{T}_i^j(e-1) \in T_j^a(e-1)$, and $\tau_j^A(e)$ as inputs from $\mathbb{P}_i$ or $\mathbb{P}_j$ the $\mathbb{V}_{\not\subset}$ last passive transfer delivery challenge procedure for transfers where the recipient's authorization is not mandatory requires that the operator provide a satisfying $\tau_i^A(e)$ and $\lambda(\mathfrak{T}_i^j(e-1) \in T_i^p(e-1))$ to $\mathbb{V}_{\not\subset}$ before an *epoch* passes. Otherwise, $\not\subset$ is shut down, and all transactions since the beginning of $e-1$ are reverted.

Preconditions:

- $\not\subset$ must not have entered recovery

Input: $\text{update}_j(e-1), \mathfrak{T}_i^j(e-1), \lambda(\mathfrak{T}_i^j(e-1) \in T_j^a(e-1))$

- Verify $\tau_j^A(e)$
- Validate that $\tau_j^A(e)$ ratifies $\text{update}_j(e-1)$
- Verify $\lambda(\mathfrak{T}_i^j(e-1) \in T_j^a(e-1))$
- Store $\text{offset}_{\mathfrak{T}} = S_j^P(e)$ in $\mathbb{X}_i^d(e)$
- Store $\mathfrak{T}_i^j(e-1)$ in $\mathbb{X}_i^d(e)$

D.3.13 Close Authorized Transfer Delivery Challenge $\mathbb{X}_i^d(e)$

Given a valid $\tau_i^A(e)$, $\text{update}_i^a(e-1)$ and $\lambda(\mathfrak{T}_i^j(e-1) \in T_i^a(e-1))$ as input from $\mathbb{O}_{\not\subset}$, $\mathbb{V}_{\not\subset}$ marks $\mathbb{X}_i^d(e)$ as closed if it were open within the last *epoch*.

Preconditions:

- $\not\subset$ must not have entered recovery
- $\exists\, \mathbb{X}_i^d(e)$ not older than an *epoch*

Input: $\tau_i^A(e), \text{update}_i^a(e-1), \lambda(\mathfrak{T}_i^j(e-1) \in T_i^a(e-1))$

1. Verify $\tau_i^A(e)$
2. Validate $Sig_i(\text{update}_i^a(e-1))$
3. Validate that $\tau_i^A(e)$ ratifies $\text{update}_i^a(e-1)$
4. Validate $\lambda(\mathfrak{T}_i^j(e-1) \in T_i^a(e-1))$
5. Mark $\mathbb{X}_i^d(e)$ closed

D.3.14 Close Passive Transfer Delivery Challenge $\mathbb{X}_i^d(e)$

Given a valid $\tau_i^A(e)$, $\text{update}_i(e-1)$ and $\lambda(\mathfrak{T}_i^j(e-1) \in T_i^P(e-1))$ as input from $\mathbb{O}_{\not\subset}$, $\mathbb{V}_{\not\subset}$ marks $\mathbb{X}_i^d(e)$ as closed if it were open within the last *epoch*.

Preconditions:

- $\not\subset$ must not have entered recovery
- $\exists\, \mathbb{X}_i^d(e)$ not older than an *epoch*

Input: $\tau_i^A(e), \text{update}_i(e-1), \lambda(\mathfrak{T}_i^j(e-1) \in T_i^P(e-1))$

1. Verify $\tau_i^A(e)$
2. Validate that $\tau_i^A(e)$ ratifies $\text{update}_i(e-1)$
3. Validate $\lambda(\mathfrak{T}_i^j(e-1) \in T_i^P(e-1))$ with stored $\text{offset}_{\mathfrak{T}}$ from $\mathbb{X}_i^d(e)$
4. Mark $\mathbb{X}_i^d(e)$ closed

D.3.15 Deposit Staked Collateral

$\mathbb{O}_{\not\subset}$ can top-up the collateral staked in favor of a single $\mathbb{P}_i$ by simply transferring it to $\mathbb{V}_{\not\subset}$ in favor of $\mathbb{P}_i$.

Preconditions:

- $\not\subset$ must not have entered recovery

Input: $\mathbb{BC}$ transfer T from $\mathbb{O}_{\not\subset}$ to $\mathbb{V}_{\not\subset}, \mathbb{P}_i$

1. Set $\mathbb{U}_i(e)$ to $\mathbb{U}_i(e) + \mathrm{T}.\text{amount}$

D.3.16 Withdraw Staked Collateral

If the total amount maintained by $\mathbb{V}_{\not\subset}$ as insurance collateral deposit, denoted as $\mathbb{C}$, is greater than $\mathbb{C}(e)$, $\mathbb{C}(e+1)$ and $\mathbb{C}(e+2)$, then $\mathbb{O}_{\not\subset}$ can request to withdraw the excess collateral from $\mathbb{V}_{\not\subset}$.

Preconditions:

- $\not\subset$ must not have entered recovery

Input: amount to be withdrawn w

1. Verify $\mathbb{C} - \mathrm{w} \geq \mathbb{C}(e)$
2. Verify $\mathbb{C} - \mathrm{w} \geq \mathbb{C}(e+1)$
3. Verify $\mathbb{C} - \mathrm{w} \geq \mathbb{C}(e+2)$
4. Transfer w to $\mathbb{O}_{\not\subset}$

D.3.17 Recover Funds

Had any $\mathbb{X}_i^b(e-1)$ or $\mathbb{X}_i^d(e-1)$ not been closed for any i within one *epoch*, or if $\mathbb{O}_{\not\subset}$ fails to commit to $\mathbb{T}_{\not\subset}^A(e)$ within the first *epoch* of e, $\not\subset$ is considered to have shut down and gone into recovery mode, whereby any $\mathbb{P}_i$ may withdraw all their commit-chain funds as of the end of $e-3$, and all their parent-chain deposits starting from $e-2$ by providing $\tau_i^A(e-2)$ to $\mathbb{V}_{\not\subset}$.

Preconditions:

- $\not\subset$ must have entered recovery
- $\mathbb{P}_i$ may not have previously recovered its funds

Input: $\tau_i^A(e-2)$, $\tau_i^C(e-2)$

1. Validate $\tau_i^A(e-2)$, $\tau_i^C(e-2)$
2. Transfer $A_i(e-2) + \mathbb{D}_i(e-2) + \mathbb{D}_i(e-1)$ to $\mathbb{P}_i$
3. Transfer $C_i(e) + \mathbb{U}_i(e-1) + \mathbb{U}_i(e)$ to $\mathbb{P}_i$
4. Mark $\mathbb{P}_i$ as recovered

D.4 $\mathbb{O}_{\not\subset}$ Commit-chain Operator

The commit-chain operator $\mathbb{O}_{\not\subset}$ acts as the facilitator of transfers between members of $\mathbb{P}$, and is designed to behave as follows:

D.4.1 Admit $\mathbb{P}_i$

On request to enter the managed $\not\subset$ instance from a participant, $\mathbb{O}_{\not\subset}$ need only append the participant to $\mathbb{P}$ and acknowledgement its $\text{update}_i^a(e)$ reflecting an empty balance by providing a countersignature on it.

D.4.2 Create $\mathbb{T}_{\not\subset}^{A,C,E}(e)$

After an *eon* $e-1$ is over, $\mathbb{O}_{\not\subset}$ creates $\mathbb{T}_{\not\subset}^{A,C,E}(e)$ by using all confirmed transfer information in $e-1$. This means that for each $\mathbb{P}_i$, the last $\text{update}_i(e-1)$ ratified by $\mathbb{O}_{\not\subset}$ would be used to construct $\mathbb{T}_{\not\subset}^A(e)$ as described in Section 4.3.1. Tree $\mathbb{T}_{\not\subset}^C(e)$ is constructed per $\mathbb{O}_{\not\subset}$'s requirements, and $\mathbb{T}_{\not\subset}^E(e)$ is createt per the $\mathbb{E}$ values in $\mathbb{B}^G(e)$.

In case the consistency verification methods from Section 4.5 are utilized, $\mathbb{O}_{\not\subset}$ will also have to compute $\mathbb{O}(\mathbb{P}\log|\mathbb{P}| + T\log|T|)$ verification subroutines, where:

$$T = \bigcup_{P_i \in \mathbb{P}} T_i^a(e)$$

D.4.3 Commit $\mathbb{T}_{\not\subset}^{A,C,E}(e)$

After the creation of $\mathbb{T}_{\not\subset}^A(e)$ and $\mathbb{T}_{\not\subset}^C(e)$, $\mathbb{O}_{\not\subset}$ needs to commit each $t_{root}(e)$ to $\mathbb{V}_{\not\subset}$ within the first *epoch* of e, or be halted in $\mathbb{V}_{\not\subset}$.

D.4.4 Provide $\tau_i^{A,C,E}(e)$

After constructing $\mathbb{T}_{\not\subset}^{A,C,E}(e)$, $\mathbb{O}_{\not\subset}$ communicates each $\tau_i^{A,C,E}(e)$ to its respective $\mathbb{P}_i$ such that $\mathbb{P}_i$ can verify the integrity of its commit-chain balance or issue a challenge if need be. $\tau_i^C(e)$ is to be provided to each $\mathbb{P}_i$ to prevent them from assuming that no collateral will be available in *eon* $e+2$ to cover their transfers in case of failure.

D.4.5 Deliver Transfers

$\mathbb{O}_{\not\subset}$ requires a transfer $\mathfrak{T}_j^i(e)$ from a $\mathbb{P}_i$ to a $\mathbb{P}_j$ to proceed as follows:

1. $\mathbb{P}_i$ sends a new signed $\text{update}_i^a(e)$ to $\mathbb{O}_{\not\subset}$ with $T_i(e) \cup \mathfrak{T}_j^i(e)$.
2. $\mathbb{P}_j$ sends a new signed $\text{update}_j^a(e)$ to $\mathbb{O}_{\not\subset}$ with $T_j(e) \cup \mathfrak{T}_j^i(e)$.
3. $\mathbb{O}_{\not\subset}$ ratifies both $\text{update}_i^a(e)$ and $\text{update}_j^a(e)$ and sends its signatures to $\mathbb{P}_i$ and $\mathbb{P}_j$ respectively.

$\mathbb{O}_{\not\subset}$ must enforce that a $\mathbb{P}_i$ may only have one transfer ongoing at a time. Abortion prior to the last confirmation by $\mathbb{O}_{\not\subset}$ may be signaled via peripheral messages.

D.4.6 Deliver Passive Transfers

$\mathbb{O}_{\not\subset}$ requires a passive transfer $\mathfrak{T}_j^i(e)$ from a $\mathbb{P}_i$ to a $\mathbb{P}_j$ to proceed as follows:

1. $\mathbb{P}_i$ sends a new signed $\text{update}_i^a(e)$ to $\mathbb{O}_{\not\subset}$ with $T_i^a(e) \cup \mathfrak{T}_j^i(e)$.
2. $\mathbb{O}_{\not\subset}$ inserts $\mathfrak{T}_j^i(e)$ into $T_j^p(e)$
3. $\mathbb{O}_{\not\subset}$ sets $S_i^p(e)$ to $R_j^p(e)$
4. $\mathbb{O}_{\not\subset}$ adds $\mathfrak{T}_j^i(e)$.amount to $R_j^p(e)$
5. $\mathbb{O}_{\not\subset}$ ratifies $\text{update}_i^a(e)$ and sends its signature to $\mathbb{P}_i$.

$\mathbb{O}_{\not\subset}$ must enforce that a $\mathbb{P}_i$ may only have one transfer ongoing at a time. Abortion prior to the last confirmation by $\mathbb{O}_{\not\subset}$ may be signaled via peripheral messages. The new $\text{update}_i^a(e)$ must set $\text{offset}_{\mathfrak{T}}$ of the last outgoing passive transfer in e to the previous value of $S_i^p(e)$ in addition to authorizing the new $\mathfrak{T}_j^i(e)$.

D.4.7 Credit Deposits $\mathbb{D}_i(e)$

$\mathbb{O}_{\not\subset}$ is required to monitor $\mathbb{V}_{\not\subset}$ and properly credit all deposits $\mathbb{D}_i(e)$ made by every $\mathbb{P}_i$ or face balance update challenges in the next *eon*. This is done by simply increasing the allotment $A_i(e+1)$ for a deposit made in e such that Equation 23 holds for $e+1$.

D.4.8 Proxy Withdrawals $\mathbb{W}_i(e)$

A $\mathbb{P}_i$ can request instant liquidity from $\mathbb{O}_{\not\subset}$ after initiating a withdrawal using $\mathbb{V}_{\not\subset}$. The $\mathbb{O}_{\not\subset}$ should then, if it decides that the withdrawal is valid, and if it possess sufficient funds to cover it, use $\mathbb{V}_{\not\subset}$ to act as a proxy for this withdrawal, where $\mathbb{O}_{\not\subset}$ directly sends $\mathbb{P}_i$ its requested withdrawal amount in exchange for being the recipient of the withdrawal it requested in $\mathbb{V}_{\not\subset}$ once two *eons* pass and it is confirmed.

D.4.9 Close challenges $\mathbb{X}_i^b(e)$, $\mathbb{X}_i^d(e)$

A $\mathbb{P}_i$ may issue a challenge via $\mathbb{V}_{\not\subset}$ at any moment. $\mathbb{O}_{\not\subset}$ needs to monitor $\mathbb{V}_{\not\subset}$ for these challenges and issue appropriate responses to close them, or risk being halted. It is guaranteed that an honest $\mathbb{O}_{\not\subset}$ will always have the information required to construct a valid call to $\mathbb{V}_{\not\subset}$ to close invalid challenges.

D.4.10 Manage Instant Finality Collateral $\mathbb{C}$

$\mathbb{O}_{\not\subset}$ is in charge of re-assigning the instant finality collateral every *eon*. It should withdraw from, or deposit to, the insurance collateral pool using $\mathbb{V}_{\not\subset}$ to expand or shrink it to cover more users to reclaim un-utilized collateral.

D.5 $\mathbb{P}$ Users

Members of $\mathbb{P}$ are the main parties interested in transferring funds to each other in $\not\subset$, and are designed to behave as follows:

D.5.1 Join $\not\subset$

A $\mathbb{P}_i$ wishing to join a $\not\subset$ instance during *eon e* need only do so through $\mathbb{O}_{\not\subset}$ by providing a signed $\text{update}_i^a(e)$ and waiting for acknowledgement in the form of a countersignature. The update should reflect an empty account within the $\not\subset$ instance.

D.5.2 Audit $\tau_i^A(e)$ and $\tau_i^C(e)$

$\mathbb{P}_i$ must ensure that it always receives a valid $\tau_i^A(e)$ (acceptable by $\mathbb{V}_{\not\subset}$) every *eon e* from $\mathbb{O}_{\not\subset}$ to maintain custody of its funds throughout the time progression and enforce correct transfer delivery by $\mathbb{O}_{\not\subset}$.

$\mathbb{P}_i$ must also ensure that $\tau_i^C(e)$ is valid, or otherwise infer $C_i(e+2)$ to be equal to zero, to verify the amount guaranteed to be delivered to it in case of failure.

D.5.3 Send Transfer

A $\mathbb{P}_i$ wishing to enact a $\mathfrak{T}_j^i(e)$ to a $\mathbb{P}_j$ during eon e sends a signed $\text{update}_i^a(e)$ to $\mathbb{O}_{\not\subset}$ and notifies $\mathbb{P}_j$ to send a signed $\text{update}_j^a(e)$ to $\mathbb{O}_{\not\subset}$ that authorizes the transfer's receipt. $\mathbb{P}_i$ should expect $\mathbb{O}_{\not\subset}$ to return its own signature on $\text{update}_i^a(e)$, after $\mathbb{P}_j$ submits its receipt to $\mathbb{O}_{\not\subset}$, before proceeding with sending or receiving further transfers. Moreover, $\mathbb{P}_i$ may not attempt to initiate any other transfers until $\mathbb{O}_{\not\subset}$ countersigns $\text{update}_i^a(e)$.

D.5.4 Receive Transfer

A $\mathbb{P}_i$ notified of a transfer $\mathfrak{T}_i^j(e)$ by a $\mathbb{P}_j$ should hand over a signed $\text{update}_i(e)$ to $\mathbb{O}_{\not\subset}$ reflecting receipt and wait for a countersignature on $\text{update}_i(e)$ by $\mathbb{O}_{\not\subset}$ to confirm delivery commitment before proceeding with further transfers. Again, $\mathbb{P}_i$ may not initiate any other transfers until $\mathbb{O}_{\not\subset}$ countersigns $\text{update}_i(e)$.

D.5.5 Send Passively Delivered Transfer

A $\mathbb{P}_i$ wishing to enact a $\mathfrak{T}_j^i(e)$ to a $\mathbb{P}_j$ that can be delivered without requiring $\mathbb{P}_j$ to come online during eon e sends a signed $\text{update}_i^a(e)$ to $\mathbb{O}_{\not\subset}$. $\mathbb{P}_i$ should expect $\mathbb{O}_{\not\subset}$ to return its own signature on $\text{update}_i^a(e)$ inserting it into $T_j^p(e)$. $\mathbb{P}_i$ may not attempt to initiate any other transfers until $\mathbb{O}_{\not\subset}$ countersigns $\text{update}_i^a(e)$ and reveals the new value of $S_i^p(e)$.

Moreover, $\mathbb{P}_i$ should send the countersigned $\text{update}_i^a(e)$ to $\mathbb{P}_j$ directly to notify it that it should expect an incoming transfer to be credited in $T_j^p(e)$.

D.5.6 Deposit $\mathbb{D}_i(e)$

Users that wish to deposit into $\not\subset$ must do so only while in possession of a $\tau_i^A(e)$, or a ratified $\text{update}_i(e)$ if this is the first *eon* for $\mathbb{P}_i$ in $\not\subset$, and only if $\not\subset$ is not in recovery. The deposit is done through sending a $\mathbb{BC}$ transaction to $\mathbb{V}_{\not\subset}$.

D.5.7 Withdrawal $\mathbb{W}_i(e)$

To withdraw funds during *eon e*, users utilize their $\tau_i^A(e-1)$ and not attempt to overdraw beyond their minimum within the current and past *eon*, or face their withdrawals being cancelled by an honest $\mathbb{O}_{\not\subset}$, or cancelled by the halt of $\not\subset$. After the first *epoch* of $e+2$ passes successfully, users may claim $\mathbb{W}_i(e)$ on $\mathbb{BC}$ using $\mathbb{V}_{\not\subset}$.

D.5.8 Issue $\mathbb{X}_i^b(e)$

If $\mathbb{O}_{\not\subset}$ does not provide a valid $\tau_i^A(e)$ after commitment to $\mathbb{T}_{\not\subset}^A(e)$, a $\mathbb{P}_i$ should issue a $\mathbb{X}_i^b(e)$ using $\mathbb{V}_{\not\subset}$.

D.5.9 Issue $\mathbb{X}_i^d(e)$

When shown proof of debit (signed $\text{update}_j(e)$ by $\mathbb{O}_{\not\subset}$) not reflected by an authorized credit in $\tau_i^A(e)$, a $\mathbb{P}_i$ should issue a $\mathbb{X}_i^d(e)$ to $\mathbb{V}_{\not\subset}$. Unless $\mathbb{P}_j$ is malicious, $\mathbb{O}_{\not\subset}$ will not be able to close $\mathbb{X}_i^d(e)$. $\mathbb{P}_j$ should also issue the challenge in case of $\mathbb{P}_i$'s noncooperation or if a receipt confirmation is not provided.

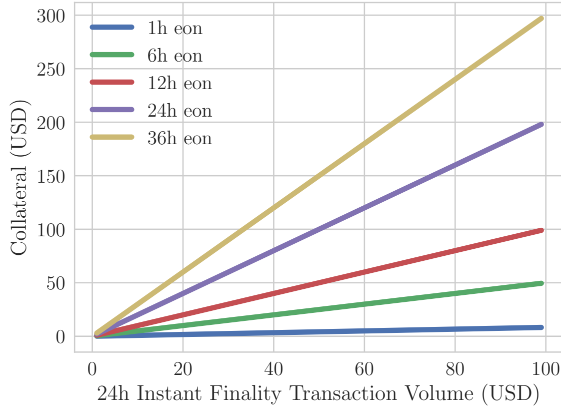

Figure 9: Collateral by $\mathbb{O}_{\not\subset}$ to provide a given transaction volume within 24 hours, with instant finality. Given e.g. an *eon* interval of 24h, 20M USD volume, $\mathbb{O}_{\not\subset}$ must stake 40M USD to provide instant transaction finality.

More verbosely, given $\tau_i^A(e)$ and $\tau_j^A(e)$, along with the corresponding $\text{update}_i(e-1)$, $\text{update}_j(e-1)$, $\mathbb{P}_i$, the recipient, can determine in advance whether $\mathbb{O}_{\not\subset}$ would be able to close a corresponding $\mathbb{X}_i^d(e)$ for a $\mathfrak{T}_i^j(e-1)$. Since $\mathbb{P}_i$ is always aware of all the contents of $\text{update}_i(e-1)$, it can determine if a corresponding credit authorization in $\text{update}_i(e-1)$ is missing for the debit authorization in $\text{update}_j(e-1)$. If so, from the definition of $\mathbb{V}_{\not\subset}$, a $\mathbb{X}_i^d(e)$ against this $\mathfrak{T}_i^j(e-1)$ would not be closeable by $\mathbb{O}_{\not\subset}$.

D.5.10 Recover Funds

Upon $\mathbb{O}_{\not\subset}$'s failure to close any challenge within one *epoch* and before e ends, $\not\subset$'s time progression stops at e and it enters into recovery. Every $\mathbb{P}_i$ will need to recover its confirmed commit-chain funds through $\mathbb{V}_{\not\subset}$.

E Extended Evaluation

E.1 Instant Transaction Collateral Discussion

In Figure 9, we visualize the collateral for $\mathbb{O}_{\not\subset}$ to stake to provide instant transaction finality to its users.

At an *eon* interval of 24 hours, the operator must stake e.g. 20M USD to provide instant transaction finality of 10M USD towards its users within 24 hours, at e.g. $n = 1$M users, each receiving at most 10 USD. The operator could now choose to halve the *eon* interval (from 24h to 12h), to halve the required stake (from 20M to 10M USD), while still providing the same instant transaction finality volume within 24 hours. The reduction in *eon* time, however, also reduces the maximum transaction amount that a user can instantly receive within an *eon*. Following the previous example, at an eon interval of e.g. 12h, a user would be able to accept instantly at most 5 USD.

Note that stake must be *individually* allocated within a checkpoint commitment (cf. Section 4.3.4).

Users	*eon*	Per user/*eon*	Per user/24h	Stake
1M	24 hours	1000 USD	1000 USD	2B USD
1M	12 hours	1000 USD	2000 USD	2B USD
1M	24 hours	100 USD	100 USD	0.2B USD
1M	12 hours	100 USD	200 USD	0.2B USD

Table 1: Example numbers to quantify the potential stake for $\mathbb{O}_{\not\subset}$ to provide instant transaction finality to its users.

E.2 Libsnark Evaluation

Table 2 presents the respective constraints, variables, inputs, and utilization frequency of our libsnark implementation. The measured generation and proving times are taken on a locally running machine with an Intel i7-7700K 4.20GHz. The total time required to generate a complete proof of checkpoint consistency can be estimated using the utilization frequency column in conjunction with the proving time for each procedure.

Procedure	**Constraints**	**Variable**	$\mathbb{V}$ **Inputs**	**Generation**	**Proving**	**Recurrence**	$\lvert\pi\rvert$
Merkle Membership	132 109	132 079	5	5.48s	3.83s	$\lvert T^a\rvert - \lvert T^p\rvert$	2 690 bits
+ *Wrapper*	45 129	45 121	5	2.80s	2.59s	$\lvert T^a\rvert - \lvert T^p\rvert$	2 988 bits
Exclusive Allotment	229 377	229 088	8	8.71s	4.88s	$\lvert T^p\rvert + \lvert\mathbb{P}\rvert$	2 690 bits
+ *Wrapper*	54 954	54 946	8	3.15s	2.93s	$\lvert T^p\rvert + \lvert\mathbb{P}\rvert$	2 988 bits
Transfer Inclusion	100 085	100 067	7	5.33s	5.07s	$\lvert T^a\rvert$	2 988 bits
Transfer Delivery	179 394	193 462	5	6.19s	6.89s	$\lvert T^a\rvert$	2 690 bits
+ *Wrapper*	45 129	45 121	5	2.82s	2.57s	$\sum 2\lvert T_i^a\rvert - 1$	2 988 bits
+ *Combiner*	148 349	162 425	5	5.28s	5.91s	$\sum 2\lvert T_i^a\rvert - 1$	2 690 bits
Deposit/Withdrawal	110 828	110 798	5	5.96s	5.39s	$\lvert\mathbb{P}\rvert$	2 988 bits
Exit Notification	91 174	91 146	4	4.47s	4.67s	$\lvert\mathbb{P}\rvert$	2 988 bits
Chain Accumulator	220 407	241 520	6	7.55s	8.03s	$\lvert\mathbb{P}\rvert$	2 690 bits
+ *Wrapper*	48 404	48 396	6	2.89s	2.63s	$\lvert\mathbb{P}\rvert$	2 988 bits
Account Integrity	240 922	272 350	12	8.24s	8.24s	$\lvert\mathbb{P}\rvert$	2 690 bits
+ *Wrapper*	68 054	68 046	12	3.70s	3.65s	$2\lvert\mathbb{P}\rvert - 1$	2 988 bits
+ *Combiner*	197 839	211 894	5	6.75s	7.50s	$2\lvert\mathbb{P}\rvert - 1$	2 690 bits

Table 2: zkSNARKs implemented in libsnark. *Wrapper* circuits convert the generated snark from one verifiable in the MNT6 curve to one verifiable in the MNT4 curve. zkSNARKs proving and generation times expressed in seconds. Measuremed on an Intel i7-7700K 4.20GHz CPU with 4x 8GB 2400 MT/s DDR4 RAM. Measured verification times were $\leq 0.03s$.