Proof of Necessary Work: Succinct State Verification with Fairness Guarantees

Assimakis A. Kattis
New York University
kattis@cs.nyu.edu

Joseph Bonneau
New York University
jcb@cs.nyu.edu

Abstract

Blockchain-based payment systems utilize an append-only log of transactions whose correctness can be verified by any observer. In almost all of today's implementations, verification costs grow linearly in either the number of transactions or blocks in the blockchain (often both). We propose a new distributed payment system which uses Incrementally Verifiable Computation (IVC) to enable constant-time verification. Since generating the succinct proofs needed to verify correctness is more expensive, we introduce the notion of Proof of Necessary Work (PoNW), in which proof generation is an integral part of the proof-of-work used in Nakamoto consensus, effectively producing proofs using energy that would otherwise be wasted. We implement and benchmark a prototype of our system using recent recursive SNARK-based constructions, enabling stateless "light" clients to efficiently verify the entire blockchain history in about 40 milliseconds.

1 Introduction

Balancing throughput with decentralization is a major challenge in modern cryptocurrencies. In this work, our goal is to design a payment system supporting efficient verification of the system's entire history by any participant *without* trusting any third parties. Participants can join the system at any time and need only to obtain some fixed public parameters of modest size from a trusted source (e.g. the genesis block and the system's rules).

Current systems such as Bitcoin require participants to process the entire system history to verify that the current state (the most recent block in the chain) is correct. This requirement makes joining the system prohibitive for many clients, as downloading and verifying over 300 GB of system history takes days on an ordinary laptop. In practice, most clients don't perform independent verification and rely on a trusted third party to assert the current state of the system.

We address this problem using succinct proofs of state validity. These enable clients to verify any snapshot of the system using minimal bandwidth and time, even if they have no other information except the genesis state and transaction validity rules. For any *block* in the system, these proofs demonstrate both that there exists a sequence of valid transactions from the genesis state $\mathcal{S}_0$ to the current block, and that the block's *branch* (the sequence of predecessor blocks) is of a claimed quality q according to the consensus protocol. In this work we focus on aggregate proof-of-work difficulty as the measure of branch quality, which is the same as in Bitcoin.

Currently, systems such as Bitcoin or Ethereum require $O(t+h)$ work to completely verify a branch containing t transactions and h blocks. We are able to achieve optimal asymptotic performance of $O(1)$ verification costs even for a client joining the system at an arbitrary point in its history. Our techniques cannot help a client that is separated (or *eclipsed*) from the genuine system by a network partition. We assume a client can reach at least one node which will provide the most recent block and a proof. The client may also communicate with arbitrarily many attacker-controlled nodes; efficient verification means the client can quickly tell which block is canonical in the system.

Another major issue facing Bitcoin and related cryptocurrencies is their energy consumption. These systems employ proof-of-work-based Nakamoto consensus, which provides system security by publicly verifying energy consumption. This energy consumption, while necessary for the consensus protocol, is not used for anything else and hence is often described as wasted. We design a proof-of-work puzzle which produces correctness proofs for each block as a useful byproduct, thus recycling the expended energy to enable efficient verification. Achieving this requires carefully designing the proof-of-work to replicate the properties of Bitcoin's non-useful puzzle. Our main technical contribution is a method to deeply embed a nonce into the proof computation process, making it suitable as a progress-free proof-of-work puzzle. We formalize this intuition by introducing the notion of ε-amortization resistance, and propose a PoNW design based on this. Our formalization of efficient proof-tagging techniques may be of independent interest.

1.1 Our Contributions

(1) We design and prove the correctness and security of a protocol satisfying *succinct state verification*. This ensures negligible computational requirements for any observer to verify the current system state. Our construction rests on recent advances in Incrementally Verifiable Computation (IVC) [7] [6] using Succinct Non-Interactive ARguments of Knowledge (SNARKs) [29] [5].

(2) We propose a variant of Nakamoto consensus, which we call Proof of Necessary Work. This allows us to compute proofs of block/transaction validity as part of the consensus process, making the work performed useful.

(3) We implement the proof system in (1) with the consensus protocol variant in (2) at an 80-bit security level, benchmark its performance and establish feasibility. Our system:

- produces block headers of size $<$ 500 bytes for any number of transactions per block.
- allows stateless clients to verify a block in $<$ 40ms.
- achieves throughput of 50 tx/block.

Current limitations. In terms of throughput and block header size, our prototype is about an order of magnitude worse than Bitcoin. Bitcoin block headers are 80 bytes and throughput is about 1,000 transactions per block. However, our system allows a stateless client to rapidly verify a block (and thus its complete history) in 40 ms with 500 bytes of data downloaded. In Bitcoin, a comparable full verification of a block requires many hours of computation time and downloading hundreds of gigabytes of data.

Our approach also does not reduce the burden on full/mining nodes. As with Bitcoin, these nodes must download and track the full state of the system and process all transactions. The efficient block verification provided by our system does assist miners in quickly validating new blocks broadcast on the network, which may reduce the risk of block collisions and enable faster block frequencies.

Our payment system is also slightly simpler than Bitcoin, supporting only signature-based payments by individual public keys. It does not replicate Bitcoin's rudimentary script system nor support features such as multisig transactions. However, we do provide a commitment to state in each block (like Ethereum but unlike Bitcoin) enabling efficient proofs of a user's current balance.

2 Prior Work

2.1 Light Client Verification

The idea of providing portions of the blockchain to light clients for verification began with Bitcoin [28], where Simple Payment Verification (SPV) clients download only block headers and Merkle inclusion proofs for specific transactions to be convinced of their validity. While this approach forgoes downloading the whole blockchain, clients must trust that the downloaded blocks contain only valid transactions due to incentives provided to miners which discourage mining invalid blocks. This approach also still requires a linear amount of memory with respect to chain length. For Bitcoin downloading only block headers requires about 10 kB per day in bandwidth, which is reasonable for up-to-date clients but non-trivial for new clients which must download the entire chain of headers (currently about 40 MB and growing).

Sublinear memory complexity in SPV clients through skip-lists was first formally analyzed in [22] [23] [21] by Kiayias et al., where the authors propose keeping pointers to multiple previous blocks at every step to allow for fast verification. This allows the protocol to check for high-difficulty previous blocks (or 'superblocks'), of which verifying a logarithmic number suffices to ensure security for the whole chain. This approach, however, is only feasible in the regime of fixed difficulty and thus cannot be implemented as is in current decentralized networks.

Flyclient [14] also guarantees logarithmic complexity for transaction and proof-of-work verification, and is secure with high probability under variable difficulty even if fractions of the network are adversarially controlled. This is achieved by using Merkle Mountain Range Commitments to achieve memory improvements, and a random block sampling protocol to ensure security. However, Flyclient still requires resources linear with respect to each new transaction, and storage requirements still grow with blockchain size.

Neither the work of Kiayias et al. or Flyclient enable efficient verification of transaction validity, both rely on the same argument from Bitcoin that economic incentives discourage mining a long chain of blocks containing incorrect transactions.

By contrast, while Mimblewimble [30] does not provide sublinear verification guarantees with respect to block header size h, it contributes an innovative framework for transaction verification. By compressing state in a 'UTXO set' of $u \leq t$ transactions that adaptively updates with each block, it provides asymptotically better transaction verification. In Mimblewimble, history verification is linear only in the number of currently unspent coins, not the total number of transactions. While we could theoretically adapt these techniques in our work to decrease proving costs, we choose not to given practical observations that the number of unspent coins is not much smaller than the total number of transactions. The more complex predicate to verify state in MimbleWimble would likely negate any gains in proving time from compressing some transaction history.

In Table 1, we provide a comparison of the asymptotic time and memory requirements of existing SPV protocols implementing transaction and/or proof-of-work verification. Given that transaction volume and chain length both grow linearly

Technique	**Transaction Verification**	**PoW Verification**	**Memory Requirements**
Bitcoin [28]/Ethereum [12]	$\Theta(t)$	$\Theta(h)$	$\Theta(h+t)$
Mimblewimble [30]	$\Theta(u) = O(t)$	$\Theta(h)$	$O(\log^4(h))$
NIPoPoW [23]	$\Theta(t)$	$\text{polylog}\,(h)$	$\log h \cdot (\log t + \log\log h)$
FlyClient [14]	$\Theta(t)$	$O(\log^2 h)$	$O(\log^2 h)$
This work	$O(1)$	$O(1)$	$O(1)$

Table 1: Previous Work on Light-Client Verification: Asymptotic state and proof-of-work verification times for clients verifying t transactions in h Blocks alongside client memory requirements.

over time, ideally we can enable verification that is constant with respect to both. The only other work we are aware of with this goal is the Coda Protocol framework [1]. Coda takes a similar high-level approach as our work, encoding state transitions in a recursive proof system to asymptotically optimal verification time. Our approaches are independent and vary in a number of technical details around predicate structure, with Coda choosing a different design for transaction proof aggregation. Most importantly, the main conceptual differences lie in our choice of consensus protocol. Coda implements a proof-of-stake [25] system, which must be carefully adapted for the succinct proof setting. By contrast, we implement a proof-of-work system, which requires tackling an orthogonal set of design challenges to adapt to the succinct proof setting.

2.2 Useful Proof of Work

The proof-of-work process in Bitcoin and most modern cryptocurrencies is based on HashCash [2] and involves solving a hard puzzle for which the difficulty can be adaptively set according to the number of participants. Hardness here is taken to mean that no adversary can compute solutions to the puzzle faster than randomly guessing. An important property of such systems is that they are *memoryless*, or that the probability of winning does not depend on time spent computing a solution. It is important to ensure that the proof-of-work process is fair, meaning that a miner's hashrate is directly proportional to their computational power and hence that large miners do not enjoy algorithmic efficiency gains with growth. This is necessary to ensure that the network remains decentralized; without this property there would be a mathematical incentive for miners to consolidate. Of course, there may be economic and logistic incentives for miner consolidation (e.g. reduced administrative overhead) but we consider these out-of-scope.

It has long been an open challenge to design a proof-of-work puzzle that is both suitable for Nakamoto consensus and also *useful* for some independent purpose [8]. In addition to being memoryless, the puzzle must satisfy several other properties such as being generated from public parameters and allowing fine-tuned adjustment of difficulty. To date, the only candidates for useful proof-of-work puzzles are highly structured problems of questionable public utility, such as finding long Cunningham chains of related prime numbers [24] or tables of relations for solving discrete log computations [20].

In this work, we identify a new approach to useful proof work in proposing that the work not solve an external problem, but for aiding in verification of the system itself. We denote this as *Proof of Necessary Work* and show how it can be used concurrently with the succinct blockchain architecture proposed above as a suitable proof-of-work puzzle.

3 Preliminaries

We first define a distributed payment system in general form and then discuss required security properties. We model the processing of a payment between Alice and Bob as a transition of some state $\mathcal{S}_i$ to a new state $\mathcal{S}_{i+1}$ that represents this change of funds in a state machine. A state machine is comprised of a set of states, their transitions, and the current system state. We can assign to each state machine a state transition function, which allows the transition from one state to another given some information as input. Moreover, we work under the assumption that this is a *replicated state machine*, requiring local copies of the state machine in each client in order to achieve fault tolerance.

Definition 1. A replicated state machine Σ^n with n state transitions is a tuple $\Sigma^n = (\mathcal{S}_i, \mathbf{t}_i, z_i)_{i=1}^n$ of states $\mathcal{S}_i \in \mathcal{S}$, sets of transactions $\mathbf{t}_i \in 2^{\mathcal{T}}$ (where $\mathcal{T}$ is the set of valid transactions), and witnesses $z_i \in \{0,1\}^*$. We denote $\mathcal{S}_n$ as the *current* state of Σ^n and $\mathcal{S}_0$ as its *genesis* state.

3.1 Model Setup

We define our payment system state machine as follows: we have a set of participants who share a broadcast communication channel, and who may join or leave the system at will. There are two types of nodes we concern ourselves with here: miners and light clients.

Miners: A mining (or full) node has access to the current state $\mathcal{S}_i \in \mathcal{S}$ at timestep i, and is responsible for performing proof-of-work and verifying state transitions.

Light Clients: Light clients (or end-users) can issue transactions $t \in \mathcal{T}$ and verify their inclusion, but do not need to keep mutable state.

We investigate how the system transitions from $\mathcal{S}_i$ to $\mathcal{S}_{i+1}$ while retaining consensus over state. Transitions between states happen through the processing of transactions by a model-specific transition function NewState. We also require a transition validation function VerifyState that ensures the state update was done correctly. By defining the notion of *validity* between state transitions, we differentiate between legitimate and illegitimate transactions and only permit processing of the former. Moreover, we require that such tuples are also internally consistent, namely that all new states are correctly validated. These are specified below.

Definition 2. A tuple of efficiently computable algorithms $(\mathsf{VerifyState}, \mathsf{NewState})$ is considered a *transition* tuple if the following conditions hold:

- $\mathsf{VerifyState} : 2^{\mathcal{T}} \times \mathcal{S} \times \mathcal{S} \times \{0,1\}^* \to$Yes/No
- $\mathsf{NewState} : 2^{\mathcal{T}} \times \mathcal{S} \times \{0,1\}^* \to \mathcal{S}$

and moreover we consider such a tuple *consistent* if $\forall \mathcal{S}_i, \mathcal{S}_{i+1} \in \mathcal{S}, \mathbf{t} \in 2^{\mathcal{T}}$:

$$\exists z_i \text{ s.t. } \mathsf{VerifyState}(\mathbf{t}, \mathcal{S}_i, \mathcal{S}_{i+1}, z_i) = \text{Yes}$$
$$\iff \mathsf{NewState}(\mathbf{t}, \mathcal{S}_i) = \mathcal{S}_{i+1}.$$

Definition 3. An RSM $\Sigma^n = (\mathcal{S}_i, \mathbf{t}_i, z_i)_{i=1}^n$ is considered *valid* with respect to a consistent transition tuple $(\mathsf{VerifyState}, \mathsf{NewState})$ if $\mathsf{VerifyState}(\mathbf{t}_i, \mathcal{S}_i, \mathcal{S}_{i+1}, z_{i+1}) = \text{Yes}$ or equivalently $\mathsf{NewState}(\mathbf{t}_i, \mathcal{S}_i) = \mathcal{S}_{i+1}$ for all $i \in [n]$.

The determination of what is considered an allowed state transition for a given system is fully specified by the $(\mathsf{VerifyState}, \mathsf{NewState})$ objects. Thus, by specifying these functions as part of the implementation, we allow our definition to encompass arbitrary types of payment systems—each based on different transition functions and notions of transaction validity. For example, the Bitcoin and Ethereum protocols both define their own transition functions between blocks (states) and each one is based on its own notion of transaction validity.

We can also define the property of *succinct verification* by requiring that VerifyState runs in restricted time and that z_i needs only constant space. This is formalized below.

Definition 4. A valid RSM $\Sigma^n = (\mathcal{S}_i, \mathbf{t}_i, z_i)_{i=1}^n$ with respect to a consistent transition tuple $(\mathsf{VerifyState}, \mathsf{NewState})$ is considered *succinctly verifiable* if $\forall i \in [n]$:

1. z_i has $O(1)$ size,
2. VerifyState runs in time $O(|\mathcal{S}_i|)$,

where the asymptotic bounds are over n, $|\mathcal{S}_i|$ and $|z_i|$.

3.2 Proof Carrying Data

3.2.1 Definitions

We now briefly introduce Proof Carrying Data (PCD), an efficient IVC primitive instantiated using (pre-processing) SNARKs. Consider a set of system states $\mathcal{S}$ with initial state $\mathcal{S}_0 \in \mathcal{S}$. We denote the system's state transition function by UPDATESTATE and construct a predicate $\Pi_{\mathcal{S}}$ that evaluates to 1 on input state $\mathcal{S}_{i+1}$ (or a commitment to it) if and only if there exists a valid transition from some $\mathcal{S}_i$ to $\mathcal{S}_{i+1}$. A prover repeatedly applies state transitions on the initial state to acquire state $\mathcal{S}_n$.

A PCD system allows a verifier that only sees (a commitment to) the last state $\mathcal{S}_n$ and a short proof π_n to be convinced that $\mathcal{S}_n$ is a valid system state, i.e. a state that can be derived from $\mathcal{S}_0$ by applying valid state transitions for all i in the chain. More specifically, a PCD system is comprised of the following three algorithms:

Setup: $\mathcal{G}(\Pi_{\mathcal{S}}, 1^\lambda) \to (pk, vk)$. Key generation takes as input a predicate $\Pi_{\mathcal{S}}$ and a security parameter λ, outputting proving and verification keys.

Prover: $\mathcal{P}(pk, \mathcal{S}_{i+1}, \mathcal{T}, \mathcal{S}_i, \pi_i, w) \to \pi_{i+1}$. The prover takes as input the proving key pk, state $\mathcal{S}_i$, a proof π_i that $\mathcal{S}_i$ is a valid state and a set of transactions $\mathbf{t} \in 2^{\mathcal{T}}$, outputting a proof π_{i+1} that $\mathcal{S}_i \to \mathcal{S}_{i+1}$ is a valid state transition.

Verifier: $\mathcal{V}(vk, B_i, \pi_i) \to$ Yes/No. When given as input the verification key vk, a proof π_i and a commitment B_i to state $\mathcal{S}_i$, the verifier outputs Yes if π_i is a valid proof that state $\mathcal{S}_i$ is valid, outputting No with very high probability otherwise.
A more complete presentation of PCD systems and their security properties can be found in Appendix B.

4 Transaction Layer

Given a pair $(\mathsf{VerifyState}, \mathsf{NewState})$ characterizing some RSM Σ, we can define a distributed payments system (DPS) over Σ as a tuple of algorithms that together ensure the basic functionality of payment remittance over arbitrary transaction objects. This can be thought of as the 'minimal' client that can support a full (verifying) node. In the following definitions, we require the use of a digital signature scheme along with proof-of-work for consensus.

4.1 Preliminaries

We begin with the requirement for a digital signature scheme, which end-users will use to authenticate their corresponding accounts. Its corresponding security properties are standard and formally defined in Appendix C.

In order to instantiate a proof-of-work system, we are required to include (and commit to) q_i and n_i with every proof, where q_i is the quality of state $\mathcal{S}_i$ and n_i the associated nonce. This is because these quantities are needed by miners in order to follow the longest chain and achieve consensus. In addition, we associate the monetary value $c \in \mathbb{N}$ of each account with user address values $z \in \mathcal{Z}$, of which there can be multiple in a given state. This provides us with all the ingredients needed to define the fundamental system.

4.2 Minimal Payment Semantics

We define a distributed payment system (DPS) as a tuple of algorithms necessary for minimal payment functionality. In the definitions below, we denote the supplementary information string by $*$, but make no assumptions about the type of information provided. This is done to ensure that information required by an (arbitrary) transition function is encompassed by our definition.

Definition 5. Given a consistent RSM Σ, a Distributed Payment System (DPS) is a tuple $\Delta(\Sigma)$ consisting of:
Setup : $1^\lambda \to pp$

- INPUTS: Security parameter λ
- OUTPUTS: Public parameters pp

NewCoinbase : $S \times z_a \times c \times (pk, sk)_a \times * \to t$

- INPUTS: Subset of current state $S \in \mathcal{S}_i$, z_a address of sender a, c value transferred, public-private key pair $(pk, sk)_a$
- OUTPUTS: Transaction t

NewTransaction :

$$S \times z_{\{a,b\}} \times c_{\{a,b\}} \times (pk, sk)_a \times pk_R \times * \to t$$

- INPUTS: Subset of current state $s \in \mathcal{S}_i$, $z_{a,b}$ addresses of sender/receiver, $c_{\{a,b\}}$ value transferred, public-private key pair $(pk, sk)_a$, public key pk_R
- OUTPUTS: Transaction t

VerifyTransaction : $t \times S \times * \to$ Yes/No

- INPUTS: Subset of current state $s_1 \in \mathcal{S}_i$, transaction t
- OUTPUTS: Yes/No

NewState : $S \times \mathbf{t} \times * \to \mathcal{S}_{i+1}$

- INPUTS: Subset of current state $S \in \mathcal{S}_i$, list of ordered transactions $\mathbf{t} \in 2^{\mathcal{T}}$
- OUTPUTS: State $\mathcal{S}_{i+1}$

VerifyState : $\mathbf{t} \times S_1 \times S_2 \times * \to$ Yes/No

- INPUTS: Subsets of current and next state $S_1 \in \mathcal{S}_i, S_2 \in \mathcal{S}_{i+1}$, list of ordered transactions $\mathbf{t} \in 2^{\mathcal{T}}$
- OUTPUTS: Yes/No

CreateAddress : $pp_S \to (pk, sk)$

- INPUTS: Public parameters pp_S
- OUTPUTS: New public/private keys $pk, sk \in \{0,1\}^*$

GetBalance : $S \times pk \times * \to c$

- INPUTS: Subset of current state $S \in \mathcal{S}_i$, pk corresponding to a **CreateAddress** output
- OUTPUTS: Balance c corresponding to pk

GetQuality : $S \to q$

- INPUTS: Subset of current state $S \in \mathcal{S}_i$
- OUTPUTS: Quality q of state $\mathcal{S}_i$

In terms of security, the system needs to provide both completeness and correctness guarantees. This requires that the protocol should guarantee that state transitions considered correct by VerifyState will not be rejected by compliant nodes. Similarly, satisfying correctness requires that transactions and state transitions that are invalid should not be accepted by compliant nodes. These definitions are constructed in the usual way, and we defer their formal specification to Appendix F.

4.3 Compatibility with Existing Protocols

Our model can easily be adapted to describe existing blockchain-based payment systems. We illustrate this explicitly for Bitcoin to provide intuition for what the essential components of a distributed payments system are.

Bitcoin: The Bitcoin protocol is a UTXO-based payment clearing system, for which a valid block update includes a set of valid ordered transactions and specific block header information. The components of the RSM are illustrated below:

- **Transactions:** The UTXOs in each block.
- **State:** The current block.
- **Witness:** Not required here, since validation happens by inspection of the ledger.
- **NewState:** Generation of a new block.

- **VerifyState:** Validity of a block transition requires:
  - Verifying all UTXO Merkle paths.
  - Verifying that the header is well formed.
  - Checking the nonce satisfies proof-of-work.
  - Ensuring all transactions are valid.

A similar treatment would allow us to characterize Ethereum using the same basic components. This paradigm also makes obvious that, in order to verify the state of the *whole* system without any external information, we would need to iteratively validate each state transition. We use the witness z_i to provide 'hints' to the validation function, which we will demonstrate later allows us to construct protocols tailored for much more efficient state verification.

5 A DPS Designed for Succinct Verification

Here we demonstrate a specific instantiation of a DPS for which we define a transition function tailored to fast state verification by stateless clients. To achieve this, we leverage the capabilities of IVC systems and construct a succinct proof of state validity to represent each state transition. Since we will be basing our implementation of the proofs on SNARKs, we design the transition function so as to minimize SNARK proof sizes. This is critical for efficiency and establishing feasibility.

Our system uses Nakamoto consensus, updating the difficulty d of solving a given puzzle according to block timestamps. Compliant miners and light clients follow the longest chain of state transitions, assigning a (monotonically increasing) score q_i to each new block. The chain with the highest score is then considered the valid chain. The system uses proof-of-work, requiring that the hash of the current block be less than the mining difficulty d, exactly like in Bitcoin.

Each participant in our system has a public and secret key that they generate when they first join the network. The participants use these keys to digitally sign transactions and verify other participants' signatures. The state $\mathcal{S}_i$ contains the distribution of money between the participants (stored as a tree), state quality and a nonce corresponding to the most recent proof-of-work. We also distinguish between the i-th block, which in our case will be represented by a proof π_i that the i-th state transition is valid along with the set of transactions $\mathbf{t}_i$ corresponding to the transition, and *commitments* to state, which we denote by B_i and use for client verification. We require an account-based system (unlike Bitcoin) and keep track of state with an 'Account Tree' of all account-value pairs. These building blocks are:

Account Tree: We use a Merkle tree construction with a compressible Collision Resistant (CRT) hash function $\mathcal{H} : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$. We assume a fixed size tree T with height h throughout.

State: We denote S_i the state after the i-th update:

1. Account tree T^i with leaves the accounts in state. Leaves are lexicographically ordered based on their address.
2. The block number i.
3. The quality q_i.
4. The nonce n_i.

State Commitment: Set B_i as the commitment to S_i:

1. The root rt_i of the Account tree T^i in $\mathcal{S}_i$.
2. The block number i.
3. The quality q_i.
4. The nonce n_i.

Protocol Initialization: We assume that initially all accounts in the Account tree are set to null. In every transition, the tree allows the following modifications:

1. *Account Initialization:* Set the public key to a non-null value and initialize the balance and the nonce. An account with a non-null public key is considered initialized. An account can be initialized only once. Uninitialized accounts have null public key.
2. *Balance Update:* Modify the balance bal of an account, ensuring that money is conserved.
3. *Nonce Update:* Modify the nonce n of an account to that of the current block.

We denote the initial state of the system (or "genesis state") by $\mathcal{S}_0$; this is agreed to by an out-of-band process. For example, a system might start with all addresses having a balance of zero or it might pre-populate some accounts with non-zero balance (colloquially known as "pre-mining"). Note that in the initial state, the Account tree is a full tree and contains one leaf/account for every address that can exist in the state. The genesis state can contain initialized and uninitialized accounts.

5.1 Basic Data Structures

We define the basic data structures of our system below. We require the standard assumption of a public-key signature scheme, and take all signatures below as over some common field $\mathbb{F}$.

Account: An account $\mathbf{a}$ is a tuple $(\text{addr}, \text{PK}, bal, n)$ where:

1. addr is the address of $\mathbf{a}$.
2. PK is the public key of $\mathbf{a}$.
3. bal the balance of the account (non-negative).

4. n the nonce of the block that contains the transaction that last modified $\mathbf{a}$.

Address & Public Key: Each PK does not have to have a unique address. Decoupling user keys from account addresses is done to represent each account in the tree with minimal size requirements (and regardless of public key sizes) within the proof.

Transactions: There are two types of transactions:

1. Coinbase:
$$t_c = (\text{addr}, v, \text{PK}, \sigma)$$
   (a) addr: the address of the recipient
   (b) v: the value it receives
   (c) σ: The digital signature of the transaction

2. Standard
$$t = (\text{addr}_s, \text{addr}_r, v, f, \text{PK}, \sigma, \text{PK}_R)$$
   (a) $\text{addr}_s, \text{addr}_r$: the addresses of the sender and the receiver respectively
   (b) v the value to be transferred from the sender to the receiver (it is a positive integer)
   (c) f: Total mining fee provided
   (d) σ: The signature of the transaction
   (e) PK: the public key that validates σ
   (f) PK_R: the public key of the recipient

5.2 State Transition Semantics

Below we define our semantics used for transaction and state transition validity. A full specification of the rest of the algorithms can be found in A.

Verifying Transactions: $\textsc{VerifyTx}(t, T^i) \rightarrow$Yes/No takes as input a transaction t and an Account tree T^i, outputting Yes/No (1 or 0). A transaction is considered valid if:

1. Sender and receiver are legitimate accounts in T^i.
2. Amount transferred is not more than sender's balance.
3. Signature authenticates over the sender's public key.
4. Sender and receiver accounts in the Account tree are updated correctly.
5. Recipient's public key matches that stored in the Account tree, or the address is uninitialized.

Updating System State: $\textsc{UpdateState}(\mathcal{S}_i, \mathbf{t}, n) \rightarrow \mathcal{S}_{i+1}$ is a procedure that takes as input a state $\mathcal{S}_i$, am ordered set of transactions $\mathbf{t}$ with $|\mathbf{t}| = N$ and a nonce n. It outputs the next state $\mathcal{S}_{i+1}$ and a witness w of objects proving the update was done correctly. A system transition is considered valid if:

1. All transactions in $\mathcal{T}$ are valid.
2. The previous state has performed proof-of-work.
3. Only last transaction t_N is of coinbase type.
4. Each transaction builds on top of a previous transaction, except the first which builds on the previous root.

All of this is registered in $\mathcal{S}_{i+1}$.

5.3 State Transition as an NP statement

In order to instantiate a DPS that is capable of verifying a given state transition function, we encode the transition function ValidState as a compliance predicate $\Pi_{\mathcal{S}}$. With every state transition, we include a proof that the transition was $\Pi_{\mathcal{S}}$ compliant. This is done by verifying the transition from the previous state and producing an attesting witness w in the process. In this context, we are interested in verifying the transition between two states of the Account tree by processing transactions between them into the system. This is achieved by tracking changes to the root rt_i of the Account tree after the input of each transaction.

We capture all requirements for transaction, proof-of-work and state validity in an NP language BLOCK-V that only accepts commitments of the form $B_i = (rt^i, q_i, i, n_i)$ that build 'correctly' on top of a previous state. At a high level, the elements of this language are state commitments that, given some previous state's root, have only processed valid transactions.

5.3.1 Compliance Predicate

Given input $B_{i+1} = (rt_{i+1}, i+1, q_{i+1}, n_{i+1})$, the compliance predicate $\Pi_{\mathcal{S}}$ evaluates to 1 if and only if all of the following are satisfied:

1. There exists a previous state $\mathcal{S}_i$ satisfying proof-of-work with nonce n_i and quality q_i.
2. There exists a tuple of ordered transactions $\mathbf{t}$ with $|\mathbf{t}| = N$. These transactions need to be sequentially valid with respect to $\mathcal{S}_i$.
3. $\textsc{UpdateState}(\mathcal{S}_i, \mathcal{T}, n_i) = \mathcal{S}_{i+1}$.

We use the compliance predicate $\Pi_{\mathcal{S}}$ to design a PCD system consisting of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, as formally defined in Appendix B, where each message z_i is a state commitment object B_i.

5.4 DPS Specification

Here we define how the system transitions from $\mathcal{S}_i \rightarrow \mathcal{S}_{i+1}$. The first method generates a new state and associated proof of compliance, along with a nonce certifying that the system performed proof-of-work. When validating, we check that the new state $\mathcal{S}_{i+1}$ is a valid next state for the system by being (a) $\Pi_{\mathcal{S}}$ compliant and (b) providing proof-of-work. Note that the validation only requires the *root* of the Account tree corresponding to $\mathcal{S}_i$, thus making it efficient enough for light clients.

Algorithm 1 NewState

```
Input: $pp, \mathcal{T}, \mathcal{S}_i, \pi_i$
Output: $\mathcal{S}_{i+1}, \pi_{i+1}$
1: procedure NEWSTATE($pp, \mathcal{T}, \mathcal{S}_i, \pi_i$)
2:     if $\mathcal{V}(vk, \mathcal{S}_i, \pi_i) = 0$ then return 0
3:     while $\mathcal{H}(\pi_{i+1}) > d$ do
4:         Pick $n_{i+1}$ uniformly at random
5:         $(\mathcal{S}_{i+1}, w) \leftarrow$ UPDATESTATE($\mathcal{S}_i, \mathcal{T}, n_{i+1}$)
6:         $\pi_{i+1} \leftarrow \mathcal{P}(pk, \mathcal{S}_{i+1}, \mathcal{T}, \mathcal{S}_i, \pi_i, w)$
7:     return $(\mathcal{S}_{i+1}, \pi_{i+1})$
```

When updating the state of the system, each participating miner receives π_{i+1} and $\mathbf{t}_{i+1}$. This allows them to update their own state to $\mathcal{S}_{i+1}$ and begin mining again. We formalize the security of our DPS scheme in the sense of the definitions presented in Appendix F.

Theorem 1. The protocol as defined in Section 5.4, is complete (c.f. Definition 8) and correct (c.f. Definition 9).

In formalizing the correctness of our system, we rely on a random oracle similar to the construction in [31], but extended to arbitrary transition functions. Proofs for the above are provided in Appendix G.

6 Proof of Necessary Work

To allow proof generation to serve as a proof-of-work puzzle, we require (a) a proof π_i whose generation algorithm $\mathcal{P}$ is moderately difficult to compute and (b) a proof-of-work puzzle $\mathsf{P}_V^{\mathcal{H},d}$ that requires the miner to fully recompute $\mathcal{P}$ to test a potential solution. The second property is necessary for the puzzle to be progress-free for fairness to miners of differing size. Indeed, if generating unique proofs π_i based on randomly sampled nonces n_i is sufficiently 'hard', then using $\mathsf{P}_V^{\mathcal{H},d}$ instead of a generic puzzle (such as computing the double SHA256 digest in Bitcoin) would allow us to not only perform proof-of-work with the same theoretical guarantees, but also compute a valid proof π_i in the process.

We do not formally analyze any consensus properties, since our goal is not to design a new consensus protocol but to retain that used by Bitcoin (and similar systems) and inherit its properties. However, we would like the work done to be useful by producing proofs of each block's validity. We introduce the notion of performing proof-of-work by proving the validity system state, which we denote as *Proof of Necessary Work* (PoNW).

6.1 Definitions

We formalize this definition below, and provide the relevant security model.

Definition 6 (Verification Puzzle)**.** Given a collision-resistant pseudorandom function $\mathcal{H}$ and a proof $\pi_i \in \mathcal{Z}$ in some RSM with transition tuple (NewState, VerifyState), we define the *verification puzzle* $\mathsf{P}_V^{\mathcal{H},d} : \mathcal{S} \times \mathcal{S} \times \mathcal{Z} \to \{0,1\}$ with difficulty d as the solution to the following function:

$$\mathsf{P}_V^{\mathcal{H}}(\mathcal{S}_i, \mathcal{S}_{i+1}, \pi_i) = \mathbf{1}\begin{bmatrix} \mathsf{VerifyState}(\mathcal{S}_i, \mathcal{S}_{i+1}, \pi_i) = 1 \\ \mathcal{H}(\pi_i) < d \end{bmatrix},$$

where $\mathbf{1}[\cdot]$ is the indicator function.

By having access to a proof generating algorithm $\mathcal{P}(\mathbf{t}, \mathcal{S}_i, \mathcal{S}_{i+1}, n) \to \pi_{i+1}$ that generates unique (yet valid) π_i for each n_i, we can generate π_{i+1} for $\mathcal{S}_{i+1} = \mathsf{NewState}(\mathbf{t}, \mathcal{S}_i, \pi_i)$ using a uniformly random sample n until the puzzle condition is satisfied:

$$\mathsf{P}_V^{\mathcal{H}}(\mathcal{S}_i, \mathcal{S}_{i+1}, \mathcal{P}(\mathbf{t}, \mathcal{S}_i, \mathcal{S}_{i+1}, n)) = 1.$$

Then π_i suffices for public verification that proof-of-work has been performed. This is because our prover will always fail with constant probability (when $\mathcal{H}(\pi_i) \geq d$), so iteratively sampling new proofs (by sampling new n) until a valid one is found can be shown, under the assumption that $\mathcal{P}$ is the most efficient way to find such an n, to be a memoryless exponential process and hence *fair*. Note that, by construction, we also guarantee that π_{i+1} is a valid witness for the RSM.

6.1.1 An Initial Approach

A natural thought would be to require the generation of proofs until $\mathcal{H}(\pi) < d$, as is proposed in the previous section. In the case that the proof is unique to the state and witness input, we can ensure that by adding a nonce in the input we will always get a different hash for π. This requires that our prover satisfy *unique simulation extractability*, which can be shown to hold for our design choices.

However, this can lead to unfair outcomes. When computing π, the adversary can retain the parts of π that don't change between nonces and therefore substantially decrease proof generation time with respect to other provers - violating the scheme's amortization resistance. This means the process is not memoryless, and so the fairness of the system is compromised. We thus require a mechanism through which to prevent this decomposition attack.

6.1.2 Amortization Resistance

Just like in Nakamoto consensus, our puzzle needs to satisfy the property that solutions are equally hard to test even after testing an arbitrary number of previous solutions. In other words, a miner should not be able to *amortize* costs while testing multiple potential solutions. We define this property more formally below. We model PoNW as a function f^O with limited access to some oracle O that performs a hard computation in some encoding of a given group $\mathbb{G}$.

Definition 7 (ε-Amortization Resistance)**.** For $\ell \in \mathbb{N}$ with $\ell = \text{poly}(\lambda)$ and inputs of length λ, a function $f^O = \{f_n^O\}_{n \in S}$ with $S = S(\lambda)$ is considered ε-hard if for all adversaries $\mathcal{A}$ performing less than $(1-\varepsilon)N\ell$ queries to the oracle O, where N number of queries required for one evaluation of f_n^O, the following is negligible in λ:

$$\Pr_{n \leftarrow S^\ell}\left[\ \forall i \in [\ell], \pi_i = f_{n_i}^O(a_i)\ \middle|\ \begin{array}{c}\{n_i\}_{i=1}^{\ell} \leftarrow n \\ \{\pi_i, a_i\}_{i=1}^{\ell} \leftarrow \mathcal{A}(1^\lambda, n)\end{array}\ \right]$$

This definition captures the fact that computing multiple proofs does not come with marginal gains: indeed, provers cannot use computational advantage to batch process proofs and achieve non-negligible performance improvements. By prohibiting large miners from achieving returns-to-scale, this property is crucial in ensuring fairness. With the above objectives in mind, we now look at how to adapt our implementation to realize such a system.

6.1.3 Prover Computational Costs

Before we look at designing an amortization-resistant PoNW system, we summarize the computationally expensive components of proof generation in Quadratic Arithmetic Program (QAP) SNARK provers such as [29]. For a proof with N variables $a \in \mathbb{F}^N$ and n constraints, the prover $\mathcal{P}$ needs to compute the following (simplified) steps:

1. Update inputs and witnesses to a.
2. Compute exponentiation $a_i \to g_i^{a_i}, \forall i \in [N]$, where g_i the corresponding element from the proving key.

Although asymptotically the exponentiations are the bottleneck as the security parameter grows, for lower regimes and large predicate sizes the polynomial computations dominate, taking time $O(n \log n)$ where n the number of constraints. Since updating variable assignments is orders-of-magnitude faster than the other two steps, for each new nonce a prover would need to be forced to recompute (almost) all modular exponentiations in order to achieve amortization resistance.

We thus want to ensure that all variables in the proof are sufficiently altered by any change in the nonce or transactions being verified. This is done to make all variables in the proof change *unpredictably* with every new attempt at satisfying difficulty. Since updating variable assignments is asymptotically negligible in the other two steps and since all subsequent proof computations are functions of the variable assignments, by uniquely altering all input variables in the proof as a function of the nonce we can hope that amortization resistance is satisfied.

6.2 A Fair and Efficient Construction

We modify the DPS predicate Π to ensure that most of the proof variables change unpredictably with modifications of the nonce or state. This gives us amortization resistance in exchange for increasing the number of variables and constraints in our predicate. The performance overhead originates from the need to commit to state and 'mask' the computation, which can be expensive for large predicates.

The naive approach would be to isolate each of the different circuits in the system and show that they can be modified to change unpredictably based on some seed. The design challenge here is how to make this happen while conserving the proof's correctness guarantees. For this, we ideally want to leverage a property specific to our predicate in order to 'mask' the computations. This would allow us to treat the proving system as a 'black box'. We show how we can leverage the Pedersen hash function to transform our predicate Π to an amortization-resistant version.

We first outline the requirement for commitment to state. Given some nonce n, the prover might only change a part of the input in order to (re)check difficulty. This is an issue if the same nonce can be used with many inputs (in our case, transactions), as an adversarial prover would compute a proof and then only switch out a single transaction (or bit!), rechecking difficulty with no expensive recomputation. This means that we need to define a seed parameter $\rho = \mathsf{PRF}(n||\mathsf{state})$ that commits to state. For our DPS, we would need to commit to all transactions in the block, ensuring that a change to one transaction would lead to a different valid ρ. PRF here denotes a pseudorandom function, meaning that ρ is semantically secure. This comes with the additional cost of verifying correct computation for one instance of PRF. However, this can be large if we exploit no information about the underlying predicate, since the PRF would (in the worst case) have to commit to every variable in the original predicate.

Fortunately, for our choice of predicate the input to PRF need not be too large: indeed, in our DPS one only needs to commit to the two addresses involved and the amounts transferred for each transaction, and not to the public keys or signatures (assuming non-malleability) which can all be substantially larger. This is because the two addresses along with the values transferred fully specify the state transition, and thus changing any other parts of any transaction would violate what miners have stored for state. Even better, we can use $\rho = \mathsf{PRF}(n, rt)$ where rt the root of the new state. Since this will be computed anyways as part of our protocol, we don't actually suffer any overhead apart from having to verify the above computation. Note that this is actually *constant* in predicate size.

We can force unique changes to the Merkle path updating the account if we require n to be part of the leaf: since a change in the block (or nonce) would lead to a new n, all update paths need to be recomputed if any transaction is changed. However, we also need to enforce change to the *old* Merkle path checking account existence. This technique is thus not ideal, since these paths do not depend on the current nonce (or state) at all, meaning that around half our variables will remain the same - giving $\varepsilon \approx 1/2$.

To get around this issue, we opt for a different approach. We 'mask' the input variables to our hash function by interaction with ρ and transform the constraints of the hash function subcircuit $C_{\mathcal{H}}$ into a new circuit that retains the original Proof of Knowledge (PoK) guarantees by verifying the same underlying computation. By ensuring that the inputs all change unpredictably, we hope to achieve upper bounds for amortization resistance based on the security of our PRF.

6.3 Quantization Effects for Slow Miners

A novel concern with our proposal when implemented with a proof-of-work consensus protocol is that checking even one proof-of-work solution can be slow (on the order of tens of seconds to minutes). When this becomes a significant fraction of the average block generation time, miners are hit with a loss of efficiency because they will be forced to discard a partially-checked puzzle solution when a block is broadcast in the middle of checking. We prove the scale of this efficiency loss in a short theorem:

Theorem 2. For a miner in a proof-of-work consensus protocol that can check a single puzzle solution in time τ (with the mean block arrival time normalized to 1), the miner will discard a fraction $1 - \frac{\tau}{e^{\tau}-1}$ of their work due to newly broadcast solutions.

A proof of this theorem is provided in Appendix H. Note that as $\tau \to 0$ (fast puzzle checking time relative to block interval), the fraction of wasted work drops to 0. Similarly, as $\tau \to \infty$ the fraction of wasted work approaches 1. For $\tau = 1$ (solutions take as long to check as the mean block interval), the fraction of wasted work is $\frac{e-2}{e-1} \approx 0.42$, suggesting that we should aim to keep the time (even for slow miners) to get a solution significantly shorter than the mean block time.

7 Implementation

7.1 Choice of Proof System and Parameters

Since we'll be broadcasting each proof π_i to the network, we would like them to be quite small (ideally < 1kB). We also require that the size of π_i does not increase with i, ideally staying the same size after every state transition. With these design choices in mind, we prototype our system using libsnark [26], a C++ library implementing the PCD system in [6] using the construction from [29]. This is done using Succinct Non-Interactive Arguments of Knowledge (SNARKs) [5], non-interactive proofs of knowledge with the additional property of *succinctness*: producing constant-sized proofs that can be instantly verified. We note that improved proof sizes can be obtained by using the verifier from [19], which has the fastest performance and smallest proof size (consisting of 3 group elements) of all current implementations.

Encoding Π_S: We can equivalently consider Π_S as an arithmetic circuit C_Π, evaluating to 1 on some input B_i if and only if $B_i \in$ BLOCK-V. In the implementation of [6], C_Π is expressed in a low-level NP-complete language similar to arithmetic circuit satisfiability called Rank 1 Constraint System (R1CS) (see [4] for definitions). The circuit is encoded over elliptic curve elements through vectors in $\mathbb{F}_p$, where the number of gates increases with the size of π_i and the time required to generate it. By manually designing a circuit C_Π for our predicate, we minimize the number of gates used and provide a deployable implementation.

We note that our system need also allow for *recursive proof composition*, or the capability of new proofs to check the validity of previous proofs efficiently. Since this construction depends on SNARKs over pairs of elliptic curves that form *PCD-friendly cycles*, we use the same pair of non-supersingular curves of prime order as [6]. They provide a security level of 80 bits with a field of size $|q| \approx 2^{298}$.

7.2 Arithmetic Circuit Requirements

The vast majority of the constraints in our predicate come from verifying the existence of accounts in the tree. This requires the serial recomputation of a hash function $\mathcal{H}$, tracing the path from the given leaf to the root of the tree. To ensure that this path uniquely identifies the leaf, it is additionally required that $\mathcal{H}$ be collision resistant.

A tree depth of 32 for our implementation allows for 4.2 billion accounts. We compare this to 32 million unique used wallets on the Bitcoin blockchain after 10 years of operation. This requires $32 \cdot 4 = 128$ hash checks for each transaction. We use the circuit provided in libsnark to verify such proofs of inclusion and modification.

7.2.1 Pedersen Hashes

Since it is desirable for $\mathcal{H}$ to be efficiently represented with a low number of gates, we opt for using Pedersen hashes [17]. In verifying a Pedersen hash, we are required to compute $\prod_{i=1}^{D} g_i^{x_i}$ where $\{x_i\}_{i=1}^{D}$ is the bit representation of the input x and $\{g_i\}_{i=1}^{D}$ is a set of primitive roots for $\mathbb{Z}_q^*$. We encode each root as a field element and, based on the sign of each input x_i, perform multiplication of an intermediate field variable c by each g_i to arrive at the digest if the corresponding $x_i = 1$. We use the same underlying group $\mathbb{Z}_p^*$ as for the SNARK with $|p| = 2^{298}$, which reduces in security to the discrete-logarithm problem (DLP).

7.2.2 Schnorr Signature Scheme

We use Schnorr signatures [32] over an elliptic curve (EC), based on the hardness of DLP. This choice is motivated by our desire to minimize the size of the verifying circuit, since this has to be built inside C_Π. The Schnorr verification circuit only requires two exponentiations, a hash computation, and a comparison between scalars. The same curve from the PCD construction is used, since it satisfies all security requirements and offers a security of 80bits, just like for the PCD system.

Key Sizes: Schnorr signatures use elliptic curve elements as public keys, resulting in key sizes of 596 bits, or $298+1 = 299$ bits with point compression. Secret keys are sampled as random 298-bit strings.

7.3 Randomizing the Pedersen Hash

We now show how to randomize the computation of the Pedersen hash function. In addition to some input x of length n bits, our evaluation requires a pseudorandom seed $\rho \in \{0,1\}^n$. We propose a modification of Pedersen, which can be thought of as masking the underlying evaluation by using two sets of input variables: $\mathcal{H}(\rho)$ and $x_i \oplus \rho_i$ for $i \in [n]$, where $\mathcal{H}(\cdot)$ the evaluation of the unaltered Pedersen function and $\oplus$ the bit-wise XOR operation.

The variable $h = \mathcal{H}(\rho)$ forms the 'starting point' of the evaluation. In the beginning, the prover will precompute some generator variables v_i for the specific instance of the problem. At every bit, the new circuit would also evaluate $z_i = x_i \oplus \rho_i$. It would then proceed normally: if $z_i = 1$, multiply the intermediate variable by v_i else by 1. By carefully choosing the v_i, we can design the circuit in such a way that unpredictability based on the seed is retained by all intermediate variables except the output y, which we ensure is equal to $\mathcal{H}(x)$.

Correctness follows from the following observation: at step 0, the variable $c_0 = w \cdot \mathcal{H}(\rho) = w \cdot \prod_{i=1}^n g_i^{\rho_i}$ is initialized as the hash of the seed. For all intermediate steps $j < n$, we have that $c_j = w \cdot \left(\prod_{i=1}^j g_i^{x_i}\right) \cdot \left(\prod_{i=j+1}^n g_i^{\rho_i}\right)$. Finally, after the n-th bit has been processed the final intermediate variable c_n is equal to the Pedersen hash of the original input x multiplied by (the unpredictable) w. By multiplying with w^{-1}, we get $\mathcal{H}(x)$. This follows easily from the fact that at every step we are performing the following operation: $c_i = c_{i-1} \cdot (g_i \cdot \mathbf{1}[\rho_i = 0] + g_i^{-1} \cdot \mathbf{1}[\rho_i = 1])^{\rho_i \oplus x_i}$. It can be quickly checked that this computation ensures the previous recursive property when initialized with $c_0 = w \cdot \mathcal{H}(\rho)$. By induction, this implies that after the n-th bit, only the w parameter and the exponentiations due to the bits of x remain in the output variable i.e. $c_n = w \cdot \prod_{i=1}^n g_i^{x_i}$.

Algorithm 2 MaskedPedersen

```
Input: $x, \rho \in \{0,1\}^n, g_0 \in \mathbb{F}_p^n$
Output: $y \in \mathbb{F}_p$
 1: procedure CacheGenerators($\rho, g_0$)
 2:     Parse $\{\rho_i\}_{i=1}^n \leftarrow \rho$
 3:     Compute $h \leftarrow \mathcal{H}(\rho)$
 4:     Compute $w \leftarrow \mathcal{H}(h)$
 5:     for $i \le n$ do
 6:         $z_i = x_i \oplus \rho_i$
 7:         if $\rho_i = 0$ then
 8:             $v_i = g_i^0$
 9:         if $\rho_i = 1$ then
10:             $v_i = (g_i^0)^{-1}$
11:     return $v, h, z, w$
12: procedure MaskedHash($z, h, v, w$)
13:     Parse $\{z_i\}_{i=1}^n \leftarrow z$, $h, w \in \mathbb{F}_p$
14:     Parse $\{v_i\}_{i=1}^n \leftarrow v$
15:     Define $q = \{q_i\}_{i=1}^n$ $c = \{c_i\}_{i=0}^n$
16:     Set $c_0 = h \cdot w$
17:     for $i \le n$ do
18:         if $z_i = 1$ then
19:             $q_i = v_i$
20:         if $z_i = 0$ then
21:             $q_i = 1$
22:         $c_i = c_{i-1} \cdot q_i$
23:     $y = c_n \cdot w^{-1}$
24:     return $y$
```

7.3.1 Measuring Amortization Resistance

The hardness of the prover's computation here reduces to exponentiating elements in the proving key $f_i \in G$ by the variables $a_i \in \mathbb{F}_p$ in the circuit. At the end of the process a sum of the form $\prod_{i=1}^m f_i^{a_i} = \prod_{i=1}^m g^{a_i \cdot b_i}$ needs to be calculated for some (randomly sampled) $b_i \in \mathbb{F}_p$. We observe that in all cases where we know that the variable a_i has very small support (when, for example, it is boolean $a_i \in \{0,1\}$), the prover can always precompute once and use the same answers without performing exponentiations. However, this is not a problem since all miners would know what the precomputed answers are from the very beginning and can incorporate them in the proving key.

Amortization resistance comes from computing the exponentiations corresponding to the intermediate variables $c_i \in \mathbb{F}_p$. Since their values can be any element of $\mathbb{F}_p$, these can not be precomputed and would always require exponentiations. Indeed, an evaluation of the above scheme would require n such computations, assuming that all potential values for q_i, ρ_i, x_i are provided in the proving key. This is because the prover would need to compute all values of c_i (that they would not have seen before) since, by definition of the modified Pedersen function used here, they are next-bit-unpredictable and thus one computation cannot provide information on the next.

Constant Hardness: Since amortization resistance depends on the number of exponentiations of 'new' (or unpredictable) *field elements* in the system, the number of times we update the intermediate variable in each Pedersen evaluation needs to be constant. To this end, we 'double' the size of the unpredictable seed ρ by creating a new seed ρ^* for which the $2i$-th and $2i+1$-th bits are set to 01 or 10 depending on the value of the i-th bit of ρ. This additional transformation ensures that there is always a fixed number of 1's in the seed and thus that a set number of unpredictable field elements exist in every update of the Pedersen function's intermediate variables. This provides constant amortization resistance for each Pedersen function by ensuring that there is a constant number of 'hard' exponentiations to perform at each evaluation.

7.4 Performance

We construct the DPS based on the above specifications and investigate its running time and memory consumption. Results are displayed in Table 2. Our benchmark machine was an Amazon Web Services (AWS) c5.12xlarge instance, with 48 vCPUs and 96GiB of RAM. The security properties of the DPS are based on the guarantee of Π-compliance provided by PCD. It is apparent that setup and proving times dominate both the running time and memory consumption in the protocol. Setup takes place once by a trusted third-party and hence is less critical for day-to-day performance of the system.

The prover in this protocol would be run by the miners, or full nodes. These would be generating proof-of-work solutions repeatedly and would thus be computing proof instances for different input nonces. Thus, larger storage requirements ($\sim$ 5.42GB key sizes) could be easily met by these nodes, as could the need for more parallelism and better computing power to bring down the proving rate.

We can normalize the block time to achieve $\tau = 1/6$ in the sense of Theorem 2 for a proof including 30 transactions. This gives us that a miner will discard in expectation 8.05% of their work for an efficiency of $\sim$ 92% if all miners operated based on the above benchmarks. Since we are keeping block times constant at 10 minutes, we note that any improvements in SNARK proof generation times will correspondingly increase system efficiency. Moreover, this does not depend on the way that the proofs are generated: distributed techniques among many participants (such as [34]) would also benefit efficiency through the corresponding decrease of average proof time.

8 Future Research Challenges

Trusted setup Using SNARKs as a building block in our system introduces the issue of the one-time trusted setup. Like in other cryptocurrency systems built using SNARKs [31], an adversary with knowledge of the secret parameters would be allowed to forge proofs and hence arbitrarily print money. One mitigating approach is to distribute the ceremony over many participants through a multiparty protocol [10, 11], with the requirement that only one would have to be honest for security. This shows great potential for the construction of a system that can perform the setup procedure with sufficiently many participants that not only verify proofs, but also continually contribute randomness they (are supposed to) destroy.

Recent work has sought to construct SNARK systems which require limited or no trusted setup. SONIC [27] uses an adaptively changing structured reference string (the proving/verification keys). More recent advances such as Marlin [15] and Fractal [16] provide structured reference strings for *all* predicates trustlessly, in addition to being the first such systems that can perform efficient recursive composition. They can thus implement transparent PCDs with proof sizes small enough to be relevant to our context. We believe our design can be adapted to work with these systems, though a careful examination of the amortization resistance properties will need to be conducted.

Quantum resistance We use SNARKs based on elliptic-curve hardness assumptions which are not quantum-resistant. Recent work on practical instantiations of SNARK constructions based on lattice assumptions [18] or point-based PCPS and IOPs [3], may offer an option for quantum-resistant SNARKs.

Privacy We did not consider transaction privacy in this work, focusing instead on a simple distributed payment ledger closely matching the properties of Bitcoin. However, while we use SNARKs for their succinctness properties, the constructions here readily extend to provide zero-knowledge succinct arguments as well (zk-SNARKs). It should be straightforward to adapt to a zk-SNARK-based privacy-preserving transaction format (such as Zerocash [31]) and provide no additional overhead for chain verification. The main cost is that users must compute SNARK proofs to post transactions, imposing a heavier burden to use the system. It would also require careful thought to achieve amortization resistance when users are computing proofs of transaction validity.

More complex transactions We implemented a simpler transaction model than Bitcoin's, enabling only payments using a signature from a single public key. Our use of Schnorr signatures supports multi-stakeholder security models using standard threshold signing techniques. Bitcoin supports more complex payment scripts, enabling applications like atomic cross-chain swaps and off-chain payment channels that our system does not. Ethereum supports fully programmable smart contracts to control payment. In principle, a programmable state machine like Ethereum's can be supported on our architecture using "universal" SNARK techniques such as TinyRAM [4]. Recent work on achieving privacy in smart contract platforms using SNARKs [9, 13] can potentially be adapted to our setting to enable a more powerful

# Tx	# Constraints	Generator $\mathcal{G}$		Prover $\mathcal{P}$		Verifier $\mathcal{V}$		Size		
		Avg. (s)	σ (%)	Avg. (s)	σ (%)	Avg. (s)	σ (%)	*pk* (GB)	*vk* (kB)	π (B)
2	1100849	24.29	1.75	12.06	1.31	0.0387	0.21	0.31	1.30	373
10	4611209	75.90	2.01	43.92	0.70	0.0387	0.18	1.16		
20	8999159	177.45	1.08	81.10	1.03	0.0386	0.32	2.19		
30	13387109	218.22	0.68	99.42	1.59	0.0387	0.35	3.35		
40	17775059	300.97	1.65	156.83	1.46	0.0387	0.28	4.30		
50	22163009	347.56	0.49	196.91	2.15	0.0387	0.49	5.42		

Table 2: Prototype Times and Key Sizes for Predicates verifying different numbers of transactions: Average running times for setup $\mathcal{G}$, prover $\mathcal{P}$ and verifier $\mathcal{V}$ over 10 iterations are shown alongside proving/verification key and proof sizes.

programming model with efficient verification.

More complex consensus rules We adapted Bitcoin's relatively simple linear longest-chain rule. Many more complex DAG-based proposals exist which improve on Bitcoin's consensus protocols. All of these involve a different formul for computing the quality of a specific block in the chain. Our approach does not preclude the use of more complex predicates for. By setting the quality accordingly and ensuring that π_i also proves that the required stage in the protocol was executed correctly, we can ensure that *any* consensus protocol can be used in this way.

Parallelism Our current construction uses relatively little parallelism in proof-construction. Recent advances [34] enable constructing larger proofs using many parallel workers, a model that adapts readily to cryptocurrencies which typically feature large mining pools. Exploring this is an important avenue for future work, especially given the order-of-magnitude improvements in the size of computable predicates.

Hardware acceleration Just like in the context of Bitcoin mining, there is potential to achieve substantial improvements in the computation of such proofs by the usage of specialized hardware. The design of Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs) that are especially created to compute the proof corresponding to a given predicate would lead to order-of-magnitude improvements in proving time and thus substantially minimize quantization effects. Such specialized hardware, however, would be expensive to construct given the purported memory-hardness of most current SNARK designs. Moreover, such hardware would also provide a large barrier to entry for most miners due to the large upfront existing costs for its design. This would have the potential to impact the fairness of the distributed system.

9 Concluding Remarks

We present the minimal transaction semantics for a distributed payment system, alongside correctness and completeness definitions. Although our framework can be used modularly with any system transition function desired, it is fundamentally an account-based model of keeping state that uses transactions to update state (or account contents) over discrete time intervals.

We could alternatively have designed a 'UTXO'-based model, but we felt that storing state in individual accounts lends itself more easily to generalization over arbitrary transition functions simply by validating the individual transactions that update state. Moreover, even though coin-based models have been historically favored in the design of privacy-focused protocols [31, 33], the expressive powers of our system are only limited by the need to verify membership of state in some NP language. Our system could be adapted to predicates can be designed for any substantially complex task, such as smart contracts.

However, the design of such a system would require more efficient proof generation. While a basic money-remittance system is feasible with today's proof systems, further work is needed to deploy substantially more complex predicates. The superlinear relationship between predicate size and proof generation time means generating such proofs for more complex predicates quickly becomes prohibitive. Recent advances in lowering proof generation times in SNARKs are very promising in bridging this gap, as is recent work in distributing proof computations over multiple participants.

Finally, we introduce Proof of Necessary Work, which performs computation necessary for system verification as part of the proof-of-work computation. We ensure fairness in our system by forcing the prover to unpredictably alter proof variables with each nonce, making it secure against 'churning' attacks. This process leverages the homomorphic properties of our hash function, and we believe that more general techniques to do this would be of great interest.

References

[1] O(1) labs. https://cdn.codaprotocol.com/v2/static/coda-whitepaper-05-10-2018-0.pdf. Accessed: 2019-04-15.

[2] Adam Back et al. Hashcash-a denial of service counter-measure. 2002.

[3] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018.

[4] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90–108. Springer, 2013.

[5] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive arguments for a von neumann architecture. *IACR Cryptology ePrint Archive*, 2013:879, 2013.

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

[7] 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.

[8] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, May 2015.

[9] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Report 2018/962, 2018. https://eprint.iacr.org/2018/962.

[10] Sean Bowe, Ariel Gabizon, and Matthew D Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *International Conference on Financial Cryptography and Data Security*, pages 64–77. Springer, 2018.

[11] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptology ePrint Archive*, 2017:1050, 2017.

[12] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. Accessed: 2016-08-22.

[13] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. Cryptology ePrint Archive, Report 2019/191, 2019. https://eprint.iacr.org/2019/191.

[14] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Super-light clients for cryptocurrencies. Cryptology ePrint Archive, Report 2019/226, 2019. https://eprint.iacr.org/2019/226.

[15] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. Cryptology ePrint Archive, Report 2019/1047, 2019. https://eprint.iacr.org/2019/1047.

[16] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. Cryptology ePrint Archive, Report 2019/1076, 2019. https://eprint.iacr.org/2019/1076.

[17] Ivan B Damgård, Torben P Pedersen, and Birgit Pfitzmann. On the existence of statistically hiding bit commitment schemes and fail-stop signatures. In *Annual International Cryptology Conference*, pages 250–265. Springer, 1993.

[18] Rosario Gennaro, Michele Minelli, Anca Nitulescu, and Michele Orrù. Lattice-based zk-snarks from square span programs. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 556–573. ACM, 2018.

[19] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 305–326. Springer, 2016.

[20] Marcella Hastings, Nadia Heninger, and Eric Wustrow. The proof is in the pudding: Proofs of work for solving discrete logarithms. Cryptology ePrint Archive, Report 2018/939, 2018. https://eprint.iacr.org/2018/939.

[21] Kostis Karantias, Aggelos Kiayias, Nikos Leonardos, and Dionysis Zindros. Compact storage of superblocks for nipopow applications. Cryptology ePrint Archive, Report 2019/1444, 2019. https://eprint.iacr.org/2019/1444.

[22] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka. Proofs of proofs of work with sublinear complexity. In *International Conference on Financial Cryptography and Data Security*, pages 61–78. Springer, 2016.

[23] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. *IACR Cryptology ePrint Archive*, 2017:963, 2017.

[24] Sunny King. Primecoin: Cryptocurrency with prime number proof-of-work. *July 7th*, 1(6), 2013.

[25] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[26] SCIPR Lab. libsnark: a c++ library for zksnark proofs.

[27] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Report 2019/099, 2019. https://eprint.iacr.org/2019/099.

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[29] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. Cryptology ePrint Archive, Report 2013/279, 2013. https://eprint.iacr.org/2013/279.

[30] Andrew Poelstra. Mimblewimble, 2016.

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

[32] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[33] Nicolas Van Saberhagen. Cryptonote v 2.0, 2013.

[34] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 675–692, 2018.

Appendices

A Transaction Semantics

Setup: This algorithm is run once by a trusted third party to initialize the parameters of the system. It takes as input security parameters λ, μ and outputs public parameters $pp = (pk, vk, pp_{sig})$ where pk, vk are the proving key and the verification key of the PCD system respectively and pp_{sig} are the public parameters for the signature scheme.

Algorithm 3 Setup

```
Input: $1^\lambda, 1^\mu$
Output: $pp$
1: procedure SETUP($1^\lambda, 1^\mu$)
2:     $(pk, vk) \leftarrow \mathcal{G}(1^\lambda, \Pi_{\mathcal{S}})$
3:     $pp_{sig} \leftarrow$ SC-SETUP($1^\mu$)
4:     return $(pk, vk, pp_{sig})$
```

Creating Transactions: Algorithm 4 creates a send transaction, while the Algorithm 5 a coinbase transaction. Note that these algorithms do not provide guarantees about the validity of the transaction created.

Algorithm 4 NewTransaction

```
Input: $pp, \mathrm{addr}_s, \mathrm{addr}_r, v, f, \mathrm{PK}, \mathrm{SK}, \mathrm{PK}_R$
Output: $t$
1: procedure NEWTX($pp, \mathrm{addr}_{\{s,r\}}, v, f, \mathrm{PK}, \mathrm{SK}, \mathrm{PK}_R$)
2:     $\sigma \leftarrow$ SIGN($\mathrm{SK}, \mathrm{addr}_s \| \mathrm{addr}_r \| v \| f \| \mathrm{addr}_s.n$)
3:     return $(\mathrm{addr}_s, \mathrm{addr}_r, v, f, \mathrm{PK}, \sigma, \mathrm{PK}_R)$
```

Algorithm 5 NewCoinbase

```
Input: $pp, \mathrm{addr}_r, v, \mathrm{PK}, \mathrm{SK}$
Output: $tx$
1: procedure COINBASETX($pp, \mathrm{addr}_r, v, \mathrm{PK}, \mathrm{SK}$)
2:     $\sigma \leftarrow$ SIGN($SK, \mathrm{addr}_r \| v \| \mathrm{addr}_r.n$)
3:     return $(\mathrm{addr}_r, v, \mathrm{PK}, \sigma)$
```

CreateAddress: $\mathrm{CreateAddress}(pp_{sig}) \rightarrow (\mathrm{PK}, \mathrm{SK})$

GetQuality: $\mathrm{GetQuality}(\mathcal{S}_i) \rightarrow q_i$

GetBalance: $\mathrm{GetBalance}(\mathrm{PK}, \mathcal{S}_i) \rightarrow \mathbf{a}.v$ where $\mathbf{a}$ is the leaf in $\mathcal{S}_i$ with $\mathbf{a}.\mathrm{PK} = \mathrm{PK}$

B Proof Carrying Data (PCD)

Define an online distributed system with state $\mathcal{S}$ and initial state $\mathcal{S}_0$. The system transitions from a state $\mathcal{S}_i$ to $\mathcal{S}_{i+1}$ through a function $\mathsf{UpdateState}$. The security goal is to ensure that all the states that the system transitions to are compliant with a predicate Π representing a valid state update. Proof Carrying Data ensures this by attaching short and easy to verify proofs of Π-compliance to each state the system transitions to.

Specifically, we assume that we have a key generator $\mathcal{G}$ that sets up a proving and verification key. Anyone can use a prover $\mathcal{P}$, which is given as input the proving key, a prior state $\mathcal{S}_i$ with a proof π_i and a new state $\mathcal{S}_{i+1}$, to generate a proof π_{i+1} attesting that $\mathcal{S}_{i+1}$ is Π-compliant. One can then use a verifier $\mathcal{V}$, which is given as input the verification key, a state $\mathcal{S}_i$ and a proof π_i to verify that $\mathcal{S}_i$ is Π-compliant.

Transcripts Given $n_{\mathcal{S}}, n_w$ and field $\mathbb{F}$, an $\mathbb{F}$ arithmetic-transcript is a triple $Tr = (k, w, \mathcal{S})$, where $k > 0$, w maps each $0 < i \leq k$ to a value in $\mathbb{F}^{n_w}$ and $\mathcal{S}$ maps each $0 < i \leq k+1$ to a value $\mathbb{F}^{n_{\mathcal{S}}}$. The output of Tr, denoted $out(Tr)$ equals $\mathcal{S}_{k+1}$.

Intuitively, w_i represents the auxiliary data used by the node that performs the i-th state transition. $\mathcal{S}_i$ represents the output state of the i-th state transition. Typically a node that performs the i-th state transition uses the auxiliary data w_i and an prior state $\mathcal{S}_i$ to compute the new state $\mathcal{S}_{i+1} \leftarrow \mathsf{UpdateState}(\mathcal{S}_i, w_i)$.

Compliance: Given field $\mathbb{F}$ and $n_{\mathcal{S}}, n_w \in \mathbb{N}$, an $\mathbb{F}$-arithmetic compliance predicate Π (for state size $n_{\mathcal{S}}$ and auxiliary input size n_w) is an $\mathbb{F}$-arithmetic circuit with domain $\mathbb{F}^{n_{\mathcal{S}}} \times \mathbb{F}^{n_w} \times \mathbb{F}^{n_{\mathcal{S}}} \times \mathbb{F}$. The compliance predicate Π specifies whether a given transcript Tr is compliant or not. Consider any transcript Tr with state size $n_{\mathcal{S}}$ and auxiliary input size n_w. We say that $Tr = (k, w, \mathcal{S})$ is Π-compliant, denoted $\Pi(Tr) = 0$, if, for every $0 < i \leq k$ it holds that:

$$\Pi\big(\mathcal{S}_i, w_i, \mathcal{S}_{i+1}, b_{base}\big) = 1,$$

where $b_{base} \in \{0,1\}$ is the base case flag (i.e., equals 1 if and only if $i = 1$). Furthermore, we say that a state s is Π-compliant if there is Tr such that $\Pi(Tr) = 0$ and $out(Tr) = s$.

We are now ready to describe the syntax, semantics, and security of a proof-carrying data system:

- $\mathcal{G}$ is a procedure that takes as input a compliance predicate Π and some security parameter λ and outputs a proving key pk and a verification key vk. We assume without loss of generality that pk contains (a description of the predicate Π).
- $\mathcal{P}$ takes as input a proving key pk, auxiliary input w, a state $\mathcal{S}_{i+1}$ and a state $\mathcal{S}_i$ with proof π_i and outputs a proof π_{i+1} for the statement that $\mathcal{S}_{i+1}$ is Π-compliant.
- $\mathcal{V}$ takes as input a verification key vk, a state $\mathcal{S}_i$ and a proof π_i and outputs $b = 1$ if $\mathcal{S}_i$ is Π-compliant.

Completeness:. This is based on an interaction between a computationally unbounded adversary $\mathcal{A}$ and the prover $\mathcal{P}$, who are both given the predicate Π and the common string (pk, vk). We build transcripts through the following:

$$\text{ProofGen}(\Pi, pk, \mathcal{A}, \mathcal{P}) \to (\mathcal{S}_i, \pi_i, T)$$

For a full definition, see [7]. The adversary will provide witnesses $w \in \mathbb{F}^{n_w}$ and output $\mathcal{S}_i \in \mathbb{F}^{n_{\mathcal{S}}}$ to the transcript for each transition. $\mathcal{P}$ then attaches a proof of compliance π_i to every output $\mathcal{S}_i$ in the transcript. Using this, we have that for every predicate Π, the following is negligible in μ:

$$\Pr\left[\begin{array}{c|c} \Pi(T) = 1 & (pk, vk) \leftarrow \mathcal{G}(1^\mu) \\ \mathcal{V}(vk, \pi, \mathcal{S}) \neq 1 & (\mathcal{S}, \pi, T) \leftarrow \text{ProofGen}(\Pi, pk, \mathcal{A}, \mathcal{P}) \end{array}\right].$$

Security Properties: If the verifier accepts a proof π for a state S, the prover "knows" a compliant transcript Tr with output S. Namely, for any constant $c > 0$ and every polynomial size adversary $\mathcal{A}$ there is a polynomial-size witness extractor E s.t. for every large enough security parameter λ, for every $\mathbb{F}$-arithmetic compliance predicate Π of size λ^c, the following probability is negligible in λ:

$$\Pr\left[\begin{array}{c|c} \mathcal{V}(vk, S, \pi) = 1 & (pk, vk) \leftarrow \mathcal{G}(1^\lambda, \Pi) \\ out(Tr) \neq S \vee \Pi(Tr) \neq 1 & (S, \pi) \leftarrow \mathcal{A}(pk, vk) \\ & Tr \leftarrow E(pk, vk) \end{array}\right]$$

B.0.1 Constant-Depth Compliance Predicates

It should be noted that PCDs face trade-offs based on the depth of any given transcript Tr. In this context, the depth is equivalent to the length of the path from $\mathcal{S}_0$ to the current state. Unfortunately, in order for the security properties of the PCD system to hold for a meaningful number of transitions, we need to limit ourselves to *constant-depth* compliance predicates. However, in this case we are interested in verifying *all* transitions in a path from the genesis block, which would require a polynomial number of nodes in the transcript.

We solve this problem using the construction of Bitansky et al. [7], wherein a polynomial depth predicate expressing an NP language $\mathcal{L}$ can be efficiently transformed into a constant-depth predicate $Tree(\mathcal{L})$. The transformation requires a CRT hash function $\mathcal{H}$ and stores all proofs (of $\mathcal{L}$ membership) in a tree to 'keep track' of their position by using only a constant number of operations.

We achieve this in our construction by keeping the state commitments B_i of each transition in an (ordered) Merkle tree, with all leaves initialized to zero. By requiring a proof of update of each state to the next at every transition, we ensure the same security guarantees as for $\mathcal{L}$ (i.e. that $x \in \mathcal{L} \iff x \in Tree(\mathcal{L})$). In practice, this transformation only requires the additional verification of a single Merkle update path and thus does not meaningfully impact performance.

C Digital Signature Schemes

We define a digital signature scheme as follows:

- SC-SETUP$(1^\mu) \to pp_{sig}$: Setup generating public parameters based on security parameter μ.

- $\text{CA}(pp_{sig}) \rightarrow (\text{PK}, \text{SK})$ takes as input pp_{sig} and outputs a public key PK and a private key SK.
- $\text{SIGN}(SK, m) \rightarrow \sigma$ takes as input a secret key SK and some message m, outputting a signature σ.
- $\text{VS}(\text{PK}, m, \sigma)$ is a procedure that takes as input a public key PK, some message m and some signature σ and outputs a bit denoting Yes/No.

The above signing scheme should satisfy the following security properties:

Completeness:

$$\text{VS}(\text{PK}, m, \text{SIGN}(\text{SK}, m)) = 1$$

Security: For all non-uniform probabilistic polynomial time adversaries $\mathcal{A}$:

$$\Pr\left[\begin{array}{c|c} \sigma \notin Q & pp_{sig} \leftarrow \text{SC-SETUP}(1^{\mu}) \\ \text{VS}(\text{PK}, m, \sigma) = 1 & (\text{PK}, \text{SK}) \leftarrow \text{CA}(pp_{sig}) \\ & (m, \sigma) \leftarrow \mathcal{A}^{\text{SIGN}(\text{SK}, \cdot)}(\text{PK}, 1^{\mu}) \end{array}\right]$$

is negligible in μ. Here $\mathcal{A}^{\text{SIGN}(\text{SK},\cdot)}$ denotes that $\mathcal{A}$ has access to the oracle, $\text{SIGN}(\text{SK}, \cdot)$ Q denotes the set of queries on SIGN made by $\mathcal{A}$, which knows the public key PK and the security parameter μ. Note that we require that any adversary cannot directly query the string, m, on SIGN.

C.1 Schnorr Implementation

Consider inputs $s, e \in \mathbb{F}_q$ and $pk = (pk_x, pk_y) \in E(\mathbb{F}_q) \subset \mathbb{F}_q \times \mathbb{F}_q$ to a circuit C_{Sig}. Verification of a signature $(s, e) \in \mathbb{F}_q \times \mathbb{F}_q$ for some m requires that $e = \mathcal{H}(G^s \cdot Q^e \| m)$, where Q the given public key and G the generator of the underlying group.

We built an exponentiation subcircuit C_{exp} to verify the computation of G^s and pk^e, with G hardcoded at setup. $C_{\mathsf{exp}}(S, x)$ parses two field elements as an elliptic curve point S, it also parses a scalar by which to multiply the curve point. We use the "double-and-add" method, consisting of serial multiplications and squarings of S to verify that exponentiation was correctly performed. We then compute the hash of the message concatenated to the bit-representation of $G^s \cdot pk^e$. This is done using an instantiation of a Pedersen hash circuit. An additional final constraint ensures that $d = e$ for the signature to verify.

D DPS Transition Functions

We specify VerifyTx in Algorithm 6 and UpdateState in Algorithm 7. Together they define the state transitions in the model set out by our DPS and are used to construct the PCD. If the above accepts, then the participant accepts $\mathcal{S}_{i+1}$ (corresponding to commitment B_{i+1}) as the new state of the system with associated compliance proof π_{i+1}.

Algorithm 6 VerifyTx

```
Input: $(t, T^i)$
Output: bit $b$
procedure VERIFYTX$(t, T^i)$
    Parse $t$
    if $t = (\text{addr}_s, \text{addr}_r, v, f, \text{PK}, \sigma, \text{PK}_R)$ then
        if $v < 0$ then
            return 0
        Let $\mathbf{a}_s$ the account in $T^i$ s.t. $\mathbf{a}_s.\text{addr} = \text{addr}_s$
        if $\mathbf{a}_s.\text{PK} \neq \text{PK}$ then
            return 0
        if $\mathbf{a}_r.\text{PK} \neq$ nil and $\mathbf{a}_r.\text{PK} \neq \text{PK}_R$ then
            return 0
        $m \leftarrow \text{addr}_s \| \text{addr}_r \| v \| f \| \mathbf{a}_r.n$
        if $\text{VS}(\text{PK}, m, \sigma) = 0$ then
            return 0
        // Check for sufficient balance
        if $\mathbf{a}_s.bal < v + f$ then
            return 0

    else if $tx = (\text{addr}, v, \text{PK}, \sigma)$ then
        if $\text{VS}(\text{PK}, \text{addr} \| v \| \text{addr}.n, \sigma) = 0$ then
            return 0
        Let $\mathbf{a}$ the account in $T^i$ s.t. $\mathbf{a}.\text{addr} = \text{addr}$
        if $\mathbf{a}.\text{PK} \neq$ nil and $\mathbf{a}.\text{PK} \neq \text{PK}$ then
            return 0
    else
        return 0
    return 1
```

E Predicate Specification

We first define the data structures used in our constructions, and then look at the NP statement that defines what a block needs to be valid.

E.1 Proof of Account Modification

Here we define a proof of modification $\pi^{mod}(\text{acc}_A, \text{acc}_B)$ of some account acc_A to acc_B in Account tree T. We also provide definitions for each of these proofs and their corresponding properties.

Paths of Inclusion: We define a path of a certain leaf to the root, along with the notion of a 'full' path that also includes the other child node. These are important in constructing proofs of inclusion of nodes in a given tree. The two objects are defined below:

1. $\mathsf{path}(\text{acc}, q) = (\text{acc}, v_0, \{v_k\}_{k=1}^{q})$ where $q \in [h-1]$
2. $\mathsf{fullpath}(\text{acc}) = (\text{acc}, v_0, \{c_k, v_k, i_k\}_{k=1}^{h-1})$

Definitions of the parameters above are given here:

Algorithm 7 UpdateState

```
Input: $(\mathcal{S}_i, \mathbf{t}, n)$
Output: $(\mathcal{S}_{i+1})$
procedure UpdateState($\mathcal{S}_i, \mathbf{t}, n$)
    Parse $\mathcal{S}_i \leftarrow (T^i, i, q_i, n_i)$, return 0 if this fails
    $N \leftarrow |\mathcal{T}|$
    $T_0^i \leftarrow T^i$
    $v_{Tfee} \leftarrow 0$
    for $j \leftarrow 1, \ldots, N-1$ do
        $t_j \leftarrow (\mathrm{addr}_s, \mathrm{addr}_r, v, f, \mathrm{PK}, \sigma, \mathrm{PK}_R)$
        if VerifyTx($t_j, T_{j-1}^i$) $= 0$ then
            return 0
        $T_j^i \leftarrow T_{j-1}^i$ // Initialize $T_j^i$
        Define $\mathbf{a}_s$ as leaf of $T_{j-1}^i$ s.t. $\mathbf{a}_s.\mathrm{addr} = \mathrm{addr}_s$
        Update $T_j^i$:
          Set $\mathbf{a}_s.bal \leftarrow \mathbf{a}_s.bal - tx_j.v - tx_j.f$
          Set $\mathbf{a}_s.n \leftarrow n$
        Define $\mathbf{a}_r$ as leaf of $T_{j-1}^i$ s.t. $\mathbf{a}_r.\mathrm{addr} = \mathrm{addr}_r$
        Update $T_j^i$:
          Set $\mathbf{a}_r.bal \leftarrow \mathbf{a}_r.bal + tx_j.v$
          Set $\mathbf{a}_r.n \leftarrow n$
          If $\mathbf{a}_r.\mathrm{PK} = \mathrm{nil}$, set $\mathbf{a}_r.\mathrm{PK} \leftarrow \mathrm{PK}_R$
        $v_{Tfee} = v_{Tfee} + tx_j.f$
    $t_N \leftarrow (\mathrm{addr}, v, \mathrm{PK}, \sigma)$
    if VerifyTx($t_N, T_{N-1}^i$) $= 0$ then
        return 0
    if $v \neq v_{mint} + v_{Tfee}$ then
        return 0
    $T_N^i \leftarrow T_{N-1}^i$
    Define $\mathbf{a}_m$ as leaf of $T_{N-1}^i$ s.t. $\mathbf{a}_m.\mathrm{addr} = \mathrm{addr}_m$
    Update $T_N^i$:
      Set $\mathbf{a}_m.bal \leftarrow \mathbf{a}_m.bal + v$
      Set $\mathbf{a}_m.n \leftarrow n$
      If $\mathbf{a}_m.\mathrm{PK} = \mathrm{nil}$, set $\mathbf{a}_m.\mathrm{PK} \leftarrow \mathrm{PK}$
    $q_{i+1} \leftarrow q_i + 1$
    $\mathcal{S}_{i+1} = (T_N^i, i+1, q_{i+1}, n)$
    return $\mathcal{S}_{i+1}$
```

Algorithm 8 VerifyState

```
Input: $pp, B_i, \pi_i, B_{i+1}, \pi_{i+1}$
Output: $\{0,1\}$
procedure VerifyState($pp, B_i, \pi_i, B_{i+1}, \pi_{i+1}$)
    if $\mathcal{V}(vk, B_{\{i,i+1\}}, \pi_{\{i,i+1\}}) = 0$ then
        return 0
    if $\mathcal{H}(\pi_{i+1}) > d$ then
        return 0
    return 1
```

1. acc is the account in question and $v_0 = \mathcal{H}(\mathrm{acc})$
2. $q \in [h-1]$ is the length of the desired path
3. v_k is the k-th node in the path from the leaf acc to the root. By construction it holds that $\mathcal{H}(v_{k-1}, c_k) = v_k$ if $i_k = 0$ and $\mathcal{H}(c_k, v_{k-1}) = v_k$ otherwise.

Leaf Order: By construction the leaves of T follow a given canonical ordering. We use the unique path from the root to a given leaf to define this ordering. More specifically, we consider the *address* $\mathrm{addr} \in \{0,1\}^h$ of some account acc to denote the i-th leaf where $i = 2^{acc}$. It is immediate that this is a unique total ordering of all accounts in T.

Proof of Modification: We now define a proof of modification of a certain leaf acc_A to acc_B:

$$\pi^{mod}(\mathrm{acc}_A, \mathrm{acc}_B) = (\mathsf{fullpath}(\mathrm{acc}_A), \mathsf{path}(\mathrm{acc}_B, h-1))$$

We say that a proof of modification is *consistent* if it satisfies Algorithm 9.
We also define the following two values related to the inclusion proof object $\pi^{mod} = \pi^{mod}(\mathrm{acc}_A, \mathrm{acc}_B)$:
$\pi^{mod} = (\mathsf{fullpath}(\mathrm{acc}_A), \mathsf{path}(\mathrm{acc}_B, h-1))$

- $\mathsf{firstroot}(\pi^{mod}) = \mathsf{root}(\mathsf{fullpath}(\mathrm{acc}_A))$
- $\mathsf{lastroot}(\pi^{mod}) = \mathsf{root}(\mathsf{path}(\mathrm{acc}_B, h-1))$

Inclusion Sequence: We are also interested in lists of inclusion proofs, of which we want *valid* sequences of proofs, which are defined for the i-th state based on the tuple below:

$$\mathsf{Seq_j} = \left(N, rt_{j-1}, \{\pi^{mod,i}_{debit}, \pi^{mod,i}_{credit}\}_{i=1}^{N}, \pi^{mod}_{CB}, rt_j\right)$$

where we define:

- N the total number of transactions to be processed (excluding the coinbase)
- rt_{j-1} and rt_j the root of T before and after all updates have been processed respectively
- $\mathrm{acc}^{1,i}_{debit}$ and $\mathrm{acc}^{2,i}_{debit}$ the debit account before and after the i-th transaction

Algorithm 9 Modification Consistency

```
Input: $\pi^{mod}(\mathsf{acc}_A, \mathsf{acc}_B) = (\mathsf{fullpath}(\mathsf{acc}_A), \mathsf{path}(\mathsf{acc}_B, j))$
Output: $b \in \{0,1\}$
procedure CONSISTENCY($\pi^{mod}(\mathsf{acc}_A, \mathsf{acc}_B)$)
    $\mathsf{fullpath}(\mathsf{acc}_A) \leftarrow (\mathsf{acc}_A, v_0^A, \{c_k^A, v_k^A, i_k^A\}_{k=1}^{h-1})$
    $\mathsf{path}(\mathsf{acc}_B, j) \leftarrow (\mathsf{acc}_B, v_0^B, \{v_k^B\}_{k=1}^{j})$
    Check the following, return 0 if any fail:
    $j = h-1$
    $\mathcal{H}(\mathsf{acc}_A) = v_0^A$
    $\mathcal{H}(\mathsf{acc}_B) = v_0^B$
    for $k \in [h-1]$ do
        if $i_k = 0$ then
            if $\mathcal{H}(v_{k-1}^A, c_k^A) \neq v_k^A$ then
                return 0
            if $\mathcal{H}(v_{k-1}^B, c_k^B) \neq v_k^B$ then
                return 0
        else
            if $\mathcal{H}(c_k^A, v_{k-1}^A) \neq v_k^A$ then
                return 0
            if $\mathcal{H}(c_k^B, v_{k-1}^B) \neq v_k^B$ then
                return 0
    return 1
```

- $\mathrm{acc}_{credit}^{1,i}$ and $\mathrm{acc}_{credit}^{2,i}$ the credit account before and after the i-th transaction
- acc_{CB}^{1} and acc_{CB}^{2} coinbase account before and after modification
- $\pi_{user}^{mod,i} = (\mathrm{acc}_{user}^{1,i}, \mathrm{acc}_{user}^{2,i})$ the modification proofs for each transaction

We consider a *valid proof sequence* to be a tuple that satisfies the following algorithm.

Algorithm 10 Valid Proof Sequence

```
Input: $\mathsf{Seq_j} = \left(N, rt_{j-1}, \{\pi_{debit}^{mod,i}, \pi_{credit}^{mod,i}\}_{i=1}^{N}, \pi_{CB}^{mod}, rt_j\right)$
Output: $b$
procedure VALSEQ($\mathsf{Seq_j}$)
    if $rt_{j-1} \neq \mathsf{firstroot}(\pi_{debit}^{mod,1})$ then
        return 0
    for $i$ from 1 to $[N]$ do:
        if $\mathsf{lastroot}(\pi_{debit}^{mod,i}) \neq \mathsf{firstroot}(\pi_{credit}^{mod,i})$ then
            return 0
        if $i < N$ then
            if $\mathsf{lastroot}(\pi_{credit}^{mod,i}) \neq \mathsf{firstroot}(\pi_{debit}^{mod,i+1})$ then
                return 0
    if $\mathsf{lastroot}(\pi_{credit}^{mod,N}) \neq \mathsf{firstroot}(\pi_{CB}^{mod})$ then
        return 0
    if $\mathsf{lastroot}(\pi_{CB}^{mod}) \neq rt_j$ then
        return 0
    return 1
```

E.2 Proof of State Validity

We construct the following NP statement, which checks for the validity of a certain block. Instances are tuples $B_i = (rt^i, i, q_i, n_i)$ where rt^i the root of tree T^i corresponding to $\mathcal{S}_i$ with nonce n_i, while witnesses w are defined by:

$$(N, \{\pi_{debit}^{mod,j}, \pi_{credit}^{mod,j}, s^j, v_s^j, v_m^j\}_{j=1}^{N}, \pi_{CB}^{mod}, s_{CB}, \pi_{i-1}, B_{i-1}, n_i),$$

where we define:

1. N the number of transactions in the block
2. s^j signature for tx_j, s_{CB} signature for coinbase
3. v_s^j, v_m^j amounts sent to credit account and miner respectively in tx_j
4. $\pi_{debit}^{mod,i} = \pi^{mod}(\mathrm{acc}_{debit}^{1,j}, \mathrm{acc}_{debit}^{2,j})$ proof of modification of debit account
5. $\pi_{credit}^{mod,i} = \pi^{mod}(\mathrm{acc}_{credit}^{1,j}, \mathrm{acc}_{credit}^{2,j})$ proof of modification of credit account
6. $\pi_{CB}^{mod} = \pi^{mod}(\mathrm{acc}_{CB}^{1}, \mathrm{acc}_{CB}^{2})$ proof of modification of coinbase account
7. $B_{i-1} = (rt^{i-1}, i-1, q_{i-1}, n_{i-1})$ previous state commitment and π_{i-1} previous proof
8. Current n_i proof-of-work nonce

We define the language BLOCK-V with state commitment objects $B_i = (rt^i, i, q_i, n_i)$ as candidate elements:

$$\text{BLOCK-V} = \{B_i | \exists w \text{ s.t. VALIDBLOCK}(B_i, w) = 1\}$$

where VALIDBLOCK is defined below:

VALIDBLOCK(B_i, w)

1. Check that $\mathcal{H}(\pi_{i-1}) \leq d$
2. Define $v_{fees} = 0$
3. For $j \in [N]$, return 0 if any of the following fail:
   (a) $\mathrm{acc}_{debit}^{j,1} \leftarrow (\mathrm{addr}_{debit}^{j,1}, \mathrm{PK}_{debit}^{j,1}, v_{debit}^{j,1}, n_{debit}^{j,1})$
   (b) $\mathrm{acc}_{debit}^{j,2} \leftarrow (\mathrm{addr}_{debit}^{j,2}, \mathrm{PK}_{debit}^{j,2}, v_{debit}^{j,2}, n_{debit}^{j,2})$
   (c) $\mathrm{acc}_{credit}^{j,1} \leftarrow (\mathrm{addr}_{credit}^{j,1}, \mathrm{PK}_{credit}^{j,1}, v_{credit}^{j,1}, n_{credit}^{j,1})$
   (d) $\mathrm{acc}_{credit}^{j,2} \leftarrow (\mathrm{addr}_{credit}^{j,2}, \mathrm{PK}_{credit}^{j,2}, v_{credit}^{j,2}, n_{credit}^{j,2})$
   (e) $\textsc{Consistent}(\pi_{debit}^{mod,j}) = 1$
   (f) $\textsc{Consistent}(\pi_{credit}^{mod,j}) = 1$
   (g) $\mathrm{VS}(\mathrm{PK}_{debit}^{j,1}, \mathrm{addr}_{debit}^{j,1} \| \mathrm{addr}_{credit}^{j,1} \| v_m^j \| v_s^j \| n_{debit}^{j,1}, s^j) = 1$
   (h) $n_i = n_{debit}^{j,2} = n_{credit}^{j,2}$
   (i) $\mathrm{PK}_{debit}^{j,1} = \mathrm{PK}_{debit}^{j,2}$
   (j) $\mathrm{PK}_{credit}^{j,1} = \mathrm{PK}_{credit}^{j,2}$ or $\mathrm{PK}_{credit}^{j,2} = \perp$
   (k) $v_{debit}^{j,1} = v_m^j + v_s^j + v_{debit}^{j,2}$

(l) $v^{j,2}_{credit} = v^j_s + v^{j,1}_{credit}$

(m) Set $v_{fees} = v_{fees} + v^j_m$

4. $\text{acc}^1_{CB} \leftarrow (\text{addr}^1_{CB}, \text{PK}^1_{CB}, v^1_{CB}, n^1_{CB})$
5. $\text{acc}^2_{CB} \leftarrow (\text{addr}^2_{CB}, \text{PK}^2_{CB}, v^2_{CB}, n^2_{CB})$
6. Verify $\text{PK}^1_{CB} = \text{PK}^2_{CB}$
7. Verify $\text{addr}^1_{CB} = \text{addr}^2_{CB}$
8. Verify $v^2_{CB} = v^1_{CB} + v_0 + v_{fees}$
9. Verify $\textsc{Consistent}(\pi^{mod}_{CB}) = 1$
10. Verify that $n^2_{CB} = n_i$
11. Verify that the following equal 1:

$$\text{VS}(\text{PK}^1_{CB}, \text{addr}^1_{CB} \| v_{fees} + v_0 \| n^1_{CB}, s_{CB})$$

$$\textsc{ValSeq}\left(N, rt_{i-1}, \{\pi^{mod,j}_{debit}, \pi^{mod,j}_{credit}\}^N_{j=1}, \pi^{mod}_{CB}, rt_i\right)$$

F Security Properties

F.1 Completeness

It is important for the protocol to allow transactions formed by honest parties to be accepted and processed. This requires that it satisfy the property of completeness. We define completeness in the standard way, by requiring that no polynomial-sized adversary can win the incompleteness experiment with non-negligible probability. We formalize this below, where the experiment is an interaction between an algorithm $\mathcal{A}$ and a challenger $\mathcal{C}$.

Definition 8. We say that a DPS Π is *complete* if, for all poly(λ)-size algorithms $\mathcal{A}$ and large enough λ, the adversary wins INCOMP with at most negligible probability:

$$Pr[\text{INCOMP}(\mathcal{A}, \Pi, \lambda) = 1] \le negl(\lambda)$$

INCOMP$(\Pi, \lambda, \mathcal{A})$:

1. $\mathcal{C}$ samples $pp \leftarrow \text{Setup}(1^\lambda)$, sending pp to $\mathcal{A}$
2. $\mathcal{A}$ sends $\mathcal{C}$ the following:
   (a) A state $\mathcal{S}_i$
   (b) Three addresses z_a, z_b, z_{CB}
   (c) Positive integer values c_s, c_f, c_m
   (d) A key pair (PK, SK) corresponding to address **a**
   (e) A public key PK_B
   (f) A key pair (PK_{CB},SK_{CB})
   (g) Two signatures σ, σ_{CB}
   (h) Information strings $\mathsf{info}_S, \mathsf{info}_\sigma, \mathsf{info}_{\sigma_2}$
3. $\mathcal{C}$ checks that the following hold, outputting 0 if any test fails:
   (a) Check that the key pairs are well formed
   (b) Check that all addresses are different
   (c) Check that
   $$\text{VS}(\text{PK}, z_a \| z_b \| c_s \| c_f \| \mathsf{info}_\sigma, \sigma) = 1$$
   (d) Check that
   $$\text{VS}(\text{PK}_{CB}, z_{CB} \| c_m \| \mathsf{info}_{\sigma_2}, \sigma_{CB}) = 1$$
   (e) $\text{VerifyState}(pp, \mathcal{S}_i, \mathsf{info}_S) = 1$
   (f) Account **a** with **a**.addr $= z_a$ exists in $\mathcal{S}_i$ and is non-null with public key PK
   (g) If account **b** with **b**.addr $= z_b$ is initialized, check that it has public key PK_B
   (h) $\text{GetBalance}(pp, \mathcal{S}_i, \text{PK}) \ge c_s + c_f$
   (i) $c_f \le c_m$
4. $\mathcal{C}$ constructs a send transaction t with the given parameters:
$$\text{NewTransaction}(pp, z_a, z_b, c_s, c_f, (\text{PK}, \text{SK}), \text{PK}_B)$$
5. $\mathcal{C}$ constructs a coinbase transaction t_{CB} with the given parameters:
$$\text{NewCoinbase}(pp, z_{CB}, c_m, (\text{PK}_{CB}, \text{SK}_{CB}))$$
6. $\mathcal{C}$ checks that the following hold, outputting 0 if any test fails:
   (a) $\text{VerifyTransaction}(pp, t, \mathcal{S}_i) = 1$
   (b) $\text{VerifyTransaction}(pp, t_{CB}, \mathcal{S}_i) = 1$
7. Compute the state transition:
$$(\mathcal{S}_{i+1}, \mathsf{info}_{S_2}) = \text{NewState}(pp, \{t, t_{CB}\}, \mathcal{S}_i, \mathsf{info}_S)$$
8. Output 1 if any of the following hold:
   (a) $t \ne (z_a, z_b, c_s, c_f, \text{PK}, \sigma, \text{PK}_R)$
   (b) $t_{CB} \ne (z_{CB}, c_m, \text{PK}_{CB}, \sigma_{CB})$
   (c) $\text{GetBalance}(\text{PK}_B, \mathcal{S}_{i+1}) \ne \text{GetBalance}(\text{PK}_B, \mathcal{S}_i) + c_s$
   (d) $\text{GetBalance}(\text{PK}_{CB}, \mathcal{S}_{i+1}) \ne \text{GetBalance}(\text{PK}_{CB}, \mathcal{S}_i) + c_m$
   (e) $\text{GetBalance}(\text{PK}, \mathcal{S}_i) \ne \text{GetBalance}(\text{PK}, \mathcal{S}_{i+1}) + c_s + c_f$
   (f) $\text{VerifyState}(pp, \mathcal{S}_{i+1}, \mathsf{info}_{S_2}) = 0$

F.2 Correctness

In order to correctly set up the threat model, we need to define an oracle $\mathcal{O}$ that will initialize the DPS based on some public parameters pp, keep the state of the system $\mathcal{S}_i$ along with an information string $\mathsf{info}_\mathcal{S}$, and allow queries from the adversary. It will also keep counters C, D, E initialized to zero and an initially empty set ADDR. The possible queries are:

CreateAddress:

1. Generate $(\mathrm{PK}, \mathrm{SK}) \leftarrow \mathsf{CA}(pp_{sig})$
2. Add $(\mathrm{PK}, \mathrm{SK})$ to ADDR
3. Return PK

LookupAddress:(z_a)

1. Find the public key PK for account z_a
2. If PK is not in ADDR, return zero
3. Return $\mathrm{GetBalance}(\mathrm{PK}, \mathcal{S})$

RequestTransactions:

$$(z_a, z_b, c_s, c_f, \mathrm{PK}, \mathrm{PK}_B, z_{CB}, c_m, \mathrm{PK}_{CB})$$

1. Check that $\mathrm{PK}, \mathrm{PK}_{CB} \in \mathsf{ADDR}$
2. Retrieve SK and SK_{CB} and obtain σ, σ_{CB}
3. $t \leftarrow (z_a, z_b, c_s, c_f, \mathrm{PK}, \sigma, \mathrm{PK}_B)$
4. $t_{CB} \leftarrow (z_{CB}, c_m, \mathrm{PK}_{CB}, \sigma_{CB})$
5. Check that $\mathrm{VerifyTransaction}(pp, t, \mathcal{S}) = 1$
6. Check that $\mathrm{VerifyTransaction}(pp, t_{CB}, \mathcal{S}) = 1$
7. $E_0 = \mathrm{GetBalance}(\mathrm{PK}_B, \mathcal{S})$
8. Update the state and information string:
$$(\mathcal{S}_2, \mathsf{info}_{\mathcal{S}_2}) \leftarrow \mathrm{NewState}(pp, \{t, t_{CB}\}, \mathcal{S}, \mathsf{info}_\mathcal{S})$$
9. If $\mathrm{VerifyState}(pp, \mathcal{S}_2, \mathsf{info}_{\mathcal{S}_2}) = 1$, set
$$\mathcal{S} = \mathcal{S}_2, \mathsf{info}_\mathcal{S} = \mathsf{info}_{\mathcal{S}_2}$$
10. Check that $c_s = \mathrm{GetBalance}(\mathrm{PK}_B, \mathcal{S}) - E_0$
11. If $\mathrm{PK}_B \notin \mathsf{ADDR}$, set
$$E = E + c_s$$

AddTransactions:(t, t_{CB})

1. $(z_a, z_b, c_s, c_f, \mathrm{PK}, \sigma, \mathrm{PK}_B) \leftarrow t$
2. $(z_{CB}, c_m, \mathrm{PK}_{CB}, \sigma_{CB}) \leftarrow t_{CB}$
3. Check that $\mathrm{VerifyTransaction}(pp, t, \mathcal{S}) = 1$
4. Check that $\mathrm{VerifyTransaction}(pp, t_{CB}, \mathcal{S}) = 1$
5. $C_0 = \mathrm{LookupAddress}(z_b)$
6. $D_0 = \mathrm{GetBalance}(\mathrm{PK}_{CB}, \mathcal{S})$
7. Update the state and information string:
$$(\mathcal{S}_2, \mathsf{info}_{\mathcal{S}_2}) \leftarrow \mathrm{NewState}(pp, \{t, t_{CB}\}, \mathcal{S}, \mathsf{info}_\mathcal{S})$$
8. Check that $c_m = \mathrm{GetBalance}(\mathrm{PK}_{CB}, \mathcal{S}) - D_0$
9. If $\mathrm{PK}_B \in \mathsf{ADDR}$, check that
$$c_s = \mathrm{LookupAddress}(z_b) - C_0$$
10. If $\mathrm{VerifyState}(pp, \mathcal{S}_2, \mathsf{info}_{\mathcal{S}_2}) = 1$, set
$$\mathcal{S} = \mathcal{S}_2, \mathsf{info}_\mathcal{S} = \mathsf{info}_{\mathcal{S}_2}$$
11. Set $C = C + \mathrm{LookupAddress}(z_b) - C_0$
12. Set $D = D + c_m$

We are now ready to define the correctness experiment, which will prove security for our system.

Definition 9. We say that a DPS Π is *correct* if, for all poly(λ)-size adversaries $\mathcal{A}$ and large enough λ, the adversary wins INCOR with at most negligible probability:

$$Pr[\mathrm{INCOR}(\mathcal{S}, \Pi, \lambda) = 1] \leq negl(\lambda)$$

The game below refers to an interaction between an adversary $\mathcal{A}$ and challenger $\mathcal{C}$.

$\mathrm{INCOR}(\Pi, \lambda, \mathcal{A})$:

1. $\mathcal{C}$ samples $pp \leftarrow \mathrm{Setup}(1^\lambda)$, sending pp to $\mathcal{A}$
2. $\mathcal{C}$ instantiates an oracle $\mathcal{O}$ based on Π
3. $\mathcal{A}$ issues queries to $\mathcal{O}$
4. $\mathcal{A}$ sends a set of addresses $\{z_i\}_{i=1}^K$ to $\mathcal{C}$
5. $\mathcal{C}$ then adds together in a variable v all the balances of the addresses PK_i corresponding to z_i for which $\mathrm{PK}_i \notin \mathsf{ADDR}$
6. $\mathcal{C}$ outputs 1 if $v + C > D + E$

G Proof of Theorem 1

G.1 Completeness

Proof. In step (3), $\mathcal{C}$ ensures the transaction and state provided are valid. Since the transactions involve different addresses, we do not worry about one referencing the other.

We look at all the ways the adversary can win. Firstly, (a) and (b) are impossible since NewTransaction signs and creates the given transactions without changing parameters. By the completeness property of the signature scheme, the valid signatures will always verify.

Since the new state will be built according to UpdateState by exactly matching the balances of the participants according to the transaction amounts, we also ensure the equalities in (c), (d) and (e).

Finally, we know by completeness of the PCD that in the case of a valid state transition the verifier will reject with negligible probability. In our case, $\mathsf{info}_S = \pi_i$ and $\mathsf{info}_{S_2} = \pi_{i+1}$. Since Setup runs $\mathcal{G}$, NewState runs $\mathcal{P}$ and VerifyState runs $\mathcal{V}$, we know that:

$$\Pr\left[\begin{array}{c} \Pi(T) = 1 \\ \mathcal{V}(vk, \pi_{i+1}, \mathcal{S}_{i+1}) \neq 1 \end{array} \middle| \begin{array}{c} (pk, vk) \leftarrow \mathcal{G}(1^\mu) \\ (\mathcal{S}_{i+1}, \pi_{i+1}, T) \leftarrow \mathrm{PG}(\Pi, pk, \mathcal{A}, \mathcal{P}) \end{array}\right]$$

We can think of $\mathcal{A}$ as providing $\mathcal{S}_i$, $\mathcal{S}_{i+1}$ and local information $\{t, t_{CB}\}$, with $\mathcal{P}$ providing proof π_{i+1} through ProofGen (PG). From (6), we know that this is a valid state transition for our predicate, so remains to show that the transcript up to $\mathcal{S}_i$ is compliant. By the Proof of Knowledge property there exists an extractor $\mathbb{E}_{\mathcal{P}}(pk, \mathcal{S}_i) \rightarrow T_i$ with $out(T_i) = \mathcal{S}_i$ and $\Pi(T_i) = 1$ with high probability $1 - \mathrm{negl}(\mu)$. This means that $\Pi(T) = 1$ as all transitions are valid with high probability and so since $\mathcal{V}(vk, \mathcal{S}_i, \pi_i) = 1$ the adversary cannot activate (f) to win the game non-negligibly.

□

G.2 Correctness

Proof. The adversary only changes the state when they call **AddTransactions** and **RequestTransactions**. We assume they perform N calls and submit $\{t^i, t^i_{CB}\}_{i=1}^n$ at each step to set $\mathcal{S}_i \rightarrow \mathcal{S}_{i+1}$. Since NewState calls $\mathcal{P}$ to generate the transition at every step and we know that $\forall i \in [N]$, VerifyState$(pp, \mathcal{S}_i, \pi_i) = 1$, the transcript T recording these updates satisfies $\Pi(\mathrm{T}) = 1$ with high probability. From here we presume without loss of generality that every state $\mathcal{S}_i$ is consistent with Π.

C measures how much value $\mathcal{A}$ has transferred to honest parties in ADDR, D how many coinbase transactions $\mathcal{A}$ has won, and E how much money they have received from honest nodes. Since this cannot hold if balance is always conserved, we look at the possible options:

Option 1: The adversary has been able to use some PK $\in$ ADDR in **AddTransactions**. However, $\mathcal{A}$ only has oracle access to signatures of *other* messages from PK $\in$ ADDR so recreating one it has not seen non-negligibly would violate the security of the signature scheme. In order to validate a transaction, the adversary needs to sign a message including the nonce n of the last block that modified the account, which is always different with high probability. Therefore they would have to generate a signature for a unique message $m = \mathrm{addr}_s \| \mathrm{addr}_r \| c_1 \| c_2 \| n$, which has not been seen before in a query to **RequestTransactions**, as n always updates when a transaction is processed requiring a new m even if all other parameters remain the same.

Option 2: Balance is not conserved in at least one transaction. Since $\Pi(T) = 1$, this is immediately false with high-probability since all transactions added were valid and thus conserve balance between accounts by the requirements of ValidState.

Option 3: There exists a different compliant transcript T_2 with the same value for $\mathcal{S}_i$ for some i. This would mean that transactions valid for T_2 could be verified by T at the i-th step and thus violate the required expression. However, by design $\mathcal{S}_i$ is the root of a Merkle tree based on $\mathcal{H}$, which is a collision-resistant hash function. Suffices to show that a different transcript would imply a collision in $\mathcal{H}$.

Since the two transcripts differ, there exists a first node j in which they have at least one differing transaction. Let the two resulting Merkle tree roots be $\mathsf{root}(T^*(j))$ and $\mathsf{root}(T(j))$ after that update. Since the transactions were different, the two trees will differ in at least one leaf by $l \neq l^*$ and have that $\mathsf{root}(T^*(j)) \neq \mathsf{root}(T(j))$. Since the predicate recognizes Tree(BLOCK-V), we store the account Merkle tree root $\mathsf{root}(T(i))$ in the i-th leaf of another Merkle tree whose root is $\mathcal{S}_i$ for the i-th step. This means that there exists an authentication path from $\mathsf{root}(T^*)$ and $\mathsf{root}(T)$ to the same $\mathcal{S}_i$. Since $l \neq l^*$ and $\mathsf{root}(T^*(j)) \neq \mathsf{root}(T(j))$, this is a contradiction. □

H Proof of Theorem 2

Proof. Assume that a blocks are found in a Poisson process with a mean of $\lambda = 1$ and an individual miner can check one puzzle solution in time τ. Consider the expected number of blocks this individual miner is able to check before the network broadcasts a solution. A block will be found by the network in less than time τ with probability:

$$\int_0^\tau e^{-x} dx = 1 - e^\tau.$$

In this case, the miner will not even finish checking a single block. If the network does not broadcast a block within time τ, the miner will check at least one block. The Poisson process then repeats, since it is memoryless. So the expected number of blocks checked is:

$$\begin{aligned} \mathrm{E}_{blocks} &= (1 - e^\tau) \cdot 0 + e^{-\tau} \cdot (1 + \mathrm{E}_{blocks}) \\ e^\tau \cdot \mathrm{E}_{blocks} &= 1 + \mathrm{E}_{blocks} \mathrm{E}_{blocks} \qquad = \frac{1}{e^\tau - 1}. \end{aligned}$$

If no partially-checked solutions were wasted, the miner would always expect to check $\frac{1}{\tau}$ solutions. Thus, the fraction of wasted work is:

$$1 - \frac{\frac{1}{e^\tau - 1}}{\frac{1}{\tau}} = 1 - \frac{\tau}{e^\tau - 1}.$$

□