Coda: Decentralized cryptocurrency at scale

Izaak Meckler[1] and Evan Shapiro[1]

[1]O(1) Labs

May 10, 2018

Abstract

We present Coda, the first cryptocurrency protocol that remains decentralized at scale. "Scalability" refers to Coda's ability to handle throughput of thousands of transactions per second. "Decentralization" refers to the accessibility of verifying the chain state and synchronizing as a new user. Synchronizing the chain state with Coda requires receiving less than a megabyte of data, allowing devices like smartphones to securely perform transactions independent of how long the protocol has been running or how many transactions have been performed. Transactions on Coda can also be verified independent of their complexity, allowing complex computations on its blockchain without burdening the network. Coda achieves these features without sacrificing either scaling or decentralization through the use of recursive composition of zk-SNARKs in a novel architecture that implements a decentralized ledger. The resulting consensus protocol is consistent and responsive as long as at most 1/2 of the mining power is malicious.

1 Introduction

While cryptocurrencies have achieved notable financial and public recognition, they have had practicality problems due to poor scaling and high resource requirements. Resource efficiency is particularly concerning, as existing scaling solutions will raise the resource requirements even further, harming accessibility and decentralization.

These problems not only present practicality issues at cryptocurrency's present level of usage (e.g., prohibitively-high transaction fees, long sync times, high storage requirements for verification), but make scaling cryptocurrency to millions of users to be entirely infeasible without sacrificing decentralization.

We would like a cryptocurrency protocol that achieves the following properties

1. **Scalable.** In order to be used by millions of participants, the system must be capable of processing thousands of transactions per second.

2. **Decentralized.** In order to minimize centralizing barriers to entry, the protocol must have a constant cost of participation. The amount of computational resources (storage, computation, and network usage) required to join and interact with the network should be low and grow as slowly as possible.

Historically cryptocurrencies have faced a tradeoff between scaling and decentralization [Jor18]. As cryptocurrencies scale, their blockchains increase in size, causing increasing burden for participants to verify balances and data on the blockchain. These cryptocurrencies tend towards inaccessibility and more centralized validation with usage.

We present a new protocol, Coda, which overcomes this previous limitation and achieves both scaling and decentralization through the use of what we term a **succinct blockchain**.

A succinct blockchain is a blockchain for which verifying the current world state requires only a constant amount of data and a constant amount of time. Concretely, a user of Coda requires only around 20 kilobytes and about 10 milliseconds to verify their balance. These requirements are independent of the number or complexity of the transactions processed by the blockchain. This allows Coda to scale to large numbers of users while staying accessible to those with relatively limited computing resources.

Coda's succinct blockchain is enabled by recursive composition of zk-SNARKs [BCCT12]. Recursive composition of SNARKs enable constant-sized proofs of arbitrary, incremental computations. This allows the Coda network to construct a proof of the validity of blockchain state that can be updated incrementally as more blocks are added.

In this paper, we overview the cryptographic preliminaries needed for our succinct blockchain, and present the succinct blockchain construction we use in Coda.

2 Preliminaries

We begin with some preliminary cryptographic definitions.

Definition 2.1. A function f is called negligible if for any polynomial $p(\lambda)$, for all sufficiently large λ we have $f(\lambda) < \frac{1}{p(\lambda)}$. This will often be denoted $f(\lambda) < \mathsf{negl}(\lambda)$.

We review the definition of *collision-resistant hash functions* (CRH) which are used extensively in the recursive composition construction.

Definition 2.2. A collision-resistant hash function (CRH) consists of a probabilistic-polynomial-time algorithm Pgen that on input 1^λ outputs a circuit h such that for every non-uniform polynomial-size adversary $\mathcal{A}$,

$$\Pr\left[h \leftarrow_\$ \mathsf{Pgen}(1^\lambda); (x,y) \leftarrow_\$ \mathcal{A}(1^\lambda, h); x \neq y \text{ and } \mathsf{H}(x) = \mathsf{H}(y)\right] < \mathsf{negl}(\lambda)\,.$$

In our protocol, in order to ensure maximal efficiency of our SNARK construction, we use a Pedersen hash function [CDv88] [CvP92] [BGG94]. The Pedersen CRH has particularly low arithmetic complexity (and so is particularly efficient to verify with a SNARK).

Definition 2.3 (Pedersen CRH)**.** Fix an elliptic curve family $G = G(1^\lambda)$ and an input length $s(\lambda)$. The G-Pedersen CRH is the CRH given by

$$\begin{array}{l}\mathsf{Pgen}_G(1^\lambda)\\ \hline G \leftarrow_\$ G(1^\lambda);\\ (g_1,\ldots,g_s) \leftarrow_\$ G;\\ \mathsf{return}\left[(b_1,\ldots,b_s) \mapsto \prod_{i=1}^{s} g_i^{b_i}\right]\end{array}$$

where the b_is are boolean.

The collision-resistance of the Pedersen CRH reduces to the hardness of finding the discrete logarithm of a random element of G (with respect to a random base). For state-of-the-art SNARK performance optimizations, please see [HBHW17].

Finally, we review the definition of SNARKs for arithmetic constraint systems. For simplicity, we define an $\mathbb{F}$-arithmetic constraint system to simply be a list of polynomials over $\mathbb{F}$ in some variables $(x_1,\ldots,x_r,y_1,\ldots,y_s)$.

A satisfying assignment for a given constraint system will be an assignment of $\mathbb{F}$ elements to each variable which causes each polynomial in the system to evaluate to 0. For $a \in \mathbb{F}^r, w \in \mathbb{F}^s$, we use the notation $C(a,w)$ to indicate that assigning the x_i to a and the y_i to w is a satisfying assignment for C.

Definition 2.4. A succinct non-interactive argument of knowledge (SNARK) for $\mathbb{F}$-arithmetic constraint systems is is a tuple of algorithms (Setup, Prove, Verify) such that the following conditions are satisfied:

1. **Completeness.**

   For any constraint system C and a, w with $C(a,w)$, we have

   $$\Pr\left[(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Setup}(C,1^\lambda);\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk},a,w);\ \mathsf{Verify}(\mathsf{vk},a,\pi) = \top\right] = 1$$

   For every $(x,w) \in R$.

2. **Knowledge soundness.** For every polynomial-size prover P, there exists a polynomial-size extractor Ext such that for every $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$ we have

$$\Pr[(a,\pi) \leftarrow \mathsf{P}(\mathsf{pk},\mathsf{vk},z); w \leftarrow \mathsf{Ext}(\mathsf{pk},\mathsf{vk},z); \mathsf{Verify}(\mathsf{vk},a,\pi) = \top \text{ and not } C(a,w)] < \mathsf{negl}(\lambda)\,.$$

3. **Succinctness.** For every $\mathbb{F}$-arithmetic constraint system C, every $(\mathsf{pk},\mathsf{vk})$ generated by setup, and every $a \in \mathbb{F}^r$, we have that every honestly generated proof has size $\mathsf{poly}(\lambda)$ and $\mathsf{Verify}(\mathsf{vk},a,\pi)$ runs in time $\mathsf{poly}(\lambda) \cdot r$.

3 Recursive composition of SNARKs for state-transition systems

We now describe at a high level the recursive-composition technique described variously in [Val08], [BCCT13] and [BCTV14] for "bootstrapping" a preprocessing SNARK which can only certify execution of fixed-size non-deterministic computations into a SNARK capable of certifying essentially arbitrarily long non-deterministic computations.

Instead of phrasing the construction in the language of incrementally verifiable computation as in [Val08] or in the language of PCD as is done in [BCCT13] and [BCTV14], we opt to describe it in terms of state-transition systems as it maps more clearly onto the application of producing a compact blockchain.

Definition 3.1 (State transition system)**.** A state transition system consists of a type Σ of states, a type T of transitions, and a (non-deterministic) poly-time computable function $\mathsf{update}\colon \mathrm{T} \times \Sigma \to \Sigma$. update may also "throw an exception" (i.e., fail to produce a new state for certain inputs).

Moreover we ask that Σ and T be poly-sized, in the sense that their members can be represented by bitstrings of length $\mathsf{poly}(\lambda)$.

We now define SNARKs for state transition systems. At a high level, we would like $\mathsf{poly}(\lambda)$-size proofs (which are verifiable in $\mathsf{poly}(\lambda)$ time) which prove statements of the form "there exist a sequence of transitions $t_1,\ldots,t_k\colon \mathrm{T}$ such that $\mathsf{update}(t_k,\mathsf{update}(t_{k-1},\ldots,\mathsf{update}(t_1,\sigma_1),\ldots)) = \sigma_2$". In other words, we would like succinct certificates of the existence of state-transition sequences joining two states. The application to blockchains is the following: we will take our state to be the database of accounts (along with some metadata needed for correctly validating new blocks) and transitions to be blocks containing transactions.

Definition 3.2 (Incremental SNARKs for state transition systems (informal))**.** An incremental SNARK for a state transition system $(\Sigma,\mathrm{T},\mathsf{update})$ is a tuple of algorithms $(\mathsf{Pgen},\mathsf{Prove},\mathsf{Verify})$ such that (suppressing parameter generation and passing the parameters to Prove and Verify)

1. **Completeness.**

   For $\sigma_1\colon \Sigma$ and $t_1,\ldots,t_k\colon \mathrm{T}$, letting $\sigma_2 = \mathsf{update}(t_k,\mathsf{update}(t_{k-1},\ldots,\mathsf{update}(t_1,\sigma_1),\ldots))$, we have $\mathsf{Verify}(\sigma_1,\sigma_2,k,\mathsf{Prove}(\sigma_1,\sigma_2,(t_1,\ldots,t_k))) = \top$.

2. **Incrementality**

   If $\mathsf{Verify}(\sigma_1,\sigma_2,k,\pi) = \top$ and $t\colon T$, then letting $\sigma_3 = \mathsf{update}(t,\sigma_2)$, we have $\mathsf{Verify}(\sigma_1,\sigma_3,k+1,\mathsf{Prove}(\sigma_1,\sigma_3,(\pi,t))) = \top$.

3. **Knowledge soundness**

   For $T = (t_1,\ldots,t_k)$ a sequence of transitions, define

   $$\mathsf{update}(T,\sigma) = \mathsf{update}(t_k,\cdots,\mathsf{update}(t_1,\sigma),\cdots).$$

   For every polynomial-size prover P, there exists a polynomial-size extractor Ext such that for every $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$,

   $$\Pr[(\sigma_1,\sigma_2,\pi) \leftarrow \mathsf{P}(z); T \leftarrow \mathsf{Ext}(z); \mathsf{Verify}(\sigma_1,\sigma_2,k,\pi) = \top \text{ and } \mathsf{update}(T,\sigma) \neq \sigma_2] < \mathsf{negl}(\lambda)\,.$$

4. **Succinctness** Every honestly generated proof has size $\mathsf{poly}(\lambda)$ and for any proof π, $\sigma_1,\sigma_2\colon \Sigma$, and k, we have that $\mathsf{Verify}(\sigma_1,\sigma_2,k,\pi)$ runs in time $\mathsf{poly}(\lambda) \cdot \log k$.

The incrementality is very useful, as it implies the proving process can be executed in parallel to compress k state transitions with span (or "wall-clock time") $O(\log k)$.

Naive recursive composition is extremely expensive. To address this, we use the "cycle of elliptic curves" technique (as described in [BCTV14]) in which two SNARK constructions (let's call them Tick and Tock) verify each other. As a first step, let's define a set of typing rules such that the existence of a term $\pi : \sigma_1 \to_{\mathsf{Tock}} \sigma_2$ implies the existence of a sequence of transitions taking σ_1 to σ_2. The typing rules are as follows:

$$\frac{\sigma_1 : \Sigma \quad \sigma_2 : \Sigma \quad t : \mathrm{T} \quad \mathsf{update}(t, \sigma_1) = \sigma_2}{\mathsf{base}(\sigma_1, \sigma_2, t) : \sigma_1 \to_{\mathsf{Tick}} \sigma_2}\ \text{base} \qquad \frac{\sigma_1 : \Sigma \quad \sigma_2 : \Sigma \quad \pi : \sigma_1 \to_{\mathsf{Tick}} \sigma_2}{\mathsf{wrap}(\sigma_1, \sigma_2, \pi) : \sigma_1 \to_{\mathsf{Tock}} \sigma_2}\ \text{wrap}$$

$$\frac{\sigma_1 : \Sigma \quad \sigma_2 : \Sigma \quad \sigma_3 : \Sigma \quad \pi_1 : \sigma_1 \to_{\mathsf{Tock}} \sigma_2 \quad \pi_2 : \sigma_2 \to_{\mathsf{Tock}} \sigma_3}{\mathsf{merge}(\sigma_1, \sigma_2, \sigma_3, \pi_1.\pi_2) : \sigma_1 \to_{\mathsf{Tick}} \sigma_3}\ \text{merge}$$

From a term $\pi : \sigma_1 \to_{\mathsf{Tock}} \sigma_2$ we can obtain a sequence of transitions taking σ_1 to σ_2. Indeed, a term of type $\sigma_1 \to_{\mathsf{Tock}} \sigma_2$ here is essentially an explicitly parenthesized sequence of transitions leading from σ_1 to σ_2.

Now, we will implement each of these deduction rules with a SNARK, such that the proof-of-knowledge assumption for each SNARK will correspond to a kind of inversion-lemma for the corresponding deduction rule. This will then allow us to apply essentially the same reasoning as above to conclude that a SNARK which purportedly witnesses the existence of transitions taking σ_1 to σ_2 in fact implies the knowledge of such transitions.

We will have 3 SNARKs, one corresponding to each deduction rule.

1. A Tick-based SNARK for certifying single state transitions, which we'll call the "base" SNARK.

   Input: (The hash of) a tuple (σ_1, σ_2, y) where $\sigma_i : \Sigma$ and y is an arbitrary string.

   Statement: There exists $t : \mathrm{T}$ such that $\mathsf{update}(t, \sigma_1) = \sigma_2$.

2. A Tick-based SNARK for merging two Tock proofs, which we'll call the "merge" SNARK.

   Input: (The hash of) a tuple $(\sigma_1, \sigma_3, \mathsf{vk})$ where $\sigma_i : \Sigma$ and vk is a Tock verification key.

   Statement: There exists $\sigma_2 : \Sigma$ and Tock-proofs π_1, π_2 such that $\mathsf{Verify}_{\mathsf{Tock}}(\mathsf{vk}, (\sigma_1, \sigma_2, \mathsf{vk}), \pi_1)$ and $\mathsf{Verify}_{\mathsf{Tock}}(\mathsf{vk}, (\sigma_2, \sigma_3, \mathsf{vk}), \pi_2)$

   In plain English, it will prove that there exists a SNARK certifying the existence of transitions from σ_1 to σ_2 and a SNARK certifying the existence of transitions from σ_2 to σ_3.

3. A Tock-based SNARK for wrapping a Tick proof, which we'll call the "wrap" SNARK. Let $\mathsf{vk}_{\mathsf{base}}, \mathsf{vk}_{\mathsf{merge}}$ be "base" and "merge" verification keys respectively.

   Input: An arbitrary string x.

   Statement: There exists a Tick proof π and $\mathsf{vk} \in \{ \mathsf{vk}_{\mathsf{base}}, \mathsf{vk}_{\mathsf{merge}} \}$ such that $\mathsf{Verify}_{\mathsf{Tick}}(\mathsf{vk}, x, \pi)$.

   This SNARK merely wraps a Tick SNARK into a Tock SNARK so that another Tick SNARK can verify it efficiently.

Various tricks are applied in the actual implementation for efficiency's sake, but for simplicity of exposition we omit them here.

One obtains the following proposition regarding the above proof system.

Proposition 3.1 (SNARKs for state transition systems, informal)**.** *Assume the existence of collision resistant hash functions and a pair of preprocessing SNARK constructions with linear-time knowledge extractors.*

Let $(\Sigma, \mathrm{T}, \mathsf{update})$ *be a state transition system. Then the above construction instantiated with those primitives yields a state transition system SNARK for* $(\Sigma, \mathrm{T}, \mathsf{update})$.

As mentioned, the proof-of-knowledge assumption for each SNARK corresponds to a kind of inversion-lemma for the corresponding deduction rule. A full argument (in the language of PCD) is provided in [BCTV14].

4 Succinct blockchains

A **succinct blockchain** is a blockchain with verification complexity essentially independent of chain length. Instead of preserving the entire chain, one merely holds onto the current state along with a SNARK proving that *there exists* a blockchain explaining the current state. In fact, it is even better: the SNARK certifies the existence of a blockchain explaining a state with Merkle root h. Then, users who are interested in only part of the state (e.g., their own balance) can be given this SNARK along with a small Merkle-path corresponding to root h into the part of the state that they are interested in.

Blockchain verification and extension can be expressed as a state transition system, and thus SNARKs for state transition systems as described above enable the construction of succinct blockchains.

We handle the permissioning mechanism (whether it be proof-of-work, proof-of-stake, or something else) abstractly.

Definition 4.1 (Permission mechanism)**.** A permission mechanism consists of types PermissionState and PermissionProof along with polytime-computable functions

1. checkPermission : PermissionState $\times$ PermissionProof $\to \{ \top, \bot \}$
2. updatePermission : PermissionState $\times$ PermissionProof $\to$ PermissionState.

Let us give some examples of how this definition can be instantiated. For proof-of-work, the permission state would contain several previous difficulty targets and block-times (from which to compute the current difficulty target) and a permission proof would contain the proof-of-work itself along with a new time to update the state with.

For a Praos-style [DGKR17] proof-of-stake mechanism, the permission state would contain the current random seed, the (Merkle root of) the current epoch's stakes, and some information about the previous blocks and block times. A permission-proof would contain a public-key and a VRF evaluation proof meeting the difficulty target corresponding to that public-key and the stakes indicated in the permission state.

This definition omits mention of generating the permission proofs, dealing only with their verification. The following definition of a succinct blockchain will assume some permissioning mechanism. We also assume the existence of a collision-resistant hash function H, and whenever we speak of hashes, we are referring to outputs of H.

We can now describe in more detail the precise transition system giving rise to a succinct blockchain. Our blockchain will maintain consensus on a "ledger", which is a list of accounts with balances. Users submit "transactions" which are instructions to transfer balance from one account to another. We discuss the structure of the ledger and transactions in more detail in section A.

Instead of containing transactions directly, blocks will include a SNARK proving the existence of transactions which when applied to the previous ledger leave it in some new state. That is, we assume a function verifyLedgerProof which given two hashes r_1 and r_2 along with a proof π returns $\top$ (computationally) iff there exists a sequence of transactions taking a ledger with Merkle root r_1 to a ledger with Merkle root r_2.

Definition 4.2 (Blockchain-state)**.** The blockchain's state type Σ will be a record consisting of

1. previousStateHash: The hash of the previous blockchain state.
2. ledgerHash: The Merkle root of the database of accounts with balances.
3. permissionState: A value of type PermissionState.

Definition 4.3 (Blockchain transition)**.** The transition type T will be a record consisting of

1. permissionProof: A value of type PermissionProof.
2. ledgerProof: A SNARK proof.
3. nextLedgerHash: The purported hash of the ledger after applying the transactions indicated by ledgerProof.

Finally, we define the update function for a succinct blockchain's transition system.

$\mathsf{update}(s, t)$

$\mathsf{assert}\ (\mathsf{checkPermission}(\mathsf{permissionState}(s), \mathsf{permissionProof}(t)) = \top);$

$\mathsf{assert}\ (\mathsf{verifyLedgerProof}(\mathsf{ledgerHash}(s), \mathsf{nextLedgerHash}(t), \mathsf{ledgerProof}(t)) = \top);$

$\mathbf{return}\ (\mathsf{H}(s), \mathsf{nextLedgerHash}(t), \mathsf{updatePermission}(\mathsf{permissionState}(s), \mathsf{permissionProof}(t)))$

Simply put, to update a blockchain with a given transition, we check that the transition is permissioned to act on the blockchain, then we check the correctness of the transactions the transition wants to apply to the ledger, and finally we return a new blockchain whose previousStateHash, ledgerHash, and permissionState have been set appropriately.

Instantiating the construction of SNARKs for state-transition systems with this update function yields a SNARK construction for certifying statements of the form "there exist a sequence of appropriately-permissioned transactions which when applied to blockchain-state σ_1 result in blockchain-state σ_2." We fix an arbitrary "genesis state" $\sigma_0 \colon \Sigma$ and define a "succinct blockchain" (which we may refer to simply as a "blockchain") as follows.

Definition 4.4 (Succinct blockchain)**.** In Coda, a blockchain is a blockchain-state $\sigma \colon \Sigma$ along with a state-transition SNARK which verifies against the input (σ_0, σ). I.e., a SNARK proving the existence of a sequence of valid transitions linking the genesis state to σ.

Nodes can participate in a succinct blockchain protocol without storing anything except for the strongest blockchain and a full or partial state. If a node has these items, they can be certain that the information in whatever state they hold is backed by a blockchain with the strength indicated and that balances have been updated only via a sequence of valid transactions contained in that blockchain.

5 Permissioning mechanism and network model

At a high level, Coda works by simply instantiating the succinct blockchain construction with a permissioning mechanism such as Ouroboros Praos proof-of-stake protocol[DGKR17] and the same distributed system assumptions. The result would be a succinct proof-of-stake blockchain.

In practice, nodes have additional local state (beyond that explicitly described in the blockchain) which they use in the consensus process. This process is described in detail in [DGKR17] and is essentially unchanged in Coda.

5.1 Ledger properties

The succinct blockchain construction instantiated with a particular permissioning-mechanism inherits all of the consensus properties provided by a typical verbose blockchain instantiated with the same permissioning-mechanism.

In particular, assuming Coda is instantiated with a permissioning-mechanism such as Ouoroboros Praos, it provides persistence and liveness.

To properly define permission and liveness, we first define the notion the history of a blockchain.

Definition 5.1. Say σ_1, σ_2 are blockchain-states. We say that σ_1 is the predecessor of σ_2 (or that σ_2 is the successor of σ_1) if $\mathsf{H}(\sigma_1) = \mathsf{previousStateHash}(\sigma_2)$.

By collision-resistance of H, each valid blockchain-state σ has a computationally-unique predecessor which we denote $\mathsf{pred}(\sigma)$. This is with exception of the genesis state σ_0, which is chosen so that it (computationally) has no predecessor.

Definition 5.2. Let (σ, π) be a blockchain, and let k be the (computationally) unique integer such that $\mathsf{pred}^k(\sigma) = \sigma_0$. We define the history of (σ, π) to be the set $\left\{ \mathsf{pred}(\sigma), \mathsf{pred}(\mathsf{pred}(\sigma)), \ldots, \mathsf{pred}^k(\sigma) \right\}$.

We are now in a position to define the persistence and liveness properties provided by Coda.

Persistence. Suppose an honest node claims at some point that current blockchain is B_1 and then claims at a later point that the current blockchain is B_2. Then with high probability B_1 is in the history of B_2.

Liveness. Suppose all honest nodes in the system attempt to include a given transaction t when updating the ledger. Then with high probability, after a constant amount of time t will be used in creating a "ledger proof", and thus its effect on the ledger will be confirmed by the network.

The ledger functionality implemented by Coda has several additional highly desirable properties which have not been achieved by any other cryptocurrency. Let A be the number of accounts in the ledger.

Decentralization and universal availability. Any node can verify their balance in the ledger by downloading an amount of data which has size $O(\log A)$ and which requires time $O(\log A)$ to verify. Note that this is asymptotically optimal as $O(\log A)$ is the amount of data required to even write down an index into a ledger of size A.

Sustainability. The amount of data stored by any node in the system is constant in the number of transactions that have been processed. In particular it is $O(A)$.

References

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks . . ., 2012.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. pages 111–120, 2013.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. pages 276–294, 2014.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. pages 216–233, 1994.

[CDv88] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. pages 87–119, 1988.

[CvP92] David Chaum, Eugène van Heijst, and Birgit Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. pages 470–484, 1992.

[DGKR17] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. Cryptology ePrint Archive, Report 2017/573, 2017. http://eprint.iacr.org/2017/573.

[HBHW17] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification: Version 2018.0-beta-19 [overwinter+sapling]. https://github.com/zcash/zips/blob/master/protocol/sapling.pdf, 2017.

[Jor18] Raul Jordan. How to scale ethereum: Sharding explained. https://medium.com/prysmatic-labs/how-to-scale-ethereum-sharding-explained-ba2e283b7fce, 2018.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. pages 1–18, 2008.

Appendix A Ledger and ledger updates

The succinct blockchain construction described in section 4 assumes the ability to certify the existence of transactions taking a ledger in one state to a ledger in another state. Here we describe the exact construction in more detail. We assume the existence of a digital signature scheme.

A "ledger" will be a list of "accounts". An "account" will be a record containing a signature public-key, an integer representing the account balance, and an integer nonce which functions to prevent transaction replay.

As described in section 4, the ledger is represented within the blockchain's transition system state as its Merkle root. This is important for the efficiency of the system as representing the

entire ledger explicitly is both prohibitively costly. Representing the ledger as its Merkle root also enables the possibility of checking the validity of portions of the state (e.g. only one's own balance) without having to see the whole state itself.

Let us describe how we can "compress" away transactions again using the transition-system SNARK construction. A "transaction" will be a record consisting of a "sender" public-key, a "receiver" public-key, an integer indicating the amount of the transaction, an integer nonce, and a signature. We omit discussion of fees.

We instantiate the state-transition SNARK construction to produce SNARKs proving statements of the form "there exists a sequence of transactions which when applied to the ledger with Merkle root r_1 yield the ledger with Merkle root r_2."

The state type will simply consist of a hash (interpreted as the Merkle root of the ledger). A transition will simply be a transaction. The (non-deterministic) update function is given by the following, where `request` represents non-determinstic choice and `assert` aborts the computation if the given condition is not true.

```
let update t s =
  let { signature; body = { sender; receiver; amount; nonce } } = t;
  Signature.assert_verifies signature sender t.body;
  let s' =
    find_and_modify s sender (fun account ->
      assert (account.balance >= amount);
      assert (account.nonce = nonce);
      return { account with balance = account.balance - amount; nonce = nonce + 1 });
  let s'' =
    find_and_modify s' receiver (fun account ->
      return { account with balance = account.balance + amount });
  return s''
where
  find_and_modify s public_key f =
    request account : Account;
    assert (account.public_key = public_key);
    request auth_path : List (Bool * Hash);
    Merkle_tree.assert_membership s account auth_path;
    request s' : Merkle_root;
    Merkle_tree.assert_membership s' (f account) auth_path;
    return s'
```


It is important to reiterate that the incremental nature of the transition-system SNARK construction enables one to compose individual transactions along a binary tree. This means that a sequence of k transactions can be compressed into a single SNARK in span $C \cdot \log k$ (for some constant C).

What this means is that even though the requirement that transactions be verified with a SNARK incurs a computational overhead, it does not incur a "wall-clock time" overhead. To be specific, if k transactions per second are sent to the system, over s seconds, we will need to SNARK sk transactions. This will take time $C \log(sk) = C \log k + C \log s$, which for large enough s is less than s. Thus the time required to SNARK the transactions will asymptotically be exceeded by the time taken to produce the transactions, allowing the system to function at any throughput assuming sufficient parallelism.