The DZCK Whitepaper

Abstract

This paper introduces a blockchain-based distributed ledger protocol called DZCK secured via a novel Proof-of-Stake-based consensus mechanism, enabling permission-less participation in the process of state transition execution, validation and addition to the global state while simultaneously providing strong finality guarantees for the said state transitions. The protocol is built to preserve privacy when transacting with the native protocol asset called DZCK and with native support of zero-knowledge proof-related primitives on the generalized compute layer.

Contents

1 Introduction **3**
- 1.1 Our Contributions . . . 5

2 Overview **5**

3 Use Cases **6**

4 Notation **6**

5 Abstract Protocol **8**

6 Cryptographic Primitives **9**
- 6.1 Hash Function . . . 9
- 6.2 Merkle Tree . . . 9
- 6.3 Elliptic Curve . . . 10
- 6.4 Stealth Address Scheme . . . 10
- 6.5 Encryption Scheme . . . 10
- 6.6 Commitment Scheme . . . 11
- 6.7 Signature Scheme . . . 11
- 6.8 Zero-Knowledge Proof Scheme . . . 12

7 Building Blocks **13**
- 7.1 Segregated Byzantine Agreement . . . 13
  - 7.1.1 Proof-of-Blind Bid . . . 15
  - 7.1.2 Generation Phase . . . 17
  - 7.1.3 Reduction Phase . . . 18
  - 7.1.4 Agreement Phase . . . 19
  - 7.1.5 Main Consensus Loop . . . 19
- 7.2 Phoenix . . . 20
- 7.3 Zedger . . . 22
- 7.4 Rusk VM . . . 23
- 7.5 Genesis Contracts . . . 25
  - 7.5.1 DZCK Contract . . . 25
  - 7.5.2 Bid Contract . . . 26
  - 7.5.3 Stake Contract . . . 27
  - 7.5.4 Reward Contract . . . 27
- 7.6 Kadcast . . . 27

8 Concrete Protocol **27**

9 Related Works **30**

10 Conclusion **30**

1 Introduction

The notion of digital currencies deployed in a distributed network secured via cryptographically and game-theoretically sound primitives rather than trust has been a point of discussion in limited circles of enthusiasts for decades before being formalized for the first time by David Chaum [Cha82]. Between then and the introduction of Bitcoin [Nak08] in 2008, numerous researchers [Cha82; LSS96; Wei98; VCS03; Sza05] in the field attempted to propose a viable digital currency protocol.

The first noticeable breakthrough happened with the release of the Bitcoin whitepaper [Nak08], which ushered in a new era of research and enthusiasm in the field. Built on top of a novel digital ledger called *blockchain* and secured via a Proof-of-Work-based consensus mechanism referred to as *Nakamoto consensus* in modern literature, inspired by the works of [DGN04] and [Bac02], Bitcoin became the first truly decentralized digital currency, inspiring the work on other decentralized applications, such as decentralized DNS [Nam11] and a generalized distributed state machine [Woo21]. Soon after the release of Bitcoin, researchers have begun uncovering numerous issues previously unbeknownst to the creator/s[1] of Bitcoin. [KCW13; ES18; GKL15; SSZ17; PSS17; Bon16] have discovered deficiencies in the assumptions outlined in the Bitcoin whitepaper with regards to the consensus mechanism and the accompanying economic model. Additionally, the paper [RS12] published by Ron and Shamir has been the first of many to demonstrate the relative triviality of transaction graph analysis and the lack of anonymity that the Bitcoin users maintain.

The issue of excessive energy consumption required to retain the security guarantees of the data stored in the ledger as well as other consensus mechanism-related protocol inefficiencies has been another point of contention for the Bitcoin protocol. Throughout the years, multiple researchers have tackled the issue with various solutions, the majority of which revolved around a concept of *one-vote-per-share* as a substitute for *one-vote-per-CPU* acting as a Sybil attack protection mechanism. The idea of Proof-of-Stake was first formalized in the Peercoin whitepaper [KN12] and since has been expanded upon by others such as [Ben+14b; Goo14; Mic16; But+20]. A more formal approach has been taken by [DPS16; Kia+17; Dav+18]. The protocols referenced above belong to the family of *chain-based* Proof-of-Stake mechanisms, which attempt to emulate the Proof-of-Work-based family of mechanisms, specifically the *longest chain* rule. The downside of probabilistic finality of chain-based mechanisms was tackled by Algorand [Mic16], which utilized various novel techniques to guarantee near-instant transactional finality (with a statistically negligible probability of a fork) while retaining the *permission-less* properties of the underlying protocol. Unfortunately, the mechanism came with disadvantages, mainly revolving around the committee (2000+) and certificate, attesting to the validity of the block,

[1]

sizes.

Understanding the importance of anonymity, researchers began working on techniques to convert Bitcoin into an anonymity-preserving protocol. The initial idea was to utilize mixers, trusted services which combine the inputs and outputs of multiple users into a single transaction. The downsides of the service was the reliance on trust as well as the lack of obfuscation of the amounts involved. The initial surge of interest in anonymity-preserving digital currencies was followed by the publications of [Ben+14a; Sab13; Hop+21; Max15; NMM16; Poe16; Fau+18; Jiv19; Bun+19], which took differing approaches to the problem with differing outcomes. The resulting rise of interest, has seen multiple projects, such as Monero and Zcash, rise in popularity with anonymity preservation being the main selling point of the aforementioned protocols.

Additionally, the inquiry into the expansion of the smart contract functionality has begun since the introduction of Bitcoin. Originally proposed in [Sza96] and expanded upon in [Sza97], smart contracts represent a set of codified rules automatically encforced without the need for trusted intermediaries. While Bitcoin's Script language is capable of facilitating limited smart contract functionality, it is not Turing complete and is unable to facilite arbitrary computations without off-chain solutions such as [Ler19]. The idea of expanding Bitcoin's limited scripting functionality was originally proposed by [But13], later to be formalized as [Woo21]. Ethereum includes a custom-designed Turing complete Virual Machine called *Ethereum Virtual Machine* (EVM), which supports [Eth21], [Tea19] as well as other programming languages and is currently considered to be the default virtual machine standard for blockchain-related applications.

In this paper we present DZCK , a blockchain-based distributed ledger protocol secured via a novel Proof-of-Stake-based consensus mechanism, enabling permission-less participation in the process of state transition execution, validation and addition to the global state while simultaneously providing strong finality guarantees for the said state transitions. The protocol is built to preserve privacy when transacting with the native protocol asset called DZCK and with native support of zero-knowledge proof-related primitives on the generalized compute layer. DZCK protocol is conceptually split into two non-overlapping layers: native protocol asset (i.e. DZCK) and general compute layer. Though both share the same state space, DZCK can be conceptualized as a separate layer due to the numerous privileges the asset retains within the DZCK protocol, such as being a sole asset permitted to be utilized for staking and for computational cost reimbursement for transaction execution costs as well as the contract accommodating DZCK -related logic, eponymously called DZCK *Contract*, being the singular entry point to state transition initiation. [Mah20] expands on the token economics of DZCK and the DZCK protocol economic model as a whole.

1.1 Our Contributions

Our contributions include the following:

- We formalize a novel privacy-preserving leader extraction procedure called *Proof-of-Blind Bid*. Proof-of-Blind Bid belongs to the family of *Private Proof-of-Stake* mechanisms such as [Ker+19; GOT19] and forms the basis to *Segregated Byzantine Agreement* (*SBA* from hereon) consensus mechanism.
- We introduce a novel consensus mechanism called SBA. SBA is permissionless *committee-based* Proof-of-Stake protocol which provides near-instant transactional finality guarantees with a negligible probability of a fork.
- We present a *UTxO-based* privacy-preserving transaction model called *Phoenix*. Phoenix expands on the transaction model proposed in [Hop+21] to enable the users to spend non-obfuscated outputs confidentially, a requirement for quasi-Turing complete systems where the final cost of the execution is unknown until the termination of said execution.
- We introduce a hybrid privacy-preserving transaction model called *Zedger*. Zedger is a model created to comply with the regulatory requirements of security tokenization and lifecycle management. Zedger utilizes a novel structure called *Sparse Merkle-Segment Trie* as a basis for *private memory* representing a user account, in which the respective owner of the said account can log the balance changes per segment while only revealing the change to *some* Sparse Merkle-Segment Trie root publicly.
- We propose a *WebAssembly*[Ros21]-based Virtual Machine called *Rusk VM*. Rusk VM includes native zero-knowledge proof verification functionality as well as support for efficient creation of Merkle Tree structures.
- We formalize the concrete version of the protocol as DZCK protocol.

2 Overview

The paper is organized as follows. **Section 3** outlines the main use cases for the DZCK protocol. We define the notation utilized in the paper in **Section 4**. **Section 5** describes an idealized abstract version of the protocol $\mathcal{P}_{\mathcal{E}}^{\mathsf{ideal}}$. We delve into the cryptographic fundamentals that make up the building blocks for the concrete version of the protocol in **Section 6**. We define the building blocks for the concrete version of the protocol in **Section 7**. **Section 8** formalizes the concrete version of the protocol $\mathcal{P}_{\mathsf{Chain},\mathcal{A}}^{\mathsf{real}}$. In **Section 9** we discuss the related works and conclude our thoughts in **Section 10**.

3 Use Cases

While a protocol enabling arbitrary on-chain computation such as DZCK can act as a host of a virtually unlimited number of unique applications, we designed DZCK with a specific set of use cases in mind. More specifi-cally, DZCK was primarily conceived with regulatory compliant security tokenization and lifecycle management in mind. The details of the security tokenization standard are beyond the scope of this paper. We encourage readers to refer to [Mah21] for an in-depth look into this Confidential Security Contract (XSC) Standard. Additionally, a Confidential Token Standard was created to enable seamless interaction between non-regulated and regulated assets within the protocol, without compromising on the privacy of the interacting users. This also enables the DZCK protocol to act as a privacy-preserving sidechain for any other existing Layer 1 protocol via trusted or trust-minimized interoperability solutions [Zam+19].

4 Notation

This section defines the common notation utilized throughout this paper. Some standard notations are not denoted for the sake of brevity.

A `leftarrow` and alteranrively `rightarrow` symbol denotes an assignment function, e.g. $A \leftarrow B$ or $B \rightarrow A$ stands for the assignment of B to A. The equivalency function is denoted with a = symbol, e.g. $A = B$ stands for A is equivalent to B. A `mapsto` symbol defines a mapping function, e.g. $A \mapsto B$, where B is mapped to A. The || symbol denotes an appension of two byte sequences, e.g. $a||b$ denotes b being appended to a. / symbol denotes division without the remainder, e.g. $6/4 = 1$, while % symbol denotes the remainder of the rounded division, e.g. $6\%4 = 2$.

We denote a tuple containing 1 or more element with round brackets, e.g. $\mathsf{Tuple} \leftarrow ()$. A set is defined with a $\mathsf{Set}[]$ function, which denotes the element type the set is comprised of, e.g. $\mathsf{ExampleSet} \leftarrow \mathsf{Set}[\mathsf{bool}]$. A list is defined with a $\mathsf{List}[]$ function, which denotes the element type the list is comprised of, e.g. $\mathsf{ExampleList} \leftarrow \mathsf{List}[\mathsf{bool}]$ stands for a list comprised of bool elements. The difference a set and a list is that a set can only be comprised of unique elements whereas a list can be comprised of identical elements. We define a mapping functions $\mathsf{Mapping}[]$ which maps a key key to a value value, i.e. $\mathsf{ExampleMap} \leftarrow \mathsf{Mapping}[\mathsf{byte32} \mapsto \mathsf{bool}]$. A binary sequence is defined with a $\mathsf{BinarySeq}[\mathsf{N}]$ funtion, where N ($N > 0$) is the number of bytes the defined binary sequence is comprised of, e.g. $\mathsf{ExampleBinarySeq}_{\bigstar} \leftarrow \mathsf{BinarySeq}[32]$ stands for a binary sequence comprised of 32 bytes. A byte sequence is defined as string of bytes. The member elements of the tuple, list, set, binary sequence and byte sequence can be obtained by providing the index to their position in the structure (where index always begins with 0), e.g. $\mathsf{bool} \leftarrow$

$\mathsf{ExampleList}[2]$. A structure name enclosed in $||$ defines the cardinality of the said structure, i.e. $|\mathsf{ExampleList}| = 4$. A key provided as a substitute for an index in case of a mapping would return the value corresponding to the said key, e.g. $\mathsf{bool} \leftarrow \mathsf{ExampleMap}[\mathsf{byte32}]$.

We denote a Merkle Tree structure with a $\mathsf{MerkleTree}[]$ function, which defines the element type the Merkle Tree leaves are comprised of and the depth of the Merkle Tree, e.g. $\mathsf{ExampleTree} \leftarrow \mathsf{MerkleTree}[\mathsf{bool}, 32]$, where 32 is the depth of the Merkle Tree of arity 2 (i.e. $\mathsf{LeafCount}(\mathsf{ExampleTree}) = 2^{32}$). A Merkle Tree with a superscript capital R defines the root of the Merkle Tree, e.g. $\mathsf{ExampleTree}^R$. We define $\mathsf{LeafCount}()$ as a function that returns the maximum number of leaves the said Merkle Tree can have. Function $\mathcal{O}()$ takes the Merkle Tree root $\mathsf{ExampleTree}^R$ and the Merkle Tree path from a specified leaf $\mathsf{exampleLeaf}^P$ and returns a `true` is the provided path is a valid opening to the specified root or `false` otherwise, i.e. $\mathsf{bool} \leftarrow \mathcal{O}(\mathsf{ExampleTree}^R, \mathsf{exampleLeaf}^P)$.

We denote $\mathsf{Generators}$ to be a mapping, which maps the hash of secret $\mathsf{secretHash}$ to a tuple comprised of the cryptographic commitment to the bid balance c_{bid}, bid eligibility height $h_{\mathsf{bidEligibility}}$ and bid expiration height $h_{\mathsf{bidExpiration}}$, i.e.:

$$\mathsf{Generators} \leftarrow \mathsf{Mapping}[\mathsf{secretHash} \mapsto (c_{\mathsf{bid}}, h_{\mathsf{bidEligibility}}, h_{\mathsf{bidExpiration}})].$$

Additionally, we denote $\mathsf{Provisioners}$ to be a mapping, which maps the BLS public key pk_{BLS} to a tuple comprised of the stake balance b_{stake}, bid eligibility height $h_{\mathsf{stakeEligibility}}$ and bid expiration height $h_{\mathsf{stakeExpiration}}$, i.e.:

$$\mathsf{Provisioners} \leftarrow \mathsf{Mapping}[pk_{\mathsf{BLS}} \mapsto (b_{\mathsf{stake}}, h_{\mathsf{stakeEligibility}}, h_{\mathsf{stakeExpiration}})].$$

We assume that there exist functions $\mathsf{TotalActiveStake}()$, $\mathsf{TotalHonestStake}()$ and $\mathsf{TotalByzantineStake}()$ which return the total amount of DZCK staked by the active (the owners of the bids/stakes for which the current block height is not smaller than $h_{\mathsf{bidEligibility}}/h_{\mathsf{stakeEligibility}}$ and smaller than $h_{\mathsf{bidExpiration}}/h_{\mathsf{stakeExpiration}}$) Generators/Provisioners, the total amount of DZCK staked by the honest Generators/Provisioners and the total amount of DZCK staked by the Byzantine Generators/Provisioners (**NOTE:** the three functions exist only in the theoretical model [aside from $\mathsf{TotalActiveStake}()$ applicable only to Provisioners] and do not exist in the concrete instantiation of the protocol).

$\mathcal{F}$ denotes a function. $\mathcal{F}$ with a superscript lettering defines name of the given function, e.g. $\mathcal{F}^{\mathsf{Function}}$ stands for a function named Function.

Greek letter σ denotes a signature. σ with subscript lower case lettering defines the signature scheme utilized to generate said signature, e.g. $\sigma_{\mathsf{schnorr}}$, is a signature generated with the Schnorr signature scheme. Greek letter ν denotes a nullifier. ν with a subscript capitalized letter defines a nullifier corresponding to the given structure, e.g. ν_I, is the nullifier of the identity I. Greek letter π denotes a zero-knowledge proof. π with a subscript lower case lettering defines the

name of the given zero-knowledge proof, e.g. π_{plonk} stands for a zero-knowledge proof named $\mathcal{PLONK}$. π with a superscript lower case lettering defines the name of function being proven, e.g. π^{send} stands for a zero-knowledge proof capable of satisfying the requirements of function send.

5 Abstract Protocol

The goal of this section is to establish the protocol functionality requirements and formalize a real-life protocol based on the said functionality in the later sections of the paper. Specifically, the protocol has to be capable of supporting the following functionalities:

1. privacy-preserving leader extraction procedure,
2. permission-less access to the consensus mechanism,
3. near-instant transactional finality,
4. transactional confidentiality,
5. quasi-Turing complete state transition function with native zero-knowledge proof verification capabilities.

We define an idealized abstract protocol as $\mathcal{P}_{\mathcal{E}}^{\mathsf{ideal}}$, instantiated under environment $\mathcal{E}$. We assume that $\mathcal{E}$ is responsible for handling communications with the participants of $\mathcal{P}_{\mathcal{E}}^{\mathsf{ideal}}$. $\mathcal{E}$ is assumed to follow the protocol rules, have unbounded computational resources and be 'omniscient' (i.e. $\mathcal{E}$ has access to the internal state of every participant of $\mathcal{P}_{\mathcal{E}}^{\mathsf{ideal}}$).

An interaction between protocol participant and $\mathcal{E}$ is handled via request-response communication method commonplace amongst distributed system protocol literature. We assume that the communication channel between protocol participant i and $\mathcal{E}$ is secure. To initiate a request, i is to communicate a message Req to $\mathcal{E}$, which, in its turn, would respond with a message Res. Specifically, $\mathcal{P}_{\mathcal{E}}^{\mathsf{ideal}}$ is instantiated with support for the following Req and Res message types:

1. REGISTER, a message type responsible for requesting $\mathcal{E}$ to create a new key pair (sk, pk) and register a new account.
2. SEND, a message type responsible for requesting $\mathcal{E}$ to send a defined number of the native tokens to a defined public key pk_r of the receiver.
3. CREATE, a message type responsible for requesting $\mathcal{E}$ to create an application Application. We assume that the applications can have arbitrary functionality and can interact with one another, with their state accessible to any protocol participant, with an exception of data related to a specific set functionalities such as asset transfers, voting, dividend claims, etc.

4. CALL, a message type responsible for requesting $\mathcal{E}$ to execute a call to a defined application Application.

Additionally, we assume that each request is immediately executed by $\mathcal{E}$ with the outcome becoming irreversible as soon as the execution is completed.

With that we have defined an abstract protocol capable of facilitating the functionality outlined in the beginning of the section.

6 Cryptographic Primitives

6.1 Hash Function

$\mathcal{H}()$ is a hash function which takes message m of an arbitrary size as an input and produces constant-size output x:

$$x \leftarrow \mathcal{H}(m).$$

To be considered cryptographically secure, hash functions are required to comply with the following requirements:

1. **Pre-image resistance**. The probability of a Probabilistic Polynomial-Time (PPT) Adversary $\mathcal{A}$ finding m given x (i.e. $m \leftarrow \mathcal{H}^{-1}(x)$) is negligible.

2. **Second preimage resistance**. The probability of $\mathcal{A}$ finding m_2 given m_1, where $\mathcal{H}(m_1) = \mathcal{H}(m_2)$ and $m_1 \neq m_2$, is negligible.

3. **Collision resistance**. The probability of $\mathcal{A}$ finding m_1 and m_2, where $\mathcal{H}(m_1) = \mathcal{H}(m_2)$ and $m_1 \neq m_2$, is negligible.

Specifically, $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain},\mathcal{A}}$ is instantiated with Blake2b hash function [Aum+14] as $\mathcal{H}_{\mathsf{blake2b}}$ for general-purpose computations and with Poseidon hash function [Gra+19] as $\mathcal{H}_{\mathsf{poseidon}}$ for zero-knowledge proof friendly computations.

6.2 Merkle Tree

Merkle Tree [Mer80] is a tree-like cryptographic structure which is constructed through recursive hashing of the child nodes beginning with leaf nodes and ending with a single root node. In order to prove the inclusion of a certain leaf node in a Merkle Tree $\mathsf{merkleTree}$, prover P has to provide verifier V with the Merkle Tree path opening the leaf node N, N^P, which includes the aforementioned leaf node as well as the neighboring node for every level of the tree.

Specifically, $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain},\mathcal{A}}$ is instantiated with Merkle Trees computed with $\mathcal{H}_{\mathsf{blake2b}}$, denoted with a $\mathsf{blake2b}$ subscript, e.g. $\mathsf{merkleTree}_{\mathsf{blake2b}}$, for general-purpose structures and with $\mathcal{H}_{\mathsf{poseidon}}$, denoted with a $\mathsf{poseidon}$ subscript, e.g. $\mathsf{merkleTree}_{\mathsf{poseidon}}$, for zero-knowledge proof friendly structures.

6.3 Elliptic Curve

Elliptic curves are algebraic structures constructed over finite fields. The security of elliptic curves relies on the hardness of *elliptic curve discrete logarithm problem* (ECDLP) The goal of ECDLP is to find a scalar s, given points G and H on the curve, such that $s \cdot G = H$, where $\cdot$ is the scalar multiplication in group $\mathcal{G}$.

Specifically, $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain},\mathcal{A}}$ is instantiated with JubJub [Hop+21] for general-purpose computations and with BLS12-381 [BLS02] for pairing computations.

6.4 Stealth Address Scheme

Stealth address is a one-time public key generated via a scheme based on Diffie-Hellman Key Exchange (DHKE) [DH76], proposed in [Sab13]. The scheme conceives three key pair types:

1. **secret spend key**, $ssk \leftarrow (a, b)$; where a and b are randomly generated scalars and represent a pair of secret keys.
2. **public spend key**, $psk \leftarrow (A, B)$; where $A = a \cdot G$ (G is a generator of a JubJub group $\mathcal{G}$) and $B = b \cdot G$ are compact representations of points on elliptic curve and represent a pair of public keys.
3. **view key**, $vk \leftarrow (a, B)$; where a is a randomly generated scalar and $B = b \cdot G$ is a compact representation of a point on elliptic curve and represent a secret and public key respectively.

To generate a one-time key (i.e. stealth address), the receiver R is required to share his public spend key, psk, with the sender S, after which S is to proceed with following steps:

1. Generate a random scalar r.
2. Compute a one-time public key $pk \leftarrow \mathcal{H}_{\mathsf{poseidon}}(r \cdot A) \cdot G + B$.
3. Compute $R \leftarrow r \cdot G$ and propagate (pk, R) to receiver R.

To detect a message addressed to R, R is required to scan through the incoming messages using view key, vk, to check whether $pk = \mathcal{H}_{\mathsf{poseidon}}(R \cdot a) \cdot G + B$ holds true for one of the transactions.

To compute the spend key, sk, corresponding to the one-time public key, pk, R is required to complete the following computation using his secret spend key, ssk: $sk \leftarrow \mathcal{H}_{\mathsf{poseidon}}(R \cdot a) + b$.

6.5 Encryption Scheme

$E()$ is an encryption function which takes plaintext m and encryption key k_e as an input and produces ciphertext e as an output:

$$c \leftarrow E(m, k_e).$$

To decrypt, decryption function $D()$ is utilized which takes ciphertext e and decryption key k_d as an input and produces plaintext m as an output:

$$m \leftarrow D(e, k_d).$$

For *symmetric encryption*, encryption and decryption keys are equivalent (i.e. $k_e = k_d$), whereas for *asymmetric encryption* encryption and decryption keys are different, though the two share a mathematical relationship (such as $k_e = k_d \cdot G$, where G is a generator of group $\mathcal{G}$).

Specifically, $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain},\mathcal{A}}$ is instantiated with ElGamal encryption scheme [El 85] as $E_{\mathsf{elgamal}}, D_{\mathsf{elgamal}}$ for asymmetric encryption and a permutation-based AEAD, concretely setup with Poseidon-SpongeWrap [Kho20] as $E_{\mathsf{poseidon}}, D_{\mathsf{poseidon}}$ for symmetric encryption.

6.6 Commitment Scheme

$\mathcal{C}()$ is a commitment scheme which takes value v and a random blinder b as an input and produces commitment c as an output:

$$c \leftarrow \mathcal{C}(v, b),$$

where

$$\mathcal{C}(v, b) = v \cdot G + b \cdot H,$$

where G and H are two non-identical generators with an unknown mathematical relationship (i.e. $s \cdot G =^{?} H$) for JubJub group $\mathcal{G}$.

Commitment scheme enables prover P to commit to a value privately while having a capability to reveal the value P has committed to at a later point. Formally, a secure commitment scheme must be hiding (the probability of $\mathcal{A}$ extracting value v from the commitment is negligible) and binding (the probability of $\mathcal{A}$ finding another value v and blinder b capable of opening commitment c is negligible).

Specifically, $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain},\mathcal{A}}$ is instantiated with Pedersen commitment scheme [Ped91] as $\mathcal{C}$.

6.7 Signature Scheme

$\Sigma^S()$ is a a signature scheme which takes message m and secret key sk as an input and produces signature σ as an output:

$$\sigma \leftarrow \Sigma^S(m, sk),$$

where σ can be verified as:

$$\Sigma^V(\sigma, m, pk) = \texttt{true},$$

where $pk = sk \cdot G$ (G is a generator for a group G).

Signature scheme enables prover P to authenticate a message by binding his secret key to the message in a verifiable procedure. To process a long message, a cryptographic hash function is deployed. To be considered cryptographically secure, signature schemes are to adhere to the following requirements:

1. **Unforgeability**. The probability of $\mathcal{A}$ being able to reproduce signature σ given message m is negligible.
2. **Message binding**. The probability of $\mathcal{A}$ being able to find message m_2 given message m_1 to produce $\sigma_1 = \sigma_2$ where $\sigma_1 \leftarrow \Sigma^S(m_1, sk)$ and $\sigma_2 \leftarrow \Sigma^S(m_2, sk)$ is negligible.
3. **Non-malleability**. The signature value can not be modified to another value valid for the same message.

Specifically, $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain},\mathcal{A}}$ is instantiated with Schnorr signature scheme [Sch95] as $\Sigma_{\mathsf{schnorr}}$ for general-purpose signatures and with BLS signature scheme as Σ_{bls} [BLS01] for aggregatable signatures.

6.8 Zero-Knowledge Proof Scheme

$\Pi^P()$ is a zero-knowledge proof scheme which takes public values p, private values w and prover key pk as an input and produces proof π as an output:

$$\pi \leftarrow \Pi^P(p, w, \mathsf{pk}),$$

where π can be verified as:

$$\Pi^V(p, \pi, \mathsf{vk}) = \mathtt{true},$$

where vk is a verifier key.

To be considered cryptographically secure, zero knowledge proof of knowledge protocol has to adhere to the following requirements:

1. **Completeness**. An honest prover P succeeds in convincing the verifier V of the statement.
2. **Soundness**. The probability of $\mathcal{A}$ proving an invalid statement to verifier V is negligible.
3. **Zero-knowledgeness**. The proof reveals no information other than the fact that the statement is true.

Specifically, $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain},\mathcal{A}}$ is instantiated with $\mathcal{P}\mathtt{lonk}\mathcal{K}$ [GWC19] zero-knowledge proof scheme as Π_{plonk}.

Additionally, $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain},\mathcal{A}}$ is instantiated with modified Schnorr proof scheme [Sch95] as Π_{schnorr} for discrete logarithm relationship proofs.

7 Building Blocks

7.1 Segregated Byzantine Agreement

SBA is the consensus mechanism utilized to secure the DZCK protocol.
SBA is a permission-less Proof-of-Stake-based mechanism with statistical finality guarantees. The mechanism segregates the consensus participants into two distinct roles:

1. **Generator.** Generators are responsible for proposing blocks. In other words, Generators perform a similar role to *leaders* from classic distributed systems literature. Generators are extracted through the privacy-preserving leader extraction procedure called Proof-of-Blind Bid.

2. **Provisioner** Provisioners are responsible for validating and finalizing the proposed blocks. In other words, Provisioners perform a similar role to *replicas* from classic distributed systems literature. Provisioners are extracted into the committees through the committee extraction procedure called deterministic sortition.

The mechanism is secure under the honest majority of money assumption, meaning that for n DZCK being the cumulative amount staked eligible to participate in the consensus, the ratio of h DZCK under control of honest participants must hold as $h > n - f$, where f is the ratio of DZCK under the control of Byzantine participants and $h \geq 2f$. Specifically, we formalize the honest majority of money assumption for the DZCK protocol as:

$$\frac{\mathsf{TotalActiveStake(Generators)} \cup \mathsf{TotalByzantineStake(Generators)}}{\mathsf{TotalActiveStake(Generators)} \cup \mathsf{TotalHonestStake(Generators)}} > \frac{1}{3},$$

and

$$\frac{\mathsf{TotalActiveStake(Provisioners)} \cup \mathsf{TotalByzantineStake(Provisioners)}}{\mathsf{TotalActiveStake(Provisioners)} \cup \mathsf{TotalHonestStake(Provisioners)}} > \frac{1}{3}.$$

Additionally, we assume that there exists a probabilistic polynomial-time (PPT) Adversary $\mathcal{A}$ capable of corrupting a consensus participants with a maximum control of f DZCK . $\mathcal{A}$ is assumed to be a mildly adaptive Adversary, meaning that corruption takes place $\Delta_{\mathsf{corrupt}}$ after $\mathcal{A}$ has selected a participant to corrupt .Concretely, $\Delta_{\mathsf{corrupt}} > \mathsf{epoch}$.

The network conditions are assumed to be synchronous, meaning that there exists a publicly known Δ_{delay} defining the maximum delay by which the message propagation can be delayed to the honest participants of the consensus. In other words, after the initial propagation time of $t_{\mathsf{initPropagation}}$, the message will be delivered to every honest consensus participant no later than $t_{\mathsf{initPropagation}} + \Delta_{\mathsf{delay}}$. We additionally assume that every honest participant will receive the messages in the order that they were propagated.

The consensus is divided into *epochs*, *rounds* and *steps*. An epoch is defined as a set of rounds, concretely as $\mathsf{epoch} \geq c \times \mathsf{round}$, where c is denoted as $(f)^c \leq 2^{-100}$, for the duration of which the Generators and Provisioner sets remain unchanged and the epoch seed $\mathsf{seed}_{\mathsf{epoch}}$ corresponding to the epoch is utilized. A round is defined as a synonym to block height in the context of the consensus, with every round corresponding to a unique block in Chain. A step denotes a consensus step. Specifically, each consensus round is comprised of iterations (i.e. loops) of 4 step.

The consensus mechanism is comprised of three phases:

1. *Generation phase*, utilizing Proof-of-Blind Bid to extract a leader who is to forge and propagate a candidate block to the consensus participants.
2. *Reduction phase*, based on [TA84], the two-step phase is responsible for the agreement on a single candidate block to be finalized.
3. *Agreement phase*, an asynchronous phase running in parallel with the two previous phases and responsible for the finalization of a candidate block. Specifically, an agreement vote is initiated after the successful termination of the Reduction phase.

Definition. A consensus protocol has statistical finality when the probability of a fork during a single execution round is negligible.

Proof. In case of SBA, the fork can be produced by "double-voting" during the three consequent steps of the execution round, specifically two Reduction steps and an Agreement step). A "double-vote" occurs when a node votes for two separate candidate blocks in the same voting step. Taking the node assumptions defined above into account, a "double-vote" can only be produced by a Byzantine node, meaning that in order to produce a fork, an Adversary has to receive the control of the supermajority of the cumulative committee stake size in three successive consensus steps. The probability of the aforementioned event happening can be defined below.

Failure Rate

The failure rate is the probability of the safety guarantees of the protocol step (exluding Generation phase) being breached. In particular, the failure rate indicates the probability of an Adversary $\mathcal{A}$ obtaining a supermajority in a committee. The probability function is outlined in a formula below, where $N \leftarrow |\mathsf{committee}_{\mathsf{round,step}}|$ is the committee size, $\tau \leftarrow \mathsf{vote}_{\mathsf{threshold}}$ is the threshold of votes in a committee required to proceed to the next step and h is the ratio of the honest participants:

$$\sum_{k=\texttt{floor}(\tau\cdot N+1)}^{N} \binom{N}{k} \cdot (1-h)^k \cdot h^{N-k} = \sum_{k=\texttt{ceil}(\tau\cdot N)}^{N} \frac{N!\cdot(1-h)^k\cdot h^{N-k}}{k!\cdot(N-k)!} \leq f_{\mathsf{failure}}^{\mathsf{step}}$$

Figure 1: The formula calculating the failure rate per step f_s^{step}.

The probability of an Adversary $\mathcal{A}$ successfully creating a fork is equal to the probability of an Adversary $\mathcal{A}$ obtaining a supermajority in the two successive Reduction steps $(([Pr]_{\mathsf{Reduction}}^{\mathsf{failure}})^2 \leq (f_{\mathsf{failure}}^{\mathsf{step}})^2)$ and a Agreement step $(([Pr]_{\mathsf{Agreement}}^{\mathsf{failure}} \leq f_{\mathsf{failure}}^{\mathsf{step}}))$ or $([Pr]_{\mathsf{Reduction}}^{\mathsf{failure}})^2 \times [Pr]_{\mathsf{Agreement}}^{\mathsf{failure}} = (f_{\mathsf{failure}}^{\mathsf{step}})^3 \leq f_{\mathsf{failure}}^{\mathsf{round}}$.

Liveliness Rate

The liveliness rate indicates the probability of an honest majority being obtained in a committee. The probability function is outlined in a formula below:

$$1 - \sum_{k=1}^{\texttt{floor}(\tau\cdot N)} \binom{N}{k} \cdot h^k \cdot (1-h)^{N-k} = 1 - \sum_{k=1}^{\texttt{floor}(\tau\cdot N)} \frac{N!\cdot h^k\cdot(1-h)^{N-k}}{k!\cdot(N-k)!} \leq f_{\mathsf{liveliness}}^{\mathsf{step}}$$

Figure 2: The formula calculating the failure rate per step f_l^{step}.

The probability of a successful consensus round termination is equal to the probability of a supermajority obtained in the successive Generation $([Pr]_{\mathsf{Generation}}^{\mathsf{liveliness}} \geq h)$, two Reduction $(([Pr]_{\mathsf{Reduction}}^{\mathsf{liveliness}})^2 \geq (f_{\mathsf{liveliness}}^{\mathsf{step}})^2)$ and Agreement $([Pr]_{\mathsf{Agreement}}^{\mathsf{liveliness}} \geq f_{\mathsf{liveliness}}^{\mathsf{step}})$ steps or $[Pr]_{\mathsf{Generation}}^{\mathsf{liveliness}} \times ([Pr]_{\mathsf{Reduction}}^{\mathsf{liveliness}})^2 \times [Pr]_{\mathsf{Agreement}}^{\mathsf{liveliness}} = h \times (f_{\mathsf{liveliness}}^{\mathsf{step}})^3 \leq f_{\mathsf{liveliness}}^{\mathsf{round}}$.

7.1.1 Proof-of-Blind Bid

Proof-of-Blind Bid (PoBB from hereon) is a novel privacy-preserving leader extraction procedure. PoBB is utilized in the Generation phase to probabilistically extract the leader for the round who is responsible for forging the candidate block proposal for the respective round.

PoBB utilizes a Merkle Tree structure comprised of the bids. The bids contain the obfuscated amounts of DZCK being staked along with accompanying data, which permits the prospective round leaders to prove the validity of their bids in zero-knowledge without revealing their identities or amounts being staked. Specifically, we define the Merkle Tree as:

$$\mathsf{bidTree}_{\mathsf{poseidon}} \leftarrow \mathsf{MerkleTree}[\mathsf{bid}],$$

where bid is:

$$\mathsf{bid} \leftarrow (c, \mathsf{secretHash}, \mathsf{nonce}, \mathsf{secret}^{enc}, R, pk, h_{\mathsf{eligibility}}, h_{\mathsf{expiry}}).$$

c is defined as the commitment to the amount of DZCK being staked, $\mathsf{secretHash}$ is the hash of secret integer secret utilized to compute the extraction score, nonce is utilized for the encryption of secret^{enc}, secret^{enc} is the secret encrypted to the derivative of pk, (R, pk) is the stealth address and $(h_{eligibility}, h_{expiration})$ defines the height after which the bid becomes eligible to participate in the consensus and during which the bid expires respectively.

Assuming that for round round the consensus participant i has an eligible bid bid_i (i.e. $h_{\mathsf{eligibility}} \leq \mathsf{round} < h_{\mathsf{expiration}}$) for which i knows the commitment opening (v, b) (i.e. $\mathsf{bid}_i.c = \mathcal{C}(v, b)$) and the secret corresponding to $\mathsf{secretHash}$ (i.e. $\mathsf{secretHash} = \mathcal{H}_{\mathsf{poseidon}}(\mathsf{secret})$), i is capable of computing the score score determining whether or not i is the leader of round round, step step ($\mathsf{leader}_{\mathsf{round,step}}$). Specifically, we compute the score in the following steps:

1. $y \leftarrow \mathcal{H}_{\mathsf{poseidon}}(\mathsf{secret}||\mathcal{H}_{\mathsf{poseidon}}(\mathsf{bid}_i)||\mathsf{round}||\mathsf{step})$
2. $x_1 \leftarrow y/2^{128}$
3. $y' \leftarrow y\%2^{128}$
4. $\mathsf{score}_{\mathsf{round,step}} \leftarrow (v \times 2^{128})/y'$, where $z \leftarrow (v \times 2^{128})$ if $y' = 0$ and v is the commitment opening corresponding to $\mathsf{bid}_i.c$

If the $\mathsf{score}_{\mathsf{round,step}} \geq \mathsf{score}_{\mathsf{threshold}}$, then i is probabilistically presumed to be the leader for round round and step step. We assume that there exists a function that is capable of dynamically computing the score threshold for every epoch. The details of the concrete instantiation of the aforementioned function are beyond the scope of this paper. The function can be defined as:

$$\mathsf{score}_{\mathsf{threshold}} \leftarrow \mathsf{Threshold}(\mathsf{Generators}, \lambda),$$

where λ defines the average number of leaders to be extracted per round-step combination.

To be capable of verifying the validity of the $\mathsf{score}_{\mathsf{round,step}}$ generated by i, the consensus participant j is required to receive the following tuple:

$$(\mathsf{score}^{i}_{\mathsf{round,step}}, \mathsf{seed}^{i}_{\mathsf{round,step}}, \pi^{\mathsf{score}}_{\mathsf{plonk}}),$$

where $\mathsf{seed}_{\mathsf{round,step}}$ is the seed for $\mathsf{round}, \mathsf{step}$ round-step combination and $\pi^{\mathsf{score}}_{\mathsf{plonk}}$ is the zero-knowledge proof attesting to the validity of the score generation. To pass the verification, $\pi^{\mathsf{score}}_{\mathsf{plonk}}$ is required to satisfy the following steps:

1. $\mathcal{O}(\mathsf{bidTree}^{R}, \mathsf{bid}^{P}_{i})$,
2. $r \geq \mathsf{bid}_i.h_{\mathsf{eligibility}}$,

3. $r < \mathsf{bid}_i.h_{\mathsf{expiration}}$,
4. $\mathsf{bid}_i.c = \mathcal{C}(v, b)$,
5. $0 \leq v < 2^{64}$,
6. $\mathsf{bid}_i.\mathsf{secretHash} = \mathcal{H}_{\mathsf{poseidon}}(\mathsf{secret})$,
7. $seed_{r,s} = \mathcal{H}_{\mathsf{poseidon}}(\mathsf{secret}||\mathsf{state}_{\mathsf{global}}.\mathsf{seed}_e||\mathsf{round}||\mathsf{step})$,
8. $y = \mathcal{H}_{\mathsf{poseidon}}(\mathsf{secret}||\mathcal{H}_{\mathsf{poseidon}}(\mathsf{bid}_i)||\mathsf{round}||\mathsf{step})$,
9. $y = x_1 \times 2^{128} + y'$,
10. $((x_1 < \lfloor|\mathbb{F}_p|/2^{128}\rfloor) \wedge (y^{prime} < 2^{128})) \vee ((x_1 = \lfloor|\mathbb{F}_p|/2^{128}\rfloor) \wedge (y' < |\mathbb{F}_p| \bmod 2^{128}))$, where $\mathbb{F}_p$ defines the prime field,
11. $x_2 < y'$,
12. $\mathsf{score} < 2^{120}$,
13. $\mathsf{score} \times y' + r_2 = v^{128}$.

7.1.2 Generation Phase

Generation phase is utilized in SBA to forge a candidate block for every iteration of the consensus. If, after successful termination of the PoBB execution, a Generator has obtained a score greater or equal to the predefined threshold, then the said Generator can proceed to forge and propagate the block. Specifically, we define the Generation function as:

Function $\mathcal{F}^{\mathsf{Generation}}$

$\mathcal{F}^{\mathsf{Generation}}(\mathsf{bid}_i, v, b, \mathsf{secret})$:

1. $(\mathsf{score}, \mathsf{seed}_{\mathsf{round},\mathsf{step}}, \pi^{\mathsf{score}}_{\mathsf{plonk}}) \leftarrow \mathcal{F}^{\mathsf{PoBB}}(\mathsf{bid}_i, v, b, \mathsf{secret})$,
2. If $\mathsf{score} \geq \mathsf{score}_{\mathsf{threshold}}$,
3. $\mathsf{candidateBlock} \leftarrow \mathsf{ForgeCandidateBlock}(\mathsf{score}, \mathsf{seed}_{\mathsf{round},\mathsf{step}}, \pi^{\mathsf{score}}_{\mathsf{plonk}})$,
4. $\mathcal{F}^{\mathsf{Propagate}}(\mathsf{score}, \pi^{\mathsf{score}}_{\mathsf{plonk}}, \mathsf{candidateBlock})$.

where ForgeCandidateBlock() is function responsible for the generation of a candidateBlock.

7.1.3 Reduction Phase

The Reduction phase utilized in SBA to reach agreement on a single candidate block to be finalized, is based on [TA84], which reduces the multivariable inputs to a single variable output before proceeding to Binary Agreement. However, unlike BBA★ [Mic17], the Turpin and Coan algorithm is not utilized as a reduction function for the Binary Agreement protocol. Reduction phase is a two-step phase defined as:

Function $\mathcal{F}^{\mathsf{Reduction}}$

$\mathcal{F}^{\mathsf{Reduction}}(\mathsf{stake}_i, sk_{\mathsf{BLS}}, \mathsf{candidateBlock}_i)$:

1. Start timer for step step, i.e. $\mathsf{timer}_{\mathsf{round,step}} \leftarrow \mathsf{StartTimer}(\mathsf{REDUCTION})$
2. If $\mathsf{stake}_i.pk_{\mathsf{BLS}} \in \mathsf{Provisioners}$,
3. If $\mathsf{stake}_i.h_{\mathsf{eligibility}} \leq \mathsf{round} < \mathsf{stake}_i.h_{\mathsf{expiration}}$.
4. If $\mathsf{stake}_i.pk_{\mathsf{BLS}} \in \mathsf{committee}_{\mathsf{round,step}}$, where $\mathsf{committee}_{\mathsf{round,step}} \leftarrow \mathsf{DeterministicSortition}()$,
5. $\mathcal{F}^{\mathsf{Propagate}}(\sigma_{\mathsf{BLS}}, pk_{\mathsf{BLS}}, \mathsf{round}, \mathsf{step}, \mathcal{H}_{\mathsf{blake2b}}(\mathsf{candidateBlock}_i))$, where $\sigma_{\mathsf{BLS}} \leftarrow \Sigma^S_{\mathsf{BLS}}(\mathcal{H}_{\mathsf{blake2b}}(\mathsf{candidateBlock}_i)||\mathsf{round}||\mathsf{step}, sk_{\mathsf{BLS}})$,
6. If $\mathsf{vote}_{\mathsf{threshold}}$ messages for a single any candidate block $\mathsf{candidateBlock}$ are received before the expiration of $\mathsf{timer}_{\mathsf{round,step}}$ set $\mathsf{candidateBlock}_i \leftarrow \mathsf{candidateBlock}$, otherwise set $\mathsf{candidateBlock}_i \leftarrow \emptyset$,
7. Start timer for step $\mathsf{step} + 1$, i.e. $\mathsf{timer}_{\mathsf{round,step+1}} \leftarrow \mathsf{StartTimer}(\mathsf{REDUCTION})$,
8. If $\mathsf{stake}_i.pk_{\mathsf{BLS}} \in \mathsf{committee}_{\mathsf{round,step}} + 1$, where $\mathsf{committee}_{\mathsf{round,step}} + 1 \leftarrow \mathsf{DeterministicSortition}()$,
9. $\mathcal{F}^{\mathsf{Propagate}}(\sigma_{\mathsf{BLS}}, pk_{\mathsf{BLS}}, \mathsf{round}, \mathsf{step} + 1, \mathcal{H}_{\mathsf{blake2b}}(\mathsf{candidateBlock}_i))$,
10. If $\mathsf{vote}_{\mathsf{threshold}}$ messages for any single candidate block $\mathsf{candidateBlock}$ are received before the expiration of $\mathsf{timer}_{\mathsf{round,step+1}}$ set return $\mathsf{candidateBlock}$, otherwise return $\mathsf{candidateBlock} \leftarrow \emptyset$.

where $\mathsf{StartTimer}()$ is a function responsible for the instantiation of a timer for a corresponding step, $\mathsf{committee}_{\mathsf{round,step}}$ is a set comprised of Provisioner public keys pk_{BLS} who have been extracted into a committee for round and step, $\mathsf{DeterministicSortition}()$ is a function responsible for the extraction of committee members for round and step and $\mathsf{vote}_{\mathsf{threshold}}$ is a threshold of committee votes required to reach a quorum for round and step.

7.1.4 Agreement Phase

Agreement is an asynchronous phase running in parallel with the main consensus loop. Successful termination of the phase indicates that a candidate block for round round has been finalized. Agreement phase is defined as:

Function $\mathcal{F}^{\mathsf{Agreement}}$

$\mathcal{F}^{\mathsf{Agreement}}()$:

1. If $\mathsf{vote}_{\mathsf{threshold}}$ messages for any single candidate block candidateBlock are received for step step (if step mod $4 = 0$), then return candidateBlock

7.1.5 Main Consensus Loop

The main loop for SBA is defined as: where StartThread() is a function respon-

Algorithm 1 Main Consensus Loop

chain = Chain
round = 1
while do
    step = 1
    thread = StartThread($\mathcal{F}^{\mathsf{Agreement}}$)
    while Running(thread) **do**
        $\mathsf{timer}_{\mathsf{round,step}}$ = StartTimer(GENERATION)
        $\mathcal{F}^{\mathsf{Generation}}(\mathsf{bid_i}, v, b, \mathsf{secret})$
        if a valid candidateBlock is collected before the expiration of $\mathsf{timer}_{\mathsf{round,step}}$ **then**
            $\mathsf{candidateBlock}_i$ = candidateBlock
        else
            $\mathsf{candidateBlock}_i = \emptyset$
        end if
        step = step + 1
        candidateBlock $\leftarrow \mathcal{F}^{\mathsf{Reduction}}(\mathsf{stake_i}, sk_{\mathsf{BLS}}, \mathsf{candidateBlock}_i)$
        step = step + 2
        if candidateBlock $\neq \emptyset$ **then**
            $\sigma_{\mathsf{BLS}} = \Sigma^{S}_{\mathsf{BLS}}(\mathcal{H}_{\mathsf{blake2b}}(\mathsf{candidateBlock})||\mathsf{round}||step, sk_{\mathsf{BLS}})$
            $\mathcal{F}^{\mathsf{Propagate}}(\sigma_{\mathsf{BLS}}, \mathsf{round}, \mathsf{step}, \mathcal{H}_{\mathsf{blake2b}}(\mathsf{candidateBlock}))$
        end if
        step = step + 1
    end while
    chain.append(candidateBlock)
    round = round + 1
end while

sible for starting a new concurrent execution thread and Running() is a function responsible of notifying whether the concurrent thread is being executed or has terminated.

7.2 Phoenix

Phoenix is a UTxO-based transaction model which expands on the model proposed in [Hop+21] to enable the users to spend non-obfuscated outputs confidentially, a requirement for quasi-Turing complete systems where the final cost of the execution is unknown until the termination of said execution. Phoenix avoids the possibility of transaction deanonymization arising from the limited anonymity sets [Kum+17; Mös+18] of ring-signature-based models and from the miner behaviour as well as the relationships between transparent and shielded transactions [Que17; Kap+18; Zha+20; BF19] in [Hop+21] (**NOTE:** Zcash has enabled shielded block reward payments recently, decreasing the attack surface on the anonymity set). The theoretical anonymity set size for Phoenix is equivalent to the number of outputs created since the genesis block. In other words, the anonymity set grows as more transactions are added to the blockchain.

On an abstract level, a Phoenix transaction is comprised of a set of inputs $\mathsf{Inputs} \leftarrow \mathsf{Set}[\mathsf{Input}]$, a set of outputs $\mathsf{Outputs} \leftarrow \mathsf{Set}[\mathsf{Output}]$ and zero-knowledge proof attesting to the correctness of said transaction, $\pi^{\mathsf{spend}}_{\mathsf{plonk}}$. The input is defined as:

$$\mathsf{Input} \leftarrow (\nu, \mathsf{anchor}),$$

where ν is a unique identifier of an input called *nullifier* utilized to prevent double-spending of the outputs and anchor is the root of the Merkle Tree comprised of *notes* with which the spending proof for the said input was computed. $1 \leq |\mathsf{Inputs}| \leq 4$ is required to hold. We define the output as:

$$\mathsf{Output} \leftarrow \mathsf{note},$$

where note is a structure defined as:

$$\mathsf{note} \leftarrow (\mathsf{type}, c, \mathsf{data}, R, pk).$$

type stands for the note type, c is the commitment to value of the note, data is a type-dependent entry and R, pk represent the stealth address identifying said note. We define two note types as *transparent* and *obfuscated*. If the note is transparent ($\mathsf{note}_{\mathsf{transparent}}$), then $\mathsf{note}_{\mathsf{transparent}}.\mathsf{type} = 0$ and $\mathsf{note}_{\mathsf{transparent}}.\mathsf{data} = v$, where v is the value of the commitment $\mathsf{note}_{\mathsf{transparent}}.c \leftarrow \mathcal{C}(v, b)$ and $b = 0$. Otherwise (i.e. $\mathsf{note}_{\mathsf{obfuscated}}$), $\mathsf{note}_{\mathsf{obfuscated}}.\mathsf{type} = 1$ and $\mathsf{note}_{\mathsf{obfuscated}}.\mathsf{data} = (d^{enc})$, where d^{enc} are the encrypted value and blinder of the commitment $\mathsf{note}_{\mathsf{obfuscated}}.c$ to the derivative of $\mathsf{note}_{\mathsf{obfuscated}}.pk$ k_{enc} utilized for encryption, i.e. $d^{enc} \leftarrow E_{\mathsf{poseidon}}(v||b, k_{enc})$. $0 \leq |\mathsf{Outputs}| \leq 2$ is required to hold.

We define $\mathsf{noteTree}_{\mathsf{poseidon}}$ as a Merkle Tree structure utilized to store output notes as leaves. Specifically, noteTree is formalized as follows:

$$\mathsf{noteTree}_{\mathsf{poseidon}} \leftarrow \mathsf{MerkleTree}(\mathsf{note}, 34).$$

Additionally, nullifierSet is defined as a set responsible for storing nullifiers corresponding to the spent outputs, i.e. $\mathsf{nullifierSet} \leftarrow \mathsf{Set}[\nu]$.

The zero-knowledge proof attesting to the correctness of the transaction $\pi^{\mathsf{spend}}_{\mathsf{plonk}}$ is required to satisfy the following conditions:

1. For every member of Inputs, a Merkle Tree path corresponding to the respective note note^P is required to be a valid opening to the corresponding anchor, i.e. $\forall \mathsf{Input} \in \mathsf{Inputs}[\mathcal{O}(\mathsf{Input.anchor}, \mathsf{note}^P) = \texttt{true}]$.

2. For every member of Inputs, a valid Schnorr proof π_{schnorr} of the respective note is required to be provided, i.e. $\forall \mathsf{Input} \in \mathsf{Inputs}[\Sigma^V_{\mathsf{schnorr}}(pk', \pi_{\mathsf{schnorr}}, \mathsf{note}.pk) = \texttt{true}]$.

3. For every member of Inputs, a nullifier ν is required to correspond to the hash of the respective nullifier-deriving public key pk' and note position note^{pos} in the noteTree, i.e. $\forall \mathsf{Input} \in \mathsf{Inputs}[\nu = \mathcal{H}_{\mathsf{poseidon}}(pk' || \mathsf{note}^{\mathsf{pos}})]$.

4. For every member of Inputs, a commitment opening tuple (v, b) is required to be a valid opening of the corresponding commitment c of a respective note, i.e. $\forall \mathsf{Input} \in \mathsf{Inputs}[c = \mathcal{C}(v, b)]$.

5. Verify that the sum of v corresponding to the opening of the commitment for the corresponding note of every member of Inputs is equal to v_{in}, i.e. $v_{in} = \sum v$.

6. For every member of Input, commitment opening v corresponding commitment c of the respective note is required to be within the valid range, i.e. $\forall \mathsf{Input} \in \mathsf{Inputs}[0 \leq v < 2^{64}]$.

7. For every member of Outputs, commitment opening tuple (v, c) is required to be a valid opening of the corresponding commitment c of the respective note, i.e. $\forall \mathsf{Output} \in \mathsf{Outputs}[\mathsf{note}.c = \mathcal{C}(v, b)]$.

8. For every member of Outputs, commitment opening v corresponding commitment c of the respective note is required to be within the valid range, i.e. $\forall \mathsf{Output} \in \mathsf{Outputs}[0 \leq v < 2^{64}]$.

9. Verify that the sum of v corresponding to the opening of the commitment for the corresponding note of every member of Outputs is equal to v_{out}, i.e. $v_{out} = \sum v$.

10. The result of v_{in} being substracted from v_{out} is required to be 0, i.e. $v_{out} - v_{in} = 0$

7.3 Zedger

Zedger is a hybrid privacy-preserving transaction model created to comply with the regulatory requirements of security tokenization and lifecycle management. In Zedger, an expanded account model is utilized to track user balances and a Phoenix-like UTxO model is utilized to facilitate user-to-user transfers.

Below, we formalize the requirements for a privacy-preserving transaction model capable of complying with the requirements of security tokenization:

1. There can only exist 1 account per user.
2. Only whitelisted users are permitted to transact.
3. The receiver is required to explicitly approve incoming transactions.
4. The transferred asset amount should be accounted for in the sender's balance until the receiver has explicitly approved the transaction.
5. The account is required to log every balance change since the creation of the account.
6. The account is required to log transactional, voting and dividend-eligible balances separately.
7. The asset operator-appointed party is required to be capable of reconstructing the capitalization table for any snapshot point.

Since pure UTxO-based models are incapable of satisfying the requirements outlined above and neither do the previous attempts at creating a privacy-preserving account model [Fau+18; Gua+20], we have formalized Zedger.

Zedger introduces a novel cryptographic structure called Sparse Merkle-Segment Trie (SMST from hereon), which forms the basis for the account storage. SMST combines two structures known as a Sparse Merkle Tree [DPP16] and a Segment Tree [BW80] into a single structure. Specifically, SMST utilizes the Sparse Merkle Tree cryptographic accumulator property and the Segment Tree's capability of storing information about the intervals. Each node in the SMST is defined as:

$$\mathsf{node} \leftarrow (\mathsf{balance}_{\mathsf{max}}, \mathsf{balance}_{\mathsf{transactional}}, \mathsf{balance}_{\mathsf{voting}}, \mathsf{balance}_{\mathsf{dividend}},$$
$$\mathcal{H}_{\mathsf{poseidon}}(\mathsf{childNode}_1), \mathcal{H}_{\mathsf{poseidon}}(\mathsf{childNode}_2)),$$

where $\mathsf{balance}_{\mathsf{max}}$ is maximum balance logged in the segment, $\mathsf{balance}_{transactional}$ is the balance permitted to be transaction with, $\mathsf{balance}_{\mathsf{voting}}$ is the balance permitted to be voted with, $\mathsf{balance}_{\mathsf{dividend}}$ is the dividend-eligible balance, $\mathcal{H}_{\mathsf{poseidon}}(\mathsf{childNode}_1)$ is the hash of the first child node and $\mathcal{H}_{\mathsf{poseidon}}(\mathsf{childNode}_2)$ is the hash of the second child node. If node is a leaf, then $\mathsf{childNode}_1, \mathsf{childNode}_2 \in \emptyset$.

Additionaly, Zedger incorporates some features from Phoenix to facilitate user-to-user transfers. We define the following structures as part of Zedger:

1. $\mathsf{whitelistTree}_{\mathsf{poseidon}}$ is a Merkle Tree structure comprised of identities $I \leftarrow (R, pk)$, i.e. $\mathsf{whitelistTree}_{\mathsf{poseidon}} \leftarrow \mathsf{MerkleTree}[I]$.

2. $\mathsf{memorySlotTree}_{\mathsf{poseidon}}$ is a Merkle Tree structure comprised of memory slots M, which represent the roots for each account SMST along with accompanying data, i.e. $\mathsf{memorySlotTree}_{\mathsf{poseidon}} \leftarrow \mathsf{MerkleTree}[M]$.

3. $\mathsf{coinTree}_{\mathsf{poseidon}}$ is a Merkle Tree structure comprised of coins C *minted* equivalent to the value being sent when transferring an assets from user-to-user, i.e. $\mathsf{coinTree}_{\mathsf{poseidon}} \leftarrow \mathsf{MerkleTree}[C]$

4. $\mathsf{memorySlotNullifierSet}$ is a set of unique nullifiers ν_M corresponding to consumed memory slots, i.e. $\mathsf{memorySlotNullifierSet} \leftarrow \mathsf{Set}[\nu_M]$

5. $\mathsf{coinNullifierSet}$ is a set of unique nullifiers ν_C corresponding to consumed coins, i.e. $\mathsf{memorySlotNullifierSet} \leftarrow \mathsf{Set}[\nu_C]$.

Zedger supports the following functions:

1. CREATE, enabling a whitelisted user to register a new account.

2. SEND, enabling a whitelisted user with an account to initiate a transfer of an asset to another user.

3. ACCEPT, enabling a whitelisted receiver to accept the transfer prior to its expiration.

4. SETTLE, enabling a whitelisted sender to update his balance in accordance with the acceptance time of the transfer (if the transfer has been accepted).

5. CLAIM, enabling a whitelisted sender to claim the transfer previously initiated if not accepted by the sender prior to its expiration.

6. VOTE, enabling an eligible whitelisted user to vote on a predefined set of subjects.

7. PUSH_DIVIDEND, enabling a contract operator to push a dividend to an eligible whitelisted user.

7.4 Rusk VM

We define Rusk VM, a WebAssembly-based virtual machine deployed in the DZCK protocol. The concrete protocol state transition function is instantiated with Rusk VM. Every VM function is priced, meaning that there exists an internal abstract accounting currency called *gas* which is utilized to assign a cost to every VM function. As a result, Rusk VM is a quasi-Turing complete virtual machine, meaning that each state transition is computationally bounded to the maximum allocated gas. The computational bound represents a workaround to the *halting problem* [Tur37], which proves the impossibility of guaranteeing the termination of an execution in a Turing complete machine.

Rusk VM inherits the majority of the native WASM OPCODEs (OPerational CODEs) alongside the addition of a variety of host functions encompassing cryptographic functionality, such as hashing, elliptic curve scalar multiplication and point addition, signature verification and zero-knowledge proof verification functions as well as the functionality exposing the protocol state, such as contract storage read and write, current block height, current timestamp, etc. Specifically, we define the state transition function as:

$$\mathcal{S}(\mathsf{data}, \mathsf{state}_{\mathsf{global}}) \rightarrow (\mathsf{state}'_{\mathsf{global}}, \mathsf{bool}),$$

where data is the state transition-initiating data included in the transaction (i.e. $\mathsf{Tx.stateTransitionInit}$) and $\mathsf{state}_{\mathsf{global}}$ is the global state of the protocol at the time of the beginning of the state transition exeuction:

$$\mathsf{state}_{\mathsf{global}} \leftarrow (\mathsf{stateTree}, \mathsf{timestamp}, \mathsf{height}, \mathsf{seed}, \mathsf{blockGasLimit}, \mathsf{blockGasUsed},$$

$$\mathsf{contractsCreated}, \mathsf{Tx.type}, \mathsf{Tx.gasLimit}),$$

where:

1. $\mathsf{stateTree}$ is a Merkle Tree comprised of contract accounts, i.e. $\mathsf{stateTree} \leftarrow \mathsf{MerkleTree}[\mathsf{account}, 32]$, where $\mathsf{account} \leftarrow (\mathsf{codeHash}, \mathsf{contractState})$, $\mathsf{codeHash}$ is the hash of the contract code and $\mathsf{contractState}$ is a Merkle Tree comprised of the mapped contract storage values.
2. $\mathsf{timestamp}$ is the *UNIX* timestamp of the current block.
3. height is the height of the current block.
4. seed is the seed of the current epoch.
5. $\mathsf{blockGasLimit}$ is the gas limit of the current block.
6. $\mathsf{blockGasUsed}$ is the amount gas used in the current block.
7. $\mathsf{contractsCreated}$ is the amount of contracts created in the current block.
8. $\mathsf{Tx.type}$ is the transaction type of the state transition-initiating transaction.
9. $\mathsf{Tx.gasLimit}$ is the gas limit defined in the state transition-initiating transaction.

Additionally, Rusk VM supports for efficient creation of zero-knowledge proof-friendly Merkle Tree structures.

7.5 Genesis Contracts

The DZCK protocol is instantiated with a set of 4 native contracts called *Genesis Contracts*. The Genesis Contracts are deployed in the genesis block, meaning that every protocol participant running the DZCK protocol has the contracts natively deployed. We formally define the functionality of the Genesis Contracts in an iterative order in the following 4 subsections of the paper.

7.5.1 DZCK Contract

DZCK Contract $\mathsf{Contract}^{\mathsf{DZCK}}$ forms the backbone of the DZCK protocol. The contract is responsible for the accounting of the native protocol asset called DZCK . DZCK is utilized as a Sybil-resistant participation token in SBA consensus mechanism, as well as the medium for subsidizing computation costs of the consensus participants (i.e. medium for *transaction fee* payment). $\mathsf{Contract}^{\mathsf{DZCK}}$ utilizes Phoenix as the underlying transaction model. The $\mathsf{Contract}^{\mathsf{DZCK}}.\mathcal{F}^{\mathsf{Execute}}$ function acts as the entry point to state transition initiation for non-Coinbase transactions. Specifically, $\mathsf{Contract}^{\mathsf{DZCK}}.\mathcal{F}^{\mathsf{Execute}}$ function takes the following input:

$$\mathsf{Inputs}, \mathsf{Crossover?}, \mathsf{Outputs}, \mathsf{Fee}, \pi_{\mathsf{plonk}}^{\mathsf{execute}}, \mathsf{calldata},$$

where Inputs and Outputs are previously defined in the Phoenix section, Crossover is an optional entry represting a unique note acting as bridge for DZCK between the transactional and generalized compute layers, Fee is the fee allocated as a reimbursment for the transaction computation costs, $\pi_{\mathsf{plonk}}^{\mathsf{execute}}$ is a zero-knowledge proof attesting to the validity of the transaction and calldata is the data attached to the transaction enabling it to initiate a contract call. We define Crossover as:

$$\mathsf{Crossover} \leftarrow (c, d^{enc}),$$

where c is the commitment to the value v of DZCK being bridged and d^{enc} are the encrypted openings of commitment c to the derivative of $\mathsf{Fee}.pk$. Fee is defined as:

$$\mathsf{Fee} \leftarrow (\mathsf{gasPrice}, \mathsf{gasLimit}, R, pk),$$

where gasPrice is the price per gas in DZCK , gasLimit is the maximum allowance of gas for the execution of said function call and (R, pk) is the stealth address to which a refund will be issued if the not all allocated gas is consumed. If, for some reason, a Crossover is not consumed during the exection of the call, then then an additional refund note is issued to (R, pk).

$\mathsf{Contract}^{\mathsf{DZCK}}.\mathcal{F}^{\mathsf{Execute}}$ function requires the following steps to be amended to the zero-knowledge proof conditions outline in the Pheonix section:

1. $\mathsf{Crossover}.c = \mathcal{C}(c, v)$,
2. $\mathsf{fee} = \mathsf{gasPrice} \times \mathsf{gasLimit}$,

3. Step 10 to be substituted with $v_{in} - v - v_{out} - \mathsf{fee}$.

In total, $\mathsf{Contract}^{\mathsf{DZCK}}$ is comprised of the following non-spurious functions:

1. $\mathcal{F}^{\mathsf{SendToContractTransparent}}$, enabling a user to send DZCK in a transparent form to a contract.
2. $\mathcal{F}^{\mathsf{SendToContractObfuscated}}$, enabling a user to send DZCK in an obfuscated form to a contract.
3. $\mathcal{F}^{\mathsf{WithdrawFromTransparent}}$, enabling a contract to withdraw transparent DZCK to a user in an obfuscated form.
4. $\mathcal{F}^{\mathsf{WithdrawFromTransparentToContractTransparent}}$, enabling a contract to withdraw transparent DZCK to a contract in a transparent form.
5. $\mathcal{F}^{\mathsf{WithdrawFromTransparentToContractObfuscated}}$, enabling a contract to withdraw transparent DZCK to a contract in an obfuscated form.
6. $\mathcal{F}^{\mathsf{WithdrawFromObfuscated}}$, enabling a contract to withdraw obfuscated DZCK to a user in an obfuscated form.
7. $\mathcal{F}^{\mathsf{WithdrawFromObfuscatedToContractTransparent}}$, enabling a contract to withdraw obfuscated DZCK to a contract in a transparent form.
8. $\mathcal{F}^{\mathsf{WithdrawFromObfuscatedToContractObfuscated}}$, enabling a contract to withdraw obfuscated DZCK to a contract in a transparent form.
9. $\mathcal{F}^{\mathsf{Execute}}$, enabling a user to pay for the transaction fee, send DZCK to another user and initiate a contract call.

7.5.2 Bid Contract

$\mathsf{Contract}^{\mathsf{Bid}}$ is utilized to enable prospective Generators to lock their bids in order to join the consensus, update the expiration height of their existing bids and withdraw the said bids once the expiry height has elapsed.

$\mathsf{Contract}^{\mathsf{Bid}}$ is comprised of the following functions:

1. $\mathcal{F}^{\mathsf{Bid}}$, enabling a prospective Generator to submit a bid.
2. $\mathcal{F}^{\mathsf{ExtendBid}}$, enabling an existing Generator to extend the bid expiration block height.
3. $\mathcal{F}^{\mathsf{WithdrawBid}}$, enabling a Generator with an expired bid to withdraw the said bid.

7.5.3 Stake Contract

$\mathsf{Contract}^{\mathsf{Stake}}$ is utilized to enable prospective Provisioners to lock their stakes in order to join the consensus, update the expiration height of their existing stakes and withdraw the said stakes once the expiry height has elapsed. Additionally, the contract enforces slashing, which enables any protocol participant to report a Provisioner who has committed slashable offence and is consequently eligible to have his/her stake revoked. The participant responsible for reporting the offence of is eligible for a reward comprised of a portion of the slashed Provisioner's stake.

$\mathsf{Contract}^{\mathsf{Stake}}$ is comprised of the following functions:

1. $\mathcal{F}^{\mathsf{Stake}}$, enabling a prospective Provisioner to submit a stake.
2. $\mathcal{F}^{\mathsf{ExtendStake}}$, enabling an existing Provisioner to extend the stake expiration block height.
3. $\mathcal{F}^{\mathsf{WithdrawStake}}$, enabling a Provisioner with an expired stake to withdraw the said stake.
4. $\mathcal{F}^{\mathsf{Slash}}$, enabling a user to report a slashable offence.

7.5.4 Reward Contract

$\mathsf{Contract}^{\mathsf{Reward}}$ is utilized to distribute rewards to the Provisioners responsible for the finalization of the blocks as well as to the Generators responsible for forging the finalized blocks and enabling the Provisioners to withdraw their accrued rewards.

$\mathsf{Contract}^{\mathsf{Reward}}$ is comprised of the following functions:

1. $\mathcal{F}^{\mathsf{Distribute}}$, enabling the leader for the round to distribute the rewards.
2. $\mathcal{F}^{\mathsf{WithdrawReward}}$, enabling a Provisioner to withdraw the accrued rewards.

7.6 Kadcast

Kadcast [RT19] is a structured overlay network based on [MM02] utilized in the DZCK protocol. Kadcast handles the propagation of messages in the DZCK protocol. Kadcast's message propagation function acts as a concrete instantiation of $\mathcal{F}^{\mathsf{Propagate}}$ defined in the previous section. For in-depth look into Kadcast, readers are referred to [RT19].

8 Concrete Protocol

In this section we define a real-life concrete protocol as $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain}}$, instantiated with chain Chain. The protocol is assumed to be instantiated with Segregated Byzan-

tine Agreement consensus mechanism, Kadcast, Rusk VM and the Genesis Contracts. We also assume that the protocol participants have loosely synchronized internal clocks with a permitted clock drift of Δ_{clock}.

Chain is defined as a set of blocks iteratively bounded to each other in a descending order via hashing. Once a block is added to the chain, the said block is considered to be irreversible (i.e. final). Formally, the chain is defined as $\mathsf{Chain} \leftarrow \mathsf{Set}[\mathsf{Block}]$, where for a random block index $r \leftarrow^{\$} \pmod{|\mathsf{Chain}|}$ the following condition holds:

$$(\mathsf{PreviousBlockHash}(\mathsf{Chain}[r]) = \mathcal{H}_{\mathsf{blake2b}}(\mathsf{Chain}[r-1].\mathsf{Header})) \wedge$$

$$\wedge(\mathsf{Height}(\mathsf{Chain}[\mathsf{r}]) = \mathsf{Height}(\mathsf{Chain}[r-1]) + 1),$$

where PreviousBlockHash() is a function that returns the previousBlockHash entry for the corresponding block, and Height() is a function that returns the height entry for the corresponding block. We define Chain[0] as a special block called *genesis block* for which $\mathsf{PreviousBlockHash}(\mathsf{Chain}[0]) = 0$ holds true. Genesis block is instantiated with the Genesis Contracts, has predefined Generators and Provisioners sets as well as hardcoded seed_0 and seed_1 for corresponding epoch 0 and 1.

Block in $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain}}$ is comprised of the metadata called *header*, *body* composed of a set of *transactions* bundled into the respective block and the *certificate* attesting to the validity of Block. In other words, the block Block is defined as:

$$\mathsf{Block} \leftarrow (\mathsf{Header}, \mathsf{Body}, \mathsf{Certificate}).$$

The Header is denoted as:

$$\mathsf{Header} \leftarrow (\mathsf{version}, \mathsf{height}, \mathsf{timestamp}, \mathsf{previousBlockHash}, \mathsf{seed}, \mathsf{blockReward}, \mathsf{TxRoot}, \mathsf{stateRoot}),$$

where version is the version of the corresponding Block, $\mathsf{Block.Header.height} \leftarrow \mathsf{Chain.Index(Block)}$, timestamp is the time of the corresponding Block generation in *Unix time*, $\mathsf{Block.Header.previousBlockHash} \leftarrow \mathcal{H}_{\mathsf{blake2b}}(\mathsf{Chain}[\mathsf{Block.Header.height}-1].\mathsf{Header})$, seed, blockReward is the cumulative reward to be distributed to the consensus protocol.

Certificate is defined as: $\mathsf{Certificate} \leftarrow (\mathsf{round}, \mathsf{step}, \pi^{\mathsf{score}}_{\mathsf{plonk}}, \mathsf{score}, \sigma^{\mathsf{BLS}}_{\mathsf{agg}}, \mathsf{validatorSeq}_{\bigstar})$, where round is the block height for which the certificate was produced, step is the first step of the Reduction phase of the successful iteration of the consensus, $(\pi^{\mathsf{score}}_{\mathsf{plonk}}, \mathsf{score})$ are the entries attesting to the validity of the score of the Generator responsible for the production of the candidate block corresponding to the finalized block for round round, $\sigma^{\mathsf{BLS}}_{agg}$ is the aggregated BLS signature of the committee validators of the successful iteration of the consensus for round round attesting to the validity of the finalized block and $\mathsf{validatorSeq}_{\bigstar}$ is a binary mapping corresponding to the validators in each of three respective committees who had their signatures aggregated into $\sigma^{\mathsf{BLS}}_{agg}$. Certificates are constructed locally

for every consensus participant, meaning that there exists no uniform certificate for a consensus round.

We define a transaction Tx as:

$$\mathsf{Tx} \leftarrow (\mathsf{version}, \mathsf{type}, \mathsf{stateTransitionInit}),$$

where $\mathsf{version}$ is the transaction version, type is the transaction type and $\mathsf{stateTransitionInit}$ is the data responsible for the initialization of state transitions.

We define a Coinbase transaction $\mathsf{Tx}^{\mathsf{coinbase}}$ as a transaction responsible for distributing the rewards to the consensus participants who have successfully attested to the validity of the block. Coinbase transaction is always included as the last transaction in the block and is defined as $\mathsf{Tx.type} = 0$. State transition initialization data for a Coinbase transaction is defined as:

$$\mathsf{Tx}^{\mathsf{coinbase}}.\mathsf{stateTransitionInit} \leftarrow (\mathsf{note}_{\mathsf{transparent}}, \mathsf{provisionerReward}, \mathsf{validatorSet}),$$

where $\mathsf{note}_{\mathsf{transparent}}$ is the reward payment to Generator responsible for forging the finalized block for the corresponding round, $\mathsf{provisionerReward}$ is the total reward to payed out to the Provisioners responsible for successfully validating the block for the corresponding round and $\mathsf{validatorSet}$ is a set Provisioner public keys corresponding to the Provisioners responsible for successfully finalizing the block for $\mathsf{round} - 1$. Coinbase transaction $\mathsf{stateTransitionInit}$ data is passed through the $\mathsf{Contract}^{\mathsf{Reward}}.\mathcal{F}^{\mathsf{Distribute}}$ function, i.e.
$\mathsf{Contract}^{\mathsf{Reward}}.\mathcal{F}^{\mathsf{Distribute}}(\mathsf{stateTransitionInit})$.

The other transaction type is called Standard transaction $\mathsf{Tx}^{\mathsf{standard}}$. Standard transaction is defined as $\mathsf{Tx.type} = 1$. State transition initialization data for a Standard transaction is defined as:

$$\mathsf{Tx}^{\mathsf{standard}}.\mathsf{stateTransitionInit} \leftarrow (\mathsf{Inputs}, \mathsf{Crossover}, \mathsf{Outputs}, \mathsf{Fee}\pi^{\mathsf{execute}}_{\mathsf{score}}, \mathsf{calldata}),$$

where Inputs is a set of inputs being spent in $\mathsf{Tx}^{\mathsf{standard}}$, $\mathsf{Crossover}$ is a unique note type acting as bridge for DZCK between the transactional and general compute layer, $\mathsf{Outputs}$ is a set of outputs being created, Fee is the fee allocated as a reimbursment for the transaction computation costs and $\mathsf{calldata}$ is the data attached to the transaction enabling it to initiate a contract call. The first 32 bytes of $\mathsf{calldata}$ ($\mathsf{calldata}[0:32]$) are required to denote a contract address. If $\mathsf{calldata}[0:32] = 0$, then the transaction is treated as a *contract creation* transaction.

We define the contract address as

$$\mathsf{contractAddress} \leftarrow \mathcal{H}_{\mathsf{blake2b}}(\mathsf{contractCode}||\mathsf{state}_{\mathsf{global}}.\mathsf{height}||\mathsf{state}_{\mathsf{global}}.\mathsf{contractsCreated}),$$

where $\mathsf{contractCode}$ is the code of the contract being instantiated, $\mathsf{state}_{\mathsf{global}}.\mathsf{height}$ is the current height of $\mathcal{P}^{\mathsf{real}}_{\mathsf{Chain}}$ and $\mathsf{state}_{\mathsf{global}}.\mathsf{contractsCreated}$ is the total number of contracts created during the current block.

9 Related Works

Bitcoin [Nak08] introduces the notion of a blockchain as well as the Nakamoto consensus mechanism. An alternative mechanism known as Proof-of-Stake is subsequently proposed by [KN12]. Numerous other works build upon the concept introduced in [KN12] to instantiate variations of Proof-of-Stake-based protocols such as [Ben+14b; Goo14; DPS16; Mic16; Kia+17; Dav+18; GOT19; Ker+19; But+20]. Related to the work introduced in this paper is [Mic16], which expands on [TA84; Mic17] to create the first permission-less Proof-of-Stake-based protocol with a negligible probability of a fork.

Monero, the largest privacy-preserving digital currency based on the market capitalization[2] at the time of the publication of this paper, was originally based on [Sab13], before being amended with [Max15; Bac15; NMM16] and later with a novel zero-knowledge proof scheme called Bulletproofs [Bun+18]. Zcash, the second largest privacy-preserving digital currency based on market capitalization, proposed in [Ben+14a], utilizes [Gro16] zk-SNARK (zero-knowledge Succinct Non-Interative ARgument of Knowledge) proof system in conjuction with a UTxO-based transaction model specified in [Hop+21]. Alternatively, Firo (formerly known as Zcoin) utilizes a different transaction model called Lelantus [Jiv19]. Additionally, [Fau+18] and subsequently [Gua+20] posit two different approaches to creation of a privacy-preserving account model.

[Bun+19] proposes a notion of privacy-preserving token contracts, emulating the transaction model proposed in [Hop+21] in an Ethereum smart contract. [PSS19] builds on the approach discussed in [Bun+19] to enable private transactions on Ethereum mainnet.

Other protocols, such as [Goo14; Dos+20; Fou20] propose solutions to create compliant security token frameworks. [Goo14] adapts the transaction model introduced in [Hop+21], making limited anonymity-preserving functionality available to the smart contract layer of Tezos. Polymath, after initially launching as an Ethereum-based protocol, subsequently switched to developing a separate protocol called Polymesh [Dos+20], which natively supports a restricted set of privacy-preserving functions, while other features are perfomed off-chain. [Fou20] introduces a protocol with a *two-channel* consensus mechanism comprised of a Proof-of-Stake-based optimistic channel and federated BFT-based fallback channel. Findora supports limited privacy-preserving functionality on-chain.

10 Conclusion

In this paper, we have established the requirements for a protocol in **Section 5** and subsequently formalized a concrete instantiation of the protocol called DZC

K 2

Network around the aforementioned requirements. The concrete protocol was instantiated with a novel permission-less Proof-of-Stake-based consensus mechanism called Segregated Byzantine Agreement, featuring a privacy-preserving leader extraction procedure called Proof-of-Blind Bid, as well as with two novel transaction models: Phoenix, a UTxO-based transaction model enabling the confidential spending of non-obfuscated outputs and Zedger, a hybrid transaction model designed with regulatory compliance in regards to security tokenization and lifecycle management in mind. Additionally, we established a new WebAssembly-based virtual machine called Rusk VM, which includes the native support for cryptographic primitives such as zero-knowledge proof verification, as well as an efficient approach to creating Merkle Tree inside contract storage.

References

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230. eprint: https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[DH76] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography". In: *IEEE Trans. Information Theory* 22 (1976), pp. 644–654.

[BW80] Jon Bentley and Derick Wood. "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles". In: *IEEE Trans. Computers* 29 (July 1980), pp. 571–577. DOI: 10.1109/TC.1980.1675628.

[Mer80] Ralph Merkle. "Protocols for Public Key Cryptosystems". In: Apr. 1980, pp. 122–134. DOI: 10.1109/SP.1980.10006.

[Cha82] David Chaum. "Blind Signatures for Untraceable Payments". In: (1982). DOI: 10.1007/978-1-4757-0602-4_18.

[TA84] Russell Turpin and Brian A. Coan. "Extending binary Byzantine agreement to multivalued Byzantine agreement". In: *Information Processing Letters* 18 (1984), pp. 73–76. DOI: 10.1016/0020-0190(84)90027-9.

[El 85] Taher El Gamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE Transactions on Information Theory* 31 (Jan. 1985), pp. 469–472.

[Ped91] Torben P. Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *CRYPTO*. 1991.

[Sch95] Claus Schnorr. "Efficient Identification and Signatures for Smart Cards". In: Jan. 1995, pp. 239–252. ISBN: 978-0-387-97317-3. DOI: 10.1007/0-387-34805-0_22.

[LSS96] Laurie Law, Susan Sabett, and Jerry Solinas. "How to Make a Mint: The Cryptography of Anonymous Electronic Cash". In: (1996).

[Sza96] Nick Szabo. "Smart Contracts: Building Blocks for Digital Markets". In: 1996.

[Sza97] Nick Szabo. "Formalizing and Securing Relationships on Public Networks". In: *First Monday* 2 (1997). DOI: 10.5210/fm.v2i9.548.

[Wei98] Dai Wei. "b-money". In: (1998).

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short Signatures from the Weil Pairing". In: ASIACRYPT '01 (2001), pp. 514–532.

[Bac02] Adam Back. “Hashcash - A Denial of Service Counter-Measure”. In: (2002).

[BLS02] Paulo Barreto, Ben Lynn, and Michael Scott. “Constructing Elliptic Curves with Prescribed Embedding Degrees”. In: vol. 2576. Sept. 2002. ISBN: 978-3-540-00420-2. DOI: 10.1007/3-540-36413-7_19.

[MM02] P. Mayamounkov and D. Mazieres. “Kademlia: A peer-to-peer information system based on the XOR metric”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2429 (Jan. 2002).

[VCS03] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Sirer. “KARMA: A Secure Economic Framework for Peer-To-Peer Resource Sharing”. In: (2003).

[DGN04] Cynthia Dwork, Andrew Goldberg, and Moni Naor. “On Memory-Bound Functions for Fighting Spam”. In: *Lecture Notes in Computer Science* (2004). DOI: 10.1007/978-3-540-45146-4_25.

[Sza05] Nick Szabo. *Bit Gold*. 2005.

[Nak08] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: *Cryptography Mailing list at https://metzdowd.com* (2008).

[Nam11] Namecoin. *Namecoin*. 2011. URL: https://namecoin.org.

[KN12] Sunny King and Scott Nadal. “PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake”. In: (2012).

[RS12] Dorit Ron and Adi Shamir. “Quantitative Analysis of the Full Bitcoin Transaction Graph”. In: (2012). DOI: 10.1007/978-3-642-39884-1_2.

[But13] Vitalik Buterin. “Ethereum Whitepaper”. In: 2013.

[KCW13] Joshua A. Kroll, Ian C. Davey, and Edward W. Felten. “The Economics of Bitcoin Mining, or Bitcoin in the Presence of Adversaries”. In: (2013).

[Sab13] Nicolas van Saberhagen. “CryptoNote v 2.0”. In: (2013).

[Aum+14] Jean-Philippe Aumasson et al. “BLAKE2”. In: Dec. 2014, pp. 165–183. ISBN: 978-3-662-44756-7. DOI: 10.1007/978-3-662-44757-4_9.

[Ben+14a] Eli Ben-sasson et al. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: May 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[Ben+14b] Iddo Bentov et al. “Proof of Activity: Extending Bitcoin’s Proof of Work via Proof of Stake”. In: *SIGMETRICS Performance Evaluation Review* 42 (2014), pp. 34–37.

[Goo14] L. M. Goodman. “Tezos : A Self-Amending Crypto-Ledger Position Paper”. In: 2014.

[Bac15] Adam Back. *Ring signature efficiency*. 2015. URL: https://bitcointalk.org/index.php?topic=972541.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: (2015), pp. 281–310. DOI: 10.1007/978-3-662-46803-6_10.

[Max15] Gregory Maxwell. "Confidential Transactions". In: (2015).

[Bon16] Joseph Bonneau. "Why Buy When You Can Rent?" In: 9604 (2016), pp. 19–26. DOI: 10.1007/978-3-662-53357-4_2.

[DPP16] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. "Efficient Sparse Merkle Trees". In: Nov. 2016, pp. 199–215. ISBN: 978-3-319-47559-2. DOI: 10.1007/978-3-319-47560-8_13.

[DPS16] Phil Daian, Rafael Pass, and Elaine Shi. "Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake". In: (2016).

[Gro16] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: May 2016, pp. 305–326. ISBN: 978-3-662-49895-8. DOI: 10.1007/978-3-662-49896-5_11.

[Mic16] Silvio Micali. "ALGORAND: The Efficient and Democratic Ledger". In: (2016).

[NMM16] Shen Noether, Adam Mackenzie, and The Monero Research Lab. "Ring Confidential Transactions". In: *Ledger* 1 (2016), pp. 1–18. DOI: 10.5195/LEDGER.2016.34.

[Poe16] Andrew Poelstra. "Mimblewimble". In: (2016).

[Kia+17] Aggelos Kiayias et al. "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: (2017), pp. 357–388. DOI: 10.1007/978-3-319-63688-7_12.

[Kum+17] Amrit Kumar et al. "A Traceability Analysis of Monero's Blockchain". In: Aug. 2017, pp. 153–173. ISBN: 978-3-319-66398-2. DOI: 10.1007/978-3-319-66399-9_9.

[Mic17] Silvio Micali. "Byzantine Agreement, Made Trivial". In: (2017).

[PSS17] Rafael Pass, Lior Seeman, and Abhi Shelat. "Analysis of the Blockchain Protocol in Asynchronous Networks". In: (2017), pp. 643–673. DOI: 10.1007/978-3-319-56614-6_22.

[Que17] Jeffrey Quesnelle. "On the linkability of Zcash transactions". In: (2017).

[SSZ17] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. "Optimal Selfish Mining Strategies in Bitcoin". In: (2017), pp. 515–532. DOI: 10.1007/978-3-662-54970-4_30.

[Bun+18] Benedikt Bunz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: May 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.

[Dav+18] Bernardo David et al. “Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain”. In: (2018), pp. 66–98. DOI: 10.1007/978-3-319-78375-8_3.

[ES18] Ittay Eyal and Emin Sirer. “Majority Is Not Enough: Bitcoin mining is vulnerable”. In: *Communications of the ACM* 61 (2018), pp. 95–102. DOI: 10.1145/3212998.

[Fau+18] Prastudy Fauzi et al. “QuisQuis: A New Design for Anonymous Cryptocurrencies”. In: *IACR Cryptology ePrint Archive* 2018 (2018).

[Kap+18] G Kappos et al. “An Empirical Analysis of Anonymity in Zcash.” In: Jan. 2018.

[Mös+18] Malte Möser et al. “An Empirical Analysis of Traceability in the Monero Blockchain”. In: *Proceedings on Privacy Enhancing Technologies* 2018 (June 2018), pp. 143–163. DOI: 10.1515/popets-2018-0025.

[BF19] Alex Biryukov and Daniel Feher. “Privacy and Linkability of Mining in Zcash”. In: June 2019, pp. 118–123. DOI: 10.1109/CNS.2019.8802711.

[Bun+19] Benedikt Bunz et al. “Zether: Towards Privacy in a Smart Contract World”. In: *IACR Cryptology ePrint Archive* 2019 (2019).

[GWC19] A. Gabizon, Zachary J. Williamson, and Oana Ciobotaru. “PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge”. In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 953.

[GOT19] Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. “Proof-of-Stake Protocols for Privacy-Aware Blockchains”. In: (2019), pp. 690–719. DOI: 10.1007/978-3-030-17653-2_23.

[Gra+19] Lorenzo Grassi et al. “Starkad and Poseidon: New Hash Functions for Zero Knowledge Proof Systems”. In: *IACR Cryptology ePrint Archive* 2019 (2019).

[Jiv19] Aram Jivanyan. “Lelantus: A New Design for Anonymous and Confidential Cryptocurrencies”. In: (2019).

[Ker+19] Thomas Kerber et al. “Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake”. In: May 2019, pp. 157–174. DOI: 10.1109/SP.2019.00063.

[Ler19] Sergio Demian Lerner. “RSK: Bitcoin Powered Smart Contracts”. In: (2019).

[PSS19] Alexey Pertsev, Roman Semenov, and Roman Storm. “Tornado Cash Privacy Solution”. In: 2019.

[RT19] Elias Rohrer and Florian Tschorsch. “Kadcast: A Structured Approach to Broadcast in Blockchain Networks”. In: 2019, pp. 199–213. ISBN: 978-1-4503-6732-5. DOI: 10.1145/3318041.3355469.

[Tea19] Vyper Team. “Vyper Documentation”. In: 2019.

[Zam+19] Alexei Zamyatin et al. “SoK: Communication Across Distributed Ledgers”. In: 2019.

[But+20] Vitalik Buterin et al. *Combining GHOST and Casper*. 2020.

[Dos+20] Adam Dossa et al. “Polymesh Whitepaper”. In: 2020.

[Fou20] Findora Foundation. “Findora Litepaper”. In: 2020.

[Gua+20] Zhangshuang Guan et al. “BlockMaze: An Efficient Privacy-Preserving Account-Model Blockchain Based on zk-SNARKs”. In: *IEEE Transactions on Dependable and Secure Computing* PP (Sept. 2020), pp. 1–1. DOI: 10.1109/TDSC.2020.3025129.

[Kho20] Dmitry Khovratovich. *Encryption with Poseidon*. Available at https://drive.google.com/file/d/1EVrP3DzoGbmzkRmYnyEDcIQcXVU7GlOd/view. 2020.

[Mah20] Toghrul Maharramov. “DZCK Economic Model Paper”. In: (2020).

[Zha+20] Zongyang Zhang et al. “A Refined Analysis of Zcash Anonymity”. In: *IEEE Access* 8 (Feb. 2020), pp. 31845–31853. DOI: 10.1109/ACCESS.2020.2973291.

[Eth21] Ethereum. “Solidity Documentation”. In: 2021.

[Hop+21] Daira Hopwood et al. “Zcash Protocol Specification, Version 2019.0.6”. In: (2021).

[Mah21] Toghrul Maharramov. “Confidential Security Contract (XSC) Standard V2.0”. In: (2021).

[Ros21] Andreas Rossberg. *WebAssembly Specification*. 2021.

[Woo21] Gavin Wood. “ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER”. In: (2021).