Phoenix transaction model

Version 1.0

Dusk

Last update: September 20, 2024

Abstract. Recent advances in blockchain technologies have led to their increased usage in different use cases, with one prominent case being cryptocurrencies. Often, this involves large amounts of transaction data being shared over the network. Even when the public keys of the users are not directly associated with identities, as is the case in Bitcoin, sending this data in plaintext can lead to traceability and de-anonymization. To solve this problem, projects like Zcash introduced a private-by-design blockchain capable of issuing obfuscated (private) transactions, where the recipient, the sender, and the amount of exchanged money remain private. This is possible thanks to zero-knowledge proofs, with the drawback of the high computing resources they require. This document gives an overview and discussion of Phoenix, the transaction model used by Dusk. Phoenix is inspired by the Zcash transaction model, sharing many of its privacy features, and includes a novel delegation model in which users can delegate proof computations to partially trusted third parties.

Table of Contents

1 Introduction

One classical problem in cryptocurrencies is anonymity. Since the deployment of the Bitcoin blockchain in 2009 [25], many concerns about the security and privacy guarantees of such networks have arisen. In most transparent blockchains, transaction information like the addresses of sender and recipient, and the amount of money being transferred, are public. Although an address is not directly linked to a user's identity, the transparency of the whole system makes transactions traceable. Furthermore, questions about Bitcoin's privacy have spanned an increasing body of research, and many studies show that it is feasible to identify and track participants of the Bitcoin network [29,18]. Whereas there have been attempts at mitigating the traceability of coins through the use of multiple addresses and mixers [4], this approach has only found limited success.

Security of transactions in this context is at higher risk than with traditional banking: if an individual gets ever linked to its address, privacy of all past and future movements associated with that account is totally lost. Furthermore, the interest of society and regulators in the topic has grown in the recent years, both for individual use and industry trade secrets. The need to guarantee the right to privacy has ended up with new legislation like the General Data Protection Regulation (GDPR) of the European Union [1] or the California Consumer Privacy Act (CCPA) [2].

However, it is a considerable challenge to use a transparent system like the Bitcoin blockchain and at the same time provide privacy to sensitive data. Recently, a great effort has been put in developing alternative models with anonymity baked into the design, the most prominent examples of this approach being Zcash [31,19] and Monero [33]. These systems allow for conducting private transactions over decentralized networks. Zcash built a solution based on the *unspent transaction output (UTXO)* model used in Bitcoin, but achieving anonymity at the same time. Zcash has an on-and-off privacy switch, whereas transactions in Monero are always private.

While initial cryptocurrencies used relatively mild cryptographic primitives, namely hashes and signature schemes, these newer private coins leverage heavy machinery, including very efficient zero-knowledge proof systems (zk-SNARKs) [17,5,14] or ring signatures [15,26]. This allows for transactions (called *shielded transactions* in the context of Zcash) in which sender address, recipient address and value are all hidden from the public, and yet the transactions can still be registered in the blockchain.

One recent project is Dusk [23], an open-source public blockchain network designed for securities trading and other financial applications. In this context, privacy is not only important, but it is essential for businesses and regulated markets. From a technical perspective, the main difference with other privacy-oriented blockchains is that Dusk aims to support the capability of executing confidential smart contracts [22].

Contributions. This document gives an overview of *Phoenix* [9], the transaction model developed by Dusk, capable of issuing obfuscated (private) transactions across Dusk's blockchain. It is a UTXO model based on Zcash [19] and CryptoNote [33], with the addition of a method for delegating the expensive proof computations to a partially trusted helper. The goal of this document is to explain the different aspects of the protocol, from the concrete cryptographic building blocks and their parameter choices, to the structure and flow of the transactions, giving an intuition on security and the reasoning behind the decisions made.

Roadmap. The paper is organized as follows. In Section 2, we give a high-level overview of the transactions flow. In Section 3, we present the set of cryptographic primitives used by Phoenix. In Section 4, we describe Phoenix within the Dusk context. We first describe the network, and then how transactions are created and processed. In Section 5, we analyze the protocol and its properties, giving an intuition on security, and its implementation, for which we provide benchmarks.

2 Model intuition

Before we introduce the Phoenix protocol in detail, we give a simplified example to illustrate the ideas behind it. Suppose there are three parties Alice, Bob, and Charlie, that want to exchange some money secretly. In particular, Alice wants to send some money v to Bob, and then Bob wants to transfer the same amount of money to Charlie using obfuscated transactions.

$$\text{Alice} \xrightarrow{\quad v \quad} \text{Bob} \xrightarrow{\quad v \quad} \text{Charlie}$$

We show how the protocol works in two steps. First, we explain how Alice would send v to Bob, without explaining how she got the money in the first place, and afterwards, we focus on how Bob can spend the money received from Alice to send it to Charlie.

Alice → Bob

If Alice wants to send v to Bob, Alice must create a new note $\mathsf{N_B}$ of value v that only Bob can spend, and sent it to the network as part of a transaction. The note $\mathsf{N_B}$ will contain the following fields:

$$\mathsf{N_B} = \{\text{value commitment, encrypted data, note public key}\}.$$

The value commitment is a commitment to the amount of money v that Alice transfers to Bob. The encrypted data is encrypted using Bob's public key, so that Bob can decrypt it to recover the opening of the value commitment. The note public key is a one-time public key that will allow Bob to spend the note; it is derived from Bob's public key in a way that only Bob (not even Alice), can compute its corresponding secret key.

Once Alice's transaction that includes $\mathsf{N_B}$ is accepted by the network, the note $\mathsf{N_B}$ is included as a new leaf in the Merkle tree of notes, a data structure that allows for efficient membership proofs.

Bob → Charlie

Assume that now Bob wants to spend the money v from $\mathsf{N_B}$. Bob must show that $\mathsf{N_B}$ is a valid note that is included in the Merkle tree of notes, and ownership of the note (i.e. knowledge of the private key associated with the note public key). Since we want the transaction to be private, we don't want to disclose which note nor how much money Bob is spending. Thus, we use a zero-knowledge proof to ensure that:

1. The note $\mathsf{N_B}$ is stored in the Merkle tree of notes (membership).
2. Bob knows the secret key associated to the note public key of $\mathsf{N_B}$ (ownership).

To prevent Bob from spending $\mathsf{N_B}$ more than once, the system requires from Bob a value, called nullifier, as part of the transaction, which is a unique deterministic value correlated to Bob's secret key and the note, but cannot be linked to the note by external parties (i.e. still hides which specific note Bob is spending). If Bob tried to spend $\mathsf{N_B}$ again, they would get the same nullifier, and since it must be included as part of the transaction, the network would detect the attempt to spend a note that is already nullified. This means that the network must keep track of all nullifiers. Bob has to guarantee that the nullifier has been computed correctly, by providing a zero-knowledge proof of the following statement:

3. The nullifier associated with $\mathsf{N_B}$ is computed as it should (nullification).

To send the money from $\mathsf{N_B}$ to Charlie, Bob will create a note $\mathsf{N_C}$ with value v that only Charlie will be able to spend. The idea is the same as in the previous section, but Bob must also prove that the transfer contains the right amount of money spent, and not more. Therefore, the zero-knowledge proof also proves the following statements:

4. The value commitment associated to $\mathsf{N_C}$ is computed as it should (minting).
5. The money sent in $\mathsf{N_C}$ is the money spent from $\mathsf{N_B}$ (balance integrity).

Thus, Bob sends a transaction to the network including the following fields:

$$\text{Transaction from Bob} = \{\text{spend proof, nullifier, } \mathsf{N_C}\}.$$

We remark that, in fact, Alice also included all the above fields in the transaction, but we only focused in the new note's minting part for the sake of clarity.

3 Cryptographic primitives

In this section, we detail the cryptographic primitives used in Phoenix. We briefly introduce Merkle trees, the commitment scheme, encryption scheme, proof system, elliptic curves and hash functions used, specifying at each step the concrete parameters with which each of the primitives is instantiated.

Notation. Throughout the document, we use the following conventions. Given a set S, we denote sampling an element x uniformly at random from S by $x \leftarrow S$. Any group $\mathbb{G}$ used is of a large prime order, and we assume that the discrete logarithm problem is hard in $\mathbb{G}$. If two elements are denoted by the same letter in upper case and lower case, e.g. a, A, this often signifies the fact that A is a public key corresponding to the secret key a.

3.1 Merkle trees

A *Merkle tree* [24] is a tree that contains at every vertex the hash of its children vertices. More precisely, we consider a perfect k-ary tree of height h. The single vertex at level 0 is called the *root* of the tree, and the k^h vertices at level h are called the *leaves*. Given a vertex in level i, the k vertices in level $i+1$ that are adjacent to it are called its *children*. Two vertices are each other's *sibling* if they are children of the same vertex.

To each vertex in the tree, we will recursively associate a value, starting from the leaves.[1] Let H be a hash function.

- Level h: leaves are initialized to a null value. Through the lifetime of the tree, they will progressively be filled from left to right with values.
- Level $0 \leq i < h$: each vertex has k children $c_1, \dots, c_k$ at level $i+1$. We set the value of the vertex to $H(c_1, \dots, c_k)$.

The tree is updated every time a new value is written into a leaf, by updating the $h+1$ elements in the path from the new value to the root. In particular, this means that the root changes after every update. A nice feature of Merkle trees is that, given a root r, it is easy to prove that a value x is in a leaf of a tree with root r. The proof works as follows:

- *Prove.* For $i = h, \dots, 1$, let x_i be the vertex that is in level i and is in the unique path from x to the root. Let $y_{i,1}, \dots, y_{i,k-1}$ be the $k-1$ siblings of x_i. Output
$$(x, (y_{1,1}, \dots, y_{1,k-1}), \dots, (y_{h,1}, \dots, y_{h,k-1})).$$
- *Verify.* Parse input as $(x_h, (y_{1,1}, \dots, y_{1,k-1}), \dots, (y_{h,1}, \dots, y_{h,k-1}))$, where x_h is the purported value and $y_{i,1}, \dots, y_{i,k-1}$ are the purported siblings at level i. For $i = h-1, \dots, 0$, compute[2]
$$x_i = H\left(x_{i+1}, y_{i+1,1}, \dots, y_{i+1,k-1}\right).$$

This allows for proving membership in a set of size k^h by sending $O(kh)$ values. This proof is sound provided that the hash function is collision resistant [21, Section 5.6.2]. For our application, we will set $k = 4$, $h = 17$, and for H we will use the Poseidon hash function, as specified in Section 3.4.

3.2 Zero-knowledge proof systems

Phoenix uses the zk-SNARK PlonK [14] as its proof system. PlonK allows anyone to prove satisfiability of any arithmetic circuit modulo a prime. Since arithmetic circuit satisfiability is an NP-complete problem, this proof system will allow us to prove any statement in NP. PlonK makes use of the KZG polynomial commitment scheme [20], as described in [14]. This requires instantiating PlonK over a pairing-friendly group, which is described in Section 3.3.

Below is a summary the efficiency of PlonK, for a circuit with n multiplication gates and ℓ public inputs.

---

[1] We will often abuse notation and write the vertex to refer to the value associated with the vertex.

[2] To be precise, the prover also has to send $\lceil \log_2 k \rceil$ bits for each level, specifying the position of x_i with respect to its siblings, so that the verifier knows in which order to arrange the inputs of the hash.

- *Proving time:* $O(n)$ group and field operations.
- *Verification time:* $O(1 + \ell)$ group and field operations.
- *Proof size:* $O(1)$ group and field elements.

PlonK is sound in the algebraic group model [13], and statistically zero-knowledge. A complete and explicit description of the scheme can be found in [14, Section 8].

3.3 Elliptic curves

BLS12-381 [6] and Jubjub [7] are the elliptic curves used. More precisely, let

$$q = 4002409555221667393417789825735904156556882819939007885332058136\\ 124031650490837864442687629129015664037894272559787,$$

$$p = 5243587517512619047944774050818596583769055250052763782260365869\\ 9938581184513.$$

Note that both are prime numbers, with bit-lengths 381 and 255, respectively. The curve BLS381-12 is the curve over $\mathbb{F}_q$ defined by the equation

$$E : Y^2 = X^3 + 4.$$

We have that $E(\mathbb{F}_q)$ has different subgroups $\mathbb{G}_1, \mathbb{G}_2$ such that $\#\mathbb{G}_1 = \#\mathbb{G}_2 = p$. This curve is pairing-friendly (with embedding degree $k = 12$), so pairings are efficiently computable. More precisely, Phoenix makes use of the bilinear group

$$\mathbb{B} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T)\,.$$

By instantiating the zk-SNARK with the bilinear group $\mathbb{B}$, we are be able to prove statements about satisfiability of arithmetic circuits over $\mathbb{F}_p$, the so-called *scalar field* of E.

Furthermore, we are interested in proving certain operations with the zk-SNARK, like the correct verification of a Schnorr signature σ. Note that σ is an element of a certain elliptic curve J, but is represented as two coordinates in the base field $\mathbb{F}_s$ of J. Therefore, the verification can be best represented as arithmetic constraints modulo s. While it is possible to represent any NP statement using arithmetic modulo p to plug it into the zk-SNARK, this incurs into a significant efficiency loss if not done carefully. The natural thing is to set $s = p$. Therefore, the signature scheme must be instantiated with an elliptic curve over $\mathbb{F}_p$. For this, let

$$d = -\frac{10240}{10241} \bmod p.$$

Phoenix uses the Jubjub curve, defined by the equation

$$J : -X^2 + Y^2 = 1 + dX^2Y^2,$$

over $\mathbb{F}_p$. In particular, it uses a subgroup $\mathbb{J}$ of order

$$t = 6554484396890773809930967563523245729705921265872317281365359162\\392183254199,$$

which is a 252-bit prime.

The primes and groups defined here will be used through the rest of the document.

3.4 Hash functions

Phoenix uses two hash functions, depending on the context. Sometimes, we will want to prove a statement involving a hash function H, e.g. given y, prove knowledge of x such that $H(x) = y$. We will do so with PlonK, which requires statements to be written as arithmetic constraints modulo a large prime p. Most hash function evaluations do not naturally translate to this language, incurring in a big efficiency loss.

To avoid this, the Poseidon hash function [16] $H^{\mathsf{Poseidon}} : \mathfrak{F}_p \to \mathbb{F}_p$, where $\mathfrak{F}_p$ is the set of tuples of $\mathbb{F}_p$-elements of any length, will be used whenever we compute a hash of which we need to produce a proof. This is because Poseidon is purposefully designed to work with modular arithmetic.

Poseidon is built by applying the sponge construction [3] to a permutation $\pi : \mathbb{F}_p^t \to \mathbb{F}_p^t$, for $t = r + c$, where

- r is the *rate*, i.e. the amount of $\mathbb{F}_p$-elements of the input that can be processed in a call to π.
- c is the *capacity*, which is a part of the permutation that is never output by the hash, and is required for security.

The permutation π is composed of linear (matrix multiplication over $\mathbb{F}_p$) and non-linear (S-boxes) operations. Some rounds are *full rounds*, and apply S-boxes to the whole input, and others are *partial rounds*, in which an S-box is applied to a single $\mathbb{F}_p$-element.

In this case, Phoenix uses Poseidon-128 to target 128-bit security. Following the recommendations of [16], parameters are set as $r = 4$ and $c = 1$, so that a hash in the Merkle tree can be computed with a single call to the permutation. Internally, a permutation performs $R_F = 8$ full rounds and $R_P = 60$ partial rounds, and uses $S(x) = x^5$ as the S-box.

When no proofs involving hashes are required, the BLAKE2 family of hash functions [30] is used, which yields better efficiency. In particular, the BLAKE2b hash function $H^{\mathsf{BLAKE2b}} : \mathfrak{F}_p \to \mathbb{F}_t$ is used, since it is optimized for 64-bit platforms.

3.5 Encryption schemes

We use the symmetric encryption AES-GCM.

3.6 Commitment schemes

Phoenix makes use of the well-known Pedersen commitment scheme [27], which we now describe.

- *Setup.* Sample and output the commitment key $\mathsf{ck} = (G, G') \leftarrow \mathbb{J}^2$.
- *Commit.* On input a value v, sample randomness $r \leftarrow \mathbb{F}_t$ and output

$$c = \mathsf{Com}_{\mathsf{ck}}(v; r) = vG + rG'.$$

- *Open.* Reveal v, r. With these values, anyone can recompute the commitment and check whether it matches the commitment previously provided.

The Pedersen commitment scheme is perfectly hiding, and computationally binding under the discrete logarithm assumption.

3.7 Signature schemes

The Schnorr Sigma protocol [32] is used, compiled with the Fiat–Shamir transformation [12,28], as a signature scheme. More specifically, Phoenix makes use of a double-key version to be able to delegate computations later in the protocol. Let $G, G' \leftarrow \mathbb{J}$.

- *Setup.* Sample a secret key $\mathsf{sk} \leftarrow \mathbb{F}_t$ and set the corresponding public key $(\mathsf{pk}, \mathsf{pk}') = (\mathsf{sk}G, \mathsf{sk}G')$. Output $(\mathsf{sk}, (\mathsf{pk}, \mathsf{pk}'))$.
- *Sign.* On input a message m and a secret key sk, sample $r \leftarrow \mathbb{F}_t$ and compute $(R, R') = (rG, rG')$. Compute the challenge $c = H(m, R, R')$, and set

$$u = r - c\mathsf{sk}.$$

  Output the signature $\sigma = (R, R', u)$.
- *Verify.* On input a public key pk, message m and signature $\sigma = (R, R', u)$, compute $c = H(m, R, R')$ and check whether the following equalities hold:

$$\begin{aligned} R &= uG + c\mathsf{pk}, \\ R' &= uG' + c\mathsf{pk}'. \end{aligned}$$

  If so, accept the signature, otherwise reject.

Since the verification of the signature is performed inside a circuit (see Section 4.3), we take as hash function H the Poseidon hash function H^{Poseidon} as described in Section 3.4, except that the output is truncated so that $c \in \mathbb{F}_t$.

The signature scheme is existentially unforgeable under chosen-message attacks under the discrete logarithm assumption, in the random oracle model [21, Section 12.5.1]. While the Schnorr signature scheme is widely known, the double-key version has not been used before, to the best of our knowledge. In the transaction protocol, this is leveraged to allow for delegation of proof computations without the need to share one's secret key with the helper.

4 Transaction model

Phoenix is the transaction model used by Dusk, an open-source public blockchain with a UTXO-based architecture that allows the execution of obfuscated transactions and confidential smart contracts. In blockchains with a state-account configuration such as Ethereum, a state machine describes the amount of money belonging to each account in the network, and the state changes after validating different sets of transactions. In privacy-preserving blockchains, there are no accounts or wallets at the protocol layer. Instead, coins are stored as a list of UTXOs with a quantity and some criteria for spending it. In this approach, transactions are created by consuming existing UTXOs and producing new ones in their place. Dusk follows this system, and UTXOs are called *notes*. From now on, we also use this terminology.

Unlike transparent transaction models, such as the Bitcoin network, where it is easy to monitor which notes were spent, this task is much harder in a privacy-preserving network, since the details of the notes must be kept hidden. In this case, the network must keep track of all notes ever created by storing their hashes in the leaves of a Merkle tree (called *Merkle tree of notes*). That is, when a transaction is validated, the network includes the hashes of the new notes to the leaves of this tree.

More specifically, all notes in the network have the same structure. More specifically, a note is a data object with the following structure:

$$\mathsf{N} = \{\mathsf{type}, \mathsf{com}, \mathsf{enc}, \mathsf{npk}, R\}.$$

The parameters above correspond to the following: type indicates the type of the note, either transparent or obfuscated; com is a commitment to the value of the note; enc is an encryption of the opening of com that can be decrypted using the recipient's view key; npk is the note's public key, whose associated private key nsk can only be computed by the recipient of the note; and R is a point in the Jubjub subgroup $\mathbb{J}$ that allows the recipient to compute nsk and also identify that he is the recipient of the transaction. A note will have an associated position, but we do not consider the position part of the note, as the position cannot be assigned by the sender.

To prevent double spending, transactions include a list of deterministic values called *nullifiers*, one for each note being spent, which invalidates these notes. The network nodes must check that nullifiers were not used before. The idea here is that the nullifier is computed in such a way that an external observer cannot link it to any specific note. This way, when a transaction is accepted, the network knows that some notes are nullified and can no longer be spent, but does not know which ones.

4.1 Transaction structure

At a technical level, transactions consist of two parts, a header $\mathsf{tx}_{\mathsf{metadata}}$ which includes metadata, and a second field tx with the actual contents of the transaction that have been set by the transaction sender. In this document we focus on the latter, but for the sake of completeness, in the following we briefly describe all parts of a transaction.

$$\begin{aligned}
\mathsf{tx}_{\mathsf{skeleton}} &= [\mathsf{root},\ \mathsf{nullifiers}\text{: } [\mathsf{nullifier}_1, \ldots, \mathsf{nullifier}_\mathsf{r}],\ \mathsf{notes}\text{: } [\mathsf{N}_1^{\mathsf{new}}, \ldots, \mathsf{N}_\mathsf{s}^{\mathsf{new}}],\\
&\quad \mathsf{cross}, \mathsf{max_fee}],\\
\mathsf{tx}_{\mathsf{payload}} &= [\mathsf{tx}_{\mathsf{skeleton}}, \mathsf{data}],\\
\mathsf{tx} &= [\mathsf{tx}_{\mathsf{payload}}, \mathsf{payload_hash}, \mathsf{tx_proof}],\\
\mathsf{tx}_{\mathsf{metadata}} &= [\mathsf{timestamp},\ \mathsf{block_height},\ \mathsf{status},\ \mathsf{type},\ \mathsf{tx_hash},\\
&\quad \mathsf{gas_price},\ \mathsf{gas_spent},\ \mathsf{fee}, \mathsf{positions}\text{:}[\mathsf{pos}_1, \ldots, \mathsf{pos}_s]].
\end{aligned}$$

Fig. 1. Elements involved in an obfuscated transaction.

The sender first sets $\mathsf{tx}_{\mathsf{skeleton}}$, which contains the following data: root is a recent root of the Merkle tree of notes; nullifiers is the list of deterministic values that nullify the notes being spent; notes is the list of new notes being minted (each of them with the structure as described above); cross is the amount that will be sent to a contract; and max_fee is the maximum fee the sender is willing to pay for the execution of the transaction.

The $\mathsf{tx}_{\mathsf{payload}}$ contains $\mathsf{tx}_{\mathsf{skeleton}}$ and other details needed for the correct execution of the transaction. That is, the field data will contain details such as the IDs of the smart contracts that need to called, the call data, or the stealth address where the change of the fee should be sent back to.

Once $\mathsf{tx}_{\mathsf{payload}}$ is set, the sender calculates payload_hash, which is the hash[3] of all elements in $\mathsf{tx}_{\mathsf{payload}}$. All these elements together with payload_hash is used to compute tx_proof, which is a zero-knowledge proof that proves that the transaction has been performed following the network rules. We provide more details about this proof in Section 4.3.

While the tx consists of all the parameters that are set by the user sending the transaction, $\mathsf{tx}_{\mathsf{metadata}}$ contains the information that is set by the network once the transaction is processed and included in a block: timestamp is the the date and time the transaction was processed; block_height is the number of the block the transaction was included in; status indicates if the transaction was successful or not; tx_hash is the hash[3] of tx; gas_price is the price at which gas was paid; gas_spent is the amount of gas spent; fee is the amount of DUSK spent in the transaction, which is the result of gas_price $\times$ gas_spent; and finally positions is the list of positions of the newly minted notes in the Merkle tree of notes[4].

[3] Here, we use H^{BLAKE2b} for hashing as described in Section 3.4, except that the output is truncated so that the result is an element of $\mathbb{F}_p$.

[4] Note that the transaction sender does not know these positions in advance, they are added by the validator when the transaction is accepted and included in a block.

4.2 Protocol keys

In this section, we introduce all the different types of keys involved in the protocol. In Phoenix, each note is associated with a unique public key, instead of using the static public key of the recipient. This hinders traceability (the approach was introduced in [33]). Also, it uses two-element keys, which allows users of the network to delegate the process of scanning for the notes addressed to them.

User static keys. Let $G, G' \leftarrow \mathbb{J}$, as defined in Section 3.3. H^{BLAKE2b} denotes the Poseidon hash function introduced in Section 3.4. On the one side, every user keeps the following static keys.

- *Secret key:* $\mathsf{sk} = (a, b)$, where $a, b \leftarrow \mathbb{F}_t$.
- *Public key:* $\mathsf{pk} = (A, B)$, where $A = aG$ and $B = bG$.
- *View key:* $\mathsf{vk} = (a, B)$.

Note keys. On the other side, we assign a one-time key pair to each note issued in the network, computed using the Diffie–Hellman key exchange protocol [8]. The recipient's Diffie–Hellman partial key will be the first part of its public key, A, whereas the sender will use a fresh key. More precisely, the note public key of a note sent to a recipient with public key pair $\mathsf{pk} = (A, B)$ is computed by the sender as follows:

- *Note public key:* npk is computed by the sender as follows.
  1. Sample r uniformly at random from $\mathbb{F}_t$.
  2. Compute a symmetric Diffie–Hellman key $\mathsf{k}_{\mathsf{DH}} = rA$.
  3. Compute a note public key $\mathsf{npk} = H^{\mathsf{BLAKE2b}}(\mathsf{k}_{\mathsf{DH}})G + B$.
  4. Compute $R = rG$.

  The sender of a note will attach to it the note public key npk and the partial Diffie–Hellman R used to create npk.[5] Given a pair (npk, R), the recipient can identify whether the note was sent to them by recomputing $\tilde{\mathsf{k}}_{\mathsf{DH}} = aR$ (using their secret a), and checking the equation

  $$\mathsf{npk} \stackrel{?}{=} H^{\mathsf{BLAKE2b}}(\tilde{\mathsf{k}}_{\mathsf{DH}})G + B.$$

  Note that this check can be performed using only the recipient's view key, and not their whole secret key. This way, a network user can delegate the job of scanning the different transactions of the network to retrieve their notes by sharing their view key vk with an external entity, which we call a *network-listening helper*. Thus, the user could delegate the scanning of all transactions to a different entity by sharing (a, B) with the helper. Even with that information, such an entity could not spend $\mathcal{R}$'s money, since they can not derive $\mathsf{sk}_{\mathcal{R}}$ without the second part of $\mathcal{R}$'s private key.

[5] The pair (npk, R) is sometimes referred to as a *stealth address* [33].

– *Note secret key:* $\mathsf{nsk} = H^{\mathsf{BLAKE2b}}(\tilde{\mathsf{k}}_{\mathsf{DH}}) + b$. Note that this key can only be computed by the recipient of the note, since they are the only ones holding the whole secret key $\mathsf{sk} = (a, b)$, and sk cannot be recovered from public information. This is due to the discrete logarithm assumption in $\mathbb{J}$. The recipient will use the note secret key when spending the note.

When spending a note, the sender will use a variation of the note public key called *nullification key* computed as $\mathsf{npk}' = \mathsf{nsk}G'$.

4.3 Phoenix

We describe Phoenix in the general scenario in which a sender wishes to send different amounts of DUSK $\mathsf{v}_1, \dots, \mathsf{v}_\mathsf{r}$ to different recipients with public keys $\mathsf{pk}_1, \dots, \mathsf{pk}_\mathsf{r}$, and/or send an amount cross to a smart contract. For that, we assume the sender has enough funds, i.e. owns a set of notes $\{\mathsf{N}_1^{\mathsf{old}}, \dots, \mathsf{N}_\mathsf{s}^{\mathsf{old}}\}$ each with an associated amount w_i such that

$$\sum_{i=1}^{s} \mathsf{w}_\mathsf{i} = \sum_{j=1}^{r} \mathsf{v}_\mathsf{j} + \mathsf{cross} + \mathsf{max\_fee},$$

where $\mathsf{max\_fee}$ is the maximum DUSK the sender will pay for the execution of their transaction. When creating the transaction to transfer the funds, the sender will have to nullify the set of old notes being spent and mint a new set of notes $\{\mathsf{N}_1^{\mathsf{new}}, \dots, \mathsf{N}_\mathsf{r}^{\mathsf{new}}\}$ with the corresponding values v_i, and assigned to the corresponding recipients. The most common case is $\mathsf{r} = 2$, where a sender generates a note $\mathsf{N}_1^{\mathsf{new}}$ with value v_1 for a recipient, and a second note $\mathsf{N}_2^{\mathsf{new}}$ for themselves with value $\mathsf{v}_2 = \mathsf{v}_1 - \sum_{i=1}^{s} \mathsf{w}_\mathsf{i} - \mathsf{cross} - \mathsf{max\_fee}$. We detail the steps the sender should follow hereunder.

Mint new notes. Set the parameters of each new note $\mathsf{N} \in \{\mathsf{N}_\mathsf{j}^{\mathsf{new}}\}_{\mathsf{j}=1}^{\mathsf{r}}$ as follows.

1. Set the type of transaction.
   – If the transaction is transparent, set $\mathsf{type} = 0$.
   – If the transaction is obfuscated, set $\mathsf{type} = 1$.
2. Set v to the amount of money being transferred to the new note N.
3. Set a blinding factor for the commitment and a nonce for the encryption.
   – If $\mathsf{type} = 0$, set $s = 0$.
   – If $\mathsf{type} = 1$, set $s \leftarrow \mathbb{F}_p$.
4. Compute the value commitment $\mathsf{com} = \mathsf{Com}_{\mathsf{ck}}(\mathsf{v}; s)$ following Section 3.6.
5. Encrypt the opening of com.
   – If $\mathsf{type} = 0$, then set $\mathsf{enc} = \mathsf{v}$.
   – If $\mathsf{type} = 1$, then set $r \leftarrow \mathbb{F}_p$ and $\mathsf{enc} = \mathsf{Enc}_{\mathsf{k}_{\mathsf{DH}}}(\mathsf{v}||s)$, where k_{DH} is the symmetric Diffie–Hellman key described in Section 4.2, and $\mathsf{Enc}(\cdot)$ is the encryption function from Section 3.4 that uses k_{DH} coordinates.
6. Compute the note public key npk of the recipient as described in Section 4.2.
7. Set the new note to

$$\mathsf{N} = \{\mathsf{type}, \mathsf{com}, \mathsf{enc}, \mathsf{npk}, R\}.$$

Nullify old notes. For each note $\mathsf{N} \in \{\mathsf{N}_i^{\mathsf{old}}\}_{i=1}^{s}$, with its respective position pos, being spent, the sender must do the following computations.

1. Compute $\mathsf{nullifier} = H^{\mathsf{Poseidon}}(\mathsf{npk}'||\mathsf{pos})$, where npk' is the nullification key of N as described in Section 4.2, and pos is the note's position in the Merkle tree of notes.
2. If $\mathsf{type} = 1$, then decrypt the encrypted information enc using the symmetric Diffie–Hellman key described in Section 4.2 to get an opening (w, t) for com.
3. Retrieve the old note private key nsk as described in Section 4.2.
4. Compute the hash of the transaction payload:
$$\mathsf{payload\_hash} = H^{\mathsf{BLAKE2b}}(\mathsf{tx}_{\mathsf{payload}}).$$
5. Use nsk to generate a signature $\mathsf{sig}_{\mathsf{tx}}$ on $\mathsf{payload\_hash}$ by running the double-key Schnorr signature scheme described in Section 3.7.

Prove that the transaction is performed correctly. Compute a zero-knowledge proof $\mathsf{tx\_proof}$ using the circuit depicted in Figure 2 to prove that the following conditions hold.

1. *Membership*: the sender must prove that every $\mathsf{N} \in \{\mathsf{N}_i^{\mathsf{old}}\}_{i=1}^{s}$ is included in the Merkle tree of notes. To do so, the sender takes the parameters $(\mathsf{type}, \mathsf{w}, t, \mathsf{npk}, \mathsf{pos})$ associated with N and uses them as inputs of the circuit. Inside the circuit, the box `com()` computes the commitment com using w and t, and together with the rest of elements, computes the hash
$$h = H^{\mathsf{Poseidon}}(\mathsf{type}, \mathsf{com}, \mathsf{npk}, \mathsf{pos})$$
of the note. The sender also provides a Merkle proof to show that h is in a leaf of the Merkle tree. The circuit takes $\mathsf{merkle\_proof}$, pos and h, to compute the Merkle root root, which is a public output of the circuit. Observe that all these inputs are private and hence, the proof will not reveal which note is being spent, only that it belongs to the Merkle tree of notes.
2. *Ownership*: the sender must prove that they hold the note secret key nsk of every note $\mathsf{N} \in \{\mathsf{N}_i^{\mathsf{old}}\}_{i=1}^{s}$. Instead of including their private key as an input to the circuit and computing npk inside, the sender proves (using the `verify_signature()` box inside the circuit) that they can sign a message with that key. In this case, they use the double-key Schnorr signature scheme to sign the hash of the transaction. Note that the signature can be kept as a private input of the circuit, since a correct execution of the circuit is proving that the sender holds a note's secret key associated with a note's public key that belongs to the Merkle tree of notes. Not only that, since the hash $\mathsf{skeleton\_hash}$ is already a public input for the circuit, it binds the details of the transaction with that particular signature.
3. *Nullification*: the sender must prove that $\mathsf{nullifier} = H^{\mathsf{Poseidon}}(\mathsf{npk}'||\mathsf{pos})$. Note that the sender provides the nullification key $\mathsf{npk}' = \mathsf{nsk}G'$ as an input to the circuit and not the note secret key nsk. As we just explained, the double-key

Schnorr signature guarantees that npk' is indeed $\mathsf{nsk}G'$. The result of the right `hash()` box is the nullifier, which is a public output circuit that is later included as part of the transaction.

4. *Minting*: the sender inputs the value v being transferred and the randomness s, and the circuit computes the corresponding value commitment. In this case, com is the public output that comes from the right `com()` box.
5. *Balance integrity*: we check that

$$\sum_{i=1}^{s} \mathsf{w_i} - \sum_{j=1}^{r} \mathsf{v_j} - \mathsf{cross} - \mathsf{max\_fee} = 0, \tag{1}$$

where $\mathsf{max\_fee}$ is the maximum amount of gas that the sender is willing to pay for the transaction. Furthermore, since each input and output is represented by an element of $\mathbb{F}_p$, we need to prevent overflows that could lead to generating money out of thin air. Since cross and $\mathsf{max\_fee}$ are public inputs to the circuit, the range check is verified (off-circuit) by the network. For the rest of private inputs, we include a range check on each value:

$$\begin{aligned} 0 \le w_i \le 2^{64} \qquad & \forall i = 1, \ldots, s, \\ 0 \le v_j \le 2^{64} \qquad & \forall j = 1, \ldots, r. \end{aligned}$$

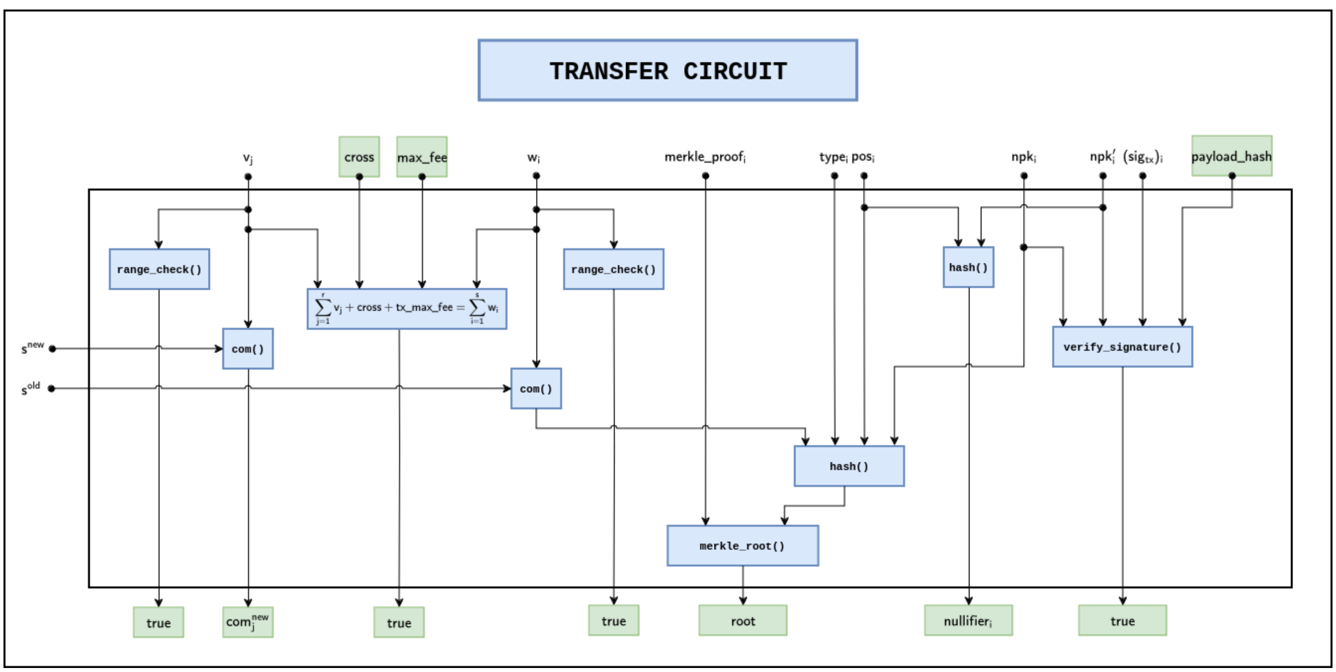

Fig. 2. Arithmetic circuit for an obfuscated Dusk transfer.

On the one side, observe that the public inputs of the circuit from Fig. 2 reveal nothing about the details of the transaction. On the other side, observe that the most important piece of information that allows a user to spend a note is nsk, which is obtained using both parts of this secret key sk, but neither key is included as an input to the circuit. Instead, Phoenix uses the double-key Schnorr signature as proof that the user holds nsk. As a consequence, users could delegate the generation of the zero-knowledge proof to a partially trusted third party, that is, a *proof helper*. This delegation would require the user to entrust the old and new values of the notes to the proof helper, but not their secret key. Benchmarks from Section 5.2 suggest that this disclosure is a reasonable trade-off with respect to the gained efficiency.

The remaining checks not verified inside the circuit are performed by the network. For instance, checking that the nullifier included in the transaction matches the output of the circuit, or that the note has the right typeset. Still, there are two computations the network cannot verify: checking that npk and enc are calculated correctly. If not, in both cases the recipient could not spend the note they own, either because they did not identify the transaction was sent to them (and can not compute nsk), or because they cannot decrypt the opening of the value commitment. Phoenix is a design in which senders can always reveal the details of their transactions at any point in time. Note that they only need to reveal the value being transferred, the randomness used in the commitment, and the public key of the recipient, but no compromised private information. Hence, in these scenarios, the sender could always prove that they computed npk and enc correctly, either by disclosing the information or producing a proof.

5 Analysis

We analyze different aspects of the Phoenix protocol described in Section 4.

5.1 Security discussion

Unforgeability. To prevent users from spending notes not in the Merkle tree (i.e. notes that do not come from an UTXO), senders are required to produce a proof that the notes being spent indeed belong to the tree. This is achieved through the Merkle proof that is part of the transaction circuit, and works as specified in Section 3.1.

Malicious helpers. Note that Phoenix allows for delegation of the work of listening to the network for transactions addressed at one, by sharing the view key $\mathsf{vk} = (a, B)$ with a network-listening helper, as explained in Section 4.2. Even with that information, it is not possible to compute the note's secret key nsk, so helpers cannot spend notes not belonging to the user being helped.

Additionally, proof computation can be delegated to a proof helper too (Section 4.3), by sharing the circuit inputs. Since nsk is not shared and cannot be computed from $\mathsf{npk}, \mathsf{npk}'$, this helper cannot spend the notes either.

Unlinkability. In the Phoenix transaction model, it is not possible to trace funds. That is, not even the sender of a note can tell when the recipient is spending the note. This is because each note uses a unique note public key. Furthermore, any revealing information is either encrypted (Section 3.4), committed in a hiding commitment (Section 3.6) or is a secret input of the transaction circuit, and thus is protected by the zero-knowledge property of the proof system (Section 3.2).

Transaction non-malleability. Phoenix prevents malicious attackers from modifying others' transactions before they are added to the ledger. This is achieved by including the hash of the transaction as a public input to the circuit, and signing it ($\mathsf{verify\_signature}()$ in Figure 2). We argue in two steps that this is enough to prevent malleability. First, an attacker cannot produce a proof of a false statement, due to the soundness of the proof system (Section 3.2). Therefore, the circuit is satisfied, which implies that the purported signature passes the verification. But then, a forged signature cannot pass the verification, due to the unforgeability of the signature scheme (Section 3.7).

Balance integrity. This property requires that no user can own more money than what he minted or received via payments from others. This is done by proving that Equation 1 holds. Such proof cannot be forged due to the soundness of the proof system (Section 3.2). Note that this is done within the circuit so that the values themselves are not leaked.

Double-spending prevention. To prevent double spending of a note in the Merkle tree, sent to a public key npk with secret key nsk and stored at position pos, a transaction includes a special value called *nullifier*, computed as

$$\mathsf{nullifier} = H^{\mathsf{Poseidon}}(\mathsf{nsk}G' \mid\mid \mathsf{pos}).$$

For a note to be spent, the nullifier must have not been used before. Note that this value is deterministic on nsk, which allows the note to be spent, and pos. Hence, if anyone tries to spend the note again, they would get the same value and the network would detect that they are trying to double spend the note. To ensure that the nullifier is really the result of this operation, the sender is required to prove correctness of this computation (right $\mathsf{hash}()$ in Figure 2).

Here G' is used instead of G because $\mathsf{nsk}G = \mathsf{npk}$ can be computed by a proof helper. This helper would realize that the public nullifier corresponds to a certain npk. Thus, they would know when a certain note is being spent, breaking unlinkability.

Anonymity and privacy. The transaction does not contain any information about the sender, recipient or value of the transfer in the clear. The notes spent by the prover are kept as secret inputs of the proof, so leakage is prevented by the zero-knowledge property of the proof system (Section 3.2). Only a unique note public key $\mathsf{npk} = H^{\mathsf{BLAKE2b}}(rA)G + B$ is revealed, which depends on the public key $\mathsf{pk} = (A, B)$ of the recipient. Due to the hash there is no way for an external attacker to link `npk` to `pk`. Finally, the value of the transfer appears only within the zero-knowledge proof, and in the commitment and encryption, which hide the information within (Sections 3.6 and 3.4, respectively). Therefore, the network does not learn any information about the sender, recipient or amount. Note that a proof helper also learns information like the values of the notes being spent and created.

5.2 Implementation

Dusk implemented the Phoenix transaction model using PlonK as its zero-knowledge proof system [9]. In particular, they use their own crate of PlonK [10]. Such libraries are coded in Rust, and are used by the different nodes of the network to generate the proof that comes out of the circuit depicted in Figure 2. Those proofs are later sent into a transaction, which calls the *transfer smart contract*. Such smart contract is executed and validated by the network validators, which will run an instance of Rusk [11], the architecture that builds the node workflows. This architecture includes a virtual machine, the Rusk VM, and one of its tasks is to execute the smart contracts, coded in Rust as well but compiled into WebAssembly. As such, the proofs computed by the users are verified by the transfer contract in the Rusk VM.

6 Conclusions

We reviewed the Phoenix transaction model implemented by Dusk. We showed that the privacy model is capable of issuing obfuscated transactions across Dusk's blockchain by means of different cryptographic primitives. We confirmed the soundness of the protocol, analyzed its features, and benchmarked the performance of the protocol using their implementation, demonstrating its feasibility both in desktop and mobile hardware. Moreover, we explained the novel delegation model that allows users to delegate the heavy part of the transactions computations to partially trusted environments.

We have also seen how a different circuit for each amount of notes to be nullified and minted is required. This is a drawback as the storage required to store all the generated proving and verifying keys grows per each additional circuit we want to support. To solve this, an interesting future work would be to find a way to merge the notes to be nullified into one, before putting them into the circuit.

References

1. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance), available online: `http://data.europa.eu/eli/reg/2016/679/oj` (accessed on 1 June 2022)

2. AB-375 Privacy: personal information: businesses. An act to add Title 1.81.5 (commencing with Section 1798.100) to Part 4 of Division 3 of the Civil Code, relating to privacy. (2018), available online: `https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375` (accessed on 1 June 2022)
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic sponges (2011)
4. Bonneau, J., Narayanan, A., Miller, A., Clark, J., Kroll, J.A., Felten, E.W.: Mixcoin: Anonymity for bitcoin with accountable mixes. In: International Conference on Financial Cryptography and Data Security. pp. 486–504. Springer (2014)
5. Bootle, J., Cerulli, A., Chaidos, P., Groth, J., Petit, C.: Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 327–357. Springer (2016)
6. Bowe, S., Grigg, J.: Implementation of the BLS12-381 pairing-friendly elliptic curve construction Available online: `https://github.com/zkcrypto/bls12_381` (accessed on 1 June 2022)
7. Bowe, S., Ogilvie-Wigley, E., , Grigg, J.: Implementation of the Jubjub elliptic curve group Available online: `https://github.com/zkcrypto/jubjub` (accessed on 1 June 2022)
8. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6), 644–654 (1976)
9. Dusk Network: Phoenix core. GitHub, available online: `https://github.com/dusk-network/phoenix-core` (accessed on 1 June 2022)
10. Dusk Network: Plonk. GitHub, available online: `https://github.com/dusk-network/plonk` (accessed on 1 June 2022)
11. Dusk Network: Rusk. GitHub, available online: `https://github.com/dusk-network/rusk` (accessed on 1 June 2022)
12. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)
13. Fuchsbauer, G., Kiltz, E., Loss, J.: The algebraic group model and its applications. In: Annual International Cryptology Conference. pp. 33–62. Springer (2018)
14. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PlonK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019)
15. Goodell, B., Noether, S., Blue, A.: Concise linkable ring signatures and forgery against adversarial keys. Cryptology ePrint Archive (2019)
16. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for zero-knowledge proof systems. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 519–535 (2021)
17. Groth, J.: On the size of pairing-based non-interactive arguments. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 305–326. Springer (2016)
18. Herrera-Joancomartí, J.: Research and challenges on bitcoin anonymity. In: Data privacy management, autonomous spontaneous security, and security assurance, pp. 3–16. Springer (2014)
19. Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash protocol specification. GitHub: San Francisco, CA, USA p. 1 (2016)
20. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: International conference on the theory and application of cryptology and information security. pp. 177–194. Springer (2010)

21. Katz, J., Lindell, Y.: Introduction to modern cryptography. CRC press (2020)
22. Maharramov, T.: Confidential security contract standard Available online: https://dusk.network/uploads/Confidential_Security_Contract_Standard_v2_0.pdf (accessed on 1 June 2022)
23. Maharramov, T., Khovratovich, D., Francioni, E.: The Dusk Network whitepaper Available online: https://dusk.network/uploads/The_Dusk_Network_Whitepaper_v3_0_0.pdf (accessed on 1 June 2022)
24. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Conference on the theory and application of cryptographic techniques. pp. 369–378. Springer (1987)
25. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review (2008)
26. Noether, S., Goodell, B.: Triptych: logarithmic-sized linkable ring signatures with applications. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology, pp. 337–354. Springer (2020)
27. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Annual international cryptology conference. pp. 129–140. Springer (1991)
28. Pointcheval, D., Stern, J.: Security proofs for signature schemes. In: International conference on the theory and applications of cryptographic techniques. pp. 387–398. Springer (1996)
29. Reid, F., Harrigan, M.: An analysis of anonymity in the bitcoin system. In: Security and privacy in social networks, pp. 197–223. Springer (2013)
30. Saarinen, M.J.O., Aumasson, J.P.: The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693 (Nov 2015), available online: https://www.rfc-editor.org/info/rfc7693 (accessed on 1 June 2022)
31. Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE symposium on security and privacy. pp. 459–474. IEEE (2014)
32. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Conference on the Theory and Application of Cryptology. pp. 239–252. Springer (1989)
33. Van Saberhagen, N.: Cryptonote v 2.0 (2013)