Dusk Whitepaper

Marta Bellés–Muñoz
Dusk, Netherlands
marta@dusk.network

Hein Dauven
Dusk, Netherlands
hein@dusk.network

Emanuele Francioni
Dusk, Netherlands
emanuele@dusk.network

Federico Franzoni
Dusk, Netherlands
federico@dusk.network

Abstract. Dusk is designed to bridge the gap between decentralized platforms and traditional finance markets by providing a privacy-focused, compliance-ready blockchain. It integrates features such as confidential transactions, auditability, and regulatory compliance into its core infrastructure. One of its key innovations is the succinct attestation protocol, which ensures transaction finality in seconds, aligning with the high throughput needs of financial systems. Combined with its efficient peer-to-peer network, and two transaction models, Dusk provides a robust, privacy-centric blockchain solution that meets both the performance and regulatory demands of the financial sector.

Last update: November 29, 2024.

Table of Contents

1 Introduction

Blockchain technology has brought new opportunities for decentralization and security in various industries, especially in financial markets. However, one of the critical issues is how to balance transparency and privacy, especially when dealing with sensitive financial information [1]. Dusk aims to solve this challenge by introducing a blockchain protocol specifically designed for regulated financial markets, providing privacy, compliance, and scalability to meet the needs of traditional financial institutions.

Dusk focuses on improving privacy in the execution of financial transactions without sacrificing regulatory compliance. Many current public blockchain platforms, such as Ethereum and Bitcoin, struggle with privacy, transaction finality, and efficiency when applied to traditional finance [2,18]. Despite their capabilities in supporting decentralized applications and financial transactions, they often face difficulties in handling private transaction details while also meeting the regulatory requirements necessary for financial institutions.

Several blockchain protocols have made significant advances in privacy, including Zcash and Monero. These platforms use advanced cryptographic techniques, such as zk-SNARKs and ring signatures, to obscure transaction details, including sender, receiver, and transaction amounts [26,29]. While these platforms are groundbreaking for personal privacy, they lack necessary features for integration with traditional finance, such as clear regulatory frameworks, auditability, and smart contract capabilities with confidential transactions. Dusk aims to integrate these features, making it suitable for regulated financial markets.

One of Dusk's contributions is its succinct attestation protocol, a new consensus mechanism that guarantees transaction finality within seconds. The consensus model is designed to meet the high-throughput and low-latency requirements of the financial sector, with mechanisms to ensure the scalability of the network while maintaining decentralization. As the underlying communication layer, Dusk uses the Kadcast [22] protocol, whose efficient and secure message propagation ensures that information is disseminated across the network quickly and reliably.

Additionally, Dusk makes use of two transaction models, Moonlight and Phoenix. While Moonlight is a transparent, account-based model, Phoenix is a UTXO-based model that supports both transparent and obfuscated transactions. The combination of these two models makes Dusk highly suitable for financial transactions that require privacy without sacrificing compliance, as regulators can access necessary data while still ensuring confidentiality for the general public.

Dusk also integrates the Zedger protocol, which is designed to support confidential smart contracts tailored for financial applications. Zedger focuses on security token offerings and financial instruments, ensuring regulatory compliance while enabling the execution of private transactions and contracts. This way, Dusk aims to provide a blockchain infrastructure that aligns with traditional finance's legal and regulatory requirements.

1.1 Related work

When compared to platforms such as Ethereum, Cardano, or privacy-focused blockchains like Monero, Dusk's focus is significantly more aligned with meeting regulatory needs while supporting private and scalable transactions. Ethereum, for instance, has become a popular choice for decentralized finance (DeFi), but its transparency and lack of built-in privacy faces challenges for handling sensitive financial data. While second-layer solutions, such as zk-rollups, have emerged to mitigate these issues, Dusk offers a more integrated approach, embedding privacy within the core of the network's protocol. Similarly, privacy-centric platforms like Zcash and Monero are optimized for individual privacy but lack the necessary infrastructure for compliance in regulated industries, such as securities trading or financial auditing.

1.2 Paper organization

The aim of this whitepaper is to give an overview of the technical foundations of the Dusk network. The paper is organized as follows. In Section 2 we discuss Kadcast, the peer-to-peer network implemented within Dusk network. In Section 3 we introduce the SA consensus algorithm. In the following Section 4 we present the Moonlight and Phoenix transaction models. In Section 5 we explore the energy-efficiency design principles embedded within the Dusk network. In Section 6 we give some implementation details about Dusk's virtual machine and genesis contracts. We conclude with some general remarks in Section 7.

2 Peer-to-peer communication

Dusk uses the Kadcast [22] *peer-to-peer* (P2P) protocol as the underlying communication layer for broadcasting blocks, transactions, and consensus votes. Kadcast is based on the Kademlia *distributed hash table* (DHT) protocol [21], which optimizes network performance by reducing redundancy and message collisions. Kadcast's implementation addresses the limitations of traditional P2P networks, such as high bandwidth consumption and delayed message delivery, by introducing a structured, efficient broadcast mechanism that ensures reliable and timely message propagation across all nodes.

These features are particularly important for Dusk, specially in environments where network resources may be constrained, and low-latency communication is a priority. Furthermore, the network's privacy requirements are well-aligned with Kadcast's structure, as the protocol naturally obfuscates message origin points by propagating messages across a network of nodes without relying on direct peer-to-peer connections.

2.1 Kadcast architecture

Kadcast builds upon the principles of the Kademlia DHT protocol, organizing network nodes in a hierarchical, tree-like structure where each node maintains a routing

table sorted by XOR distance between its node ID and other nodes. This distance metric ensures that nodes are able to efficiently locate and communicate with other peers within the network. Routing tables are divided into *buckets*, which store contact information for peers at various distances. By making use of Kademlia's XOR distance, Kadcast ensures that the farther a node is from the sender, the fewer intermediate nodes are required to relay the message, reducing message propagation time.

Kadcast's primary innovation lies in its broadcast mechanism, which optimizes the distribution of messages by limiting redundant transmissions. Instead of broadcasting to all neighboring nodes, each node forwards messages only to selected peers at increasing XOR distances, creating an efficient cascading effect. This significantly reduces bandwidth usage compared to traditional flooding or gossip-based P2P networks, where each message is broadcast to all neighbors regardless of the distance or the network structure.

Multicast trees. One of Kadcast's key features is its use of *multicast trees* to organize message dissemination. Multicast groups are formed based on node proximity in the Kademlia DHT structure. When a node sends a message, it does so through its closest peers, which in turn propagate the message to their neighbors at increasing XOR distances. This structured propagation allows for optimal coverage of the network with minimal overhead, ensuring that each node receives the message with the fewest possible relays. By structuring message dissemination this way, Kadcast drastically reduces the overall number of transmissions required to propagate data across the network.

2.2 Fault tolerance and network dynamics

Kadcast is inherently resilient to network changes and failures due to its reliance on Kademlia's DHT. In highly dynamic networks where nodes frequently join and leave, Kadcast ensures that message propagation remains robust. Nodes can dynamically update their routing tables by removing failed peers and replacing them with new ones. This constant re-evaluation and updating of routing tables allow Kadcast to maintain high availability and fault tolerance.

Additionally, the use of multiple peers within buckets means that if one node fails to forward a message, alternative paths can be used, ensuring that the message still reaches its intended recipients. This fault tolerance makes Kadcast particularly suitable for decentralized environments, where network conditions can be unpredictable, and nodes may be offline or unreliable.

2.3 Security and privacy

To mitigate the risk of malicious behavior such as Sybil attacks, Kadcast uses various mechanisms for validating message authenticity and ensuring that nodes cannot easily disrupt message propagation. In paritcular, messages are signed, and nodes verify the signatures before forwarding them, ensuring that only legitimate messages are propagated through the network.

Moreover, Kadcast also enhances privacy by obfuscating the origin of messages as they propagate. Since nodes forward messages to a select set of peers at increasing distances, it becomes difficult for adversaries to trace the exact source of a message. This is particularly important in the Dusk network, where privacy-preserving transactions and confidential data exchanges are a key focus.

3 Consensus mechanism

Succinct attestation (SA) is a permissionless, committee-based proof-of-stake consensus protocol [5]. The protocol is run by Dusk stakers, known as *provisioners*, which are responsible for generating and validating new blocks. Provisioners participate in turns, based on the *deterministic sortition* (DS) algorithm, which is used to select a unique block generator and unique voting committees among provisioners for each new block, in a decentralized, non-interactive way.

3.1 Provisioners

A provisioner is a user that locks a certain amount of DUSK as *stake*. Any user can do so by broadcasting a *stake transaction*. Formally, a stake S consists of a pair $S = (\texttt{amount}, \texttt{height})$, where `amount` is the amount of staked DUSK and `height` is the block height at which the stake transaction was included. There is a minimum stake set by the global parameter `minStake` (currently set to 1000 DUSK). Provisioners can unlock their stake by broadcasting an *unstake transaction*.

Epochs and eligibility. Only eligible stakes can participate in the DS algorithm. For that, we associate to every stake $S = (\texttt{amount}, \texttt{height})$ a *maturity period* M defined as

$$M = 2 \times \texttt{epoch} - (\texttt{height} \mod \texttt{epoch}),$$

where an `epoch` is a global parameter that corresponds to a fixed number of blocks (currently set to 2160 blocks). The stake S is eligible in round R if it satisfies the following two conditions:

$$\texttt{amount} \geq \texttt{minStake} \quad \text{and} \quad R > \texttt{height} + M.$$

That is, a stake becomes eligible at the beginning of a new epoch, after a maturity period that includes the remainder of the epoch in which the stake transaction was included and another full epoch. As a consequence, all new stakes become eligible at the beginning of an epoch.

3.2 Consensus algorithm

The SA protocol proceeds in *rounds*, with each round adding a new block to the chain. In turn, each round proceeds in *iterations*, with each iteration aiming at generating a candidate block and reaching agreement among provisioners. Each iteration is composed of three phases, or *steps*:

1. *Proposal*: in this step, a provisioner is randomly extracted via the DS algorithm and appointed as *block generator*. This provisioner is responsible for generating a new *candidate block* for the round, and broadcast it to the network using a *candidate* message. If, a candidate block is produced, or received from the network, within a certain timeout, the step outputs the block; otherwise, it outputs `NIL`. The output of this step is passed on as input to the validation step, where a committee of provisioners verifies and votes on its validity.

2. *Validation*: in this step, a committee of provisioners is randomly selected via the DS algorithm to vote on the validity of the output from the proposal step. If the result was `NIL`, the provisioner's vote will be `NoCandidate`. Otherwise, the provisioner verifies the candidate's validity against the previous block (the current *tip* of the chain). If the candidate block is valid, the provisioner votes `Valid`, otherwise it votes `Invalid`. Each provisioner in the committee broadcasts their vote using a *validation* message. A quorum is achieved with a supermajority (2/3 of the committee) of `Valid` votes or a majority ($1/2 + 1$ of the committee) of `Invalid` or `NoCandidate` votes. If this is the case, the step outputs a `ValidationResult` structure containing the quorum-reaching vote, and the aggregated signatures of the voting provisioners. If the step timeout expires before reaching a quorum, it outputs `NoQuorum`. The result of this step is passed as input to the ratification step, which ensures a majority of provisioners has accepted the validation result.

3. *Ratification*: in this step, a new committee of provisioners is randomly selected via the DS algorithm to vote on the outcome of the validation step. Each provisioner casts its vote using a *ratification* message based on the `ValidationResult` output of the previous step. If a quorum was reached, they will confirm the quorum-reaching vote, otherwise they will vote `NoQuorum`. As with validation, provisioners collect votes from the network until a quorum is reached or the step times out. As before, a quorum is achieved if a supermajority (2/3) of `Valid` votes is reached, or fails if a majority ($1/2 + 1$) of `Invalid`, `NoCandidate`, or `NoQuorum` votes is reached. If a quorum of `Valid` votes is reached, the step outputs a `Success` result; if any other quorum was reached, it outputs a `Fail` result; if no quorum was reached, the step result is `unknown`. In case of `Success` or `Fail` result, a *quorum* message is broadcast, containing the winning vote and the aggregated signatures of both validation and ratification step voters.

If the ratification step outputs a `Success` result, or if a `Success` *quorum* message is received, the corresponding candidate block is accepted as the new tip, and the round is terminated. Conversely, if the ratification result is `Fail` or `unknown`, a new iteration is executed, where a new candidate can be proposed, validated and ratified. The maximum number of iterations executed in a single round is determined by a global parameter (currently set to 50).

3.3 Voting committees

A *voting committee* consists of an array of provisioners entitled to cast votes in the validation or ratification steps. Each member of the committee is assigned credits

that define their power in the committee. Members are ordered by their inclusion, starting with index 0. Voting committees have a fixed number of credits defined by a global parameter (currently set to 64). When votes are counted, they are weighted by the number of credits each member holds. More specifically, each vote is multiplied by the power of the voter in the committee. For example, if a member has 3 credits, their vote is counted 3 times.

Each vote is signed by the provisioner using a BLS signature [3]. This allows all votes for a specific step to be aggregated into a single signature, which can be easily propagated and verified. To verify aggregated votes, a *bitset* is used to indicate which members of the committee are included. For example, if the committee consists of the array of provisioners $\mathsf{committee} = [P_0, P_1, P_2, P_3]$, the bitset $[0, 1, 0, 1]$ would indicate that the subcomittee is conformed by provisioners P_1 and P_3.

3.4 Attestations

An *attestation* is proof of a quorum being reached in a specific iteration. It contains the quorum-reaching vote and the aggregated signatures from the validation and ratification steps. Attestations proving a supermajority of `Valid` votes are called *success attestations*, while those proving a majority of `Invalid`, `NoCandidate`, or `NoQuorum` are called *fail attestations*. Note that, if more votes than the quorum threshold are cast, it is possible to have multiple valid attestations for the same iteration. For this reason, each block contains an attestation of the previous block, called *block certificate*, that the determines a unique set of voters. This unique attestation will be used to determine rewards and penalties (see Section 3.9).

3.5 Deterministic sortition

The DS algorithm is a non-interactive process used in the SA protocol to select the block generator for the proposal step and the members of the voting committees for the validation and ratification steps. The DS algorithm relies on the *deterministic extraction* (DE) algorithm, which selects provisioners with a frequency proportional to their stake. The algorithm assigns *credits* to eligible provisioners based on a pseudo-random *score* value, which is unique for each extraction. This deterministic process ensures that the selection is fair, reproducible, and weighted by the provisioners' stakes.

Deterministic extraction. The DE algorithm works by iterating through the ordered list of eligible provisioners, comparing each provisioner's weight (i.e., their stake) to the score. If a provisioner's weight is greater than or equal to the score, that provisioner is selected. If not, the provisioner's weight is subtracted from the score, and the algorithm continues with the next provisioner.

Each provisioner can receive one or more credits, depending on their stake. In particular, the algorithm follows a weighted distribution that favors provisioners with higher stakes. To balance the sortition process, the weight of each provisioner, initially set to the value of their stake, is reduced by 1 Dusk each time they are assigned a credit, reducing their chances of being selected further. This ensures that provisioners, on average, participate in committees with a frequency proportional to their stake.

Score. The score used in the DE algorithm is generated deterministically via the SHA3 hash function [28], combining several inputs such as the *seed* from the previous block, the current round and step numbers, and the number of the credit being assigned. This generates a unique score for each extraction, ensuring randomness in the process.

Seed. The seed value used in the score calculation adds unpredictability to the process. It is included in the block header and updated with each new block by the block generator. In particular, the seed for a block is the signature of the block generator on the seed of the previous block. This makes it impossible to predict future scores and, consequently, prevents pre-calculating future block generators or committee members.

3.6 Emergency mode

In extreme cases, where most provisioners are offline or isolated, multiple consecutive iterations may fail due to the absence of the appointed block generators or voting committee members. In such situations, after a certain threshold of failed iterations (currently set to 16), the SA protocol transitions into *emergency mode*. When this mode is activated, step timeouts are disabled and iterations continue indefinitely until a candidate block is generated and a quorum is achieved in both the validation and ratification steps. Each step progresses only when the previous one succeeds, meaning `NoCandidate` and `NoQuorum` votes are disabled during this mode. An ongoing emergency iteration is called an *open iteration*.

In emergency mode, new iterations are initiated after the maximum timeout for all steps elapsed, but each iteration remains active until a quorum is reached on a candidate block. Consequently, multiple open iterations can run concurrently, increasing the likelihood of producing a valid block for the round. As a downside, this also raises the possibility of *forks*, as consensus may be reached on multiple candidate blocks. These forks are resolved by selecting the candidate from the lowest iteration. Once a block is accepted for the round, all open iterations are terminated.

When the last iteration is reached, all open iterations continue indefinitely until either a candidate block achieves quorum or an *emergency block* is generated. The emergency block is a special empty block produced by a Dusk node on explicit request by provisioners. In particular, when the maximum iteration time of the last iteration has elapsed, each provisioner can broadcast an *emergency block request* (EBR) to request the production of the emergency block. The block is created if requested by a set of provisioners holding the majority of the total stake in the network. This block contains no transactions but includes a new seed signed by Dusk, verifiable with the Dusk's public key, which is a global parameter. Additionally, the block includes a proof of the EBRs through aggregated signatures of the requests. Upon receiving an emergency block, nodes accept it into their local chain and proceed to the next round.

3.7 Fallback

Due to the asynchronous nature of the protocol, *forks* can be produced when more than one candidate block reaches consensus in the same round. This is typically due to delayed or lost messages caused by network congestion.

When a fork is detected, nodes normally choose the block produced at the lowest iteration. In particular, a block B at iteration $I \geq 0$ can be replaced by a block B' with iteration $I' \leq I$. This is handled by the *fallback* procedure, which reverts the local chain to the block before B, and then accepts B'. This procedure also discards all successors of block B in the chain.

Note that due to fallback, any block produced at iteration $I \geq 0$ can potentially be reverted if a lower-iteration block also reaches consensus. On the other hand, blocks that reach consensus at iteration $I = 0$ cannot be replaced by any lower-iteration block, but they may still be reverted if one of their ancestors is reverted.

3.8 Rolling finality

To help nodes determine the level of stability of a block in their local chain, we leverage the *rolling finality* algorithm. This algorithm leverages block attestations (see Section 3.4) and is based on the following observations:

- A block can be reverted only if it has iteration $I > 0$ and there is at least a lower iteration without a fail attestation (i.e., whose result is `unknown`). Note that fail attestations, included in the candidate block, ensure the relative iterations cannot have reached consensus on a candidate.
- Any successor of a block B confirms that a set of the eligible provisioners has accepted B into their local chain.
- Since generators and voting committees are selected at random, a successor reaching consensus implies that there is a significant number of provisioners that have accepted B.
- Each further successor of B extends the set of known provisioners to have accepted B, reducing the likelihood of a competing fork progressing (the more the provisioners working on top of B the less the ones working on a competing block).
- While all attested successors confirm B, only iteration 0 successors or those with all previous iterations marked as `Fail` can safely confirm B's finality. Other blocks may still be reverted, making them unreliable indicators of B's final status.
- As more rounds/iterations are executed, the probability of receiving a competing sibling block decreases.

Consensus state. Each block in the local chain is in one of the following states:

1. `accepted`: the block has a success attestation but can be replaced by a lower-iteration block with a success attestation. A block is marked as `accepted` if it has iteration $I > 0$ and not all previous iterations have a fail attestation.
2. `attested`: the block has a success attestation, and all previous iterations have fail attestations. It cannot be replaced by a lower-iteration block. A block is `attested` if it has iteration $I = 0$ or all previous iterations failed.
3. `confirmed`: either `accepted` or `attested`, the block is unlikely to be replaced unless an ancestor is replaced. It is confirmed by the finality rules, which depend on subsequent blocks.
4. `final`: the block is `confirmed`, and its parent is `final`. It cannot be replaced under any circumstances. A block is only marked `final` if all its ancestors are also final.

Finality rules. Let n denote the count of previous non-attested iterations of a block. A new block is marked as `attested` if $n = 0$, otherwise it is marked as `accepted`. An `attested` block becomes `confirmed` when is has a successor marked as `attested` or `confirmed`. An `accepted` block becomes `confirmed` after $2 \times n$ consecutive `attested` or `confirmed` blocks. Finally, a `confirmed` block becomes `final` if its parent is `final`.

For example, if a block has iteration $I = 5$ and only two previous iterations have fail attestations, it is marked `accepted` and becomes `confirmed` after four more blocks (2×2) are either `attested` or `confirmed`. When marked as `confirmed`, it becomes `final` if its parent is also `final`. These rules are summarized in Figure 1.

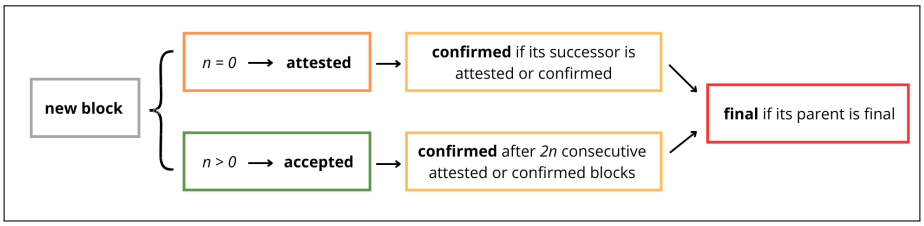

Fig. 1. Finality rules.

3.9 Incentives

Given the nature of the SA consensus protocol, it is of paramount importance that provisioners participate in the protocol when selected, as offline provisioners can slow down the network or even stop block production. To encourage participation, provisioners are rewarded for their participation and penalized for misbehavior. Rewards and penalties are applied based on attestations and proofs of faults included in each block, and handled by the virtual machine (VM) during the state transition of the accepted block.

Future-generator incentive problem. One of the problems inherent to our protocol design is due to predictability of generators for all iterations within a single round. Due to this, higher-iteration generators might be incentivized to let previous iterations fail so as to get win the block reward. To mitigate this incentive, we introduce several mechanisms:

- *Voter rewards*: provisioners earn rewards for voting in the validation and ratification steps; this gives provisioners a choice between a readily-available, more-probable reward and a potential, less-probable future reward.
- *Extra credits reward*: part of the generator's reward is conditioned by the inclusion of all known votes; this incentives the generator to include as many votes as possible, further strengthening the probability for voters to get their reward.
- *Exclusion of next-iteration generators*: at each iteration, the next-iteration generator is excluded from voting; this mitigates the extra incentive to abstain from voting due to the higher probability of getting the generator reward.

– *Limit on iterations*: the maximum number of iterations is limited to a lower value so as to reduce the number of future generators.

Rewards. Block rewards consist of newly minted DUSK and transaction fees, distributed as follows:

– 80% to the block generator,
– 10% to the voting committee,
– 10% to Dusk.

The block generator's reward is split into a fixed portion (70% of the block reward) and a variable portion (10%), which depends on the number of votes included in the block certificate. This variable portion is divided into *quotas*, which in turn depend on the voter's credits, with more votes making the generator earn a larger share. If all votes are included, the generator earns the full 80%. Voter rewards (10%) are distributed proportionally to the number of credits each voter holds in the committee, based on the previous block's certificate.

Note that rewards are proportional to voters' credits, reflecting the greater influence of those with more credits in reaching quorum. Additionally, since voters with more credits are more likely to become future block generators, a larger reward disincentivizes them from skipping voting.

Faults and penalties. Provisioners in the network can commit two types of faults: *minor* and *major faults*. Minor faults, such as failing to broadcast a candidate block, result in suspension and *soft slashing*. Major faults, which include broadcasting invalid blocks, double voting, or broadcasting two different candidate blocks for the same iteration, incur *hard slashing*.

Penalties are imposed through two mechanisms: *suspension* and *slashing*. Suspension excludes the provisioner from the DS process for a defined number of epochs, with suspensions increasing for consecutive faults. Soft slashing locks a portion of the stake, reducing the provisioner's weight in the DS process. Finally, hard slashing, applied to major faults, burns part of the provisioner's stake, with the penalty amount increasing based on the severity and recurrence of faults. These mechanisms ensure network stability and integrity by penalizing misbehavior.

4 Transactions

Transactions are used to transfer DUSK between accounts, deploy new smart contracts, or call functions on existing contracts. Dusk uses two types of transactions models, both handled by the *transfer contract*, which serves as the entry point to the Dusk blockchain. These models are *Moonlight* and *Phoenix*. Moonlight is a transparent, account-based model similar to Ethereum. Phoenix is a UTXO-based model that supports both transparent and obfuscated transactions. These models also represent the only two mechanisms by which transaction fees can be paid in the Dusk blockchain. In the following subsections we delve into the details of each model, but

we now first give an intuitive idea of how they work and the main differences between them.

At its core, any transaction on the Dusk network must prove that the following properties are satisfied:

- *Ownership*: the sender is the owner of the assets being transferred.
- *Balance integrity and payment of fees*: the transaction is supported by sufficient funds to cover the transfer amount and associated fees.
- *Double spending prevention*: the funds are not being spent more than once.
- *Malleability protection*: the transaction was not altered after its creation.

How this is handled in Moonlight and Phoenix is very different. On the one hand, in the Moonlight model, each user account is associated with a public key, whose private key is only known to the user. Ownership is proven through a digital signature with the corresponding user's public key. Malleability is also prevented via a signature on a cryptographic hash of the transaction, ensuring that any tampering would invalidate the signature. Because account states are public, the network can check account balances to ensure that there are sufficient funds to cover both the transaction amount and the required fees. At the same time, the network automatically prevents the same funds from being used in more than one transaction.

On the other hand, the Phoenix model, particularly in its obfuscated mode, uses zero-knowledge (ZK) proofs to ensure the same properties while maintaining privacy. That is, the network does not verify transaction details directly. Instead, it checks the validity of the ZK proof, which guarantees that all the required properties are met without exposing the underlying transaction data. More specifically, ownership and malleability prevention is also proven through signatures, similar to Moonlight, but verified using a ZK circuit. The balance is also verified privately within the ZK proof, ensuring that the sum of the outputs, including transaction fees, equals the sum of the input amounts. Double spending is prevented by the use of *nullifiers*, which uniquely identify each UTXO to ensure they can only be spent once.

The dual-model approach of Moonlight and Phoenix provides Dusk with the flexibility to handle both transparent and private transactions while ensuring that all fundamental transaction properties are upheld.

4.1 Moonlight

Moonlight is an account-based model for executing transactions and deploying smart contracts on the Dusk blockchain. The transfer contract maintains a global state that records the balance and state of each account.

Notation. We consider $\mathbb{G}$ the largest subgroup of points of prime order of the BLS12-381 elliptic curve and G a fixed generator of $\mathbb{G}$ [24, Section 4.2.1].

User accounts. In Moonlight, users have a public key that works as an *account* identifier. The associated private key is used to authorize transactions from the associated account. More specifically, each user holds the following pair of keys:

– Public key: $\mathsf{pk} = A$, with $A \in \mathbb{G}$,
– Secret key: $\mathsf{sk} = a$, such that $A = aG$.

The network maps each public key to a *nonce* and *balance*. The nonce is a counter that tracks the number of transactions sent from the account and prevents replay attacks. The balance is the amount of DUSK held by the account.

Moonlight transactions. A Moonlight transaction includes the following fields:

$$\mathsf{tx}^{\mathsf{moonlight}} = \{\mathsf{from}, \mathsf{to}, \mathsf{value}, \mathsf{nonce}, \mathsf{deposit}, \mathsf{data}, \mathsf{gas\_limit}, \mathsf{gas\_price}, \mathsf{signature}\}.$$

The parameters above correspond to the following: from and to correspond to the public key of the sender and the recipient, respectively; value is the amount of DUSK being transferred from the sender's to the recipient's account; nonce is a value that ensures that each transaction is unique; deposit and data are optional fields that are used to send DUSK to contracts, call contract functions, deploy new contracts, or add some data to the transaction; the field gas_limit specifies the maximum of gas the transaction is allowed to consume; gas_price is the amount the user is willing to pay per unit of gas; and finally signature is the user's signature on the hash of the previous fields signed with the public key specified in from.

When the network receives a Moonlight transaction, the network performs several checks to ensure its validity. First, they verify that the sender's account has sufficient funds to cover the transaction's value, deposit, and maximum gas costs (calculated as gas_limit $\times$ gas_price). The transaction fields are then hashed, and signature is verified against the from account's public key to ensure authenticity. The network also checks that the transaction's nonce is exactly one more than the current nonce of the from account (which will be incremented upon acceptance of the transaction). It is possible that the value or deposit fields may be left empty, which is permissible. If the smart contract execution is reverted, the corresponding amount is refunded to sender's account. Additionally, any unused gas is also refunded. Upon acceptance of the transaction, the nonce associated to the sender's account is incremented, and the balances of the accounts and contracts involved in the transaction are updated accordingly.

Security and privacy. The Moonlight transaction model satisfies the properties of unforgeability (secured by digital signatures), double-spending prevention, transaction non-malleability (secured by the digital signature on the hash of the transaction), and replay-attacks prevention (secured by the nonce).

4.2 Phoenix

The Phoenix transaction model is the mechanism used for executing obfuscated transactions on the Dusk blockchain. It is based in the unspent transaction output (UTXO) architecture similar to that used by Bitcoin but incorporates enhanced privacy features inspired by Zcash [16] and CryptoNote [29].

In the Phoenix protocol, UTXOs are referred to as *notes*, which are stored in a Merkle tree called *Merkle tree of notes*. Each note is associated with a public key, that we refer to as the *note public key*, and only the note's owner can compute the corresponding secret key. When a user spends a note, they must generate the corresponding nullifier, which is a deterministic value derived from the note's secret key that ensures the note cannot be spent again. The network keeps track of all nullifiers in a list, thereby preventing double spending. Importantly, the network does not identify which specific note within the entire Merkle tree has been spent, the only public information is that a note associated with a particular nullifier has been spent. As a consequence of this, notes are not removed from the Merkle tree. Instead, the tree continues to grow as new transactions are added to the network. Additionally, a Phoenix transaction contains enough information for the recipient to identify that they are the intended recipient of a note, what the amount is, who is the sender, and be assured that they are the only party that can spend that particular note.

In the following sections we go into the technical details of how the Phoenix protocol operates. For the sake of clarity, we have omitted certain details, but these can be found in the technical paper [6] and the security analysis report [8].

Notation. We consider $\mathbb{G}$ the largest subgroup of points of prime order of the Jubjub curve elliptic and G a fixed generator of $\mathbb{G}$ [16, Section 5.4.9.3].

User keys. In Phoenix, each user has a set of static keys for long-term identity. More specifically, the static keys of a user consist of the following tuples:

- Public key: $\mathsf{pk} = (A, B)$, where $A, B \in \mathbb{G}$.
- Secret key: $\mathsf{sk} = (a, b)$ such that $A = aG$ and $b = bG$.
- View key: $\mathsf{vk} = (a, B)$.

Phoenix notes. A *note* is a data object with the following structure:

$$\mathsf{note} = \{\mathsf{type}, \mathsf{com}, \mathsf{enc}, \mathsf{npk}, \mathsf{R}, \mathsf{enc}^{\mathsf{sender}}\}.$$

The parameters above correspond to the following: type, indicates if the note is transparent or obfuscated; com is a commitment to the value of the note, and 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, which is computed using R and the public key of the recipient; and $\mathsf{enc}^{\mathsf{sender}}$ encrypts the sender's public key.

Particularly, the note public key npk works as a one-time key to ensure unlinkability between transactions. It is computed by the sender using the formula

$$\mathsf{npk} = H(rA)G + B,$$

where (A, B) is the public key of the recipient, H is a cryptographic hash function, and r is such that $\mathsf{R} = rG$.

Everytime a transaction is sent to the network, a user can identify if it is addressed to them taking their view key (a, B) and checking if the npk specified in the

transaction satisfies the relation $\mathsf{npk} = H(a\mathsf{R})G + B$. In order to spend the received note, the user will have to compute the associated note secret key, which is

$$\mathsf{nsk} = H(a\mathsf{R}) + b.$$

This way the note secret and public keys satisfy the relation $\mathsf{npk} = \mathsf{nsk}G$ and the note secret key can only be computed using the whole user's secret key (a, b).

Phoenix transactions. A Phoenix transaction contains the following data:

$$\mathsf{tx}^{\mathsf{phoenix}} = \{\mathsf{root}, \mathsf{nullifiers}, \mathsf{new\_notes}, \mathsf{deposit}, \mathsf{data}, \mathsf{gas\_limit}, \mathsf{gas\_price}, \mathsf{proof}\}.$$

The first parameter root is a recent root from the Merkle tree of notes that is used by the network to verify that the notes being spent belong to the Merkle tree of notes; the nullifiers is a set of values that nullify the notes being spent; if the transaction is successful, new_notes are added to the Merkle tree of notes; as before, deposit and data are optional fields that are used to send DUSK to contracts, call contract functions, deploy new contracts, or add some data to the transaction; the field gas_limit specifies the maximum of gas the transaction is allowed to consume; gas_price is the amount the user is willing to pay per unit of gas; finally proof is a ZK proof that proves that the transaction has been performed following the network rules.

Unlike in Moonlight, where the network performs the checks directly, in a Phoenix transaction, the network's role is primarily to verify the ZK proof using a set of public inputs. These public inputs include those specified in the transaction, the hash of all transaction fields (excluding the proof itself), and the maximum gas costs (calculated as gas_limit $\times$ gas_price). If the proof is valid, it confirms that the input notes being spent are correctly nullified, that the nullifiers have been correctly computed, that the amount held in the input notes is enough to cover the new notes' amounts, the deposit, and the maximum gas, and that the data for the new notes has been correctly computed (that is, with the right amounts and encryptions). If the transaction is accepted and all checks pass, the nullifiers are added to the list of nullifiers, the new notes are added to the Merkle tree of notes, and the deposit and gas fees are processed accordingly. If the gas limit is not fully spent, a new note is created for the sender, who must specify their public key for this purpose.

Delegation model. Phoenix allows for the delegation of intensive computations to trusted third parties. On the one hand, view keys allow users to delegate the task of scanning the network for identifying the transaction addressed to them, but ensuring the trusted party cannot spend the notes as they do not have access to their whole secret keys. On the other hand, the use of signatures allow users to delegate the task of generating ZK proofs for the transactions without compromising the transaction integrity.

Security and privacy. The Phoenix transaction model uses stealth addresses, digital signatures, and nullifiers embedded in ZK proofs to guarantee the properties of unforgeability, double-spending prevention, unlinkability, transaction non-malleability, and balance integrity.

5 Environmental impact

Dusk is designed to provide a blockchain solution that meets the demands of regulated financial markets while minimizing environmental impact. The architecture leverages innovations across consensus, network architecture, and cryptographic operations to limit energy consumption and ensure efficient allocation of resources throughout the protocol, while balancing network security.

5.1 Consensus efficiency

Dusk's succinct attestation (SA) protocol is a committee-based proof-of-stake (PoS) consensus mechanism that achieves finality without the computational overhead associated with traditional proof-of-work (PoW) systems. Research has shown that PoS systems are significantly more resource-efficient than PoW, which relies on energy-intensive cryptographic puzzles to secure the network [12]. PoW requires solving computational puzzles that increase in difficulty as network security requirements grow, necessitating more powerful hardware and substantial energy to maintain blockchain security. By contrast, PoS achieves network security by relying on staked assets, reducing the need for computational power even as the network scales. Consequently, in a PoS network, adding additional participants to increase security imposes only minimal additional energy costs, as demonstrated by Ethereum's transition from PoW to PoS, which reduced its energy consumption by more than 99.95% [12].

The SA protocol leverages deterministic sortition to select provisioners in proportion to their staked assets, further optimizing energy use. This selection method allows for block production and validation without intensive computational operations, thus reducing the network's overall energy requirements.

Additionally, the SA protocol incorporates rolling finality, which limits the number of consensus iterations required to finalize blocks. By optimizing block finalization, the protocol reduces redundant computations, conserving network resources and minimizing the energy associated with both block propagation and validation.

5.2 Network optimization

Dusk's peer-to-peer (P2P) layer, Kadcast, provides a structured and efficient communication protocol for the propagation of blocks, transactions, and consensus messages. Unlike unstructured P2P protocols such as Gossip [19] and LibP2P [20], which broadcast messages indiscriminately to all neighboring nodes, Kadcast utilizes the Kademlia distributed hash table (DHT) protocol to organize nodes in a hierarchical structure and optimize data transmission paths using XOR distance metrics [21]. This structured approach enables Kadcast to reduce message redundancy and achieve more efficient bandwidth usage compared to unstructured protocols.

Studies indicate that Kadcast achieves an approximate *25-50% reduction in bandwidth usage* compared to Gossip protocols, as nodes forward messages only to selected peers rather than broadcasting to all neighboring nodes [22]. This bandwidth optimization not only conserves network resources but also leads to tangible energy savings at both the node and infrastructure levels. By minimizing the number of

transmissions required to propagate data across the network, Kadcast conserves energy on individual nodes by lowering the computational load associated with message processing. Moreover, reduced bandwidth usage alleviates data load on the internet backbone, indirectly reducing energy expenditure on global data routing.

In addition to bandwidth efficiency, Kadcast's structured message propagation contributes to a *10-30% reduction in stale block rates* in scenarios with faster block times, such as Ethereum-like networks [22]. Lower stale block rates mean that nodes spend less computational power on processing blocks that ultimately do not get accepted into the chain, reducing the energy overhead associated with redundant block validation and consensus efforts. In PoS networks, this reduction translates into more efficient utilization of network resources, as fewer block attestations are wasted.

5.3 Cryptographic operations

Dusk's virtual machine (VM), Piecrust, is designed for efficient and secure execution of zero-knowledge (ZK)-powered smart contracts. To optimize resource use, Dusk integrates host functions within the VM to handle computationally intensive cryptographic tasks, including ZK proof verification, signature validation, and hashing.

By performing these tasks natively via host functions, Dusk avoids the performance penalties associated with running cryptographic operations within a virtualized WebAssembly (WASM) environment. Research indicates that WASM execution can be *45-255% slower* compared to native code for complex applications, largely due to the overhead of virtualized memory management and the additional instruction handling in sandboxed environments [17].

Studies like these suggest that offloading computationally intensive tasks to native environments can substantially improve processing speeds and reduce the energy consumption associated with heavy cryptographic workloads. Although specific power savings for Dusk's VM setup have not yet been quantified, theoretical estimates indicate that such offloading may lead to significant computational efficiency gains, particularly in high-transaction networks, where the avoidance of in-VM cryptographic processing can lower energy costs across nodes.

Dusk's host functions are optimized to handle various cryptographic operations that would otherwise require significant computational resources. Examples of the exposed host functions include:

- `hash`: it computes the Blake2b [23] and Poseidon [14] hash functions, both truncated to fit a scalar.
- `verify_plonk`: it verifies a PlonK [13] ZK proof for a given circuit and public inputs.
- `verify_groth16_bn254`: it verifies a Groth16 [15] ZK proof in the BN254 [24] pairing setting.
- `verify_schnorr` and `verify_bls`: it validates Schnorr [27] and BLS [4] single and multisignature schemes.

These host functions provide secure access to cryptographic libraries and hardware capabilities, allowing Dusk to execute complex cryptographic tasks more efficiently

than would be possible within the VM alone. By handling hashing computation, proof verification, and signature validation on the host, Dusk reduces the energy expenditure associated with these operations. This structure ensures that cryptographic operations remain scalable and sustainable as network usage grows, all while maintaining the network's security, as the results are replicated across nodes.

6 Implementation

In this section, we present some details about Piecrust and the genesis contracts.

6.1 Virtual machine

Piecrust is a *virtual machine* (VM) implementation focused on creating compact WebAssembly (WASM) modules for running smart contracts in a secure, modular, and lightweight virtual machine [7]. Written primarily in Rust, Piecrust centers around two key components: the `piecrust` crate, which functions as the WASM virtual machine, and `piecrust-uplink`, a toolkit that aids in developing, deploying, and managing smart contracts.

The Piecrust VM is designed to handle the execution of smart contracts efficiently, with a focus on modularity, allowing developers to create flexible and scalable applications. By using WASM, it leverages the benefits of portability and security, ensuring that the contracts can be executed across various platforms while maintaining a high level of performance. Additionally, the modular nature of the VM enables future extensions and updates without major overhauls, making it adaptable to the evolving needs of the network. Moreover, Piecrust is built to handle complex cryptographic operations, ensuring the secure execution of contracts and integrating Dusk's privacy and security standards.

Additionally, `piecrust-uplink` offers a development framework that simplifies the process of building and testing smart contracts. It provides tools for compiling contracts into WASM modules and executing them in a controlled environment, making it easier for developers to verify the correctness and security of their code before deploying it on the Dusk network. This approach reduces the complexity of smart contract development while ensuring compatibility with the underlying infrastructure.

Overall, Piecrust represents an essential component of Dusk's ecosystem, providing a robust and scalable platform for executing smart contracts in a decentralized, secure, and efficient manner. Its focus on modularity, lightweight execution, and integration with the Dusk's cryptographic framework makes it well-suited for privacy-focused applications.

6.2 Genesis contracts

Genesis contracts are a special type of smart contract deployed during the initial launch of Dusk's blockchain. In this section, we focus on the main contracts that rule fundamental operations such as transaction validation, staking mechanisms, and initial distribution of tokens and assets.

Transfer contract. The transfer contract [9] is responsible for managing DUSK transfers between users in the Dusk Network, while also handling gas fees for these transactions. When a transfer is initiated, the contract verifies the validity of the transaction according to the transaction model rules. If the transaction includes the deployment or the call to a smart contract, this is also handled by the transfer contract. Simultaneously, the contract deducts the required gas fee from the sender's balance, which covers the computational cost of executing the transaction. This ensures that each transaction is executed securely while maintaining the network's efficiency and stability.

Stake contract. The stake contract [10] manages the staking process on Dusk, allowing users to lock their DUSK in exchange for becoming provisioners in the network's consensus mechanism. When a user stakes tokens, the contract validates the amount, ensuring it meets the minimum required stake, and then locks the tokens for a defined period. These staked tokens increase the user's likelihood of being selected in the consensus process. The contract also handles unstaking requests, where users can withdraw their tokens after the lock period has ended. Additionally, the staking contract ensures that rewards and penalties are applied correctly, reinforcing participation and maintaining the integrity of the network.

6.3 Other contracts

We introduce two foundational smart contracts that will be integral to the long-term functionality and evolution of the blockchain. These contracts are designed to be deployed on the Dusk blockchain to support important features and interactions within the blockchain ecosystem. As essential components of the platform, they will play a key role in enabling advanced capabilities, setting the groundwork for future applications.

Zedger contracts. A Zedger contract is a concrete instantiation of a smart contract built using Zedger, a protocol for managing securities and real-world assets (RWAs) on Dusk, either tokenized or natively issued. Zedger is designed for regulatory compliance and user privacy using ZK proofs and auditing capabilities. Key functions include minting, burning, corporate actions (e.g., dividends), and force transfers, ensuring legitimacy via proof validation and nullification of spent securities. It also supports issuer-initiated force transfers and transaction auditability, balancing privacy with compliance for use cases like securities trading. In essence, Zedger provides tools to create financial securities tailored to specific asset properties and jurisdictional regulations.

Citadel contract. The license contract [11] manages the issuance and validation of licenses according to the Citadel protocol [25]. This contract will allow users to acquire licenses, which are essential for enabling certain actions or privileges within the network. It will track the ownership, validity, and expiration of each license, ensuring that only users with a valid and active license can perform specific operations. The contract will also handle the revocation or renewal of licenses, ensuring that all interactions governed by licenses are secure and compliant with the network's rules.

7 Conclusions

In summary, Dusk is a blockchain platform purpose-built for traditional financial institutions, providing the infrastructure necessary for private, scalable, and compliant financial transactions. Through innovations like the SA consensus mechanism, the Phoenix transaction model, and the Zedger protocol, Dusk is set to redefine the integration of blockchain technology in financial markets, delivering enhanced privacy and compliance that aligns with the sector's complex regulatory landscape.

References

1. Azgad-Tromer, S., Garcia, J., Tromer, E.: The case for on-chain privacy and compliance. Stanford Journal of Blockchain Law & Policy (june 2023), https://stanford-jblp.pubpub.org/pub/onchain-privacy-compliance
2. Bansod, S., Ragha, L.: Challenges in making blockchain privacy compliant for the digital world: some measures. Sādhanā **47**(168) (2022)
3. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Biham, E. (ed.) Advances in Cryptology — EUROCRYPT 2003. pp. 416–432. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
4. Boneh, D., Gorbunov, S., Wahby, R.S., Wee, H., Wood, C.A., Zhang, Z.: BLS Signatures. Internet-Draft draft-irtf-cfrg-bls-signature-05, Internet Engineering Task Force, `https://datatracker.ietf.org/doc/draft-irtf-cfrg-pairing-friendly-curves/11/`
5. Dusk: Dusk consensus protocol. GitHub (2024), available online: `https://github.com/dusk-network/dusk-protocol/tree/main/consensus`
6. Dusk: Phoenix transaction model. version 2.0. GitHub (2024), available online: `https://github.com/dusk-network/phoenix/blob/master/docs/v2/protocol.pdf`
7. Dusk: π-crust. GitHub (2024), available online: `https://github.com/dusk-network/piecrust`
8. Dusk: Security analysis of Phoenix version 1.0. GitHub (2024), available online: `https://github.com/dusk-network/phoenix/blob/master/docs/protocol.pdf`
9. Dusk: Transfer contract. GitHub (2024), available online: `https://github.com/dusk-network/rusk/tree/master/contracts/transfer`
10. Dusk: Transfer contract. GitHub (2024), available online: `https://github.com/dusk-network/rusk/tree/master/contracts/stake`
11. Dusk: Transfer contract. GitHub (2024), available online: `https://github.com/dusk-network/rusk/tree/master/contracts/license`
12. Foundation, E.: Ethereum's energy expenditure (2023), `https://ethereum.org/en/energy-consumption`, accessed: 9 November 2024
13. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PlonK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019)
14. 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)
15. 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)
16. Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash protocol specification. GitHub: San Francisco, CA, USA p. 1 (2016)

17. Jangda, A., Powers, B., Berger, E., Guha, A.: Not so fast: Analyzing the performance of webassembly vs. native code. arXiv preprint arXiv:1901.09056 (2019), https://doi.org/10.48550/arXiv.1901.09056, accepted at USENIX Annual Technical Conference 2019
18. Khalilov, M., Levi, A.: A survey on anonymity and privacy in bitcoin-like digital cash systems. IEEE Communications Surveys & Tutorials (03 2018). https://doi.org/10.1109/COMST.2018.2818623
19. Kiffer, L., Salman, A., Levin, D., Mislove, A., Nita-Rotaru, C.: Under the hood of the ethereum gossip protocol. In: Borisov, N., Diaz, C. (eds.) Financial Cryptography and Data Security. pp. 437–456. Springer Berlin Heidelberg, Berlin, Heidelberg (2021)
20. Protocol Labs: Libp2p specification (2024), https://github.com/libp2p/specs, accessed: 11 November 2024
21. Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the XOR metric. In: Peer-to-Peer Systems. pp. 53–65. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
22. Rohrer, E., Tschorsch, F.: Kadcast: A structured approach to broadcast in blockchain networks. Cryptology ePrint Archive, Paper 2019/876 (2019), https://eprint.iacr.org/2019/876
23. 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)
24. Sakemi, Y., Kobayashi, T., Saito, T., Wahby, R.S.: Pairing-Friendly Curves. Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-11, Internet Engineering Task Force (Nov 2022), https://datatracker.ietf.org/doc/draft-irtf-cfrg-pairing-friendly-curves/11/
25. Salleras, X.: Citadel: Self-sovereign identities on dusk network (2023), https://arxiv.org/abs/2301.09378
26. 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)
27. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: Conference on the Theory and Application of Cryptology. pp. 239–252. Springer (1989)
28. of Standards, N.I., (NIST), T., Dworkin, M.J.: SHA-3 standard: Permutation-based hash and extendable-output functions (2015-08-04 00:08:00 2015). https://doi.org/https://doi.org/10.6028/NIST.FIPS.202, https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=919061
29. Van Saberhagen, N.: Cryptonote v 2.0 (2013)