# Elosys: A Privacy Orientated Blockchain

www.elosys.io

November 2023

v1.1


## Abstract

First popularized by Bitcoin, the blockchain-based decentralized architecture has laid the foundation for a next-generation financial system. However, the deficiency in privacy has motivated developers to establish more censorship-resistant networks.

Elosys constitutes a blockchain initiative that operates in a decentralized manner, relying on a proof-of-work (PoW) consensus. This project emphasizes resistance to censorship while maintains the public accessibility. Its core objective is to facilitate robust privacy assurances for each transaction.

We've developed Elosys as a novel cryptocurrency, built to facilitate user-friendly, fully-private payments, closely aligning with the Sapling protocol and Iron Fish. Each account is furnished with a view key, providing its holder with read-only access to the account details.

Through this protocol, we aim to redefine traditional full node usability. The Elosys networking layer seamlessly incorporates WebRTC with WebSockets, simplifying the process for users to establish genuine peer-to-peer connections without any additional setup requirements. Our initial implementation of Elosys is designed to allow future iterations to support running a full node directly in the browser. Our primary emphasis is on reducing entry barriers, ensuring that anyone with a computer can confidently run a full node.

# Introduction

Empowering individuals with the choice of what and with whom to share, privacy stands as a fundamental cornerstone for freedom, security, consent, and dignity.

Privacy goes beyond mere exclusion, offering safety, control, and the privilege to authorize access. Privacy affords you the liberty to express yourself, unleash creativity, and allocate your time and resources as you see fit, all without the prying eyes of others. It safeguards our most intimate moments, the audacity of our ambitions, unconventional ideas, and the freedom to embrace our authentic selves.

At Elosys, we take immense pride in building something noteworthy: a cryptocurrency that refuses to compromise on privacy or accessibility. Our digital cash epitomizes the concept of using privacy for good. As we transition into fully digital global citizenship, every action and transaction becomes meticulously recorded, analyzed, and traded. In this evolving landscape, privacy has never been more crucial, experiencing a surging demand worldwide.

Mainstream cryptocurrencies such as Bitcoin and Ethereum operate with the least privacy, relying on complete transparency within their protocols to verify transactions. While alternative privacy coins do exist, they often fall short of delivering the promised privacy guarantees or are so challenging to use that they see limited real-world adoption (and sometimes both).

The vast potential lies untapped in the privacy domain, contingent upon users experiencing genuine protection with their cryptocurrency and having seamless access to it. We envision that an easily accessible privacy coin could catalyze the emergence of innovative industries focused on borderless products and companies.

Within this document, we delve into the inner workings of Elosys and the rationale behind its construction. While our effort is directed towards comprehensive explanations and the inclusion of pertinent terminology, certain sections may assume a foundational grasp of elliptic curve cryptography.

# Networking

Integral to any blockchain, the networking layer plays a crucial role in sustaining the distinctive features of the protocol. More precisely, it governs the dynamics of node interactions — determining the transport layers for communication, gossiping messages among peers, and regulating the way nodes request/respond to specific messages from other peers.

In the development of any decentralized peer-to-peer (P2P) system, tackling Network Address Translation (NAT) is imperative. Given that a significant portion of our devices—machines, laptops, tablets, and phones—are situated behind routers and firewalls, establishing direct connections between peers becomes challenging. This is precisely where the networking layer assumes a pivotal role.

Unlike certain networking layers that necessitate users to configure port-forwarding on their routers to tackle the NAT issue, our implementation prioritizes accessibility. We've achieved this by leveraging a combination of WebRTC and WebSockets as our transport layer, employing a variety of techniques to facilitate direct communication between nodes.

Upon the initial launch of a node, it requires knowledge of at least one other node, referred to as a bootstrap node, which serves as an introduction point to additional peers within the network. The initial connection to a bootstrap node occurs through a WebSocket, while all subsequent peer connections rely on WebRTC. This segment will intricately delineate the process by which nodes establish connections, forming a network that underpins the Elosys protocol, beginning with the initiation of a new node.

## Startup Sequence

As a new node starts up, the following happens:

1. The new node randomly selects one bootstrap node from a provided list, and opens a WebSocket connection to it. If a user has a specific node they want to connect to during this step, they can use the config file or the command line to specify their own preferred bootstrap node(s). The bootstrap node sends its identity to the new node.

2. The bootstrap node broadcasts a peer list to the new node.

3. Using that list, the new node decides which peers to connect to.

4. The 3rd step is repeated with every new peer the node connects with, until the maximum amount of connections (up to 50) is made.

5. In order to maximize the strength of our network and to prevent network fragmentation, the node will prioritize connecting to peers that relatively few of the known peers are currently connected to.

As an end user, all this happens without you having to do anything, or even be aware of it. When you start a node, from your end, all you'll see is your node connecting to a bootstrap node, and then quickly to many others.

## Peer Connections Lifecycle

While a node is online, it tries to maintain a full and complete knowledge of the state of its peers, and all peers connected to those peers (two layers deep), checking periodically for any changes. Nodes have no knowledge whatsoever of more distant peers.

With that in mind, let's look at how nodes communicate with one another by passing messages.

## Messaging

A message is an agreed-upon format for a piece of information to be shared between peers. There are several types of messages and ways for them to interact with the nodes, depending on the situation.

## Message Types

The networking layer of Elosys currently has four different internal message types:

**Identity:** a message by which a peer can identify itself to another

**Signal:** a message used to signal a real-time communication (RTC) session between two peers

**Peer List:** a message containing the list of peers that this node is currently connected to

**Cannot Satisfy Request:** an error message when a problem occurs

All messages on the Elosys network are routed via one of the following four styles, depending on various factors.

**Gossip:** These messages reach every node in the network. When a node receives a gossip message, it validates the message and forwards it to other connected nodes. This is used to propagate changes to the blockchain to all the nodes in the network.

**Fire and Forget:** These messages are directed to a specific connected peer (it is not possible to send a message to a peer you are not connected to). No response or receipt confirmation is expected from the peer. This is useful when you don't need to make sure that the data was correctly received.

**Direct RPC:** Here, a message request is sent to a specific connected peer, and the system expects a response. A remote procedure call (RPC) stream is composed of two streams: one for the request and one for the response. This is used as a backing layer for Global RPC.

**Global RPC:** These messages are not addressed to a specific peer. The network library will choose a peer to send the message to and retry (up to a specified limited amount) with another peer if it doesn't get a response, or if the response is invalid. The selection algorithm is randomized, and weighted to favor peers who are known to be more likely to respond to the registered type of message.

## Gossip Protocol

The Gossip Protocol is primarily used to broadcast new blocks and transactions to all of the peers within the Elosys network. To help visualize this: nodes connected together form a network, and a blockchain is the data structure they agree on.

Back to the protocol. Once started, each node will then independently verify the incoming transactions before further broadcasting them out to other peers, and validate the incoming blocks before applying their encompassing transactions to the node's local copy of the ledger (a.k.a. the blockchain). There is one main objective for the gossip protocol: when a message is broadcasted, every peer should receive the message as quickly as possible.

The following section details how Elosys's gossip-based broadcast will be implemented.

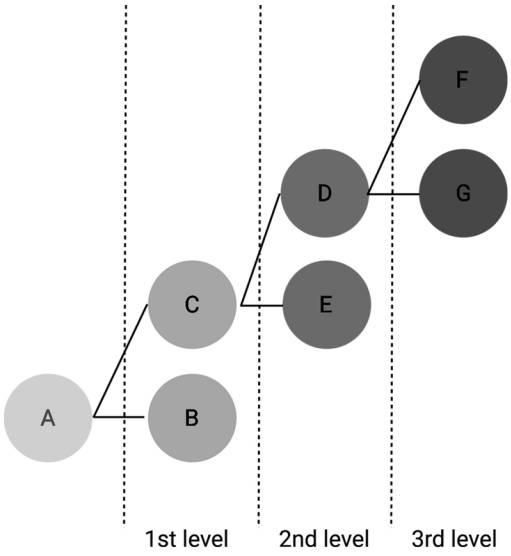
## Basic Implementation

### Step 1

When a new node comes online and connects with a peer, the peer communicates the list of their direct peers.

Let's imagine that our current network is composed of a new Node A, connected to the nodes B and C. C itself is connected to nodes D and E. D itself is connected to F and G.

Visually, this would look like the image below.

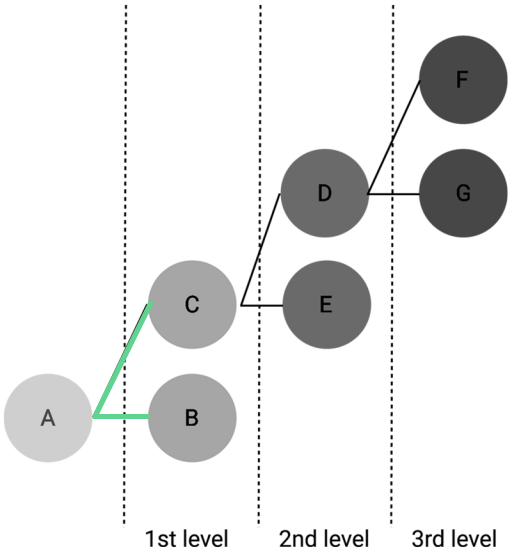



Nodes are only aware of their direct neighbors, and their neighbor's neighbors. Node A will have no knowledge of Nodes beyond D and E.

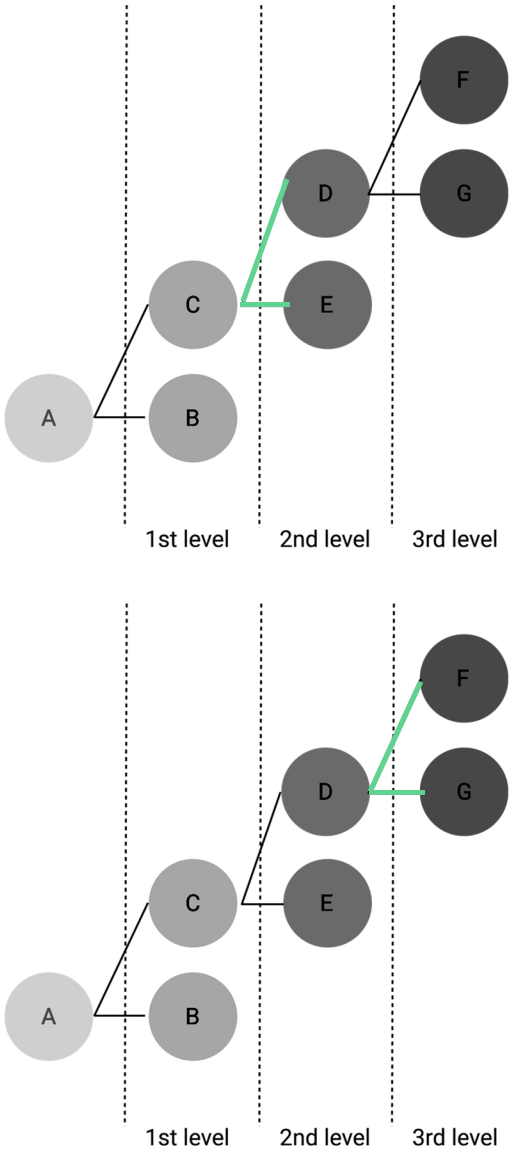
Node A can now decide to connect to some of the peers, and will store a copy of each node peers list.

### Step 2

When node A decides to broadcast a new message, it'll send out a Gossip-type message to all of its connected peers (in this example, C and B).



Then, each subsequent node will broadcast the message to their other peers, until the entire network receives the message. In this example, C broadcasts the message to D and E. Then D broadcasts the message to F and G.




## Optimization

To reduce network congestion, we implemented the following Gossip Protocol improvements.

## Local history

In an effort to avoid an infinite broadcast of the same message, each node stores a set of all the gossiped messages it has seen. When a node receives a gossip-type message already in the set (meaning it was seen before), it ignores the message. The set that keeps track of these gossip-type messages is bound to a specific size and old ones are evicted in a first-in first-out order.

## Neighbor cast

To avoid spamming the peers with duplicated messages, we implemented two other solutions:

When Node A gossips a message to Node B, Node B does not send back the message to Node A.

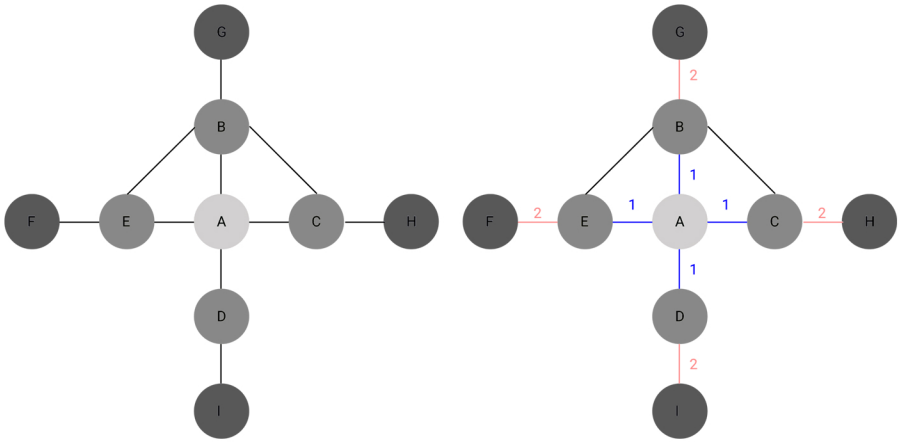
When Node A sends a message to Node B, Node B (knowing that A already took care of it) will avoid sending messages to any peer Nodes A and B are both connected to.

In the example below, Node A is connected to B, C, D, E, and stores a list of peers connections two levels deep.

When Node A gossips a message, the propagation happens in two steps:

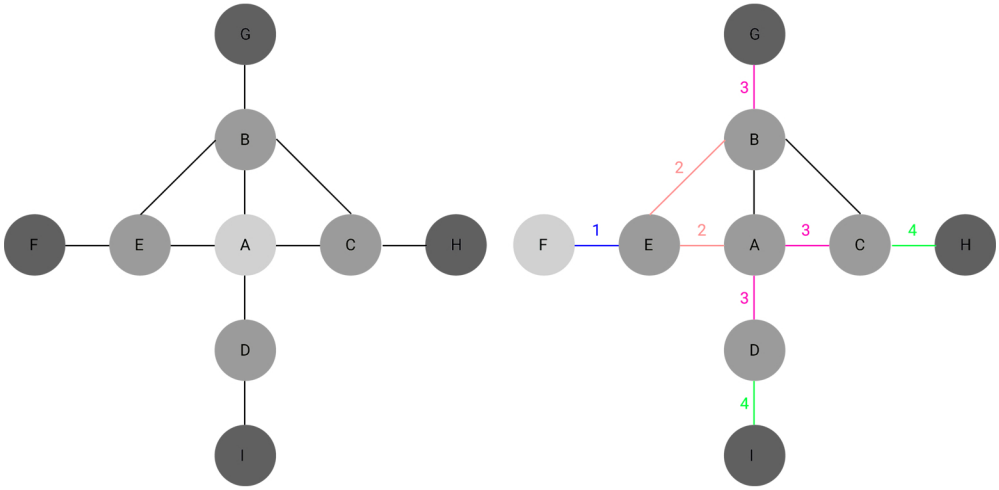
Node A broadcasts to Nodes B, C, D, and E.

Node B forwards the message to G. It does not forward it to C and E because it knows that Node A is connected to them and already sent it. Node C forwards the message to H. Node D forwards to I and Node E to F.



When Node F gossips a message, in this example, the propagation happens in four steps:

Node F broadcasts the message to Node E.
Node E forwards the message to both A and B.
Node B forward to G. It does not forward to A because it knows that E is connected to A. It does not forward to C, because it knows it is connected to A.
Node C and Node D both forward the message to H and I.




# Storage

In this section, we'll start by reviewing what we store (the data structures and models of Elosys), and then move into how we store it (using both LevelDB and IndexDB). We'll start by looking at the most basic data structures that represent the Iron Fish global state: notes and nullifiers.

## Data Structures and Models

### Note

A note is a representation of a spendable form of payment, like a bill. It is very similar to a UTXO in Bitcoin. A note is only ever referenced publicly when it is created as an output of a transaction, and only in its hashed form. The contents of the note themselves are private.

The plaintext contents of a note are:

**(pk, d):** the transmission key and the diversifier of the recipient's address (e.g. the owner of the note's public key which we'll explain in the Account section)

**v:** the plaintext value that the note holds

**rcm:** note randomness used to generate a Pedersen hash for the note

**memo:** a 32-byte memo field

## Nullifier

A nullifier is a unique identifier to a note, but is unlinkable to its note. A note can only be spent if its nullifier is revealed as part of the transaction. Once the nullifier is revealed it is saved in one of the two Elosys global data structures that all nodes keep track of — the Merkle Tree of Notes and the Merkle Tree of Nullifiers (more on these in a moment).

This ensures a note cannot be spent twice (as it would have to reveal the same nullifier twice, and such attempts would be rejected). The nullifier is unique to its note because it is derived from private information about the note — the nk (nullifier deriving key, more on this in the Account section), cm (note commitment), and position (index in the Merkle tree).

## Merkle Trees

As mentioned above, Elosys stores both notes and nullifiers in Merkle trees. A Merkle tree is an accumulator data structure, meaning it is used to represent many elements with one small identifier (a hash).

## Merkle Tree of Notes

The Merkle Tree of Notes is fixed-size, with a depth of 32; and it is used to hold all the notes that are created. Unlike in other blockchains where a UTXO is removed after it is spent, this Merkle tree is an add-only data structure where notes are added sequentially to the tree. Although a Merkle tree of depth 32 is very large (meaning it can hold 4,294,967,296 notes!), it is finite, which isn't a great solution for an ever-growing currency. When the Merkle tree gets to be completely full, it becomes a read-only tree, allowing old notes to be spent from it. A brand new Merkle tree is then constructed.

We use a Pedersen hash both on the notes, and for the intermediate nodes in the Merkle tree — using the Jubjub elliptic curve. Pedersen hashes are SNARK-friendly, meaning they can be efficiently constructed within a zero-knowledge SNARK proving circuit.

Remember that the main purpose for our Merkle Tree of Notes is to store notes that users can spend later while preserving that user's privacy. In order to do this, the notes in our Merkle tree store an encrypted note, along with other helper fields. All the information necessary to spend the note is contained here, thus the note owner doesn't need to download the specific block or transaction that resulted in this note.

## Merkle Note

A Merkle Note consists of:

***value_commitment*** A Pedersen Commitment of the note's value.

***note_commitment*** A Windowed Pedersen Commitment that is used to hash all the contents of the Merkle note.

***public_key*** A public key that gets created when the note it's associated with is created. This is used such that the recipient can decrypt the Note and spend it in the future, using a Diffie-Hellman key exchange technique.

***encrypted_note*** The encrypted note that the recipient can decrypt using the public_key mentioned above.

***note_encryption_keys*** An encrypted field to hold all the necessary info for the sender to decrypt the Note later (so the sender can reconstruct a transaction history).

## Merkle Tree of Nullifiers

Just like the Merkle Tree of Notes is an accumulator for notes, the Merkle Tree of Nullifiers is an accumulator for nullifiers. It is simply used to keep track of all the nullifiers (which are 32-byte numbers) ever revealed when their accompanying notes are spent.

We've chosen this tree to be the same size as the Merkle Tree of Notes since it'll grow in linear proportion. However, we chose a different hashing function than Pedersen because this tree is not referenced in any of the zero-knowledge proofs, and therefore can use a faster hashing function. We chose blake3.

We wouldn't have any of these notes and nullifiers if they weren't part of transactions (we cover transactions in more detail here) that get accepted to be part of the overall blockchain. Next, we'll go over exactly how all these components make up blocks, which in turn make up the Elosys blockchain.

## Block and Block Header

The main ingredient for a blockchain is a block, and each block has an accompanying block header. A block simply holds transactions that are waiting to be finalized by being added to the blockchain, and a block header gives the necessary information about the block for it to be validated and accepted by others in the network. In short, a block header validates a block, and a block holds transactions. The transactions have nullifiers from the sender spending some notes, and new notes that are created for the recipient.

With all that in mind, how does a block header help validate a block?

## Block Header

A block header consists of the following:

***sequence*** - The sequence number of this block. Blocks in a chain increase in ascending order of sequence. More than one block may have the same sequence number (indicating a fork in the chain), but only one fork is selected at a time.

***previousBlockHash*** - The hash of the previous block in the chain.

***noteCommitment*** - Commitment to the Merkle Tree of Notes after all new notes from transactions in this block have been added to it. Stored as the hash and the size of the tree at the time the hash was calculated.

***nullifierCommitment*** - Commitment to the nullifier set after all the spends in this block have been added to it. Stored as the hash and the size of the set at the time the hash was calculated.

***target*** - The hash of this block must be lower than this target value in order for the block to be accepted onto the chain.

***randomness*** - The nonce used to calculate this block's hash

***timestamp*** - Unix timestamp according to the miner who mined the block. This value must be taken with a grain of salt, but miners will want to verify that it's an appropriate distance to the previous block's timestamp.

***minersFee*** - A single (simplified) transaction representing the miners fee consisting of only one Output Description.

Note that although the block header is missing its block hash, it can be computed using the Elosys Hashing Algorithm given all the elements of the block header.

The steps necessary for another node (e.g. device or user) to validate a block are:

The previous Block that this Block is referencing exists (by using the ***previousBlockHash*** field).

The ***target*** is the one that the verifying node agrees to (more on this later on how the target and difficulty are calculated).

When all the contents of the block header are hashed, that hash is numerically less than the t***arget*** — this is largely achieved by the miner from tweaking the ***randomness*** value.

The ***timestamp*** for this Block makes sense (that its timestamp is greater than that of the previous Block by 12 seconds, +/- 10 seconds as buffer).

All the transactions in the Block are valid (more on this in the Transactions section).

The ***minersFee***, the Miner rewards for presenting this Block, is valid, meaning that it is exactly the agreed upon block reward plus all the transaction fees in the Transactions (more on this in the Mining section).

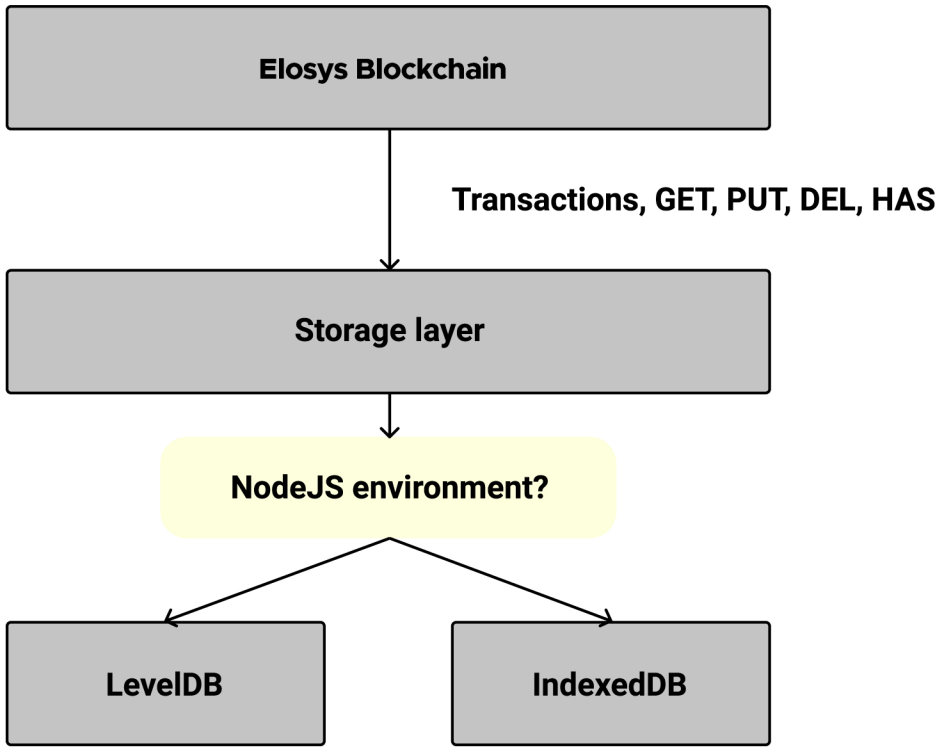
After all transactions are added to the Notes and Nullifiers Merkle trees, validate that the Notes tree's root hash matches the ***noteCommitment*** field and the Nullifier tree's root hash matches the ***nullifierCommitment*** field.

## How Elosys Stores Data

So far, we've been talking about what is necessary for an Elosys node to store, but not the how. In this section, we'll go over exactly how Elosys stores these notes, nullifiers, blocks, transactions, and block headers such that the storage layer works both as a Command Line Interface (CLI) tool running as a program on your computer, and also entirely in the browser.

Since we knew that running a full implementation of Elosys in the browser was going to be more challenging than running it in the NodeJS terminal environment, we focused on that first. The most robust database choice for applications wanting a database in the browser is IndexedDB. Unfortunately, there wasn't an accompanying IndexedDB implementation for NodeJS, so we chose LevelDB for our NodeJS implementation.

To prevent having to juggle two separate storage implementations for the two different databases, our implementation of Elosys has a generic layer of abstraction for data stores and database access based on LevelUp. This abstraction layer takes care of the specific implementations of the underlying database, and exposes a generic layer that can be used both in the browser and NodeJS environment, offering a simple datastore-agnostic API.




## The Storage Layer API

In simple terms, the storage layer is an API over its underlying data stores — it can create stores from schemas and operate on them with all the normal key-value store operations like GET, PUT, DEL, and HAS.

# Mining

Note that we use the terms nodes and peers interchangeably; a node in Elosys is always a peer. A few other quick definitions:

***Mining*** is the core mechanism that defines by which rules new blocks are created, and by which rules a peer can verify and accept an incoming block.

***Miners*** are those nodes that propose a new block to be added to the blockchain to other nodes.

A new block is said to be ***mined*** when a miner finds a hash of the block header that is below some threshold that we call the target.

The Elosys blockchain algorithm dynamically adjusts the mining difficulty to achieve 60-second average block times, by either increasing or decreasing mining difficulty if previous blocks are observed to be coming in too fast or too slow.

To be a miner, a node must have both of the global data structures synced (the Merkle Tree of Notes and the Merkle Tree of Nullifiers), and know at least the two most recent blocks.

## Block Creation

As we've mentioned, a block consists of a block header and a block body. The block body is simply a list of one or more transactions; a block containing no user-transactions is called an empty block. The block creation process is the same regardless of whether a block has user-transactions:

1. Determine the block body
2. Set the difficulty and target
3. Include the miner reward based on the coin emission schedule
4. Construct the block header

## Determine the Block Body

The transactions in the block body have been broadcasted by other peers who want their transactions to be added into the blockchain. To incentivize a miner to include such a pending transaction into the block body, transactions include a publicly visible transaction fee that goes to the miner.

A block with an invalid transaction will be rejected by other nodes — so before including it in the block body, a miner should first verify that transaction (more on transaction verification in the Transaction Creation section).

## Set the Difficulty and Target for a Block

## Difficulty

The target time for an Elosys block is currently set to 60 seconds. This is subject to change.

The purpose of the difficulty is simple. It is adjusted, if needed, at every block to make it harder or easier for miners to produce blocks such that new blocks are added to the blockchain every 55 to 65 seconds (with an average of 60 seconds). If the network (e.g. all the nodes in the network) has not produced a block in over 65 seconds, the difficulty for the upcoming block is decreased (in comparison to the previous block's difficulty). Conversely, if a miner wishes to produce a block in under 55 seconds, the difficulty for a block with that timestamp would be greater than that of the previous block.

The Elosys difficulty calculation is largely influenced by Ethereum's difficulty calculation as described in EIP-2, with a few differences.

To determine difficulty for an upcoming block, we first calculate the time “bucket” that the block belongs to. A time bucket is defined as how far away (in intervals of 10 seconds) the block's timestamp is away from the desired range of 55 to 65 seconds after the previous block. An upcoming block that is 45-55 seconds after the previous block would have a time bucket value of -1, a block that is 35-45 seconds after the previous block would have a time bucket value of -2, and so on:

| Time (in seconds) after previous block | bucket |
|---|---|
| 0-5 | -6 |
| 5-15 | -5 |
| 15-25 | -4 |
| 25-35 | -3 |
| 35-45 | -2 |
| 45-55 | -1 |
| 55-65 | 0 |
| 65-75 | 1 |
| 75-85 | 2 |
| 85-95 | 3 |
| ... and so on (max value capped at 99) | |

We use the time bucket to then adjust difficulty for the upcoming block relative to the difficulty of the last block. The pseudocode for calculating difficulty is as follows:

```
const diffInSeconds =
  (time.getTime() - previousBlockTimestamp.getTime()) / 1000;

const difficulty =
  previousBlockDifficulty -
  (previousBlockDifficulty / BigInt(2048)) * BigInt(bucket);

return BigIntUtils.max(difficulty, Target.minDifficulty());
```

Currently, `Target.minDifficulty()` is `131072`, but this is subject to change.

## Target

We've discussed adjusting difficulty to ensure 55-65 seconds between blocks — we do this by adjusting a target, which is a number that the block hash needs to fall under (e.g. be numerically less than the target).

The target is calculated from the difficulty given this formula: `target = 2**256 / difficulty`

As covered in the previous section, it has an inverse relationship with difficulty: as difficulty increases, the target decreases, and vice versa. This is because as the target decreases it becomes statistically harder to find that random number such that the hash of the block header is less than the target. Conversely, as the range of acceptable hashes decreases, difficulty increases.

## Include the Miner Reward Based on Coin Emission Schedule

The mining reward (how many coins a miner is allocated for successfully mining a block) is tied to the Elosys emission curve. The idea behind the Elosys emission curve is that after the first year post mainnet launch, the total coin supply is increased by a fourth of the genesis block value (due to block rewards). Subsequent years will have fewer and fewer newly minted coins according to a decay function, but never fully reaching 0.

The formula to determine how many new coins will be minted for a particular year after launch is:

$$g(x) = \frac{s}{4} \cdot e^{k \cdot floor(x)}$$

Where s is the initial supply of the genesis block of 42 million coins, k is the decay factor of -.05, and x is the year after mainnet launch (starting from 0).

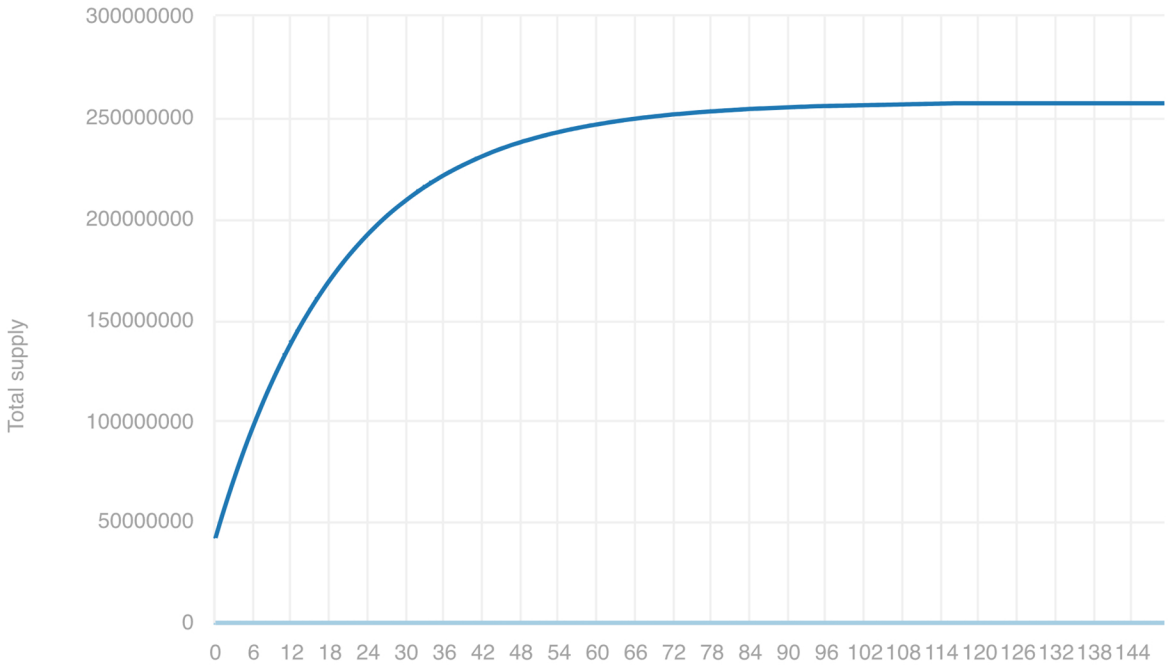
The Elosys “year” in block count is 525,600 blocks to one calendar year (assuming 60-second block times). We use the above formula to calculate the block reward using the Elosys "year", rounded to the nearest .125 of a coin:

$$blockReward = mRound(\frac{\frac{s}{4} \cdot e^{k \cdot floor(x)}}{525{,}600}, 0.125)$$

Therefore, the block reward and total supply for the first few years after launch would be:

| Years after launch | Block reward (60s block times) | Total supply |
|---|---|---|
| 0 | 0 | 42,000,000.00 |
| 0-1 | 20 | 52,512,000.00 |
| 1-2 | 19 | 62,498,400.00 |
| 2-3 | 18.125 | 72,024,900.00 |

The emissions curve using the block reward formula mentioned above, with a cap of 256,970,400 coins for total supply, would look like this:



To claim the block reward for successfully mining a new block, the miner constructs a special miner fee transaction in the block header. The value of the miner fee transaction is publicly visible so that others can verify that it is exactly the block reward plus all the transaction fees from the transactions included in that block. The recipient's address for that miner fee transaction remains hidden. Learn more about the miner reward in the Transaction Creation section.

## Construct the Block Header

After determining the block body, the miner can then construct the block header for it.

A Block Header consists of the following data:

**sequence** - The sequence is constructed using the latest block's sequence number and incrementing it by one. This field indicates the position of that block in the blockchain, starting from zero.

**previousBlockHash** - The previousBlockHash is filled out using the hash of the latest block in the blockchain, per the Elosys hashing algorithm. This indicates that this new block is building off of the latest known block in the blockchain.

**noteCommitment** - All the new notes included in the block body are applied (in order) to the Merkle Tree of Notes. The resulting new Merkle root for that tree and its size (in terms of the global number of notes) are used to build the noteCommitment for the block header.

**nullifierCommitment** - The same process is followed, but with the nullifiers revealed, to build the nullifierCommitment for the block header.

**target** - The target is determined by the difficulty and target algorithm.

**timestamp** - The timestamp is when that block is mined. The timestamp of the current block must be greater than the previous block's timestamp, and can be up to 60 seconds into the future to mitigate for different clocks for whoever is verifying the block in real time.

**minersFee** - The minersFee is a special transaction to award the block reward to any address of the miner's choosing. The value of this block reward transaction is known and can be verified, but the recipient's address is hidden. For more details, see how the block reward transaction is created (link to Miner Reward Transaction section in Transaction Creation).

**randomness** - The randomness is a 64-bit number such that when all the contents of the block header are hashed using the Elosys hashing algorithm the result is numerically less than the target.

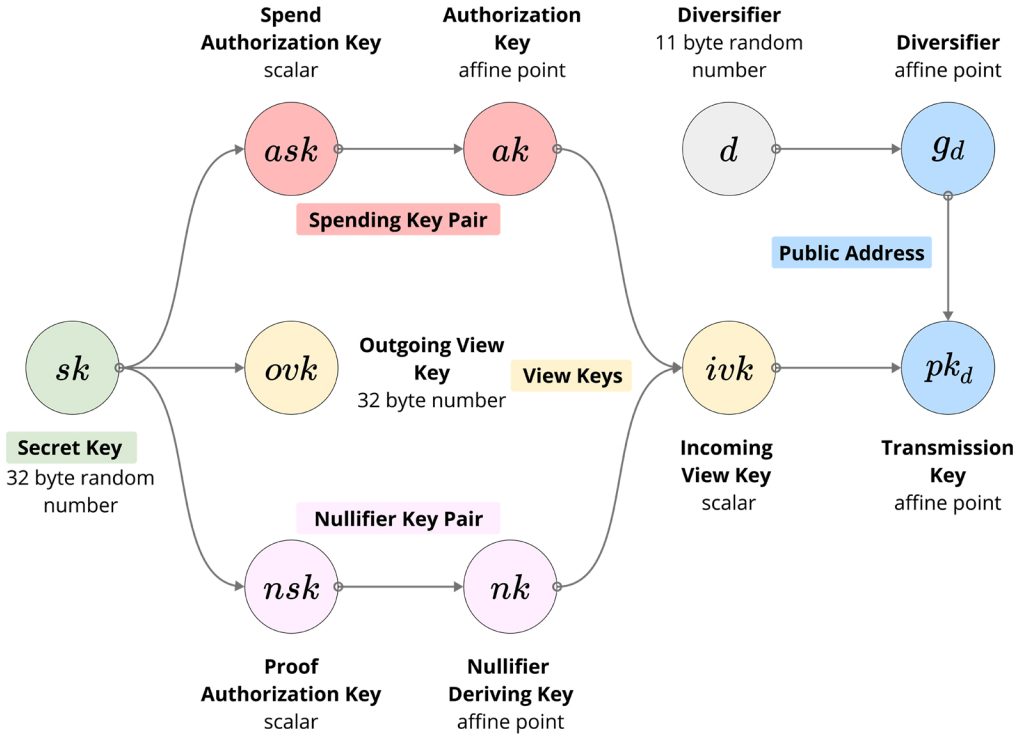
### Elosys Hashing Algorithm

Elosys currently uses Blake3 hashing algorithm, but is subject to change any time, including at mainnet launch.

# Account Creation

Accounts and transactions in Iron Fish are heavily influenced by the Sapling protocol and Iron Fish, with some differences. In this section we'll break down all the key components of an account — how they're created, how they're used, and what is relevant to be surfaced to the user.

We'll start with the key components that are used to view an account's balance, send transactions, or view past transactions. All of the key components for an Elosys account are derived from a single secret key. Though the underlying account construction may seem complex, the high level overview is that, in addition to the secret key, each account has a set of keys for spending that account's funds, viewing keys to be given to any third party for read-only access, and a public address to be used to receive funds from others.



## Secret key

The secret key is simply a 32-byte random number. This is the seed necessary to construct all other parts of your wallet.


**Secret Key**
32 byte random number

## Spending Key Pair (Spending Authorization Key and Authorization Key)

This key pair is used for spending notes associated with an account, and is derived directly from the secret key.

The Spend Authorization Key (ask) is the private key component of this key pair and is derived by hashing the secret key and a modifier using the Blake2b hashing algorithm (with personalization params) and then converting it into a scalar for the Jubjub curve.

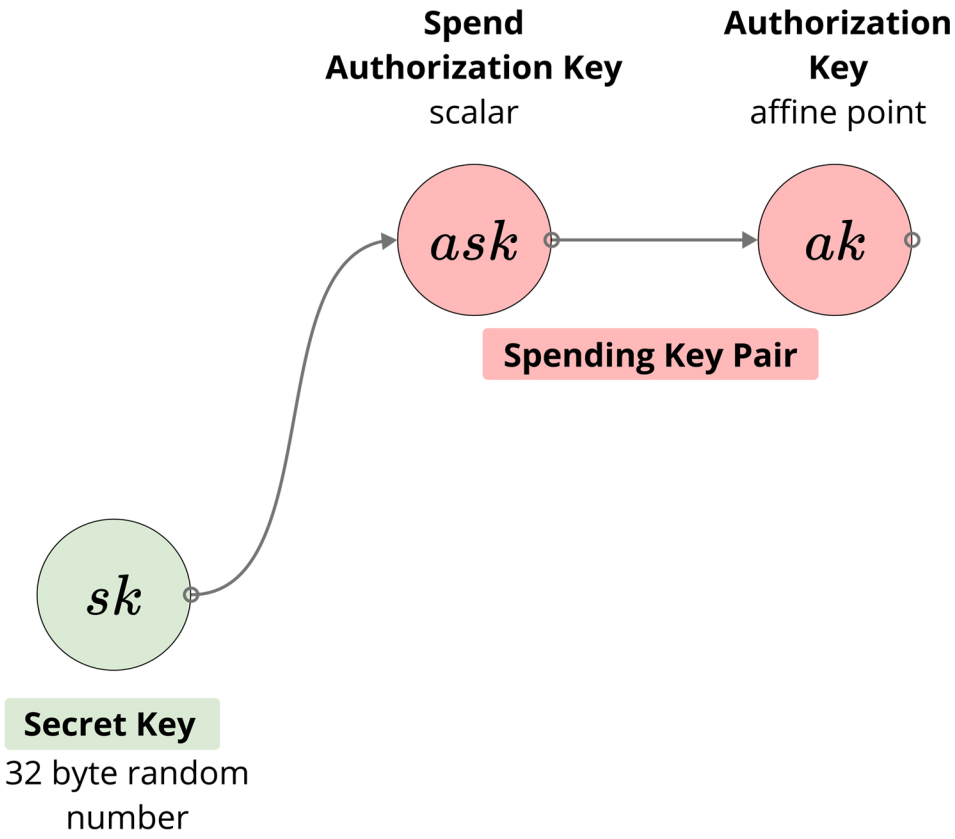
The Authorization Key (ak) is derived as the public key for the spend authorization key by multiplying the spend authorizing key with a fixed generator base point for it on the Jubjub curve.

The equation for this pair looks like:

$$ak = ask * G_{ak}$$

Where ak is the authorization key, ask is the spend authorization key, and $G_{ak}$ is the generator base point for it on the Jubjub curve.

This key is used in the Spend description of a transaction that is responsible for spending notes. More specifically, it is used in the zero-knowledge proof that is part of the Spend description to prove ownership of the note being spent. It is also used to create the signature for the Spend description.




*The secret key is used to derive the spend authorization key (ask) which in turn is used to derive the authorization key (ak)*

## Nullifier Key Pair (Proof Authorization Key and Nullifier Deriving Key)

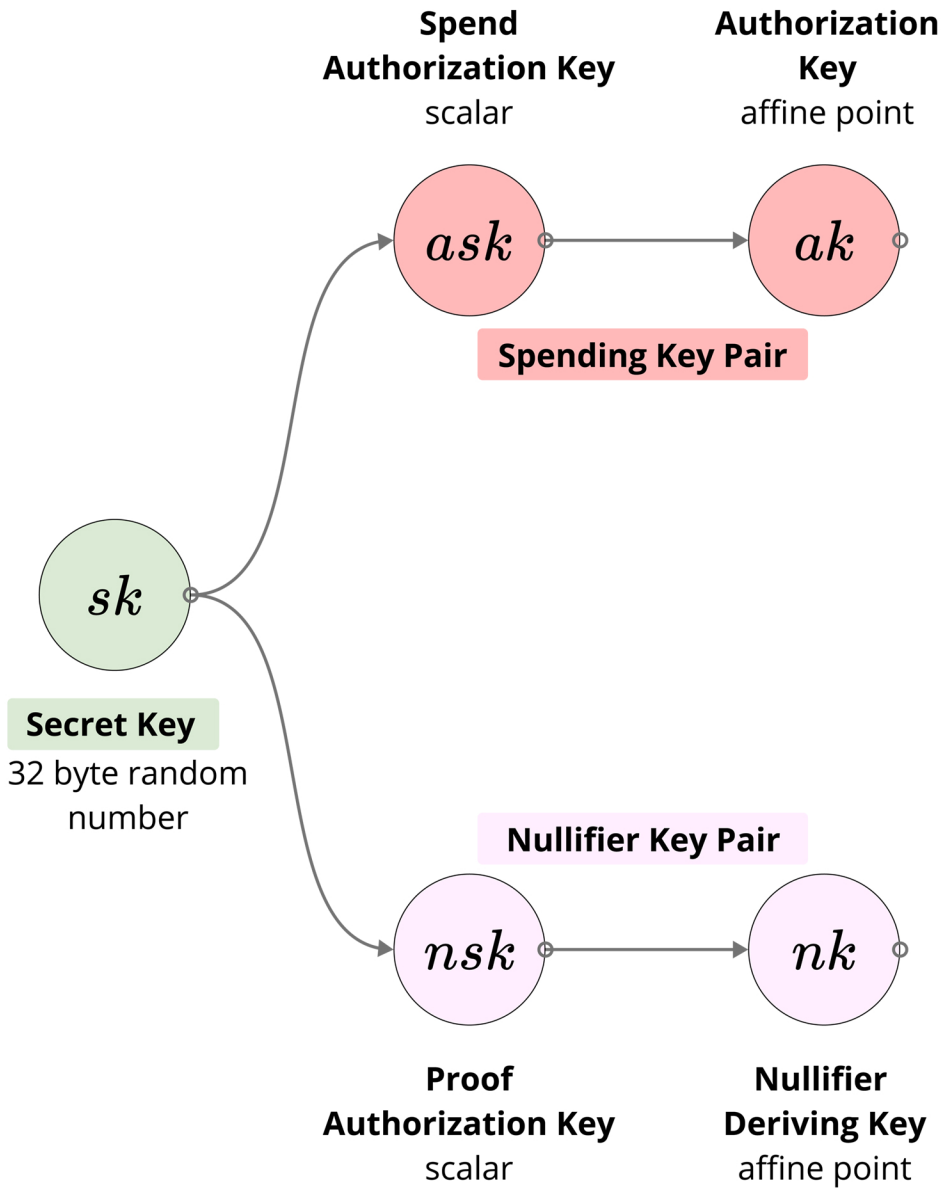
These keys are responsible for creating the nullifiers that are necessary to spend a note, derived from the secret key.

The Proof Authorization Key (nsk) is the private key component of the nullifier key pair and is derived by hashing the secret key and a modifier using the Blake2b (with parameters) hashing function and then converting it into a scalar (a.k.a. an integer) for the Jubjub curve. The proof authorization key is used in the Spend description of a transaction proving that the revealed nullifier was computed correctly. Remember that to spend a note a user must reveal its unique nullifier as part of the transaction. The zero-knowledge proof of the Spend description ensures that the revealed nullifier was properly created using the owner's proof authorization key.

The Nullifier Deriving Key (nk) is derived as the public key for the proof authorization key created by multiplying the proof authorizing key with a fixed generator base point $G_{nk}$

$$nk = nsk * G_{nk}$$

Where nk is the nullifier deriving key, nsk is the proof authorization key, and $G_{nk}$ is the generator base point for it on the Jubjub curve.



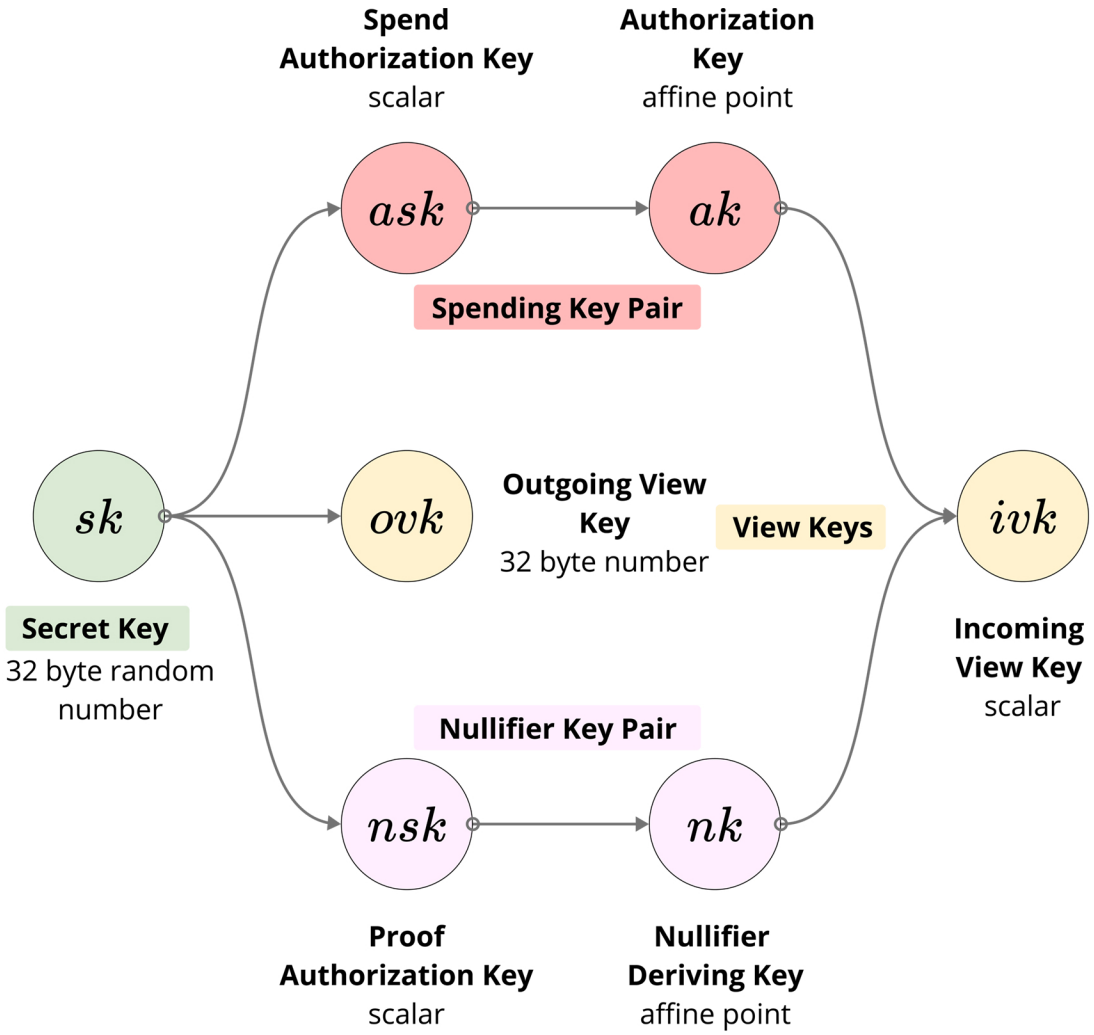

*The secret key is used to derive the proof authorization key (nsk) which in turn is used to derive the nullifier deriving key (nk)*

## View Key Pair (Incoming and Outgoing View Key)

The Outgoing View Key (ovk) allows for decrypting outgoing transactions. It is derived by hashing the secret key and a modifier using the blake2b hash function with additional params and then taking the first 32 bytes of the result.

The Incoming View Key (ivk) allows for decrypting incoming transactions. It is derived by using the blake2s hash function to hash the bytes of the authorizing key with the bytes of the nullifier deriving key and converting it into a Jubjub scalar:

ivk=blake2s(ak,nk) converted into a **jubjub scalar**




*The secret key is used to derive the outgoing view key. The incoming view key is derived from the authorization key (ak) and the nullifier deriving key (nk).*

## Public Address

The public address consists of a Transmission Key and a Diversifier. Together, they enable a single wallet with a single private key to contain up to $2^{11}$ public addresses.

## Diversifier

The diversifier (d) is a random 11-byte number used to randomize the final public address. The diversifier is converted into an affine point on the Jubjub curve ($g_d$) to be used to create the Transmission Key.

## Transmission Key

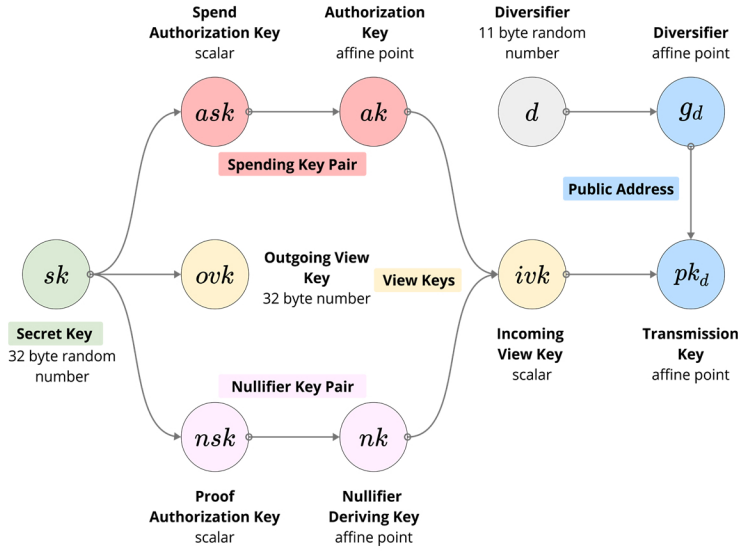
The Transmission Key ($pk_d$) is derived by multiplying the diversifier (converted to an affine point on the Jubjub curve, $g_d$) by the incoming view key:

$$pk_d = g_d * ivk$$

Concatenated together, the Diversifier and Transmission key make up the Public Address:

$$public\ \ address = (d, pk_d)$$

The public address is a 43-byte number (11 bytes for diversifier + 32 bytes for transmission key).



# Transaction Creation

Just like accounts, transactions are heavily influenced by the Sapling protocol and Iron Fish with some differences. All Elosys transactions are shielded transactions, meaning they do not reveal any information to any onlooker who does not have explicit access.

This level of privacy is achieved through the use of zero-knowledge proofs, which allow transaction details to be encrypted with an accompanying zero-knowledge proof attesting to the validity of transaction details.

There is a lot to cover in this section, so here's a quick guide to the pieces we'll be reviewing:

1. the components of a transaction

2. the Spend description component (the one that dictates how an account can spend a note)

3. the Output description component (the one that creates new notes)

4. how a transaction balances to ensure that appropriate amounts were spent and paid out

5. how a validator (such as a miner) can verify any transaction

6. a special type of transaction called the Miner Fee transaction, which is used to reward a miner for successfully mining a block

7. how notes are encrypted and decrypted such that only the appropriate parties are privy to viewing transaction details

## Transaction Components

A transaction is a list of Spend and Output descriptions:

A Spend description spends notes that are used up in a transaction.

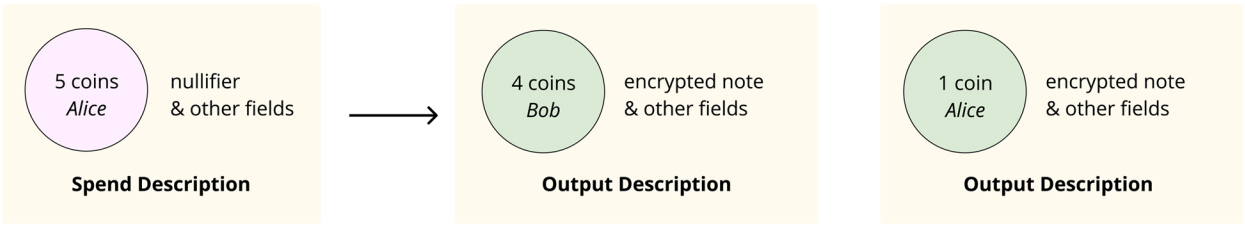
An Output description creates new notes that result as part of that transaction, including the change back to the sender if the note they've spent is greater than what is intended for the recipient.

Notes that are spent in the Spend description cannot be spent again in the future due to the unique nullifier that must be revealed when spending it as subsequent attempts will be rejected by validators (e.g. miners) if that nullifier has been revealed in the past.

For example, if Alice has a note of value five coins, and wants to send Bob four coins, then the transaction would look like this:

It would have one Spend description for the note she's spending (in this example a note with a value of five coins) along with the unique nullifier for that note.

And it would have two Output descriptions: one for Bob of four coins, and one note as change for herself of one coin since the note used in the Spend description cannot be spent again.




To ensure privacy, a Spend description spends a note without revealing which note was spent through the help of a zero-knowledge proof (specifically zk-SNARK Groth16 Sapling proof). The Output description similarly creates an encrypted note with a zero-knowledge proof that the newly created note was created correctly. The circuit construction for these proofs is taken from Sapling primitive gadgets which in turn were constructed using the bellman circuit building tool.

While explaining zk-SNARKs is outside the scope of this paper, for readers who want to learn more, zk-SNARK construction can be broken up into these 5 steps:

1. Computation
2. Arithmetic Circuit
3. R1CS (rank 1 constraint system)
4. QAP (quadratic arithmetic program)
5. SNARK

The structure of an Elosys transaction is constructed with these parts:

**Transaction Fee:** The fee (in plaintext) that'll go to any miner that successfully includes this transaction in a block.

**Spends:** The list of Spend Descriptions.

**Outputs:** The list of Output Descriptions.

**Binding Signature:** A binding signature that both signs the transaction and is used to verify that it balances — meaning that it did not destroy or create money out of thin air, and that indeed all the funds in the spend descriptions minus the funds in the output descriptions equal to transaction fee. The message that is signed here is the transaction hash, which is a blake2b hash of the serialized transaction fee, spend descriptions and output descriptions.

Remember that the miner's reward for mining a block is also a type of a transaction.

### Spend Description

The Spend description is a part of the transaction that spends notes associated with an account. The goal of the Spend description is to spend notes without revealing which notes were actually spent with the help of zero-knowledge proofs (specifically Groth16 zk-SNARK type proofs).

The high level overview of the Spend description is that it spends a note by using a zero-knowledge proof to prove the following:

1. it is attempting to spend a note that the spender can decrypt
2. this note exists in the Merkle Tree of Notes
3. the value commitment (cv) for that note was constructed using the true value of that note
4. the revealed nullifier is the unique nullifier for that note and was constructed correctly
5. the signature maps to the spender's authorization key

It does all this by having some public data that is needed to verify the proof and balance the transaction (more on this in the Transaction Balancing section), as well as the proof.

The structure of the Spend description looks like this:

| Element | Description |
|---|---|
| $cv$ | **value commitment** of the note being spent |
| $rt$ | **root anchor** – for the Merkle tree of notes root that was used to generate the proof |
| $nf$ | nullifier for the note |
| $rk$ | randomized public key for the authorization key (ak) |
| $sig$ | signature for the transaction hash |
| $proof$ | zk-SNARK proof that allows one to hide the private values needed to validate |

The cv is the value commitment (as a Pedersen commitment) for the note. It's computed during the construction of the Spend description as:

$$cv = v * G_v + rcv * G_{\{rcv\}}$$

Where v is the value of the note, $G_v$ is the generator point used for the value, rcv is the randomness to further obscure the value commitment hash, and $G_{\{rcv\}}$ ,is the generator point used for the randomness.

The rt is the root anchor to specify which Merkle root was used to construct the zero-knowledge proof. The proof will validate that there is a note that exists in the tree with that specified Merkle root. However, it is the miner's job to make sure that the Merkle root is one that is associated with a valid tree.

The nf is the nullifier, and it is unique to the note. The nullifier's construction is verified in the proof, but once again it is the miner's job to check that this nullifier has not been revealed in the past. The nullifier is computed by utilizing the blake2s hash function, the note commitment (cm), the position of the note being spent in the Merkle tree, and the nullifier deriving key (nk):

$$nf = blake2s(nk \mid cm + note\_position * G_{\{nullifierposition\}})$$

Where | denotes creating a one-byte array to hold both elements together.

The rk is the randomized public key that is used to sign the spend description. It's randomized so that nothing is revealed from a single authorization key being used multiple times to sign various spend descriptions. The proof contains information about the actual authorization key and proves it's valid transformation into a randomized key.

$$rsk == ask * \alpha$$

Where rsk is the private key of the randomized public key, ask is the private key of the authorization key, and α is the randomness

$$rk == (ask + \alpha) * G_{\{spendingKey\}}$$

The sig is the signature that signs the transaction hash of the transaction the Spend description is in using the randomized key (rk).

And finally, we have the proof, which is a Groth16 zk-SNARK proof verifying in zero-knowledge the validity of the entire Spend description.

## How is the proof generated?

The private parameters that are used to generate the proof (and are not revealed afterwards) are:

| Element | Description |
|---|---|
| $merklePath$ | the Merkle path to the commitment note being spent |
| $position$ | the position of the commitment note (e.g. index) |
| $g_d$ | the diversifier of the public address owning this note |
| $pk_d$ | the transmission key of the owner |
| $v$ | value of the note |
| $rcv$ | randomness used in the Pedersen Hash for value commitment |
| $cm$ | note commitment of the note being spent |
| $rcm$ | the randomness used in the Pedersen Hash for note commitment |
| $\alpha$ | alpha used to hide the authorization key that signs the spend |
| $ak$ | the owner's authorization key (that was randomized) |
| $nsk$ | the proof authorization key used for the nullifier |

The merkle path is the Merkle path from the given root (the rt, root anchor) to the note being spent (specifically its note commitment), using Pedersen hashes. The proof verifies that the path is valid and correct, and that the given position is the correct position for the note's commitment in the Merkle tree at the lowest level, (you can think of the position like an entry in an index).

The $g_d$ is the diversifier (converted into an affine point on the Jubjub curve) of the sender, and $pk_d$ being the transmission key of the sender. The proof checks that $g_d$ is not of small order and that $pk_d$ was properly computed.

Remember that $pk_d = g_d * ivk$ (the incoming view key). Even though the incoming view key isn't passed in here directly, we have everything we need to recompute it since ivk is derived from hashing (using the blake2s hash function) the authorization key (ak) and nullifier deriving key (nk) along with some params. We don't have the nullifier deriving key (nk) directly here either, but we can derive it using the passed in proof authorization key (nsk) since $nk = G_{\{proofGenerationKey\}} * nsk$.

Also remember that the value commitment is computed as $cv = v * G_v + rcv * G_{rcv}$ and so we pass in the value **(v)** and the randomness for the value **(rcv)** into the proof to validate the construction of the value commitment.

The note commitment (cm) is a Pedersen commitment (resulting in a full point) of the note's contents (value(v), $g_d, pk_d$) and the randomness used for the note commitment (rcm)

$$cm = pedersenHash(v, g_d, pk_d) + rcm * G_{noteCommitmentRandomness}$$

The alpha $\alpha$ along with the authorization key $ak$ is used to construct the randomized public key that is used to sign the spend description. Here in the proof we verify that the randomized key was created correctly by verifying that:

$$rk = \alpha * G_{spendingKey} + ak$$

Finally, we check that the nullifier is computed correctly. The proof first checks that $nk = nsk * G_{proofGenerationKey}$ and then checks that the nullifier was indeed computed as:

$$nf = blake2s(nk \mid cm + note\_position * G_{nullifierPosition})$$

In summary, the proof checks:

**Note Commitment Integrity** - Check that the note commitment (cm) is derived from $g_d, pk_d, value,$ and $rcm$

**Merkle Path Validity** - Check that the Merkle path is valid from the given merkle root to the leaf (that this note exists in a given tree)

**Value Commitment** - Check that the value commitment (cv) was indeed constructed as $cv == v * G_v + rcv * G_{rcv}$

**Nullifier** - Check that the nullifier is derived from nk (the owner's nullifier deriving key), the cm (note commitment) and position

**Random Authorization Key** - Check that the random authorization key (rk) that is used to sign the transaction is correctly derived from the spend authorization key (ak) and alpha ($\alpha$) for randomness.

## How is the proof verified?

In order to verify the proof, we simply need to pass in public parameters that validate all of the above mentioned statements.

The public variables necessary for the Spend description proof verification are most of the other fields of the Spend description:

| Element | Description |
|---|---|
| $rt$ | root anchor that was used for the Merkle path in the proof |
| $cv$ | Pedersen Hash (commitment) of the value |
| $nf$ | nullifier to spend the note |
| $rk$ | the randomized authorization public key |

Given these parameters, the proof will return a True/False statement whether or not all of the above statements are true given these public inputs.

## Output Description

The Output description is the part of the transaction that produces new notes. The note it produces is encrypted such that only the holder of the incoming view key for the recipient and holder of the outgoing view key for the sender can decrypt the note. It also contains a zero-knowledge proof (also Groth16 zk-SNARK proof) that verifies that the newly created note was created correctly, with the correct value.

The structure of the Output description contains these fields:

| Element | Description |
|---|---|
| $cv$ | value commitment |
| $cm$ | note commitment |
| $epk$ | ephemeral public key (more on this in Note Encryption and Decryption) |
| $C^{\{enc\}}$ | encrypted plaintext of the note |
| $C^{\{out\}}$ | encrypted blob that allows the holder of the viewing key to decrypt a decryption key for $C^{\{enc\}}$ (more on this in Note Encryption and Decryption) |
| $proof$ | the zero-knowledge proof |

Below is a deeper dive into these fields.

The cv is the value commitment (as a Pedersen commitment) of the note being created, computed as:

$$cv = v * G_v + rcv * G_{\{rcv\}}$$

Where rcv is the randomness for the value commitment and is a private parameter to the zero-knowledge proof to validate this computation.

The cm is the note commitment (as a Pedersen commitment) for the new note being created that is added to the Merkle tree of notes by the miner who mines the transaction containing this Output description. It is computed as:

$$cm = pedersenHash(v, g_d, pk_d) + rcm * G_{\{noteCommitmentRandomness\}}$$

Where rcm is the note commitment randomness used in this Pedersen commitment computation, and verified in the zero-knowledge proof.

The epk is the ephemeral public key that is used to facilitate the recipient of the note decrypting it.

The $C^{\{enc\}}$ is the actual encrypted note that results as part of this Output description. It is encrypted such that the recipient's incoming view key can decrypt it.

The $C^{\{out\}}$ is an encrypted blob of data to facilitate the holder of the sender's outgoing key to decrypt the encrypted note.

And finally we have the proof (Groth16 zk-SNARK proof) for the Outgoing description that validates all these public parameters against the private ones that were used to create them.

## How is the proof generated and verified?

The Outgoing description proof is a lot less complicated than the Spend description proof. A lot of the terms you'll see here will be familiar to the ones you've seen in the Spend description proof generation.

The private parameters that are used to create the proof are:

| Element | Description |
|---|---|
| $g_d$ | recipient's diversifier |
| $pk_d$ | public diversifier address for the recipient |
| $v$ | value |
| $rcv$ | randomness for the value |
| $rcm$ | randomness for the note commitment |
| $esk$ | ephemeral private key — a random number chosen by the sender |

And the public parameters that are used to verify the proof are:

| Element | Description |
|---|---|
| $cv$ | value commitment |
| $cm$ | note commitment |
| $epk$ | ephemeral public key |

The proof validates that:

1. $g_d$ for the recipient is not of small order and that the ephemeral public key was computed as: $epk = g_d * esk$

2. That the value commitment (cm) is properly computed as a Pedersen commitment of:
$cm = pedersenHash(v, g_d, pk_d) + rcm * G_{\{noteCommitmentRandomness\}}$

## Adding a Merkle Tree Note from the Outgoing description

The outputs in the Output description are stored as a Merkle Tree Note in addition to being part of the transaction. A Merkle Tree Note consists of these fields taken from the Output description:

| Element | Description |
| --- | --- |
| $cv$ | value commitment |
| $cm$ | note commitment |
| $epk$ | ephemeral public key |
| $C^{\{}enc\}$ | encrypted plaintext of the note |
| $C^{o}ut$ | allows the holder of the viewing key to decrypt a decryption key for $C^{\{}enc\}$ |

## Transaction Balancing

So far, we went over how zero-knowledge proofs are used to prove ownership of an existing note in order to spend it or create valid new notes, but we're still missing verifying one of the most important rules in cryptocurrencies: no coins can be created or destroyed in a transaction. Validators still need to verify that the transaction balances, meaning that the sum of all the funds being spent minus the funds being created equals the transaction fee (or zero if there is no transaction fee).

$$input\ \ values - output\ \ values = transaction\ \ fee$$

Balancing a transaction happens through the value commitments in the Spend and Output descriptions and the binding signature in the transaction. Remember that the structure of the Elosys transaction consists of these parts:

**Transaction Fee:** the fee that'll go to any miner that successfully includes this transaction in a block

**Spends:** the list of Spend Descriptions

**Outputs:** the list of Output Descriptions

**Binding Signature:** a binding signature that both signs the transaction and is used to verify that it balances

And both the list of Spend descriptions and list of Output descriptions contain their appropriate value commitments. Remember that a value commitment is constructed as a Pedersen commitment in this format:

$$cv = v * G_v + rcv * G_{\{}rcv\}$$

The transaction is cryptographically signed by a binding validating key (bvk) resulting in the binding signature in the transaction. The binding validating key is constructed by adding all the randomness (e.g. the rcv values) from the input value commitments, and subtracting all the randomness from the output value commitments. It becomes more clear why this signature is necessary to balance a transaction if we first try to balance it without it.

As an example, say we have a transaction with two inputs, and two outputs:

**Inputs**

$$cv1 = v1 * G_v + rcv1 * G_{\{rcv\}}$$

$$cv2 = v2 * G_v + rcv2 * G_{\{rcv\}}$$

**Outputs**

$$cv3 = v3 * G_v + rcv3 * G_{\{rcv\}}$$

$$cv4 = v4 * G_v + rcv4 * G_{\{rcv\}}$$

The rule that we have to follow, is that $input\ \ values - output\ \ values = transaction\ \ fee$.

Since the generator points (such as $G_v$ and $G_{\{rcv\}}$) are the same for all value commitments, we can safely add all the value commitments from the inputs and subtract them from the outputs and simplify. Remember that since these operations are on an elliptic curve, we are not going to perform division operations as that would be logarithmically hard. Instead, we'll multiply the transaction fee by the same generator point as the values (G_v) to check equality.

$$Input\ \ commitments - Output\ \ commitments =$$
$$G_v * (v1 + v2 - v3 - v4) + G_{\{rcv\}} * (rcv1 + rcv2 - rcv3 - rcv4)$$

We know that the sum of all the input values minus the sum of all the output values should be the transaction fee, meaning that:

$$G_v * (v1 + v2 - v3 - v4) = G_v * (transaction\_fee)$$

There is still that second part of the equation that deals with randomness that wasn't addressed:

$$G_{\{rcv\}} * (rcv1 + rcv2 - rcv3 - rcv4)$$

This part of the equation is the binding validating key, which acts as a public key with which we can verify the transaction signature. The private key counterpart to bvk that was used to sign the transaction is bsk (the binding signing key) which is the sum of all the randomness from the input descriptions minus the sum of all the randomness in the output descriptions.

For this example:

| Element | Description |
| --- | --- |
| $bsk\ \ (binding\ \ signing\ \ key)$ | $= rcv1 + rcv2 - rcv3 - rcv4$ |
| $bvk\ \ (binding\ \ validating\ \ key)$ | $= G_{\{rcv\}} * (rcv1 + rcv2 - rcv3 - rcv4)$<br>$= G_{\{rcv\}} * (bsk)$ |

The transaction doesn't reveal the binding validating key (bvk) directly (the transaction only contains the cryptographic signature of the transaction signed by the binding signing key), however, it can be derived from the publicly available information.

During transaction validation, the validator computes the sum of all input value commitments, minus the sum of all output value commitments, minus the transaction fee multiplied by the value commitment generator point. For our example, this step would look like this:

$$(G_v * v1 + G_{\{rcv\}} * rcv1 + G_v * v2 + G_{\{rcv\}} * rcv2) - (G_v * v3 + G_{\{rcv\}} * rcv3 + G_v * v3 + G_{\{rcv\}} * rcv3)$$

equivalent to:

$$G_v * (v1 + v2 - v3 - v4) + G_{\{rcv\}} * (rcv1 + rcv2 - rcv3 - rcv4) - G_v * (transaction\_fee)$$

Since a valid transaction would have $G_v(v1 + v2 - v3 - v4) = G_v(transaction\_fee)$ validator computes the binding validating key as:

$$G_v * (v1 + v2 - v3 - v4) + G_{\{rcv\}} * (rcv1 + rcv2 - rcv3 - rcv4) - G_v * (transaction\_fee) = bvk$$

If indeed all the values of the input descriptions minus all the values of the output descriptions equal transaction fee, then bvk must equal the leftover randomness:

$$bvk = G_{\{rcv\}}(rcv1 + rcv2 - rcv3 - rcv4)$$

To validate that the transaction balances, the validator checks that computed bvk is indeed the public key corresponding to the transaction signature that signed the transaction hash. This means that the sender of the transaction must have used the same bvk with a corresponding private key bsk to sign the transaction.

$$Final\ step:\ verify\ signature\ (bvk,\ transaction\ hash)$$

That is all that is necessary to check that the transaction balances since thebsk used to sign the transaction hash must have been the sum of all randomness from the input descriptions minus the sum of all randomness of the output descriptions (in this example bsk must have been rcv1+rcv2—rcv3—rcv4) as that is the only valid solution (called an opening) to this overall equation which is still in the format of a Pedersen commitment. It is not possible for there to be any other solution or opening since Pedersen commitments have a property that there can only be one opening per commitment.

## Transaction Verification

The prior section went over how to balance a transaction — to make sure that no coins were created or destroyed as part of that transaction. Balancing is just one step in the verification process.

In whole, a validator must perform a series of checks to validate a transaction:

Verify all the zero-knowledge proofs against the public parameters from the Spend description

Verify all the zero-knowledge proofs against the public parameters from the Output description

Check that the transaction balances

Check that every signature in the Spend description signed the transaction hash

Check that the root anchors (rt) in all the Spend transactions are valid past Merkle tree roots on the validator's Merkle tree

Check that none of the nullifiers in the spend descriptions were revealed in the past

## Miner Reward Transaction

As mentioned above, an important rule of cryptocurrencies is that no coins can ever be created or destroyed. Except in special cases where the protocol does allow for new coins to be created out of thin air. This is what happens in the special form of a transaction called the Miner Reward transaction. A Miner Reward transaction is a special transaction that awards the miner a set amount for mining the block as well as the sum of all the transaction fees for that block. The exact set amount for mining a block is variable, as described in the previous section on mining and coin emission schedule. This section will go over how such a transaction is made.

The Miner Reward transaction looks a lot like a regular transaction, except that it is stored in the block header (and not the block body), contains no Spend description, and has a negative transaction fee. This negative transaction fee contains the set amount allotted for the miner for mining the block as well as the sum of all transaction fees in the block body. Any number of Output descriptions can exist on this transaction with output values adding up to the allotted amount. Remember that a transaction fee on a transaction is in plaintext, and any validator can verify that $input\ \ values - output\ \ values = transaction\ \ fee$.

Let's say that a Miner awards itself five coins for mining a block, then that Miner Reward transaction would look like this:

| **Transaction** | **(Miner award allotted_amount = 5 $IRON)** | |
|---|---|---|
| | Spends | None |
| | Output(s): | Output Description(s) |
| | Transaction fee | -1 * allotted_amount |
| | Binding Signature | |

Note that this transaction will balance with the negative transaction fee. The miner is able to preserve their privacy by making a transaction with all the privacy guarantees of a regular transaction.

For validation, all other validators can easily check that the Miner reward transaction has the appropriate alloted_amount by checking that the values of the spend description minus the values of the output descriptions equals to the negative allotted amount. Validators can also verify that the allotted amount is exactly the block reward plus transaction fees associated with transactions in that block.

## Note Encryption and Decryption

Lastly, this section will go over how exactly the recipient of a transaction is able to decrypt a newly created note that is sent to them in the Output description. The note in the Output description is encrypted such that the recipient's incoming view key and the sender's outgoing view key is able to decrypt it. This technique shares a lot of commonalities with the Diffie-Hellman key exchange algorithm.

Remember that the note plaintext (np) is made up of:

$(pk_{d}, d)$ : the transmission key and the diversifier of the recipient's address

v: the plaintext value that the note holds

rcm: note randomness used to generate the Pedersen commitment for the note

memo: a 32-byte memo field

The Output description stores this note in its encrypted form as $C^{enc}$.

| Element | Description |
|---|---|
| $cv$ | value commitment |
| $cm$ | note commitment |
| $epk$ | ephemeral public key |
| $C^{enc}$ | encrypted plaintext of the note |
| $C^{out}$ | encrypted blob that allows the holder of the viewing key to decrypt a decryption key for C^{enc} |
| $proof$ | the zero-knowledge proof |

## Note Encryption by the Sender

The sender has to know the recipient's public key, which is a combination of the transmission key and the diversifier (d, $pk_d$). With this information, the sender's wallet can create a sharedSecret with which to encrypt the note such that the recipient's incoming view key can decrypt it. Let's go over how the sender's wallet creates this shared secret.

1. The sender's wallet generates a random number and uses it to create an ephemeral secret key (esk) by converting this number to a scalar on the Jubjub curve.

2. It then creates an ephemeral public key (epk) by using scalar multiplication between the diversifier of the recipient represented as a field point and esk. This ephemeral public key is a publicly known component of the Output description and is seen by everyone.

3. It then derives a sharedSecret using Diffie Hellman Key Exchange between esk and $pk_d$.

4. The note is then encrypted using the sharedSecret and a form of symmetric encryption (specifically ChaCha20Poly1305 symmetric encryption algorithm).

## Note Decryption by the Recipient

The recipient's wallet can then decrypt the encrypted note in the Outgoing description using the recipient's incoming view key. Remember that the recipient's transmission key ($pk_d$) is derived from the diversifier (converted to a point on the Jubjub curve as $g_d$) and the incoming view key:

$$pk_d = g_d * ivk$$

The recipient's wallet can then calculate the shared secret using the epk (ephemeral public key) provided in the Outgoing description: $sharedSecret = epk * ivk$

This is the same sharedSecret that the sender's wallet used. Note that:

$$epk = esk * g_d$$
$$pk_d = g_d * ivk$$

The recipient's wallet calculates:

$$sharedSecret = epk * ivk = esk * g_d * ivk$$

The sender's wallet calculates:

$$sharedSecret = esk * pkd = esk * g_d * ivk$$

Now the recipient's wallet can use the same symmetric encryption algorithm (ChaCha20Poly1305) to use the sharedSecret and decrypt the $C^{\{enc\}}$ field on the Output description.

## Note Decryption by the Sender Using the Sender's Outgoing View Key

If at some later time after the transaction has been sent, the sender's wallet needs to recreate the transaction history and decrypt the notes it sent in the past, it can do that with the help of the outgoing view key.

Remember that initially the sender's wallet was able to encrypt the note plaintext (using the symmetric encryption algorithm ChaCha20Poly1305) into $C^{\{enc\}}$ by calculating a shared secret as $sharedSecret = esk * pk_d$.

Since the sender's wallet doesn't have access to either esk or pkd after the transaction has been sent, that information is stored in the second encrypted field on the Outgoing description: the $C^{\{out\}}$ field. This field is created by the sender of the transaction at the time it is made and stored on the Output description.

The $C^{\{out\}}$ field is an encryption of (esk, pkd) concatenated together, also using the symmetric ChaCha20Poly1305 encryption algorithm. The symmetric key used for $C^{\{out\}}$ is calculated as:

$$symmetricEncryptionKey = blake2bHash(ovk, cv, cm, epk)$$

Where ovk is the wallet's outgoing view key, and the rest of the fields are taken from the Output description.

Output Description:

| Element | Description |
|---|---|
| $cv$ | value commitment |
| $cm$ | note commitment |
| $epk$ | ephemeral public key |
| $C^{\{enc\}}$ | encrypted plaintext of the note |
| $C^{\{out\}}$ | encrypted blob that allows the holder of the viewing key to decrypt a decryption key for $C^{\{enc\}}$ |
| $proof$ | the zero-knowledge proof |

At any given time, the holder of the outgoing view k ey (e.g. a wallet ) can recreate the $symmetricEncryptionKey$ to decrypt the $C^{\{out\}}$ field to retrieve $(esk, pk_d)$. Then, using $esk, pk_d$, the wallet can recreate the $sharedSecret = esk * pk_d$ and decrypt the $C^{\{enc\}}$ field to finally retrieve the plaintext.

# Verification and Consensus

The prior sections explained how a network gets created and how nodes construct new blocks, but not why the nodes construct blocks that certain way. Consensus is the Elosys verification layer that sets rules by which nodes accept incoming blocks. These rules implicitly force nodes to construct a block following these rules since otherwise that block won't be accepted by other nodes in the network.

An Elosys block is accepted if its header and body are valid. At a high level, verifying the header confirms that the block has enough work behind it by checking that its hash is numerically lower than the target, and that the node has performed the correct state transition by correctly applying all the transactions in that block to the two global data structures and supplying the two resulting Merkle roots. Similarly, verifying the block body confirms that all the transactions in that block are valid.

## Validating the Block Header

To validate the block header, each Elosys node receiving it checks that all of its fields are set correctly.

As a reminder, the Elosys block header consists of the following data fields:

**sequence**

**previousBlockHash**

**noteCommitment**

**nullifierCommitment**

**target**

**timestamp**

**minersFee**

**randomness**

A valid block header must follow all these verification rules:

The sequence must be one more than the sequence of that in the previous block (e.g. block number), and the previous block must be the one that has the previousBlockHash. The previousBlockHash must correspond to a valid existing block.

After the transactions in the block body are applied to the two global data structures (the Merkle Tree of Notes and the Merkle Tree of Nullifiers), the resulting Merkle tree roots for those two trees must correspond to the noteCommitment and nullifierCommitment in the block header.

The target is checked by using this block's timestamp, the previous block's timestamp, and the previous block's target by using the target formula.

The timestamp of the current block must be later than the timestamp of the previous block with a 15 second tolerance (meaning that for an incoming latest block its timestamp can be off by 15 seconds in comparison to the current time of when the block is checked).

The minersFee must follow the format of a Miner Reward transaction allocating to the miner exactly the block reward plus all the transaction fees for the transactions in that block.

Lastly, the randomness is valid if the hash of all the block header contents (including the randomness) results in a hash that is numerically less than the target specified for that block.

## Validating the Block Body

Validating the block body consists of verifying all the transactions in the block. This can be done in parallel as each transaction should be valid and independent of one another.

# Reference

Iron Fish Whitepaper

Sapling Protocol

Nakamoto Satoshi, Bitcoin Whitepaper: A Peer-to-Peer Electronic Cash System

Buterin Vitalik, Scalability, Part 1: Building on Top

Gentry Ryan, Multicoin Capital: Privacy Is a Feature, Not a Product

Buterin Vitalik, Chain Interopability