UNIVERSITY OF BUCHAREST

FACULTY OF MATHEMATICS
AND INFORMATICS

Master's Thesis in Informatics

PRIVACY-PRESERVING AND HORIZONTALLY SCALABLE BLOCKCHAIN USING ZERO KNOWLEDGE AND DISTRIBUTED COMPUTING

Budișteanu Ionuț Alexandru

Supervisor
Asoc. Prof. Cristian Kevorchian

Bucharest

Rezumat

Lucrarea de faţă îşi propunere crearea in Go a unui ledger de tipul permission-less având o reţea descentralizată şi distribuită de tipul Peer-to-Peer. Blockchain-ul dezvoltat integrează şi un protocol criptografic extins bazat pe zero-knowledge proofs pentru a permite păstrarea confidentalităţii tranzacţiilor utilizatoriilor şi un mecanism de scalabilitate orizontală folosind cloud computing. Tehnologia propusă ar putea fi adoptată la o scară largă putând procesa un număr mare de trazacţii pe secundă. Ledger-ul este multi-asset şi conţine balanţe criptate homomorfic. Reţeaua are un consensus distribuit bazat pe Proof of Stake.

Abstract

This master's thesis presents a Go implementation of a permission-less ledger with a decentralized and distributed Peer-to-Peer network. The blockchain implementation integrated an extended cryptographic protocol based on zero-knowledge proofs for privacy-preserving transactions, as well as a mechanism to achieve horizontal scalability through distributed computing. This proposed technology could be adopted on a large scale being capable to process a large number of transactions per second. The ledger is multi-asset and contains homomorphically encrypted balances. The network has a distributed consensus based on Proof of Stake.

Table of Contents

1. Introduction

1.1 Motivation

This master's thesis proposes a viable solution for creating a new distributed and decentralized network that provides a high degree of confidentiality for all its transactions and can scale horizontally for a world-wide adoption. The blockchain is a public, immutable and decentralized database that records all transactions building a ledger. It is replicated on a large number of machines, so it is reliable and secure. The ledger is built using cryptographic functions with a well-defined consensus made of rules to add new records to it in a decentralized manner. The blockchain is public and transparent by design. Protecting users' privacy is a challenge because all transactions are public and the entire history is publicly available and verifiable.

1.2 Background of the study

In early 2000, the software developers started to create the first distributed peer-to-peer (P2P) networks for data and file sharing. These distributed networks are usually categorized into three generations of networks. Napster was one of the first generation of peer-to-peer distributed networks. Napster being one of the first, it relied on a central database to organize its nodes for lookups. Gnutella was one of the second generation of peer-to-peer networks that did not rely on any central entity. In Gnutella protocol, every user would have to broadcast a query to every node in the network in order to locate the specific data they are looking for. In practice the Gnutella network was extremely slow as it was flooded all the time by its own users. BitTorrent is considered to be a third-generation P2P network that uses a distributed hash table (DHT) for its nodes to lookup for data. In short terms, BitTorrent creates a self-organized distributed peer-to-peer network which allows the exchange of information through node lookups. [1]

In 2002, Maymounkov and Mazières published a paper proposing Kademlia – a distributed hash table (DHT) that is efficient for decentralized peer-to-peer networks using XOR as a metric distance between two nodes. [2] Kademlia networks would lay the way for the first decentralized and self-organizing distributed networks that can scale to millions of online nodes. These early peer-to-peer networks were used only to locate data by a hash and for file sharing; they wouldn't be able to create and maintain an immutable ledger required by a distributed financial system. A public ledger that uses a peer-to-peer network requires guarantees of two of three CAP theorem properties namely consistency and availability.

In October 2008, a paper entitled „Bitcoin: A Peer-to-Peer Electronic Cash System" was uploaded to the internet by a pseudo anonymous author, Satoshi Nakamoto. The proposed solution [3] consists of a networking protocol to achieve a distributed and immutable public ledger that uses a consensus for a peer-to-peer network. The Bitcoin protocol is made of a set of rules that runs on multiple machines that don't trust each other. In order to assure data consistency and availability, Bitcoin requires all nodes in the network to have a backup of the entire data. Basically, every full node in the network stores all previous transactions grouped into blocks. This append-only data storage is called the Blockchain. By having the data replicated on every single machine in the network, the Bitcoin protocol solved the Byzantine Generals Problem. This allows all nodes to have an overview of entire ledger.

Another major challenge the author, Satoshi Nakamoto, solved is the creation and implementation of a decentralized rule. The network is capable for the selection of a random node which will be allowed to write data into this append-only blockchain. This was achieved by using the hash cash technique. It requires nodes (miners) to do a heavy computation to prove that they have spent enough electricity to be selected to write data into the blockchain. In short, every T seconds (10 minutes), a new block

should be created by a miner who will be proposing new transactions to be included in the Blockchain. In case the data from this block is completely valid, the block along with all its transactions will be accepted into the blockchain. The block is then broadcasted in the network. The block is then verified and accepted by all other nodes.

There are situations when two or more miners proposes different blocks at the same time which will eventually be broadcasted and accepted by various nodes located in different parts of the network. This is called a fork when the network gets split into multiple parts. Each part of the network will have different blocks of the blockchain with different views of the ledger. Eventually, the consensus makes it possible that only one branch will win by being accepted by the majority of the network (50%+1). Once the other nodes notice the existence of another branch which is "longer" (with more computation done), these nodes will eventually drop their last blocks and re-branch to the fork that has more "work" involved.

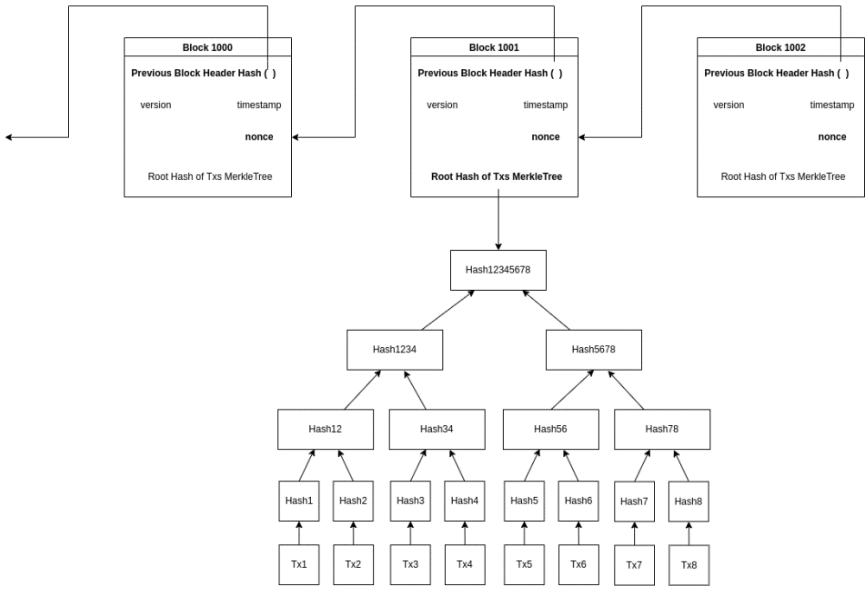


Figure 1 – The data storage scheme of the Blockchain used in the Bitcoin. The block is made of a Block Header and Merkle Tree of Transactions. The Block Header contains the hash of the previous block and the root hash of the Merkle Tree. The blockchain helps to validate the entire history make it easier to identify any forks.

The Merkle Tree is a binary tree data structure in which each leaf node is the hash of the data (serialized transaction), while each non-leaf is a hash of all its children. A Merkle Inclusion Proof is a method to prove the existence and validity of data. The proof is made by the hash of data and all hashes of the nodes climbing up the tree until we get the root hash which is included in the Block header. The Merkle Tree used in Bitcoin was presented by [3] to generate Merkle Proofs of inclusion to allow SPV (Simplified Payment Verification) clients to verify the existence and validity of any transaction in the blockchain. Satoshi stated that light nodes can verify the inclusion of any transaction in the blockchain by downloading and validating only the block headers and Merkle Inclusion Proof.

The novelty behind Bitcoin is its consensus model that has a built-in difficulty adjustment mechanism requiring Hashcash (Proof of Work) to ensure that only one block will be created every 600 seconds. In case in the past 2 weeks more than 2016 blocks were produced, the difficulty (which is a global constant in the network) will be increased to ensure that in the future (next 2 weeks) the number of blocks will be in the range of 2016. In case less than 2016 blocks were produced in the past 2 weeks, the difficulty will be lowered. This difficulty adjustment mechanism ensures that no matter how much computation power exists in the network, the number of blocks will be constant and generate a number close to 2016 blocks every 2 weeks.

To work properly and be trust-less, the nodes in the Bitcoin network will have to keep track of only the unspent outputs. By doing this, the nodes will be able to validate the status of the entire ledger and its entire history [3]. Users discovered that the Bitcoin protocol is lacking features like scalability and privacy being completely traceable. The privacy is not great because in Bitcoin the entire history of the ledger which is made of millions of transactions is publicly stored on all full nodes. Anyone can visualize the entire history of transactions using a blockchain explorer. Bitcoin

uses addresses allowing users to have pseudo anonymous identities on the network. It doesn't require registration or identification to participate in the network, but all the addresses are traceable. Bitcoin is unscalable because every single node in the network will have to process and include only a limited number of transactions in a block. Without having a limit, the network would grow without limits regardless to storage costs and bandwidth limits. Without certain limits, a decentralized network would grow to the point of being unusable as malicious nodes could create extremely large blocks with dummy data every 10 minutes. This would require other peers in the network to have extreme latency when downloading the blocks to validate the ledger.

The Bitcoin ledger relies on the concept of storing the ledger as a list of unspent transaction outputs (UTXO). By using this data structure, the nodes in the network will have to keep track of all unspent transaction outputs to avoid double spending. Once the output is consumed by a user, that output is marked as spent and cannot be used again. A transaction can have multiple inputs and multiple outputs. For each input, a digital signature must be included to prove that the sender knowns the private key. The full nodes also check that no coins were created out of thin air in any specific transaction. Thus, the full nodes will verify that the sum of all outputs is less or equal to the sum of all inputs. If the sum of outputs is less than the sum of inputs, the difference is claimed by the block producer.

The UTXO model provides a little bit more privacy especially when the addresses are rather unique each time and not being reused. When someone receives coins in the Bitcoin network, we say that one received an unspent transaction output with a small amount of digital currency. This output can be claimed by the receiver at any time in the future. The receiver will sign a transaction demonstrating to the network the knowledge of the private key of this address and confirming the way the user will consume the output.

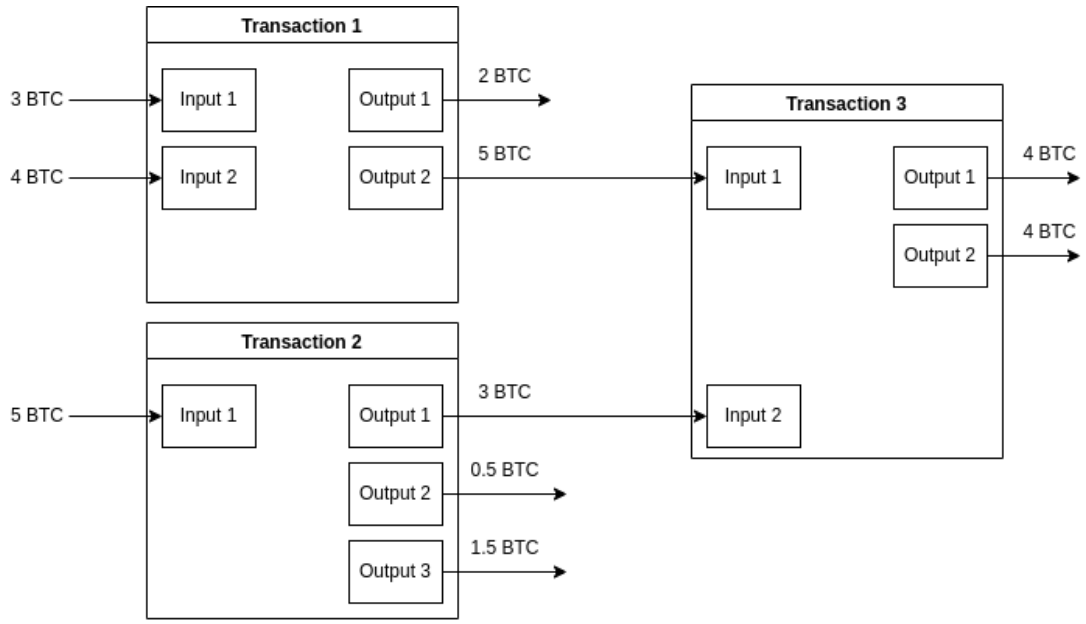


Figure 2. The UTXO model used in Bitcoin for representing the ledger. Each transaction requires the digital signature of every input. The outputs are used to send coins to other addresses or as change. Once the input is consumed, it will be rendered as spent to avoid double spending. In case a user wants to send only a fraction of the input, it is mandatory to include a change output.

In July 2014, J.D. Bruce published a paper entitled: „The Mini-Blockchain Scheme". In [4], the author proposed a new solution to store a distributed ledger in a more efficient way. The new storage scheme replaced the UTXO model proposed in [3] by Satoshi Nakamoto with an Account model. Instead of tracking (and storing) all unspent outputs, [4] proposed the creation and usage of a simple database that stores the balance of all non-zero addresses. This simple, yet ingenious storage mechanism enables the pruning of all previous stored transactions. It also solved the issue of dust transactions. The Bitcoin network is known to have a lot of small, yet positive unspent outputs which cannot be pruned and require storage space to keep them for an indefinite amount of time. Some of these dust outputs would be more expensive to be claimed in terms of fees than their intrinsic values.

The author proposed the usage of Patricia Merkle Tree data structure (a modified Radix Trie with Merkle Tree properties for security) to minimize the space requirements for storing the digital ledger. Each address (account) is a leaf node and has a deterministic and unique path in this hash tree. This data structure provides O(log N) complexities for operations like inserts, lookups and deletes. It also offers the best compression performance for storing a large Key-Value dataset.

The security considerations are still guaranteed by requiring every node in the tree to have a hash associated. The hash of each node is computed based on the hashes of its child nodes. This mechanism allows the efficient and secure verification of its entire content and could be used by the SPV light clients. At any time, the full nodes can generate a fast and succinct Merkle Proofs for the existence of an element in the Account Tree.

The proposal allows the pruning of all past transactions (inputs, outputs, digital signatures, nonce, etc.). To achieve data pruning, we only need to keep the Account Tree, while the entire list of past transactions can be completely pruned. Because the full nodes can delete all previous transactions (including their hashes), this introduced another issue: replaying transactions. A transaction could be replayed multiple times if the sender has enough funds until eventually his account gets drained. To solve this issue, an incremental account nonce can be used and stored in the Account Tree. Every time a transaction is included in the blockchain, the nonce of the sender account is increased. This would allow for the full node to detect a replay of a transaction.

The problem with dust accounts (addresses that hold 0 or a very insignificantly amount of coins) can be solved by forcing the full nodes to remove the accounts from the Account Tree after a certain amount of time.

The Mini Blockchain scheme is widely used in many blockchain implementations including Ethereum, WebDollar, Mina, etc.

In 2014, another paper with the title: „Cryptonote" was published on the internet by a pseudo anonymous author, Nicolas van Saberhagen [5]. In this paper it was possible to create a decentralized and trustless P2P cryptocurrency network that was more fungible. Cryptonote protocol introduced some degree of privacy having transactions unlinkable breaking transaction graph. This was achievable by using two new concepts namely one-time ring signatures and a variation of stealth addresses called non-reusable addresses. Ring signatures are being used to sign a message by a member of a set of addresses without revealing which member was signing the message. This is used to avoid leaking the information which account is spending the funds. Non-reusable addresses (stealth addresses) are being used to avoid reusing the same address multiple times. The recipient of every transaction in the Cryptonote protocol is a new public key that is derived from the recipient master key using a random seed generated by the sender. [5] This feature however requires users to scan the blockchain to figure it out if the receiver address is owned by them or not.

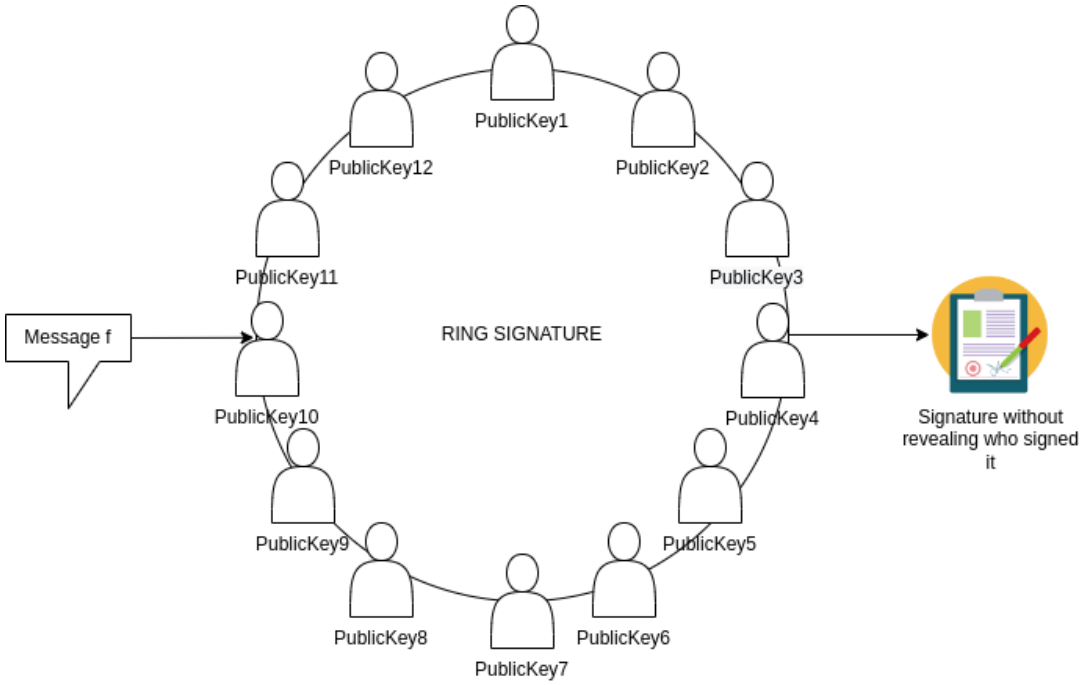


Figure 3 – Ring Signature with 12 Public Keys. One-time ring signature means that signing the same message f by the same signer would produce the same signature.

Cryptonote by design uses the UTXO model. To solve the double-spending problem when the same user creates different ring signatures spending the same input multiple times, every ring signature must contain a unique key image. This key image used in the ring signature is based on a one-way cryptographic function that is unique to every unspent input. Creating multiple ring signatures for the same unspent input will result in the creation of the same key-image. This would solve the issue or double-spending the same input by creating different ring signatures.

Because transactions' amounts are disclosed (publicly known values) it would be very hard for one user to find other unspent outputs on the chain that received the exactly same amount. To overcome this challenge, cryptonote forced users to pay the transferred amounts using different denominations (eq: 0.1, 1, 5, 10, 25, 50 units). Denominations are just like banknotes. Multiple vector attacks were discovered for cryptonote like Dust attacks, Value Based Interference Attacks, Empirical Traceability based on Deductible Transactions. These attacks can help chain analysis tools in many cases to figure out many deductive transactions. Based on the first deductible transactions, the chain analysis tools could deduce other transactions. According to a research [6] the original cryptonote protocol implementation had a significant fraction of over 91% of transactions with one or more mixins deductible.

In 2017, Monero a cryptocurrency that is a fork of Bytecoin (first implementation of Cryptonote) introduced Confidential Amounts hiding transactions values using zero knowledge proofs. This major protocol change called RingCT solved the requirement of using denominations when spending coins. The RingCT avoids entirely the problem of partitioning users' coins into denominations. The upgrade also prevented the value-based interference attacks described in the previous paragraph. Because Monero was lacking various requirements in choosing the transactions mixins used in the one-time ring signatures, Empirical Traceability was possible by creating

a chain of Deductible Transactions. According to [6], about 40% of transactions were possible to be traced using a chain analysis tool developed by the authors. This attack was possible due to the fact the many mixin members were easily deductible being weak cryptonote unspent outputs. If a mixin is made of only deductible ring members (unspent outputs) then this transaction would be deductible as well. This attack vector was later fixed by requiring transactions to have an exact number of ring members (11) and by adopting a stronger strategy of how to choose mixin members. In August 2022, a hard fork of Monero with version 0.18 increased the ring size from 11 to 16 ring members making all transactions a little bit more unlinkable.

Zcash is another privacy-centered cryptocurrency that uses zero-knowledge proofs called zk-SNARKs to provide anonymity to its users. In Zcash there is a transparent (public) pool and a shielded (anonymous) pool. In Zcash full nodes keep a list of commitments that have been created and a list of nullifiers that had been revealed when inputs (commitments) were spent. In a spend transaction, the sender will use the spending key to generate and publish the nullifier (the hash of the secret r for a commitment C) and provide a non-interactive zero-knowledge proof that he knows the secret key for spending it. Zcash protocol required a trusted setup [8] which was done by using a ceremony. The ceremony had to be redone every time there was a major upgrade of the cryptographic protocol. In the later versions of the ceremony, the Zcash developers enabled Multi-Party Computation (MPC) ceremonies allowing the participation of multiple users, making the ceremony more secured.

Basically, in Zcash protocol users create two types of zero knowledge proofs. One proof shows that a note exists in the Note Merkle Tree, while the second proof shows that no corresponding note-nullifier exists in the nullifer tree. If the proofs are valid, the note will be destroyed by appending the nullifer to the nullifer tree.

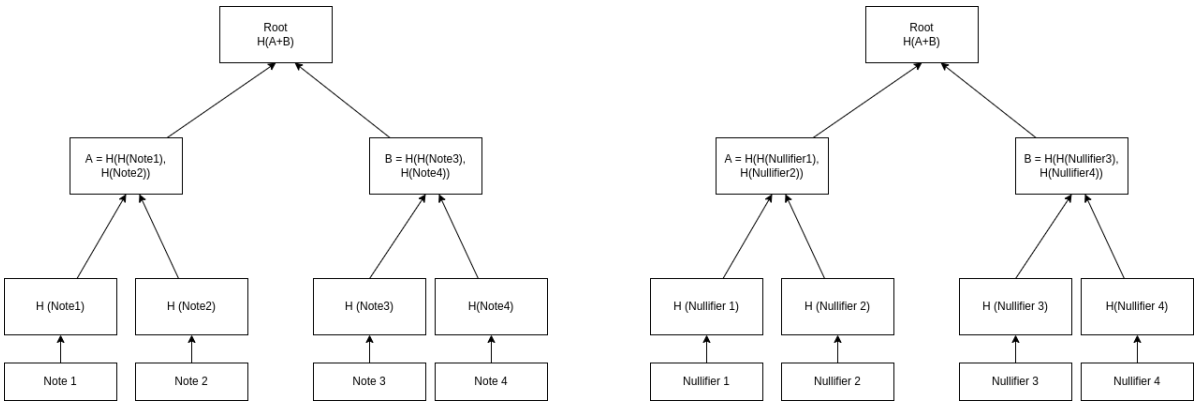


Figure 4. Zcash Merkle Trees used for Note Tree (left) and Nullifier Tree (right).

By having two sets (one public and one anonymous) Zcash users can easily be traced by user behavior. [7] This is possible especially due to an overwhelming number of users that prefer to do transactions using the public set. Most of the Zcash users use the public set because it is faster and cheaper to use.

Other projects like Aztec Protocol which is a smart contract running on the Ethereum network implements the same Zcash protocol in solidity. The Aztec Protocol enables users to deposit funds to the shielded pool. Once the funds are deposited, users can transfer each other privately notes using a 2nd layer private network. After many transactions on the 2nd layer, users are allowed to withdraw funds from the pool to their main Ethereum balance. However, the gas linkability problem still exists. Users will have to pay gas fees for interacting with this smart contract. The gas fees must be paid from a public Ethereum address.

Halo 2 protocol is a new privacy-oriented protocol published by ECC in 2019 [9]. The protocol works at the surface like the original Zcash protocol, but it does not require a trusted ceremony and has better performance.

Zcoin is another privacy-oriented cryptocurrency that uses zero-knowledge proofs. According to the Zcoin protocol paper [10], users are able to mint a new coin with a random serial number S and commit it by using a secured digital commitment

scheme. The commitment C is basically a coin that can be revealed by another secret r which will reveal the serial number S. The commitment C can be appended into an append only list of publicly known commitments C1, C2, ..., Cn. To create a valid transaction, the sender will have to create a non-interactive zero-knowledge proof that he knows a commitment in the list C1, C2, ..., Cn, without revealing which commitment was selected. The sender then will have to create another zero-knowledge proof that he knowns the secret r to the commitment C which will open serial number S. To avoid double spending (a sender reusing the same commitment C), the network will have to check that the serial number S was not previously seen in any other spend transaction on the blockchain [10]. Zerocoin has many drawbacks like it requires fixed denominations, it does not support payments of exact values and the amounts are publicly known on the network.

Firo (formerly known as Zcoin) proposed a new protocol called Lelantus [11]. One of the major improvements compared to the original Zerocoin proposal is that it no longer requires fixed denominations. By using zero knowledge range proofs the protocol supports the transfer of arbitrary values. In Lelantus protocol each coin has multiple parameters: a value v, a spending secret key, a unique serial number S, and a random r. An unspent input is represented by a commitment scheme. Once the unspent input is consumed, the commitment of serial number will be published on the blockchain to avoid double spending. The serial number behaves like a nullifier for the unspent input consuming it.

Mimblewimble is another privacy-preserving model which was initially proposed by a pseudo anonymous author under the alias of Tom Elvis Jedusor in 2016 [12]. The protocol was later improved by Andrew Poelstra and firstly used in Grin blockchain, then Beam and later adopted by Litecoin. In Mimblewimble, users' transactions are aggregated into one "super-transaction" that will be stored within

the block. All inputs and outputs from all transactions are CoinJoined together into one super-transaction before including them into a block. The amounts are confidentially encrypted using zero knowledge. The critical issue with this model is that the CoinJoin used in MW requires miners to build up one transaction at a time, while users' transactions are being created and broadcasted at different moments of time and submitted from different locations. Thus, Mimblewimble protocol is extremely vulnerable to sniffing attacks, where attackers can simply just sniff the transactions broadcasted in the network before they are aggregated into the block.

Dandelion protocol is being deployed to allow users to obfuscate their IPs, but it is also used as an extensive protection against sniffer nodes. After two transactions cross any Dandelion chain, both of these transactions will get aggregated by these nodes. By the time it gets broadcasted to every other node, the transactions get aggregated making it impossible to get disaggregated by an external observer.

However, in late 2019, Ivan Bogatty, a blockchain researcher published a medium article [13] claiming that he was able to de-anonymize the unlinkability of 96% of all transactions on the Grin network. He was able to deanonymize all these Grin transactions by using a simple sniffer node that costed him $60/mo to operate. He changed the source code making the node to connect from 8 nodes to 200 nodes out of total of 3000 nodes. By just using one single node, Ivan was able to learn the transactions before they could get aggregated. It is worth nothing that the author was able to deanonymize only the transaction linkability and not the transactions confidential amounts. Unfortunately, we believe that Mimblewimble unlinkability design is broken as an attacker could do an effective sybil attack in the network learning most, if not all transactions before they get aggregated.

Because Zcash, Zcoin, Firo does not rely on one-time ring signatures, it will not be presented in details in this paper. Monero on the other side uses one-time ring

signatures and it is very similar in design and functionality with the Zether, the model that is used in this research paper.

1.3 Zether

Zether is a new privacy-preserving scheme proposed by Bünz, Agrawal, Zamani and Boneh [14] to allow privacy-oriented payments in distributed and decentralized ledgers. The Zether model was initially designed by the authors to enable privacy-preserving transactions with confidential amounts without requiring to scan the blockchain and have no trusted setup. Moreover, Zether was designed by the authors to work as a smart contract running on a Turing-Complete smart contract platform like Ethereum. Another decision that was taken into consideration is the possibility of running Zether protocol in a private system that uses a scalable centralized database that could be operated by an entity allowing users to do anonymous payments.

Prior to Zether, all known implementation of privacy-preserving blockchain models relied exclusively on the UTXO model. Thus, all previous privacy-preserving models required users to scan the blockchain in order to determine the available funds. This process of scanning the blockchain requires users to download and run a special verification function for the meta-data of each transaction found on the blockchain. The verification function would eventually identify if the unspent output belongs to the user or not, and decrypts the available amount of coins. For example, scanning the Zcash or Monero without providing a starting block height may take even days. The UTXO model is also unscalable as the required time for scanning would only increase with number of transactions. If UTXO privacy models would get more adoption, users would need more time to sync in order to decrypt their available balance, which is made of unspent outputs. It would require users to operate their own full noes in order to ensure privacy guarantees when signing new transactions

and scanning for available balance. In addition, because it is impossible to tell if unspent outputs were consumed or not, some data of each transaction is unprunable. The full nodes must retain some information for each transaction since we don't know which unspent output was consumed or not. In case of Cryptonote, the blockchain can not be pruned since users can use funds from the network's early days to create a valid mixin transaction. The Cryptonote protocol employ a nullifer method to prevent double spending. It forces the sender of a transaction to generate a deterministic nullifer. If case of a double spending, the nullifer would be present in the list. By August 2022, Monero blockchain grew to 150 GB.

The Cryptonote blockchain also has some major issues with chain forks. Let's suppose the blockchain is forked and a new network emerges. When a user wants to spend his output on both chains, the same nullifer would appear and chain analysis could identify which unspent output was really consumed.

Each Zether account is identified by a valid ElGamal public key. The Zether account is stored in the blockchain and needs to be registered first. The registration is required as the balance is homomorphically encrypted to the attached ElGamal public key of the receiver and it should not be invalid. Zether introduces the concept of accounts that hold the available balances in an encrypted state consisting of an ElGamal ciphertext. The ciphertext encrypts the account's balance under its own public key. Only the user who owns the private key can decrypt his account balance and create transactions to transfer funds from his account.

Zether uses ElGamal homomorphic encryption which allows various mathematical operations (like addition, multiplication and subtraction) on encrypted ciphertexts. The sender of a transaction will update the encrypted balances stored in Zether accounts by performing homomorphic operations. Some on-interactive zero-knowledge proofs are required to be generated to ensure that the performed

homomorphic operations were all valid and didn't damage the encrypted values or alter them in a way that was unintended.

In order to prevent double spending, another zero-knowledge proof is required to prove that the sender has access to enough coins for this transaction. Let's say the Sender wants to transfer X coins from his ElGamal encrypted balance. The sender has to demonstrate the following conditions:

1. $X \in \{0,1,2,\ldots,2^{64}-1\}$ without revealing the actual number.
2. The sender's initial encrypted balance is greater or equal than X without revealing the initial encrypted balance.
3. The sender homomorphically subtracted amount X from his encrypted balance.
4. The sender homomorphically added amount X to the receiver encrypted balance.
5. The sender homomorphically added value 0 to all other addresses that are used as decoys in the transaction.

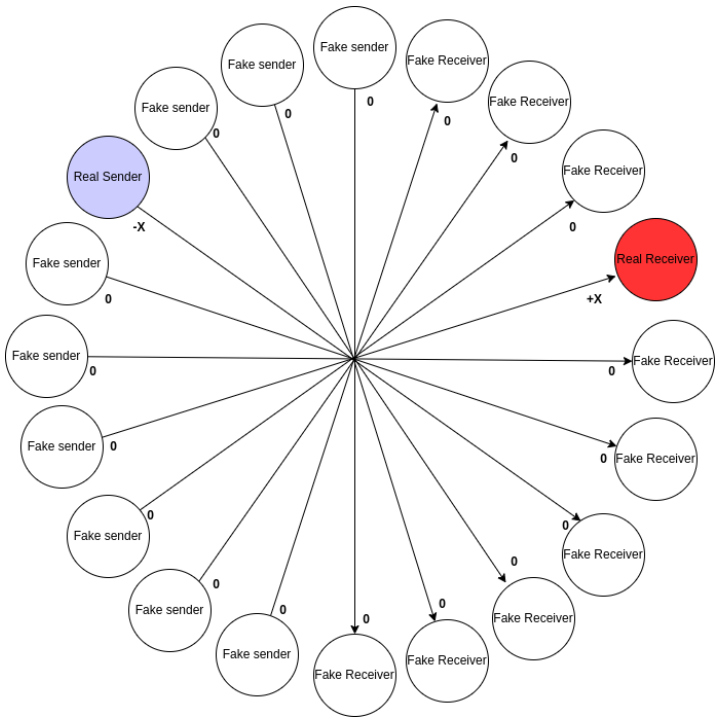


Figure 5. Zether transaction diagram. In a Zether transaction we have only one sender and one receiver. The rest of the addresses are used as "decoys" receiving zero. Using One-Time ring signature, external observers can't determine who is the sender. By using zero knowledge and ∑-Bullets, the encrypted amount X is homomorphically subtracted from the sender and homomorphically added to the receiver without revealing who these addresses are.

The above diagram might be confusing. The Zether protocol proposed by the authors does not leak the subrings configuration. It does not leak which is the sender subring and which is the receiver subring. The total number of ring members in a Zether transaction must be of power of two.

The protocol can be described in the following steps:

1. Every ciphertext is encrypted using the corresponding public key.
2. The sender has authority to spend from his account.
3. The sender ciphertext should not overflow from his account.
4. Two ciphertexts have non-zero and opposite values.
5. The rest of the ciphertexts should be zero.

Zether transactions uses one-time ring signatures and ∑-bullets to generate non-interactive zero-knowledge proofs that can be verified by any other computer in the network that the transferred amount is a positive number and nothing was created out of thin air. This last condition is important to avoid double spending. If the sender is not revealing the amount (value) he is transferring, he might try to transfer a negative amount, or simply just forget to decrease the transferred amount from his own encrypted ciphertext. The zero-knowledge proofs don't leak any other information about the transaction.

1.3.1 Front-running

One of the biggest issues with the Zether protocol is that the zero knowledge proofs are generated based on a certain view of contract's state. As it was described above, the sender has to prove that the remaining balance remains positive after the transferred amount was homomorphically subtracted. Some transactions could get front-ran by other transactions, rendering them invalid by containing an old view of the contract's state.

Let's imagine a case when Alice is transferring funds from her account to Carol generating a proof using its current account balance (encrypted ciphertext). Let's suppose another user Bob transfer some coins to Alice, and Bob's transfer gets processed first (included in the Blockchain). In this case, Alice transaction would get rejected by the network because the generated proofs will no longer be valid as the state of the contract was changed. In this case Alice may lose transaction fees and will have to redo the transaction. This problem was solved by the authors with a complex solution using a mechanism of Pending Transfers and a Rollover system. All incoming transfers remain in a pending state. After a certain amount of time (when the epoch is changed) these transfers are being rolled over into the accounts so that the funds could be spent. This method introduced a double-spending attack. An attacker could try spending funds twice in the same epoch. The authors [14] solved this issue by creating a nonce list every epoch to help detecting when a sender spends funds. Thus, all accepted private transfers in an epoch contain only unique senders and an unlimited number of recipients. The nonce list guarantees that a double spending would be impossible to be done in any given epoch. However, there could be network delays like the epoch changes sooner or later, and this could still render certain transactions invalid. We believe that the authors' proposed solution is not solving the front-running problem in a real-world environment.

1.4 Many-out-of-many-proofs and applications to Anonymous Zether

Multiple researchers at J.P. Morgan and Chase AI Research group improved the original Zether proposal. Predominately a researcher Benjamin E. Diamond dedicated a tremendous amount of time to improve the original design and make it technically feasible by reducing the size of the proofs and improve the verification speed [15]. The group from J.P Morgan and Chase AI Research did an implementation of their proposal in Solidity to run on their private Blockchain called Quorum Blockchain. The implementation can be found at this github repository. The group implemented the prover code in JavaScript to run on the user's device, while and the verifier code in Solidity to run on a smart contract platform like Ethereum or Quorum. The smart contract is made of three public functions which allow users to Deposit to the SC pool, Transfer Privately inside the SC and Withdraw funds from the SC pool.

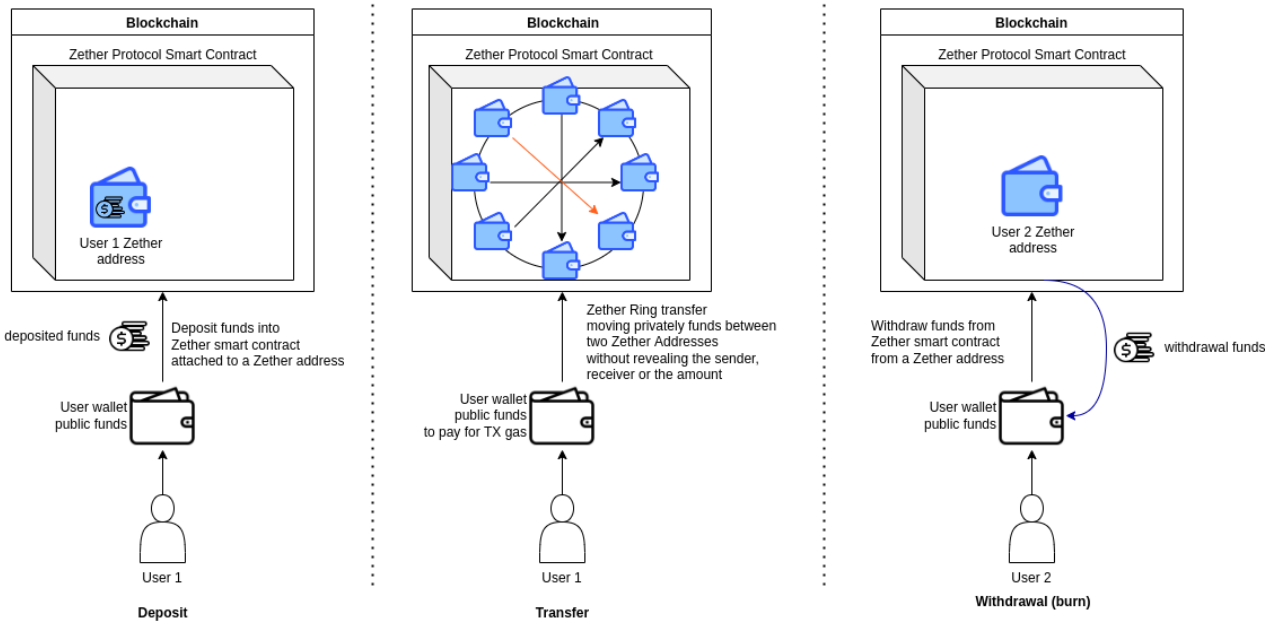


Figure 6 Diagram showing the three interactions with the Zether smart contract: Deposit, Transfer and Withdraw.

We describe below the three types of actions in the Zether smart contract:

1. Deposit Funds. It transfers user's native tokens into the Zether smart contract pool. User's encrypted ciphertext is also homomorphically increased with the transferred amount.
2. Private Transfer. It allows users to transfer funds from one Zether account to another without revealing the amount, sender and recipient.
3. Burn Funds. It allows users to withdraw tokens from the smart contract pool back to the native token. The user will burn a certain amount from his Zether account and redeem it back in the native token on the blockchain.

The author in work [15] achieved a proof protocol with the following specifications:

- Proof size: O(log N) - 3.2kb
- Prover runtime: O(N log N) – 2 seconds
- Verifier runtime: O(N log N) – 100 ms
- Transaction size: O(N) – 4.5kb

We acknowledge the fact that the authors in [15] proposed the Zether protocol to run inside a smart contract. Users to maintain their privacy will require them to conduct as much business as possible within the Zether smart contract.

According to the authors [15] the proposed zero-knowledge proof system is an extension to the Groth and Kohlweiss's one-out-of-many-proofs presented at Eurocrypt 2015 conference. It allows proving the knowledge of a secret in a subset of a public list and asserting that the commitments inside the subset satisfy various properties. This allows to prove that the sender and the receiver of a transaction are in a subset of list of public keys. This way we can create a transaction without revealing who the sender and receiver is.

2. Used technologies

The software implemented in this master thesis is divided into 2 distinct parts:

1. **Full node** – implements the blockchain, data structures and storage, cryptographic primitives, transaction creation and verification, wallet functionality, consensus rules and peer to peer networking. Although the code is implemented in Go, a special module of the code with many functions was exported to Web Assembly (WASM) to run inside the Browser. The WASM bundle was later used for building a front-end app. This way parts of the Full Node were included in the Front-end app without rewriting the code.
2. **Front-end** – implements a web wallet and a blockchain explorer. It allows connecting to a full node via HTTP Websocket to download and explore the blockchain state. The web wallet allows the creation and verification of transactions on the client side. All the privacy concerning functions are done on the client side (user's computer) without leaking any kind of sensible data to any other machines.

In both technologies, we have used multiple open source libraries. We would like to mention that except the Falcon Bootstrap 5 Theme, every technology and library used is open source and free to use by anyone.

The source code developed in this master thesis was released as open source and free to use by any entity. We also acknowledge the fact that the developed source code received some direct and indirect contributions from different people. The Zether cryptographic protocol used in this master thesis is a reimplementation of the ConsenSys's repository and later translated by DEROHE Project in Go. The developed source is available on Github with instructions for installation.

2.1 Full Node

The full node implementation can be found on Github and was done in Go. The Go programming language is a statistically typed compiled language. We have chosen Go as a viable programming language because it is a good fit offering an excellent balance between performance (program speed) and fast development time. Go is a simple, yet fast programming language that offers memory safety and CSP style concurrency. Multiple modules were developed using multi-threading techniques to leverage on the fact that many modern CPUs are multi-core. This could not have been achieved if the full node was developed in JavaScript.

Because zero knowledge is very computational heavy, it was mandatory to have the implementation of the cryptographic primitives in a fast programming language that is competitive with C/C++.

Over than 10 different modules were implemented using multi-threading to offer the best possible speed and benefits of using multi-core CPUs. Probably an implementation in C/C++ or Rust of the cryptographic primitives would increase the speed with at least 25% or so. This is because Go does not offer automated garbage collection and the compiler is more optimized.

During various benchmarks, we have noticed a considerable downgrade in terms of performance of the code when compiled and run as Web Assembly. We believe that the garbage collector is causing Go Webassembly to have poor performance.

A JavaScript implementation of the ConsenSys Zether cryptographic protocol was implemented by the author in mid-2020 on GitHub cryptography and integrated in blockchain and kernel. During the implementation, major bottlenecks were discovered: slow performance offered by JavaScript and no support for easy multi-threading. A cluster-based implementation was achieved in 2021, but it was not great.

The Go full node reference code was implemented from scratch.

Figure 7 – Screenshot of 4 Full Nodes running a private test-net for testing purposes only. All these 4 nodes were synchronized achieving consensus.

2.1.1 Non-SQL Database

The full node is required to store the Account Tree and the blockchain on the disk. Over the time, the blockchain can be completly pruned, but the Account Tree must be kept forever. For this reason, we have used a transactional non-SQL database. Moreover, we have implemented the code to be agnostic to the underlying database by using a simple generic interface. Right now we have four different implementations for the key-value storage:

1. BoltDB – embeddable key/value database powered by a B+ tree that offers in-memory and disks persistence.
2. BuntDB – embeddable key/value database powered by a single B-tree that offers in-memory and disks persistence.
3. MemoryDB – a simple key/value database powered by in-memory map data structure.
4. JSDB – a JavaScript wrapper that makes the database accessible from the Go WASM Virtual Machine. It was implemented for the project to be used by the Front-end app. Localforge is the underlying JavaScript library that stores the data in a key/value fashion and uses both IndexDB and WebSQL as fallback stores for browsers.

The main reason behind using a Key-Value non-SQL database is that all the blockchain data (blocks, transactions and accounts) can be stored at unique keys. Moreover, the data is in append only mode. Namely, the data that is stored on the hard drive can only be appended or removed in the same order from the storage. This is the reason why the blockchain is called an append-only list. The probability for collision of the keys are very small namely $\frac{1}{2^{20}}$ for accounts, while $\frac{1}{2^{32}}$ for blocks and transactions.

All the storage libraries used offer ACID properties (atomicity, consistency, isolation and durability) for database transactions. These common properties guarantee data validity and integrity despite errors or unknown hardware failures.

The full node does not do any kind of complex search queries, like JOINS, GROUP BY, etc. For this reason, we decided to keep the database using a fast Key-Value embeddable storage. Moreover, Key-Value databases are more scalable being easier to get distributed in a large cluster of machines.

2.1.2 Networking

The full nodes communicate each other (peer-to-peer) via HTTP websockets. This was considered in order to reduce the latency in the network. Websockets also offers us the possibility of notifications in the network for certain types of clients. Websockets are handy when the clients can subscribe to certain events and triggers. If the clients are subscribed then a notification is sent by the server when an event is being triggered. Multiple events were implemented to trigger different actions when transactions are added or removed, or the encrypted balance changes for one of the subscribed addresses.

The full nodes have support for three types of APIs:

1. HTTP REST API – a simple http server was used to listen to certain commands and return the results accordingly.
2. RPC API – a simple http server that follows the RPC protocol was implemented
3. HTTP Websocket – a http server listens for websockets connections and accepts them. The websocket API allows different subscriptions to certain events like (encrypted balance changed), transaction was inserted or removed from the mempool, etc.

For the HTTP REST API and RPC API the exchanged data was encoded using JSON, while for the HTTP Websocket the data was encoded using msgpack. We

acknowledge the fact that fast and low-latency implementations could be implemented using TCP/IP or UDP sockets directly. The main reason why Websocket was chosen is that the browser clients can connect directly to the full nodes and ask different queries via the public APIs. Most of the APIs are common and have the same format.

The implementation of the Peer-to-Peer networking communication protocol relies on TLS (Transport Layer Security) encryption. We acknowledge the fact that TLS relies on the trust from DNS providers to provide the right encryption certificates.

2.1.3 Web Assembly Binary

Since we wanted to run in the web wallet many functions (cryptographic primitives, transaction creation, signing, transaction verification, fee calculation, balance decryption, HD wallet etc.), we had either to reimplement these functions in JavaScript or to compile them to WebAssembly (WASM). We decided to go for the 2nd approach compiling the code using the Go WASM compiler that is included in the official distribution. Another option would be the usage of GopherJS that will compile the Go into JavaScript code.

Making the code compile to WASM was a simple task thanks to the official library and documentation. We only had to disable specific libraries to no get compiled like TCP/IP, HTTP Server, non-SQL database that uses file storage, etc.

The hardest part came after we realized that the Go WASM machine did not have support for multi-threading. We noticed that the Balance Decryptor Table initialization would take over 40 seconds in the web wallet. During this time, the main thread would entirely freeze. For this reason, we have moved the WASM instance to run inside a separate Web Worker (thread). This solved the issue with the main thread freezing, but right now multi-threading in WASM for Go is not supported. After extensive research to see different alternatives to get multi-threaded support in

Web Assembly, we have come to the conclusion that it might take a few more years until we get multi-threading support in Go WASM. During the research, we came across to this GitHub issue in which multiple discuss this problem. Multi-threading in Go WASM might get supported in the following years, but for now it is single threaded. The performance in Web Assembly was noticed to be poor as well. For example, transaction signing running as a compiled code on a x64 architecture would take only 2 seconds, while compiled for WASM architecture running in Chrome would take 50 seconds.

2.2 Front-end

The front-end is the Graphical User Interface intended to be simple to be used by a large number of users who might not have any prior experience to cryptocurrencies. The front-end can be found on GitHub and was designed to be built in JavaScript using Vue3.js - a progressive front-end framework. The chosen framework was selected because it is small, compact, efficient and it allows reactive web designs. The front-end has two different functionalities:

1. HD web wallet – wallet management.
2. Blockchain explorer – exploring the ledger.

Both functionalities were built in the same application to avoid splitting the code in too many projects (repositories). The cryptographic primitives, network communications, data storage and the wallet functionalities were all imported from the Full Node code implementation in the Go programming language. This was achieved by using the Web Assembly (WASM) compiler. This WASM bundle will enable us to run the code, enable the data storage in the Browser, and run the peer-to-peer networking protocol. DOM wrappers were used for data storage in the Brower using IndexDB/WebSQL and the WebSocket client.

Javascript libraries and technologies that were used for the Front-end:

1. **Webpack 5** – module bundler for JavaScript to transpile the multi-file project into one unified JavaScript bundle compatible to ES5 or ES6.
2. **Hotloader** – module to enable fast development without manually recompiling the source code.
3. **Localforge** – library to enable simple key-value database compatible with IndexDB and WebSQL.
4. **Vue3.js** including Vue router, Vuex state management – progressive front-end framework to render dynamic components.
5. **Bootstrap 5** framework – as responsive design for the front-end. The front-end theme was done by a 3rd party expert and we consider it to be out of the scope of this master thesis. The theme is based on this template.
6. **hCaptcha** – is used to allow a testnet faucet. The captcha was used to avoid a bad actor to abuse the faucet draining it of coins.

Web Technologies used in the front-end app:

1. **Webworkers** – the Web Assembly instances are run as separate Web Workers. This was done to avoid blocking the main thread that runs the front-end app. The balance decryptor takes 40 seconds to initialize and the signing of a 256-ring size transaction takes 50 seconds. Without using Webworkers, the entire page would freeze for the time being making the users to think that the app crashed.
2. **Web Assembly** – the cryptographic primitives and many functions were exported from the Full Node as Web Assembly. The WASM files were imported and instantiated via two separate Webworkers to avoid blocking the main thread. We have developed a small library to allow the communication between the main thread with the web workers that run the web assembly modules.

3. **IndexDB** and **WebSQL** are the storage technologies that various Browsers offer to store persistent data in the Browser. The edges cases and the real implementation is handled by the Localforge library.
4. **Websockets** are being used to communicate between the Full nodes and the Front-end client. The communication protocol is being exported from the Go full node via WASM. The exchanged data is done using msgpack.

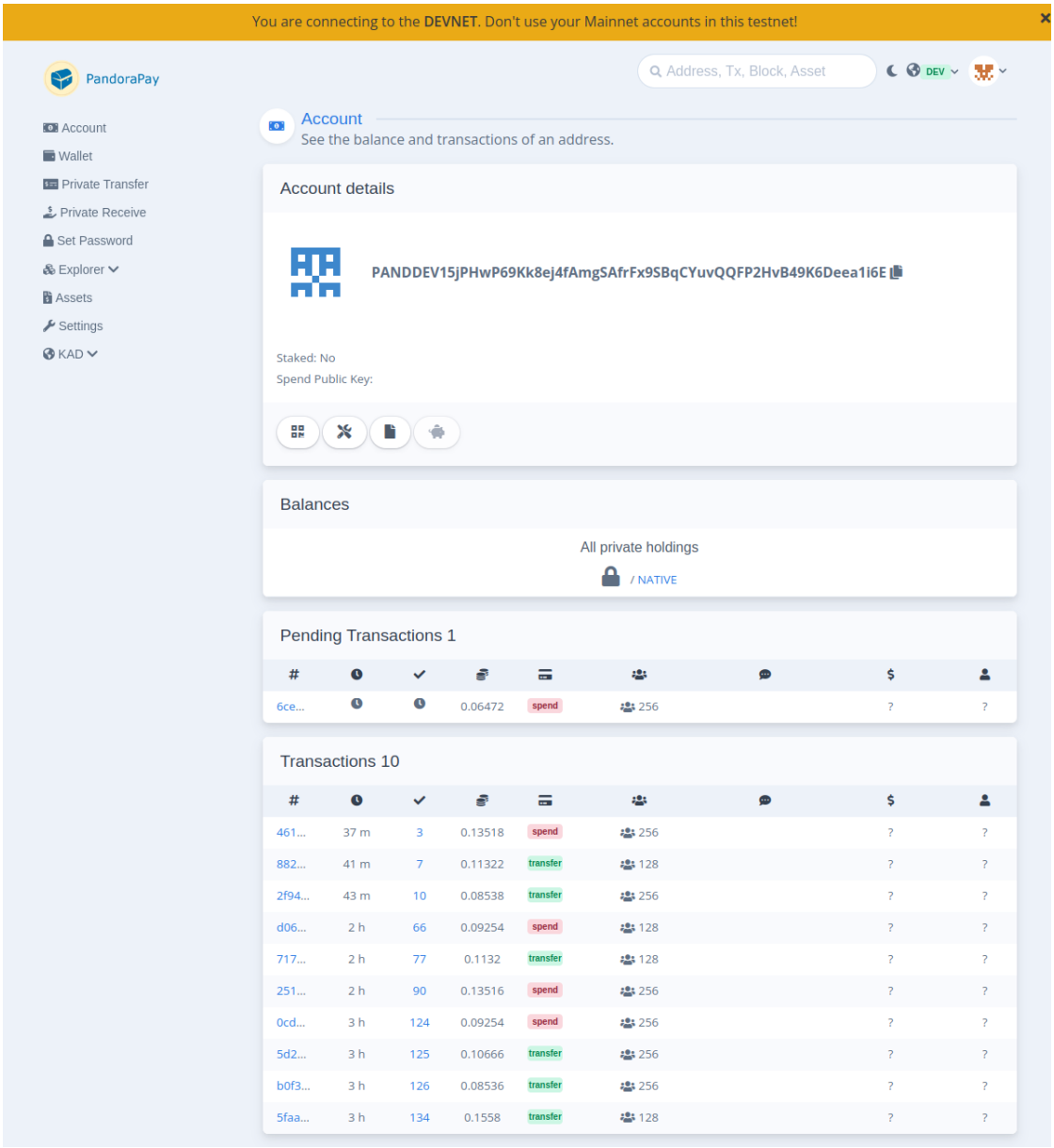


Figure 8 – Screenshot of the Web Wallet showing an Account.

2.2.1 Arguments for JavaScript based Front-end

The web technologies used in the front-end app enable us to create a simple yet cross platform application that can run on Windows, Linux, MacOS, Android, iOS and many other operating systems and distributions. The front-end can run inside of any modern Web Browser. The static JavaScript bundles can be served by a simple localhost web server or by an internet web server. In case the front-end is served by an external web server trust is required as the web host could send malicious JavaScript files that could steal the private key from the client browser.

The JavaScript Front-end could also run natively on desktop and mobile using the following frameworks:

1. **Electron.js** is a framework to enable the creation of a cross-platform desktop app that can be used to create a stand-alone executable application that will run natively on the host. The builds will run on Windows, Linux, and MacOS.
2. **Ionic.js** is a framework to enable the creation of a cross-platform mobile app that can be used to create a stand-alone mobile app that will run natively as an app on various smart phones. The builds will run on Android and iOS.

The two frameworks that were described above (Electron.js and Ionic.js) create native apps that render the front-end app using a Web View (an embedded Browser). In short, with very small changes, the Front-End can run as a Desktop app, Mobile app as well as a Web Application. This would be possible only with the help of JavaScript and the cross-platform implementation of the Web Browsers that can render our front-end app.

In late August 2022, a github user yuansong862 published on GitHub a repository which is an Electron.js app for PandoraPay making the Web Wallet run natively on any Desktop (Windows, Mac, Linux).

2.3 Open-Source Contributors

We would like to thank the many open source contributors who have helped make this project possible. Their contributions have allowed us to get closer to our goal of researching the possibility for a scalable privacy-preserving blockchain. We would categorize the open source contributors into two categories:

1 In-direct Contributors – These people contributed code or ideas to libraries and packages that were used in the development of the master's thesis. We would like to especially thank the following indirect contributors:

   1. Feb 2019 - JP Morgan & Chase, later renamed ConsenSys (after acquisition), started developing the anonymous-zether repository. The codebase consists of the implementation of the Anonymous-Zether paper [15] partially in Solidity (verifier) and partially in JavaScript (prover). It was used as the reference code.
   2. Dec 2019 – Ionut Budisteanu started the Anonymous-Zether.js repository. The codebase consists of a complete translation of the ConsenSys Anonymous Zether library in JavaScript.
   3. Dec 2020 – DERO Project started developing the derohe repository. The codebase includes the translation in Go of the Consesys's Anonymous Zether library. In this master thesis implementation, we used the derohe cryptographic primitives as a starting point to avoid having to rewrite the same cryptographic protocol in Go, translating it again from JavaScript. This was done to avoid including errors or mistakes in the cryptographic codebase.

2 Direct Contributors on github to the packages that were internally developed for this research project.

   1. go-pandora-pay contributors: zhen-kuo, xiangwei22, yuansong862, fl0ge
   2. PandoraPay-wallet contributors: yuansong862, xiangwei22, fl0ge, zhen-kuo

3. PandoraPay-website contributors: Cristian-Bejan, XuanMB, rcurelici, romeodutu, dandanidaniel, fl0ge.
4. Pandorapay-electron-js contributors: yuansong862

3. PandoraPay blockchain

In order to validate the privacy-preserving design of the Zether protocol, we have implemented the blockchain taking into consideration many of the restraints of the underlying protocol. We have decided to take in consideration all the tradeoffs in mind to maximize the privacy and efficiency of the underlying Zether protocol. For the internal storage of the state, we have chosen that the best approach will be to use the Mini Blockchain schema presented in [4]. The account-based model offers many storage benefits and it was chosen over the UTXO model. This makes a lot of sense taking into consideration that the Zether protocol creates a ledger made of encrypted ciphertexts using ElGamal homomorphic encryption

3.1 Improvements to the Anonymous Zether

The Zether protocol proposed by [14] and later improved by [15] was analyzed in many details. Some changes were required in order to maximize the privacy guarantees and make the protocol more efficient. We describe the proposed changes to the original Zether protocol in the following subchapters.

3.1.1 Account Registration using Encoded Addresses

According to [15], users first need to demonstrate the possession of their own secret keys. This is required before participating in Zether transactions to prevent a “rogue key” attack on the multi-recipient El Gamal. An attacker could craft a special key derived from an honest one which will allow him to decrypt the balance change in the honest key. To

solve this [15] recommended to simply require users to publish a signature that they know the secret key.

There are two ways to do account registrations:

1. Proof of work registration - demonstrating that some electricity was spent for the privilege to register an account. This is tricky because it will lower down the costs for an attacker to run a sybil attack that can deanonymize the linkability of the transactions. Basically, an attacker would be able to register an extraordinary number of accounts, reducing the privacy of the overall system. Moreover, this solution has a really bad user experience for regular internet users because it will require users to wait 30-60 minutes to register their accounts first time.
2. Encoded addresses – we propose the idea to encode the signature that the owner knows the secret key in his address. In case the account is registered, the address will be smaller (without including the signature), in case it is not registered, the address will be bigger (will have the signature included). When the sender wants to transfer funds to a new address or wants to use a new address as decoy, he needs to include this signature in the transaction. Because the registration will require to store extra data in the Account Tree, it will be a little bit more expensive for every account he is registering in the transaction. This way we can protect the network against the sybil attack describe above that could deanonymize the linkability of the transactions.

Encoded Address with Registration Signature embedded (not registered)
PANDDEV1URyyCUHyA1ino8zcbg5z86GGTKXe4aLriZStxqU2F8GPPPW5he9oDyhwe
rErGhxTgCpBzma8mj1ERtSJSX6fvrwAFjdoHwEW7fyyRCqebkw4sKBUKSZeJgEpYoA
TFeRRL52gNMxsfkZ

Encoded Address without registration (already registered)

PANDDEV16MqyR6ETKqUktw711BCs41jbzFcunGncBZF3JNyX1CaociwJW4uU

If a user wants to receive funds for the first time, he will have to copy the first address. Once the address is registered, the 2nd address (which is shorter) will be used by users to receive coins. The first address (the longer one) is still valid to receive funds. The address Check Sum algorithm was adapted to integrate the signature if it is encoded in the address. This way the check sum will be able to detect if a character was not copied correctly by the user.

3.1.2 Native protocol and asset

The first change was to make the Anonymous Zether protocol native to the blockchain. By integrating Zether as a native protocol of the blockchain, we would remove the need for the Deposit and Burn functions. This would make all the interactions on the blockchain private by default as all the users' transactions will be anonymous. This concept is called privacy by default and it is really helpful because most users are not experts in cryptography and network security. For this main reason it was decided to use the privacy by design architecture. The balances of all users are encrypted using ElGamal ciphertexts.

3.1.3 Solving gas linkability by subtracting Tx fee from encrypted ciphertexts

Although the authors [14] initially envisioned Zether protocol to run as a smart contract, this would introduce a new privacy issue breaking most of its benefits. All interactions on the blockchain with this smart contract would require users to pay somehow a small fee for each interaction (transaction). These fees are required to be paid

by the users with the native coin of the blockchain to avoid spamming the network. These fees are later collected by block producers (miners/forgers) who will be rewarded. The Zether protocol would require all users to have public coins available to cover various transaction fees for the interactions with the smart contract. By doing this, the Zether protocol running as a smart contract will lose most its privacy benefits requiring users somehow to pay coins from the public pool in order to cover the fees for their own transactions. By making the Zether protocol to be native to the blockchain, we still have to find a way to homomorphically subtract the fees from the encrypted ElGamal ciphertexts. The fees must be homomorphically subtracted from the sender's encrypted balance without revealing who the sender is or the transferred amount. By having Zether protocol native to the blockchain, we would solve this major issue requiring users to cover the fees from the public pool to do private transfers.

The Transaction Fees are public amounts that can deterministically be calculated based on the ring size and on transaction size in bytes. Having this assumption, the fees can homomorphically be subtracted from the encrypted ciphertext. Benjamin Diamond posted on Github and on the Quorum Slack group a modification of the original Zether protocol to accept subtracting (paying) the fees from the encrypted balance. This change will require the sender to prove that he subtracted the amount X (unknown) + fees (known) from his encrypted balance. This change would make the Zether protocol no longer requires users to have external available funds to pay the fees for their transactions. This simple, yet important change drastically improved the privacy of the original protocol removing the gas linkability problem.

3.1.4 Multi-asset Zether

It is a trivial change to enable the Zether protocol to accept and run multiple assets. Instead of keeping track of only one ledger of ElGamal ciphertexts, we can keep track of separate ledgers made of ElGamal ciphertexts. One ledger for each defined asset. We also

acknowledge that another zero-knowledge proof can be done to prove that the sender and recipient will operate on the same asset, without revealing the asset.

3.1.5 Paying multi-asset Tx fee with the underlying asset

This multi-asset feature presented brings a new major issue. How would the users pay transaction fees for transferring let's say Asset2 ? We should also take in consideration that the user might have or may not the native token of the blockchain. One naive idea is to enable multi payloads (transfers) in the same transaction. One transfer will be used to move the Asset2, while the second transfer would be used for cover the transaction fee. The issue with this solution is that the rings for both transfers would require to be identical because otherwise it would leak the sender of the transaction and break its privacy. The sender address would be in both payloads, while the other decoys will be completely different as they were randomly selected.

In PandoraPay we came up with a novel solution using a liquidity provider contract to solve this issue. Let's suppose some users might want to accept Asset2 tokens at a certain conversion rate and release native tokens in exchange. If the conversion rate is right, why wouldn't some users accept this?

Using the liquidity provider, the users would pay directly the fee using Asset2, while the liquidity provider will release the native tokens to cover the transaction fee at a certain conversion rate. We achieved this by creating a Max Heap that returns the best liquidity provider (biggest conversion rate) for the Asset2. The liquidity provider will exchange Asset2 for the native tokens at a predefined conversion rate. This way the liquidity provider will accept the Asset2 and pay back the native tokens to cover the transaction fees. These fees will be collected by the block creators (miners/forgers) when proposing a new block. In the next diagram we show the flow of the liquidity providers for a certain asset.

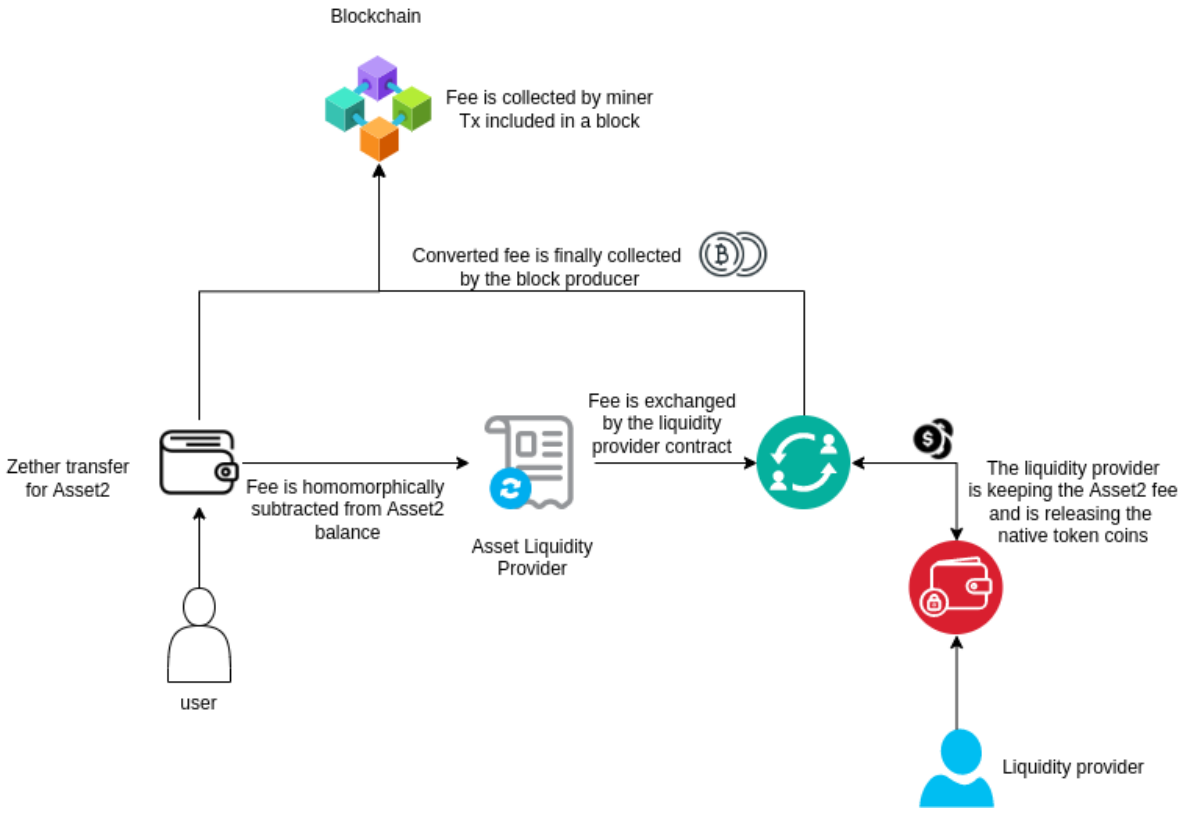


Figure 9. This diagram shows our proposed solution of allowing users to pay the transaction fee using the underlying asset.

3.1.6 Receiving multiple transfers

As it was described in the 1.3.1 front-running chapter, the Zether protocol still has one issue we need to take into consideration. An address might receive multiple transfers in the same block. There are multiple applications that makes an address to receive multiple transfers in the same block. Think of a donation address. An entity might publish its donation address and post some news regarding this. In a matter of seconds, dozens of users might rush to donate coins to the same donation address. Because of the front-running issue discussed in the previous chapters, most of these transactions will fail.

We propose a solution to separate the Zether ring members into two separate subrings. One subring for the senders (sender and decoys), and another ring for receivers

(receiver and decoys). Although this additional requirement lowers the transaction privacy as external observers will know the receiver subring and the sender subring, we are solving the above-mentioned problem. In the PandoraPay Zether implementation we have decided to go with this solution and leak the subrings configuration.

The proposed solution solves the above described problem. We can do this because all receiver subring members have encrypted ciphertexts that are all positive ($>= 0$). Since we know this fact, we could initialize the receiver ring members with 0 value encrypted ciphertexts. All the update ciphertexts for the receiver subring will contain only positive values. We will use the pending transfers solution by [14] to homomorphically add the encrypted ciphertexts to users' encrypted balances stored in the ledger.

Figure 10 Shows the way a Zether ring transaction is split into two publicly known subrings. One subring is made of the sender and decoys, while the other subring is made of the receiver and decoys. External observers can't determine who is the sender or receiver

Another advantage is that we save $64 * \text{ring\_size}/2$ bytes from the transaction size. In case a transaction has 256-ring members, this option to leak the ring configuration and initialize receiver sub-ring with 0-value ciphertexts will save 8 KB per tx.

3.1.7 Front-running transfers

As described in chapter 1.3.1, front-running is still a major issue for Zether. Let's say Alice is creating a transfer generating zero-knowledge proofs with a certain view of the ledger's state which is made of ElGamal encrypted ciphertexts. There is a chance that another transfer, let's say of Bob will get processed first and get included in the blockchain before Alice's transfer. Bob's transfer might use any of Alice's ring members as a decoy. This way, the Encrypted ElGamal ciphertexts would change (update) the ledger state, rendering Alice's transaction invalid. Although our implementation would simply invalidate Alice's transaction and require it to redo it, this would not cost anything. But in case Alice would rebuild the transaction with the updated ciphertexts, it would be important to use the same ring members because otherwise leaks could happen.

Using the solution proposed in the previous chapter 3.1.5, the front-running collision for the receiver subring was fixed. Right now, the front-running can happen only to the sender subring.

In PandoraPay we removed the authors [14] original idea of using the Nonce List and Epoch system because of two main reasons:

1. The Tx could still be invalid in case the System Epoch would change sooner or later because of network latency.

2. The probability of users to encounter the front-running transfer decreases in time as more addresses will be registered in the ledger. To illustrate this fact, we have done some simulations in the next page.

k registered addresses	Ring size	Sender ring size	Probability of front-running per one tx $\sum_{i=0}^{127} \frac{256}{k-i}$ /128
1,000	256	128	27.6%
50,000	256	128	0.5166%
1,000,000	256	128	0.0258%
10,000,000	256	128	0.00258 %

Table 1 – Probabilities to encounter the front-running collision taking into consideration only one Tx with 256-ring members was added in a block.

The probability of front-running collision increases in case multiple transactions will be included in the block. In the below table, we calculate the probability of front-running collisions in case the blockchain has already k = 1,000,000 registered accounts:

n transfers in block	Ring size	No of unique addresses used in block	Probability front-running for n-th+1 transaction $\sum_{i=0}^{127} \frac{256*n}{k-i}$ /128
5	256	1280	0.129%
20	256	5120	0.5160%

Table 2 – Probabilities of encountering the front-running collision taking into consideration that multiple transfers can be added in a block.

3.1.8 Whisper Protocol

Because the transaction contains only zero knowledge proofs and not the data itself (tx amount or receiver ring member index), the transaction is not sharing much information to the real receiver. By design, the privacy-preserving transaction defined in [15] is not sharing to the receiver such information. This means that the receiver of the transfer doesn't even know how many coins were transferred in the transaction and who sent it. This kind of information might be useful or required in many cases. Many users

may want to see the history of their account including last transactions to see who paid and how much, and who didn't pay.

The authors proposed an algorithm for the receiver to do a brute force scan in order to retrieve (discover) the number of coins transferred. This was proposed by the authors to avoid increasing the transaction size with additional space (information). But there would be many times of very large values being involved in transfers. Blockchain systems also need to use predefined usage of decimals. Especially in Zether, the number of decimals used in a unit can't be easily increased as the balances are encrypted and only the owners can decrypt them.

To illustrate this issue, we can think of the unit system used in other cryptocurrencies. In Bitcoin there are 8 decimals places for one unit. Sometimes big values like thousands of units might be involved in transactions. A transfer of 5000 units in bitcoin would be $5 * 10^{11}$. This brute force scan might take minutes or even hours in the web wallet! For this reason, in the PandoraPay blockchain we decided to allow the sender to encrypt to both the sender and the receiver the encrypted amount and which ring member received the funds. Namely, the transaction includes as a suggestion (whisper) to avoid the brute force scanning. The whisper information is a 32-byte encrypted information. The whisper information is optional and can be invalid. In case the whisper is valid, the whisper protocol can detect its validity. In just a few milliseconds the wallet can show the useful information like who received the funds and how many coins were transferred. In case it was invalid, (malformed by the sender), it would require the brute force scanning. But we decided to just show an error message.

The initial Whisper protocol was presented on the Quorum slack group by Zhou Zhiyao and Benjamin Diamond at our request. It was later proposed to be implemented in PandoraPay codebase.

Given the following:

b transfered value (funds) of the underlying transaction

i, j indexes of the sender and respectively receiver in the transaction ring

(x_k, y_k) private key, public key tuple of the k-th ring member

r shared public key of the transaction

Proof for Receiver of transfered funds

$$v = Hash(y_j^r) + b$$

Receiver can retrieve the transfered funds from proof v

$$b = v - Hash(D^{x_j})$$

Proof for Sender of transfered funds

$$v_2 = Hash(D^{x_i}) + b$$

Sender can retrieve the transfered frunds from proof v_2

$$b = v_2 - Hash(D^{x_i})$$

We need to verify the validity of b

$$g^{b-fee-burn} + g^r$$

We mentioned that both proofs v and $v_2$and computed by the sender when signing the transaction. The values v and v_2 are 32 bytes encrypted using the shared secret r. In total this extra information requires 64 bytes of extra storage per transaction. The verification is mandatory of the whisper as the sender could encode invalid (fake) whisper values.

3.1.9 Proving Tx amount and receiver without revealing the sender.

There could be many instances when a sender might want to prove the transferred amount and the receiver address to a third party without revealing other sensible information like his own address (sender's address). This could be used for multiple applications. It could be used in KYC applications. There could be cases when the sender might want to share it to a 3rd party for a dispute claiming that he sent the funds and never received the items/goods/services. It could be used in OTC transactions.

It is important to mention that this proof is not stored in the transaction and will not be required to be stored in the blockchain. The proof can be generated at any time by the sender and does not require any additional changes.

The original idea was proposed by Benjamin Diamond on the Quorum Slack group. The solution can be implemented in the Pandora Pay blockchain.

Given the following:

b transfered value (funds) of the underlying transaction

j indexes of the receiver in the transaction ring

(x_k, y_k) private key, public key tuple of the k-th ring member

r shared public key of the transaction

(C_j, D) ElGamal encrypted ciphertext

$g^b \cdot C_j = y_j^r$

$g^r = D$

Generating the proof (c, s) as following:

k a random element in F_q, we have $K_r = g^k$ and $Y_r = y_j^k$

$c = Hash(K_r, Y_r)$

$s = k + c \cdot r$

The proof (c, s) alongside with $(b,\ j)$ can be shared to the 3rd party who can verify the proof and the validity of its data.

$K_r = g^s \cdot D^{-c}$

$Y_r = y_j^s \cdot (g^b \cdot C_j)^{-c}$

if $c = Hash(K_r, Y_r)$ then the proof (c, s) is valid for given $(b,\ j)$

3.1.10 Solution against Sybil Attack deanonymization

Zether is vulnerable to Sybil attack deanonymization. Let's suppose an attacker will register 1,000 times more accounts than the honest users. The attacker can simply just register a lot of accounts. The blockchain doesn't even know when to delete zero value accounts as all accounts have encrypted ciphertexts. The attacker can just register and keep these accounts with zero balance. When an honest user will pick random decoys for the Zether ring, the user will have a high chance of picking accounts controlled by the attacker.

Let's think about a scenario where there are only 10,000 accounts registered by honest users. In this scenario an attacker does a sybil attack for deanonymization and register 10,000,000 unique accounts. The chance of picking an account controlled by the attacker is $\left(1 - \frac{10{,}000}{10{,}000{,}000}\right) = \sim 0.999$ out of 1

An approximate probability of the user to pick only accounts controlled by the attackers when creating a 256-ring size transaction is $1 - \left(\frac{10{,}000}{10{,}000{,}000}\right)^{254} = \sim 0.7755$ out of 1.

The proposed solution is to require a high fee for each unique registration. By making it extremely expensive for an attacker to register many new accounts, we reduce the chances for an attacker to do a Sybil attack for deanonymization. This registration fee

could be like 10 cents. This would require the attacker to spend 1 million USD in currency to try do this kind of sybil attack. This is just for a very small ecosystem.

For a global adoption, the costs for the attacker would be even more. According to [16] Bitcoin after 12 years of being active, it has 460 million registered addresses. To achieve the above probabilities, it would require the attacker to register 4.6 trillion addresses. For a fee of just 1 cent to register each address, it would cost the attacker 46 billion dollars. The attack would be so expensive for the attacker that it seems impossible to think that any attacker would spend that much money to try achieve a sybil attack on a PandoraPay network.

We also propose that the fee for registering a new account should be dynamic. It should be more expensive in the first months/years of the blockchain running, and it should be lowered over time when it would become too expensive for an attacker to try to do the sybil attacks.

3.1.11 Unspendable Private Proof of Stake Consensus

We have come up with a unique design to make the Zether cryptographic protocol to be used natively in the Proof of Stake consensus.

The Proof of Stake formula is well known and widely adopted by many blockchains. There are two major designs for implementation of POS blockchain, one for UTXO and one for Account based. We preset the Proof of Stake equation for the Account based blockchain used in the WebDollar cryptocurrency.

$$Hash\left(forger\_public\_key \cdot block_{timestamp} \cdot prev\_block_{kernel\_hash} \right) \leq \frac{available\_balance}{difficulty}$$

As we can see in the POS inequality, two things need to be publicly known: forger public key, and his available balance and a signature from him to confirm the proposed

block. The kernel hash is the chain hash without variable parameters that can be tweaked by the forger in a block (Merkle Tree root hash, transactions, nonce etc.)

The UPPOS consensus would require the creation of a Zether Transaction with two special payloads. One payload proves that the forger (without leaking his address) has enough available balance to solve the POS inequality. The second payload contains a new temporary account (that is not registered) which is updated with the Block reward and the user can transfer these funds to the forger address (without leaking his own address). By using Zether, we were able to achieve our POS version to run with encrypted balances and a way to avoid leaking the address of the block forger. In the UPPOS, the forger that proposes the new block is just a sender subring member without knowing exactly which is the real address of the forger.

In the original Zether implementation, the authors [14] proposed a protection mechanism to avoid transactions to be replayed by attackers. Their solution was that every account should have "associated" a nonce. Every new transaction from the same account will have to sign the latest value of the nonce. In our implementation we removed this protection because it is not possible to do replays in our network. This protection was removed because we check the encrypted ciphertexts for the sender subring to match with the ones stored in the current state.

To integrate Zether into a POS based consensus we got the idea that instead of using the *forger_public_key* disclosing his address publicly, we can use this Account Nonce that is unique per account and is committed in the zero knowledge proofs. We updated the original authors nonce to be unique also per block (using the public *prev_kernel_hash*). Any transaction the forger would build, and any sender subring configuration he would choose, it will lead to the same Nonce. This feature allowed us to solve the first challenge of having a unique hash per account, per block.

$$Hash(\ account\_nonce \cdot block_{timestamp}) \leq \frac{available\_balance}{difficulty}$$

The *account_nonce* is a unique hash that was adapted to be unique per block and forger. The *account_nonce* will be the same for the same forger (sender) without leaking the forger address, although he can choose any ring configurations. The *chain_kernel_hash* includes the commitment of the *sender_public_key* without revealing the sender.

To prove that the forger has enough available funds to stake, we would use the Burn parameter. Basically, the Burn parameter allows the forger to prove that the sender (forger) has a balance greater than a given value. This way we disclose that the payload sender subring has an available balance greater than the minimum required balance to solve the POS inequality without leaking who the sender really is and his total balance. It is important to not update the encrypted balance, otherwise the forger will lose the staked coins.

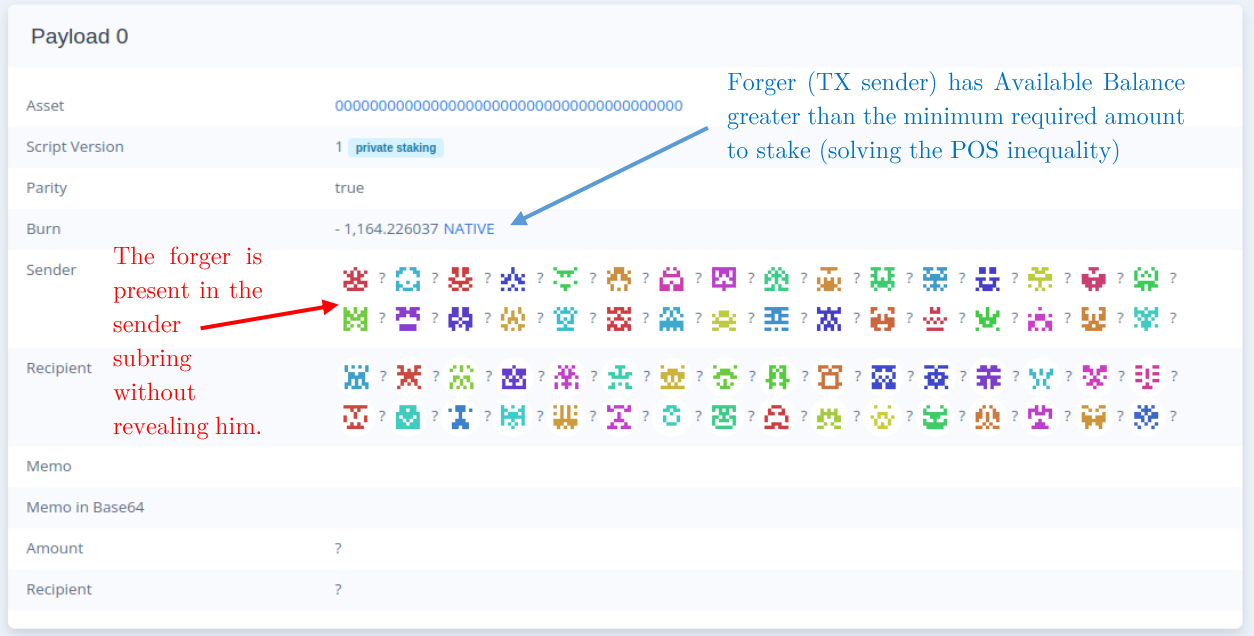


Figure 11 – Shows the Private Staking Payload where we prove that the forger (can be any sender ring member) has an available balance greater than 1,164.226037 coins (required for solving the POS inequality) without revealing his total balance.

The next question is how to reward the forger without revealing his address? We can do this using a second payload. We simply copy the sender subring and use it as the receiver subring in this payload. This payload doesn't leak any extra information about the Forger. Then in the Sender subring we require the forger to include a new temporary account (not registered) that will be credited with the Block Reward. The forger is the only one who knows the private key of this new temporary account and can create the payload to transfer the funds from the temporary account to his account.

Figure 12 – Shows the Private Reward Payload where the forger (can be any in the receiver ring member) received the block reward.

3.1.12 Mitigations for Short-Range and Long-Range attacks

We need to mention that the Available Balance must be locked for a certain amount of time against Short Range and Long-Range attacks. To illustrate the Short-Range Attack which is the most critical vector attack for POS consensus is that the Forger might generate an unlimited number of addresses and move his funds to a new temporary address that will be the lucky winner for the next block. The attacker might even try to create

small forks of 1, 2, 3 blocks and create new addresses for each block that will have very good chances to win all the blocks creating a longer chain.

The way to mitigate Short Range attack is the introduction of the concept of Available Balance for Staking. Accounts need to be registered for staking. Once these accounts are registered for Staking, their incoming balance updates will be credited only after a "grace" period. The available balance is done by having a pending state by homomorphically adding the encrypted ciphertexts of the receiver subring at a certain moment of time in the future (current block height + grace period). We can do this, because it is guaranteed that all receiver subring members have only positive values encrypted as ElGamal ciphertexts.

Long Range Attacks are mitigated by not accepting forks bigger than 10k blocks.

3.2 Overview of the infrastructure

The wallet functionalities were implemented in Go. The wallet runs directly natively in the full node via a command line interface. Moreover, there is also an application programming interface protected by a username & password to allow remote control of the wallet. Once the interface is enabled, most of the wallet functionalities can be accessed remotely via HTTP REST, RPC and Websocket. By default, the wallet APIs are disabled to protect users. In order to enable the wallet APIs, a username and password are required to be set via arguments.

The wallet functionalities were exported to Web Assembly. A user interface was built in JavaScript, HTML5 and CSS3. The web front-end is easier to be used by average users, while experts could use the command line wallet or even the APIs.

In the next diagram we present the three ways to use the PandoraPay wallet.

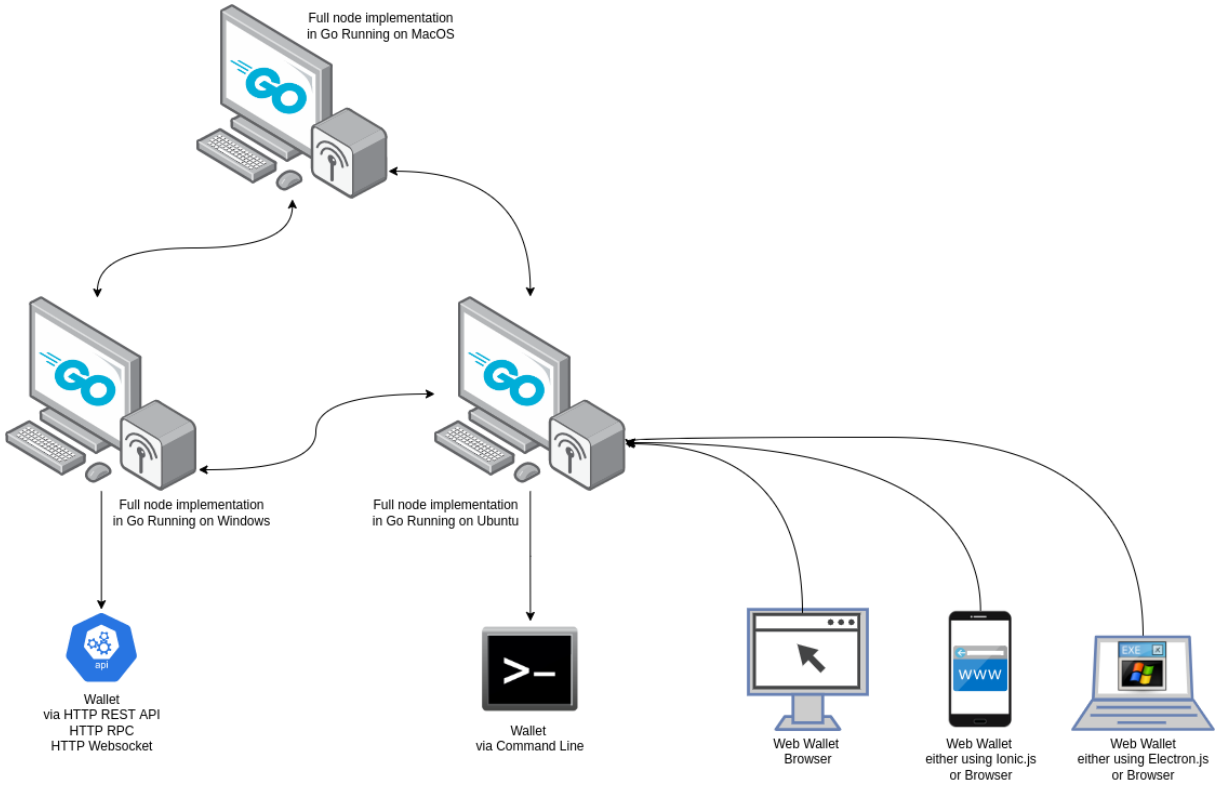


Figure 13. Diagram of the overall infrastructure for the PandoraPay network. The front-end is working right now on Desktop, Laptop, Smart Phone via the Web Browser. Using Ionic.js and Electron.js frameworks binaries can be built for apps that run natively for Mobile and respectively Desktop.

The peer-to-peer network requires a bootstrap process. A new node will connect to the seed nodes (which are publicly known) and ask them if they know other peers. A populated list is being managed by each node. This list contains an internal score which computes the probability that a particular node might be online and valid. The new node will try to connect to as many nodes as possible. Once it gets a considerable number of connections, it will try to disconnect from most seed nodes. But the consensus model requires that at least one connection should be kept to an honest node. This is needed in case of a sybil attack where an attacker start tens of thousands

of nodes undermining the number of real nodes. The global consensus will last as long as there are no connected components (splits) in the network.

We mention the fact that there are many discovery strategies that the new nodes can use. One way is to scan the entire ipv4 domain finding other online nodes. Although in theory it sounds like a bad idea to scan the entire internet, in practice it takes only 30 to 60 minutes to scan the entire ipv4 domain. Another solution is to memorize and save the contacts of the previous known nodes. In case of a malfunction (RAM, disk, CPU), the node might restart and it will know a huge list of other available nodes that might still be online.

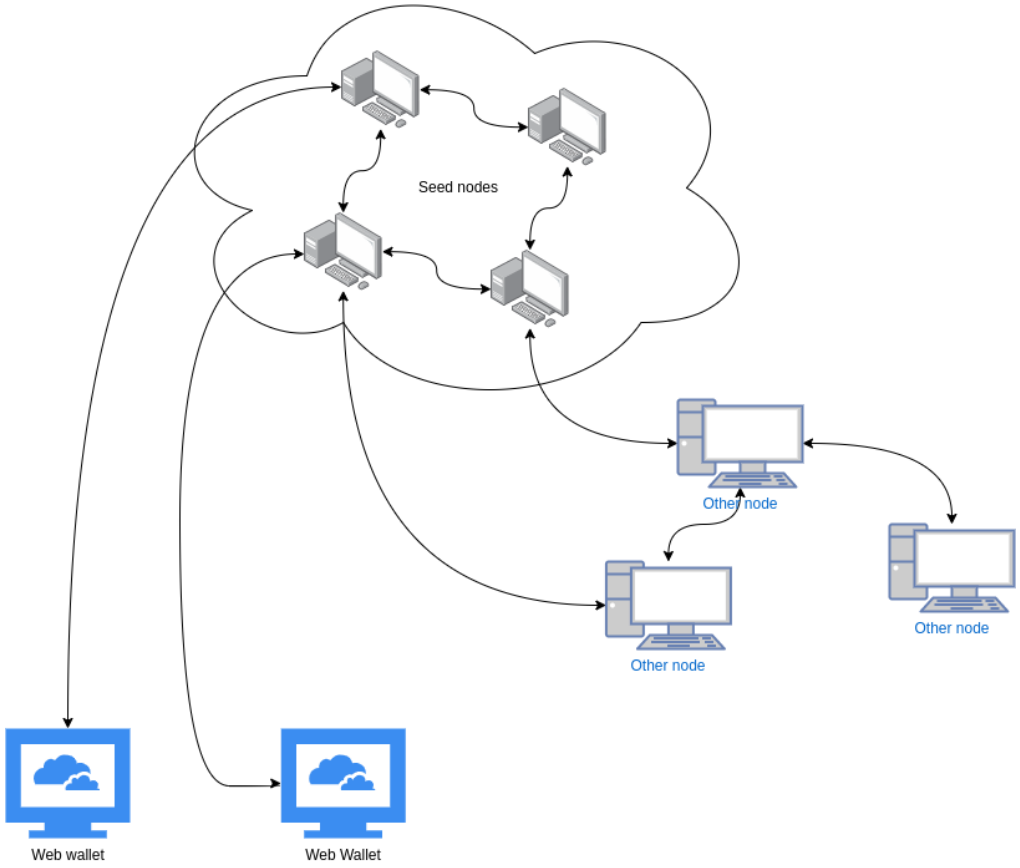


Figure 14. The peer to peer network in the current implementation. The consensus network requires at least one connection to an honest node.

3.3 Blockchain explorer

As it was mentioned in the chapter 2.2, the User Interface consists of a Web Application developed in JavaScript, HTML5 and CSS3. The user interface can be integrated using other JavaScript frameworks like Ionic.js and Electron.js to create standalone binaries that can run natively on various devices and operating system on mobile and respectively desktop.

The front-end app connects using the WebSocket protocol to a random full node. Optionally the source code can be changed to run as a full-node, but it would be slow and require a lot of unused storage space to download and synchronize the ledger. Many functions and modules including the WebSocket protocol was exported from Go to Web Assembly. The blocks and transactions are transmitted in the binary form (serialized) to save bandwidth. In the Blockchain Explorer, blocks and transactions are only deserialized and validated, but not verified. This decision was made because the blockchain explorer shows 10 blocks, and each block can contain up to 1 MB of data. This may require validating 200 Zether transactions and take a minute in a browser with the Web Assembly version.

The user interface allows any user to explore the blockchain and use it as a web wallet to receive and create transactions. The blockchain explorer allows users to view the latest blocks and transactions added to the blockchain, as well as see any assets created in the network.

The blockchain explorer was designed to be used for development and testing purposes. Regular users can also use it to search the blockchain for the existence of a specific block or transaction. The blockchain explorer doesn't download and validate transactions, it just provides some basic information.

In the web wallet, only the block headers are downloaded to explore the blockchain. The User Interface application was developed this way to reduce bandwidth and CPU use.

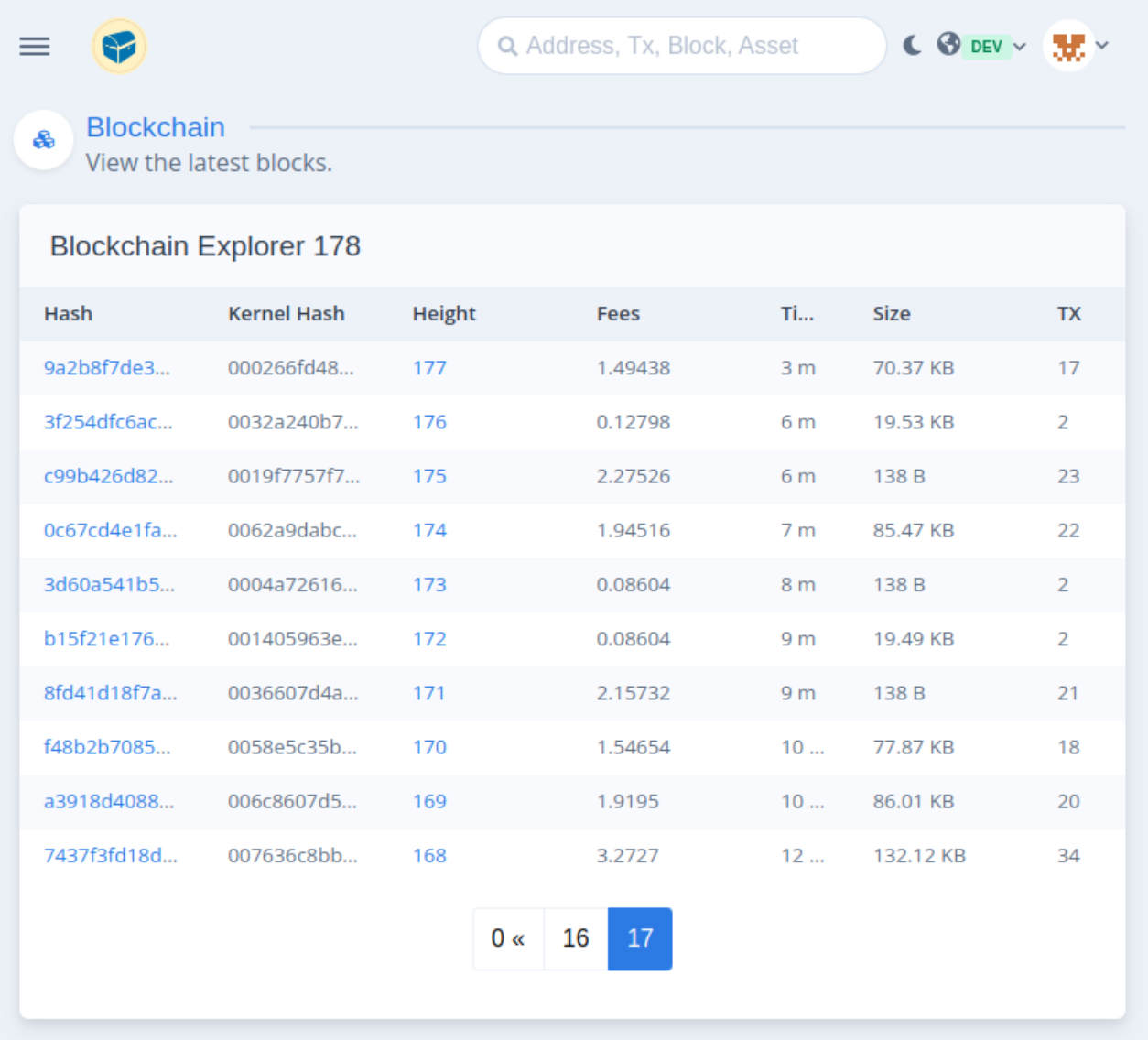


Figure 15 – Blockchain explorer. It shows the latest blocks added to the blockchain. The user can use pagination to view different blocks. The explorer is automatically update when blocks are added or removed from the blockchain using notifications APIs.

Block Explorer - Users can view blocks by height and hash by request. The block is not completely downloaded, rather only a preview is downloaded.

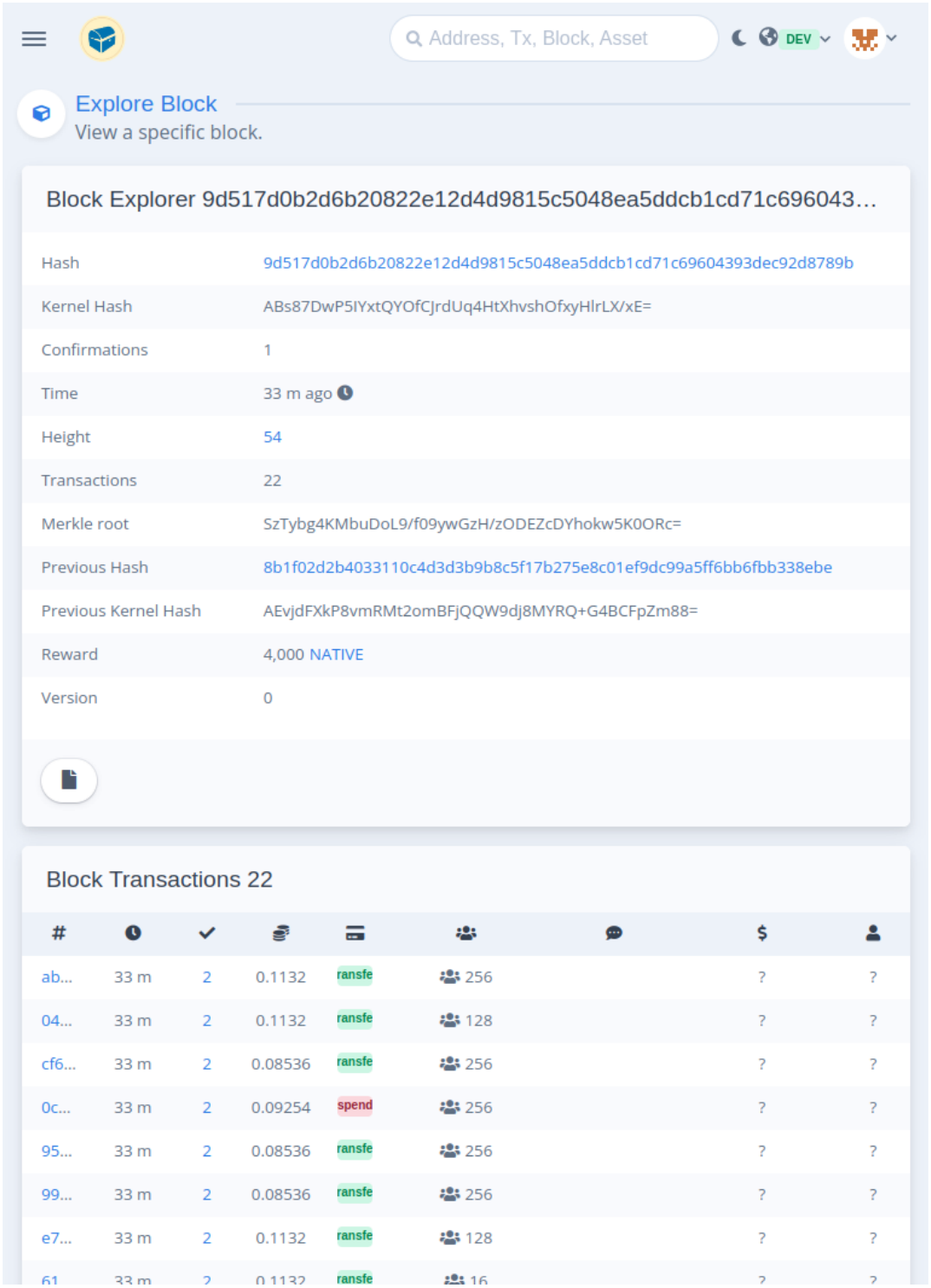


Figure 16 – Block explorer. It shows the block #54.

Transaction Explorer – it allows users to get transactions from full nodes and view different parameters (block height, timestamp, size, etc.).

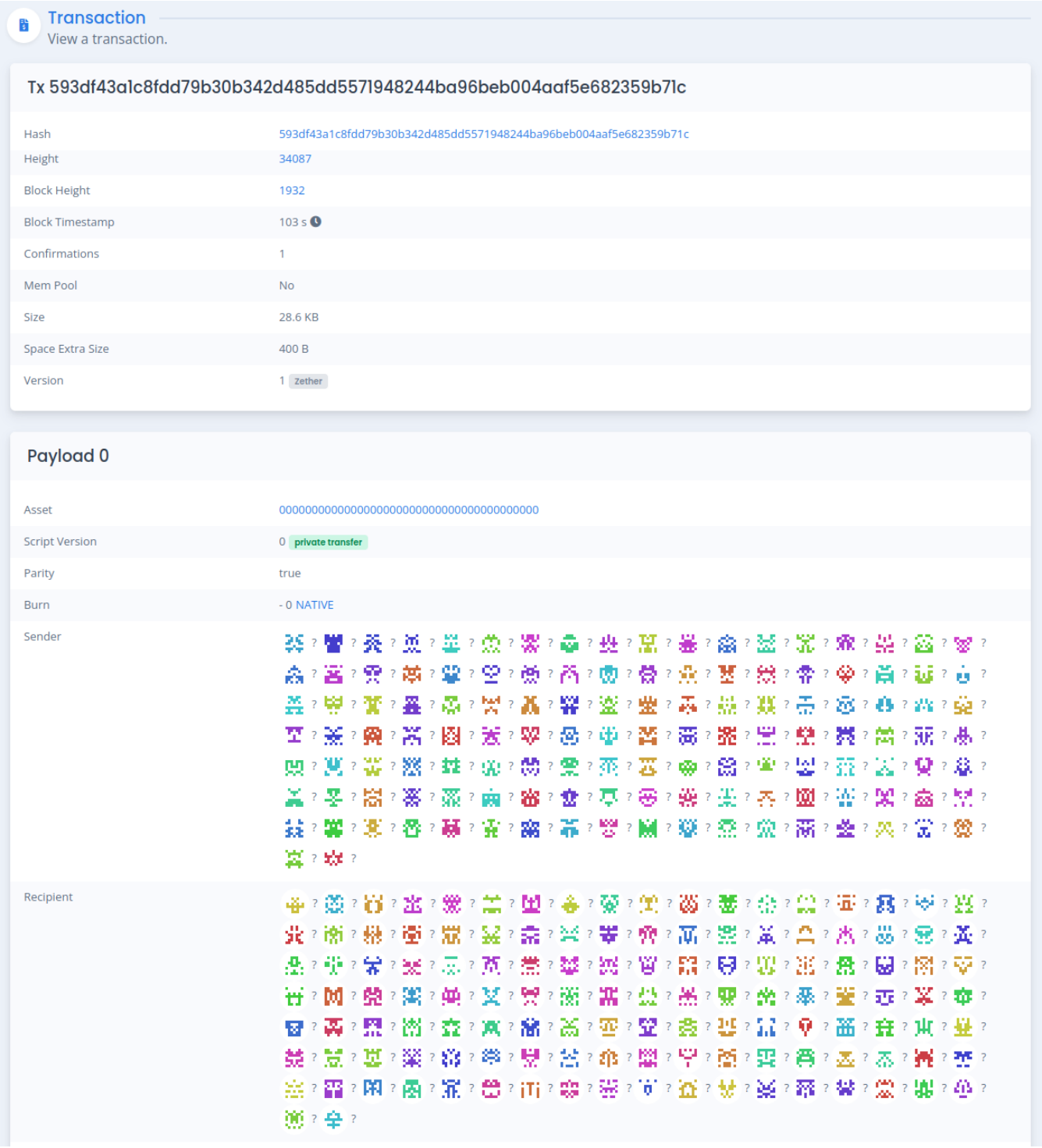


Figure 17 – A 256-ring size Zether Transaction in PandoraPay.

Account Explorer

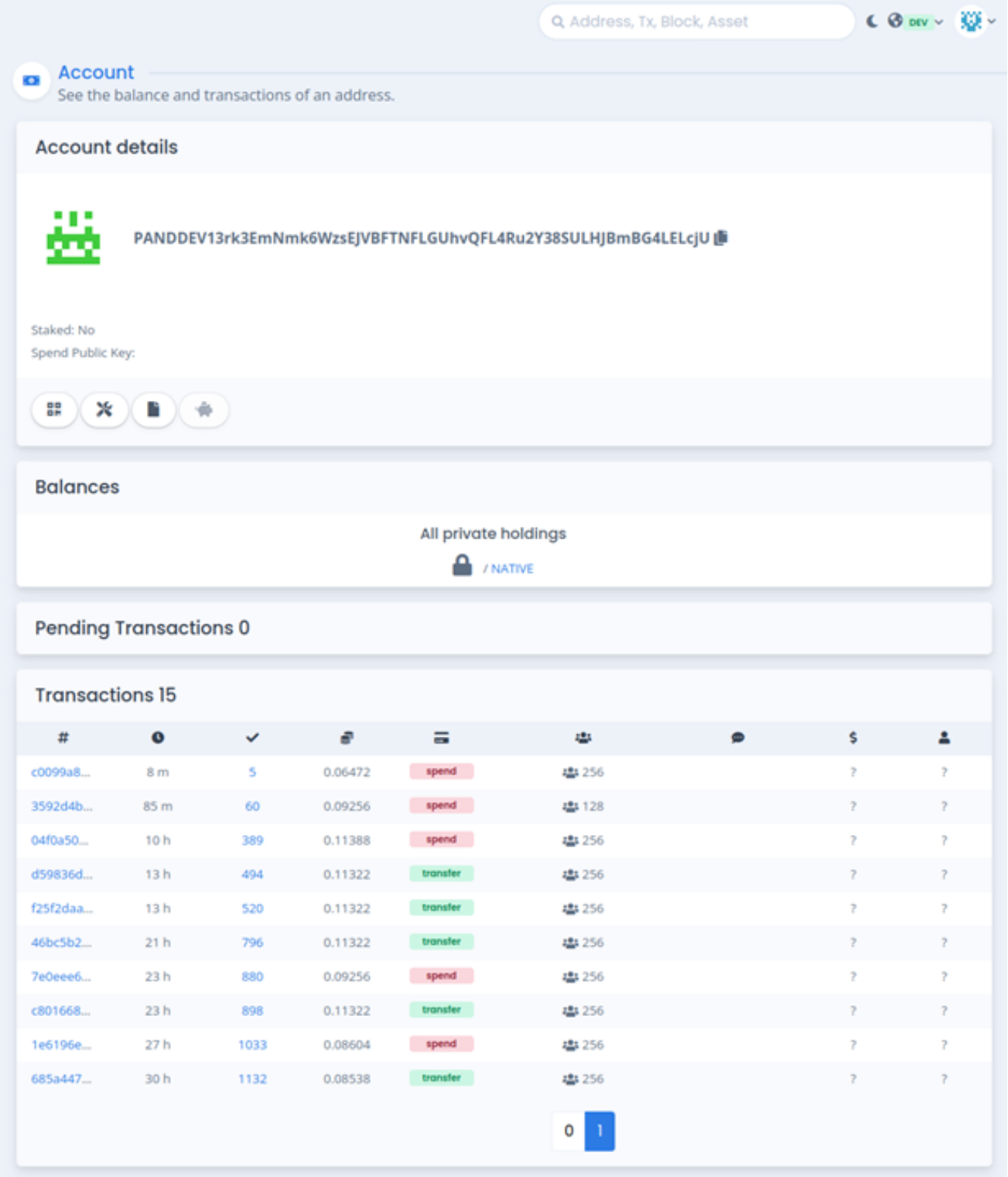


Figure 18 – Account explorer. This account is not owned by the user.

Account Explorer

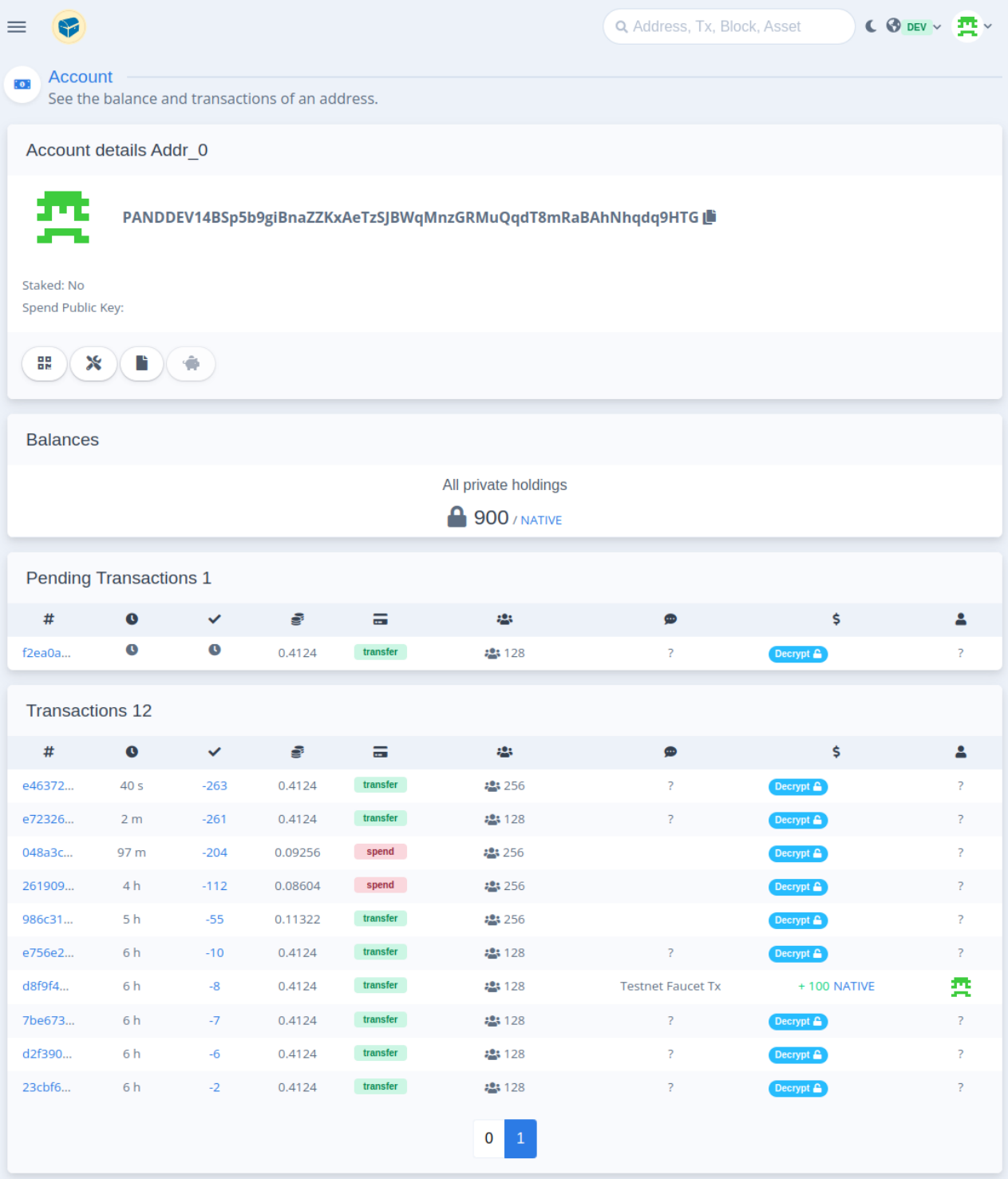


Figure 19 – Account explorer. This account is owned by the user. Balance decryption is possible and can be decrypted. A transaction was decrypted.

Users can create transactions using the CLI from the Full Node or Web Wallet.

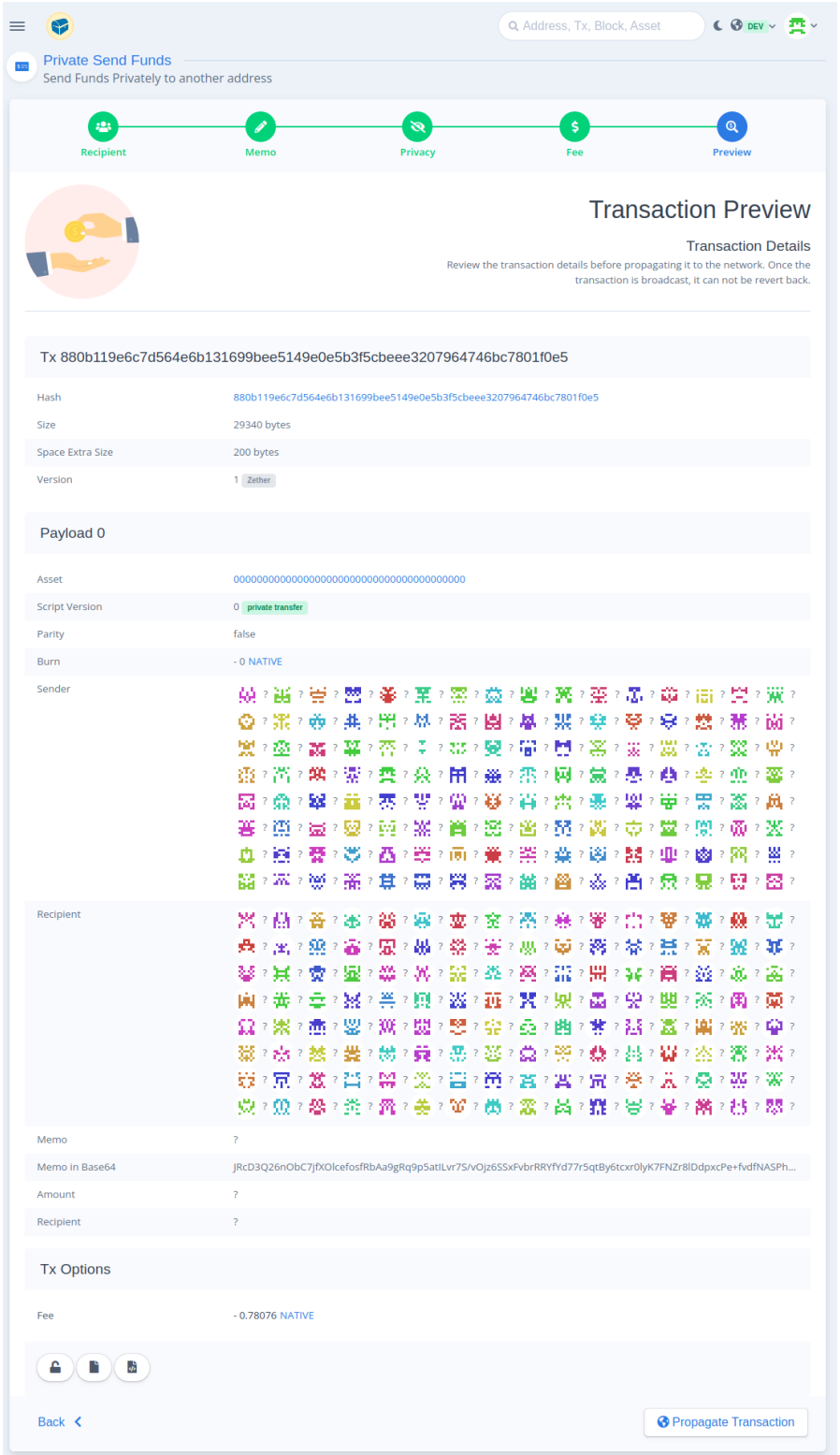


Figure 20 – Creating a transaction using the Web Wallet.

3.4 Wallet

In our front-end app, we have also created a wallet functionality. The wallet supports BIP39 (mnemonic codes) and BIP32 (hierarchical deterministic wallets). The original code for the wallet functionalities was written in Go and exported as a WebAssembly module. Several Vue3 components were created to integrate these wallet functionalities directly in the front-end app.

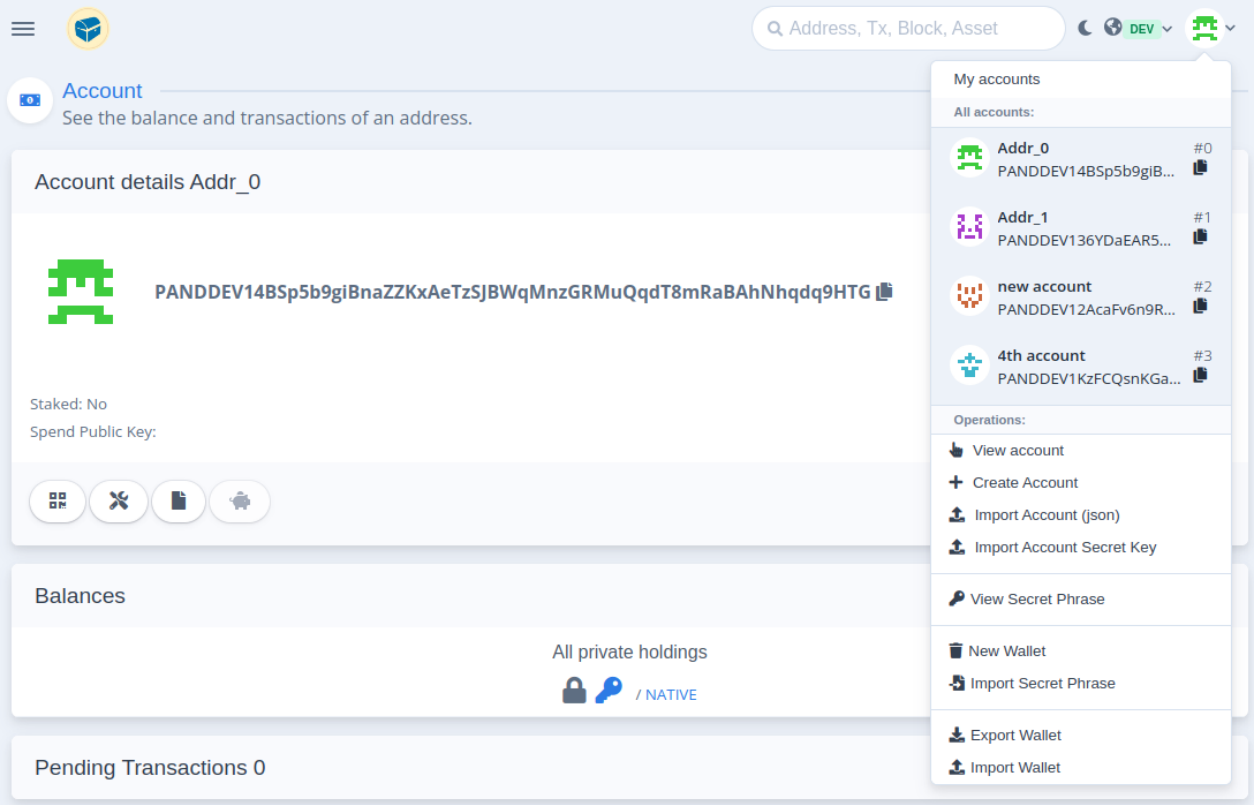


Figure 21 – The wallet menu can be found in the top right corner. Users can switch between their accounts using the menu. BIP39 allows hardening hierarchical addresses that use a sequential index.

The wallet can be imported and exported in JSON file. Addresses can be imported and exported as well in JSON.

Using BIP32 the wallet is generated using a secret phrase (mnemonic code). The secret phrase can be imported and exported as text. The mnemonic helps to create a seed for BIP39 Hierarchical Deterministic Addresses.

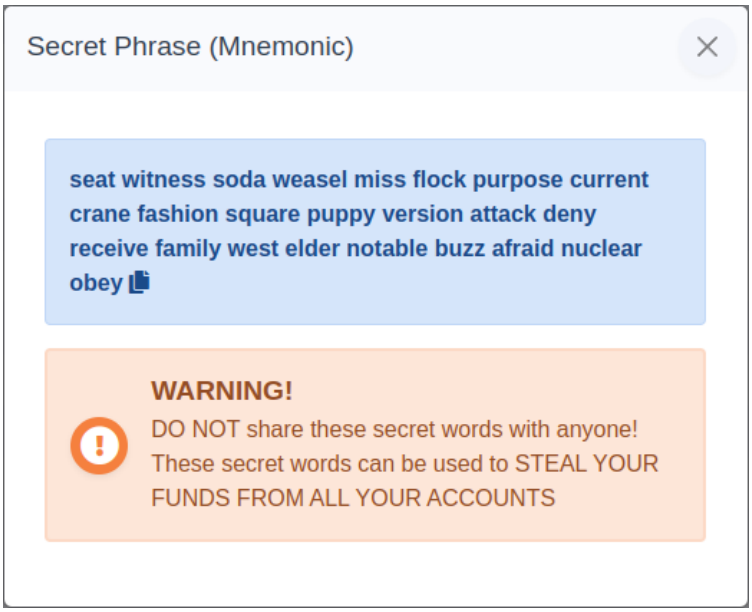


Figure 22 – Showing the Secret Phrase (Mnemonic code) of the wallet.

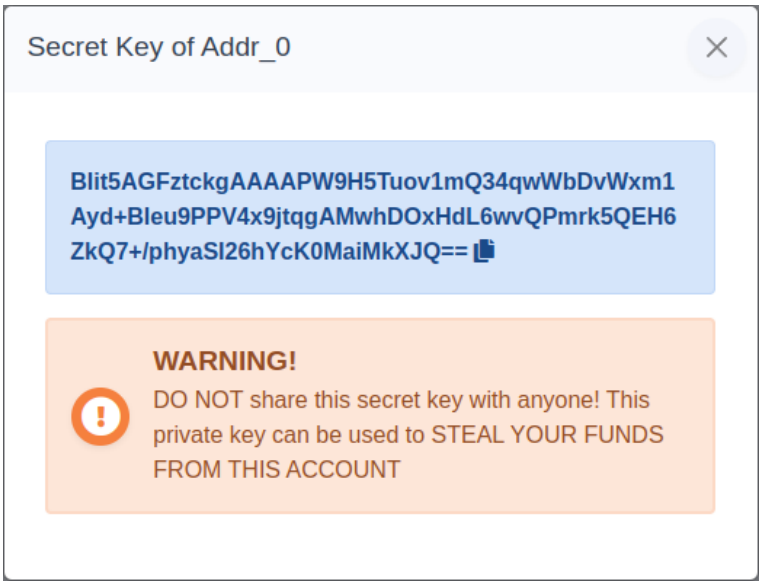


Figure 23 – Showing the Private Key of an Address. The Private key is Encoded using WIF and is shown as base64 text.

The entire wallet can be encrypted using AES-256 cipher and Argon2 (hash function). The encryption is done in the browser. When encrypted the entire wallet, the encrypted state is stored in the IndexDB. If an attacker steals the encrypted wallet (gets access to the computer), he won't be able to use it unless he can crack the password and find the password (using brute force).

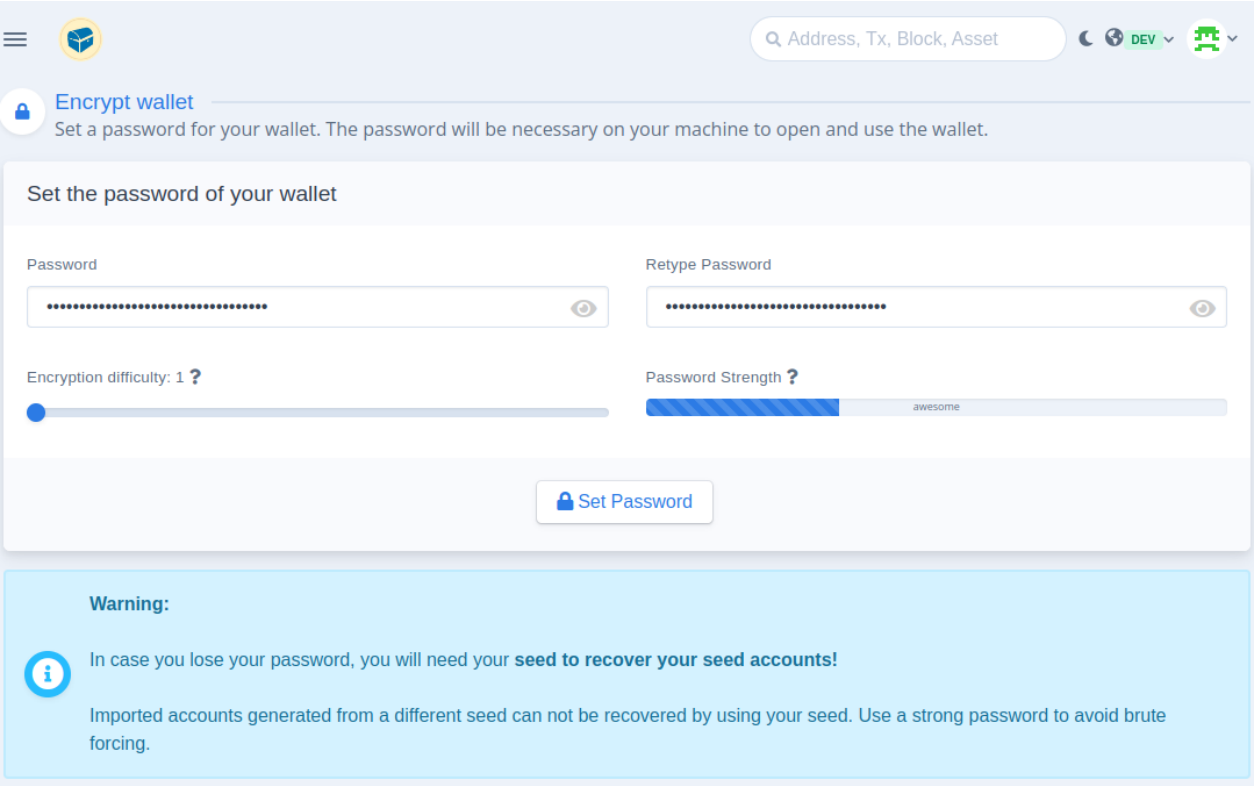


Figure 24 – Encrypting wallet with a given password. The User Interface displays the strength of the password and the encryption difficulty.

The Encryption difficulty represents how many passes the Argon2 hashing algorithm needs to make over the memory. This way we make the hashing more difficult. This parameter is really useful against brute force attacks as attackers are known to have a limited computational power. The longer time the hash function takes to calculate, the more difficulty it will be for an attacker to successfully brute force it.

4. Proposed Future Improvements

We acknowledge that there are multiple features that could be added in the future to improve the overall system developed in this master thesis.

4.1. New P2P networking protocol

4.1.1 End-to-End encryption communication

As it was mentioned in the chapter 2.1.2, the network communication encryption relies on correctness of the Transport Layer Security (TLS). This requires nodes to trust the DNS servers and their Internet Providers to send the right TLS certificates in order to establish a secured connection. This trust can be exploited by state-sponsored actors to run man-in-the-middle attacks on the HTTPS traffic by either compromising the DNS servers or forcing regular users (including data centers) to install specific certificates. We are aware of the fact that some state sponsored actors have carried out these types of attacks on ordinary internet users. One example was where the government of Kazakhstan created "a root-certificate to carry our man-in-the-middle attacks on HTTPS traffic [...] If installed [...] Kazakh government to intercept, decrypt and re-encrypt the internet traffic passing through systems it controlled." [17].

A state-sponsored actor could hijack the entire network by compromising the DNS servers of all the network nodes. This attack vector will become less likely to be exploited over time because different people can join the network from different countries that might have different internet providers and DNS servers. This is possible because PandoraPay is a peer-to-peer permission-less network.

In order to avoid trusting and relying on the DNS servers and Internet Providers, we propose the usage of an Integrated Encryption Scheme to encrypt network

communications. By using an IES, the system provides semantic security against an adversary who is able to craft special plaintexts ciphertexts to run various attacks. The security of the IES is based on the Diffie-Hellman problem. One such scheme is ECIES (Elliptic Curve Integrated Encryption Scheme) – a Public Key Authenticated Encryption Scheme. Other schemes can be used as well, but we propose the ECIES to be a simple scheme to achieve secured communication between two parties using an unsecured channel (the internet). ECIES is a scheme that uses Key Derivation Function (KDF) for deriving a MAC key and a symmetric encryption key for the ECDH shared key. [19] Using ECIES, nodes have a "Contact" information that includes a Public Key. This Public Key is used for establishing a secured peer-to-peer communication over an unsecured channel. All messages are encrypted bidirectionally between two peers. Basically, the sender will encrypt his messages using the receiver's Public Key, while the receiver will encrypt his messages using the sender's Public Key.

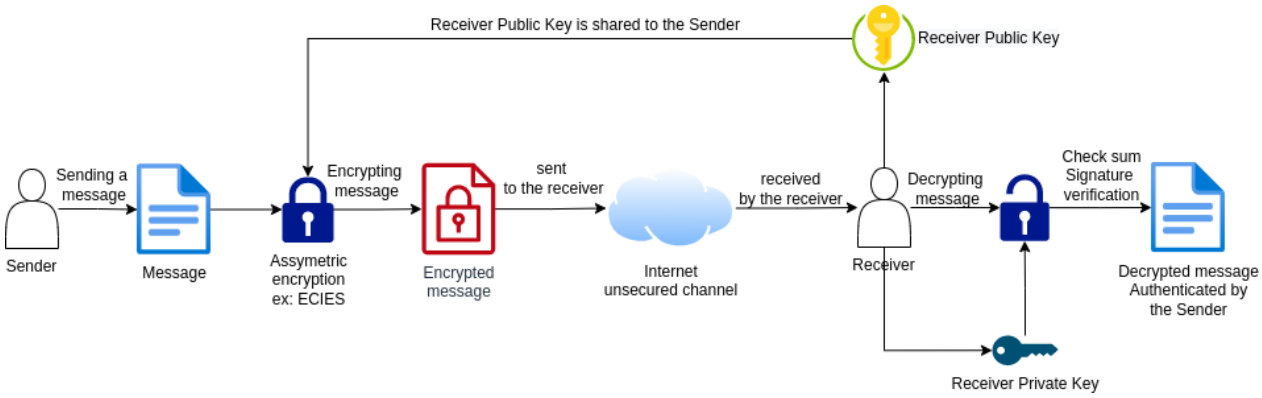


Figure 25 – Diagram of Asymmetric Encryption. The sender is using the public key of the receiver to encrypt the message. The encrypted message is sent to the receiver via the internet (an insecure channel) who can decrypt it using their secret (Private Key).

Another idea is to use Tor and I2P networks to keep network communications protected. These networks do the Asymmetric Encryption under the hood. Instead of using IPs, the nodes will use Onion v3 addresses or I2P addresses. For example, V3 onion addresses contains 56 characters, the full ED25519 Public Key.

4.1.2 Censorship resistant network

The network describe in the previous chapters is a simple peer-to-peer network. A state-sponsored attacker could try to take down the network by overwhelming the nodes with traffic. An attacker can even contain the entire network by blacklisting IPs using Domain Name System Blacklists (Great Firewall of China). To counter this, we propose a simple, yet efficient way to make the network censorship resistant. Certain full nodes could run using Tor hidden services protocol or using the I2P protocol. This way our network nodes could not be banned without taking down the Tor and I2P networks. Considering that both Tor and I2P are decentralized and self-organized networks, it is very unlikely that an entity could take both networks completely offline for a long-time without exploiting a zero-day vulnerability.

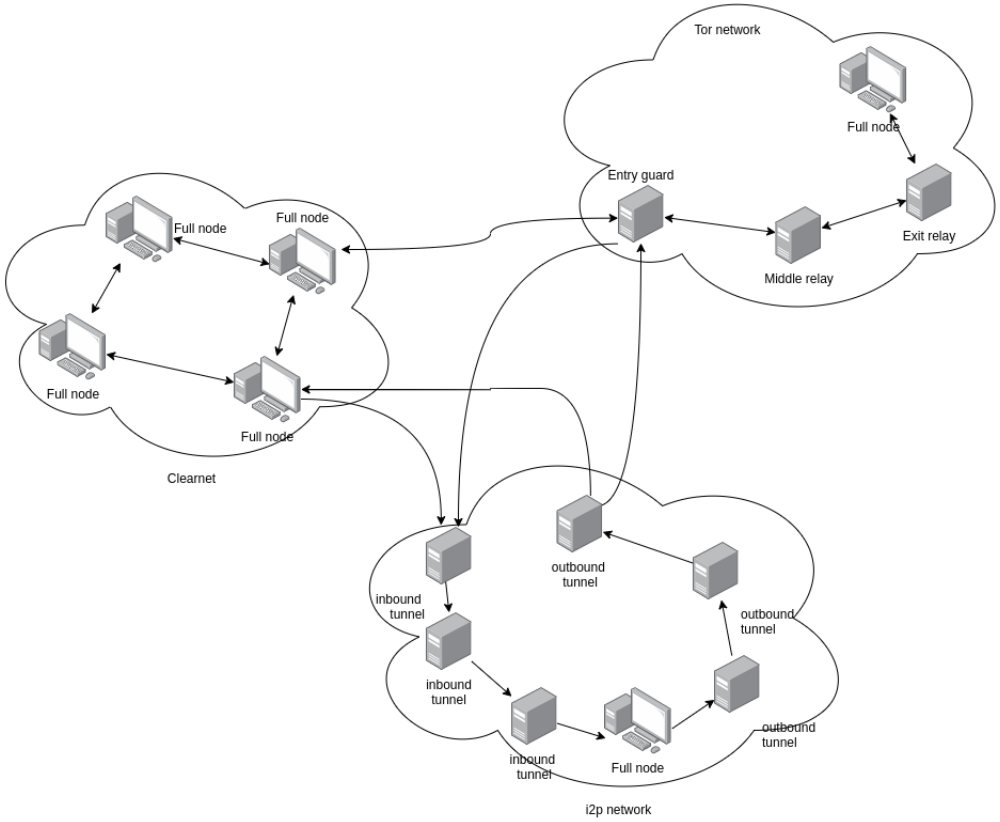


Figure 26 – Proposed censorship resistance network for PandoraPay. Data could be exchanged between Clearnet and Tor/I2P networks.

4.1.3 Integrating Dandelion++ Protocol

We propose the usage of the Dandelion++ protocol to keep users' IP addresses anonymous when they broadcast new transactions. At the moment, the code that's been developed doesn't use Dandelion or any other anonymization scheme. Users just have to "shout" the transaction they just signed broadcasting it. Users need to use a proxy or an IP anonymity protocol in order to keep their IP private when transmitting transactions. The Dandelion protocol is a simple, yet powerful network layer anonymity solution that was originally proposed to improve the Bitcoin Peer-to-peer network privacy. Multiple security risks were discovered with the original Dandelion protocol. Later an improved version called Dandelion++ was proposed. The new Dandelion++ network protocol guarantees anonymity for its users when broadcasting transactions. [18]

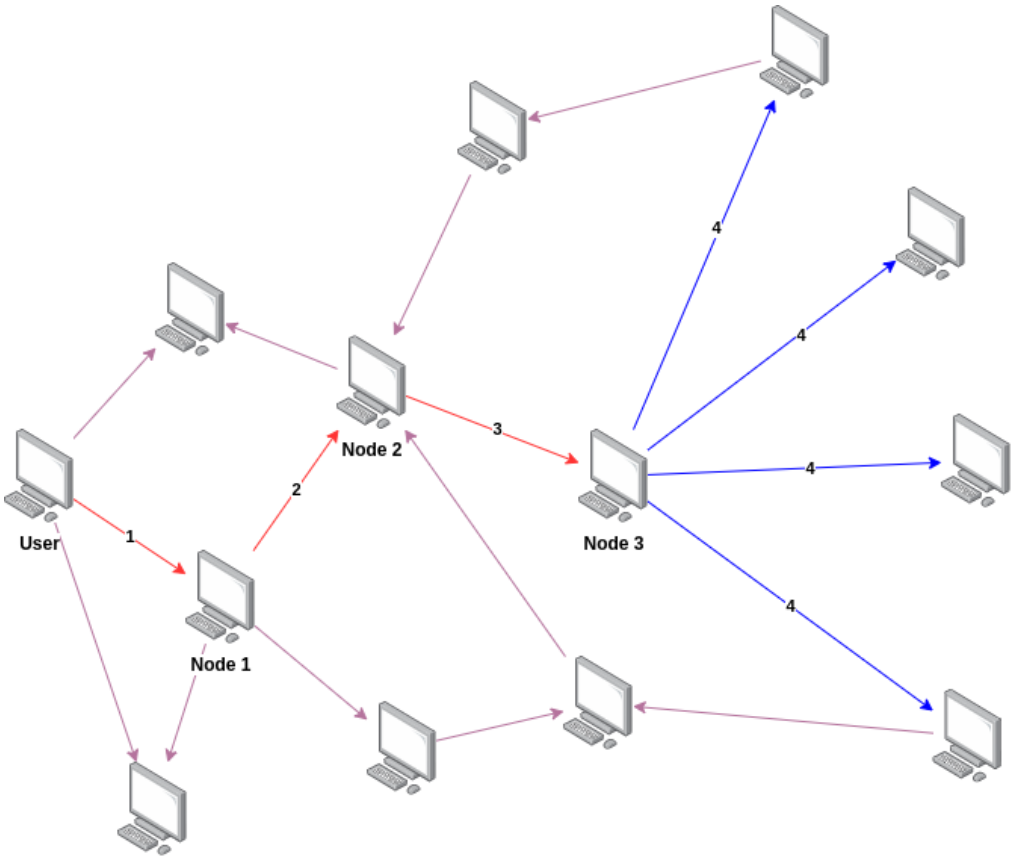


Figure 27 – Transaction propagation using Dandelion protocol

4.2 Horizontally scalable blockchain

In this section, we present a novel solution to achieve horizontal scalability for PandoraPay privacy-preserving blockchain.

The Full Node implementation is multi-threaded, and it can scale with the number of available CPU cores, but the software is still limited to a single machine. Our assumption is that in the future, we may see more processing power available for less money. This would make the software to handle more transactions per second. However, widespread adoption of a PandoraPay blockchain would outpace the hardware scalability offered by CPU manufacturers. We propose a simple, yet scalable solution for this problem.

4.2.1 Identifying the bottlenecks

The first step in solving this problem is understanding the bottlenecks of the software. We need to find the main reasons why our system simply cannot process a large number of transactions per second. We have identified the areas where the current implementation is struggling and listed them based on their importance and relevance. The chapters below provide more detail on each bottleneck.

4.2.1.1 Computation speed bottleneck

There is a limit to the speed of a CPU core. This limit imposes a finite number of instructions that a CPU core can execute per second. For example, one ECDSA signature verification takes about 150µs on Intel Xeon E-2286 CPU at 2.4Ghz with Turbo Boost Disabled. In theory, we could verify about ~7000 transactions per second. In practice we would likely get much worse performance because the transactions need to be deserialized, verified against edge cases and so on. A realistic limit for ECDSA signature will be 1000 transactions per second on a real-world blockchain. Bitcoin implementation in C/C++ can handle up to 7 transactions per second [20]. Adding more CPUs would allow the

machine to process more transactions per second. Unfortunately, most computer motherboards support two to four microprocessors. Vertical scalability is not a solution and could not be one in the near future.

However, the verification time for a Zether 256-ring size transaction is around of 150 milliseconds, which is 1000 times slower than the verification for a single ECDSA signature. In one second, a full node can process only 6.66 transactions. Besides this, the size of a 256-ring size Zether transaction would be 32kb requiring more computation for deserialization and other operations. This is likely the biggest bottleneck for widespread adoption of the proposed blockchain.

4.2.1.2 Storage Speed Bottleneck

The storage device has a limit on how much data it can read and write per second. After transactions have been verified, certain parameters from the ledger must be checked (verified) to see if the transactions share the same view of the ledger's current state. In the event the block is valid, some changes must be stored on the disk, including account tree updates, serialized transactions, and block header. According to the benchmarks in [21], on an SSD we should be able 435,000 disks operations per second for a 16kb data, and 504,000 for a 128-byte data using the libraries we used. However, one Zether 256-ring size transaction requires at least 256 operations (one read and one write) of 60 bytes of data (encrypted ElGamal ciphertext). A single Zether 256-ring transaction would require 512 key-value operations.

Until hitting storage speed bottlenecks, PandoraPay should be able to support at least 1,000 transactions per second on a high-performance NVMe SSD. Log-structured merge Tree (LSMs) in LevelDB, RocksDB and B+ tree in LMDB tend to degrade their performance as the size of the database increases. The Account Tree is unprunable because it stores encrypted ElGamal ciphertexts. Eventually, any storage libraries we might use will slow down a little bit with every read and write operation. It is possible to introduce

an annual tax to automatically release unused accounts, but this might create some account privacy issues.

According to statistics provided by [22], Bitcoin has a total of 900 million unique addresses. Most of these addresses are empty right now. PandoraPay Account Tree is unprunable and it takes about ~70 bytes to store the ElGamal encrypted ciphertext. In PandoraPay were to be adopted like Bitcoin, the blockchain would have to store 63 GB of non-prunable data. With Bitcoin's unprunable UTXO size of 343gb, a scalable homomorphic cryptographic blockchain adopted and used by millions of users might get in terms of terabytes of data.

4.2.1.3 Network bandwidth bottleneck

Eventually we would hit a network bandwidth limit. The regular network interface can only transmit up to 25 Mbps. This means that no single machine can send or receive no more than 25 Mb of data per second. Considering that a Zether 256-ring size transaction requires 32 Kb of data, that means there is a theoretical limit of up to ~750 transactions per second unless we scale up the network bandwidth. We also need to be aware that the PandoraPay sends pings to keep alive Websockets connections consuming network bytes. If data is encrypted using TLS or the ECIES scheme, we might lose some bytes per transferred packet in the network. We also need to consider that the protocol requires the nodes to gossip different information. This is necessary in order for the mempool to function as nodes must broadcast transactions.

Motherboards allow for the installation of multiple network cards. This means that vertical scalability may increase the bandwidth constraints. Although we would be able to achieve higher bandwidth limits, the network would introduce high latencies, making it difficult to reach consensus between nodes.

4.2.2 Horizontal Scaling Cluster

This master's thesis proposes a horizontal scaling technique using a cluster of machines for Blockchain. As mentioned in the previous sub-chapter, we have identified the three bottlenecks that would not allow the proposed blockchain to scale. The identified bottlenecks are: Computational Speed, Storage Speed and Network bandwidth.

The proposed solution is to scale horizontally using a cluster made of λ workers. We use distributed computing to make a cluster of machines that act as a single entity (machine) instead of just one machine doing everything. Horizontal scalability can be achieved in three separate layers. Each layer is independent and adds its own horizontal scalability. If we implement only one layer, the whole system will eventually hit the bottleneck of the other layers. Each layer can be scaled horizontally through cluster computing by using different methods and techniques.

1. CPU scalability – the system can be scaled up by running λ independently machines that are controlled by one manager, who will distribute batches of digital signatures for Zether transactions. Each machine will receive a batch of unverified state-less transactions and will verify them. Each worker will return a list of boolean results if the transactions were valid or not. If an invalid transaction is found in a batch of transactions, the machine will immediately abort the batch returning to the manager an error.
2. Storage scalability – the system can be scaled up by running τ machines that store distributively the blockchain state. We could use existing libraries like Dgraph, TiDB, CockroachDB, etc. The distributed key-value store would need to have ACID properties to avoid situations where the blockchain is in an invalid state due to unexpected hardware or software errors.
3. Network scalability – the system can be scaled up by running λ independently machines to read and write network data concurrently. This would require to

change the way a block is serialized. It would be unadvisable to increase the block size because it would just increase the network latency. Instead, the block should be split into multiple smaller chunks (called micro blocks) of 5MB that can be downloaded distributively by each worker from multiple sources. Once these chunks have been downloaded, the machines can immediately and independently verify the state-less transactions. This technique requires that the transactions stored in these separate chunks should not be affected by state changes caused by other workers.

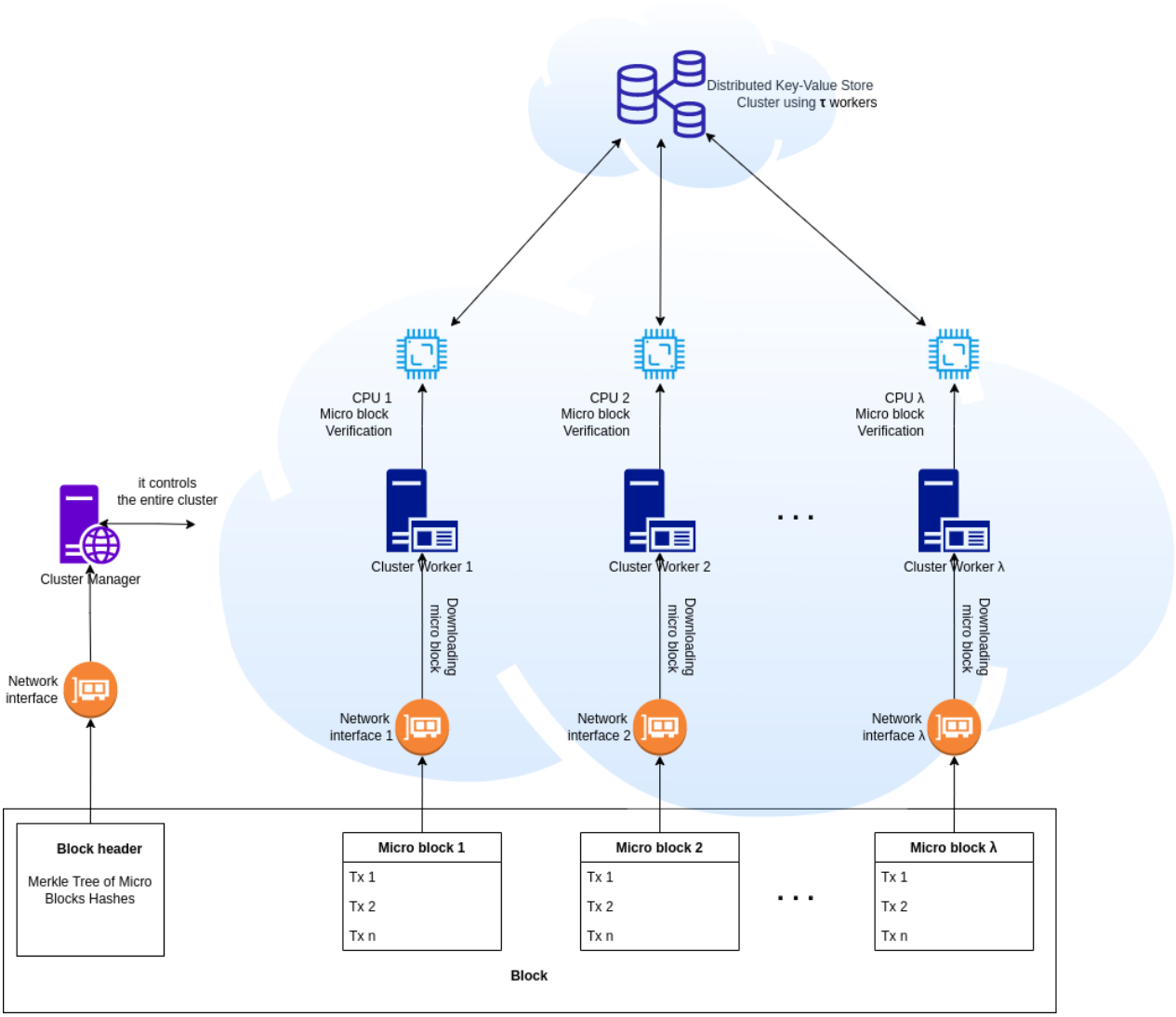


Figure 28 – Proposed Horizontal Scaling Cluster. The cluster is made of λ different machines. The λ parameter should be a global constant in the network.

The distributed Key-Value store could be run either as a separate cluster (as seen in the diagram), or on the same machines. A different cluster is recommended because these key-value stores use multi-threading and are computationally intensive in order to achieve high performance.

4.2.3 Mitigations of new vulnerabilities introduced by the cluster based horizontal scaling solution

There are a number of mitigations that can be put in place in order to minimize the risk of new vulnerabilities being introduced by the horizontal scaling solution. New vector attacks need to be taken into account when developing the scaling module.

4.2.3.1 Rogue Key attacks

An attacker could create a spam attack with specially crafted blocks. In this block all transactions addresses will be stored in the same bucket by the distributed storage library. This is possible because every full node in the network will run the same code and use the same libraries. The attacker will know the distribution of addresses stored in the Log-Structed Merge or B+ Trees. The attacker could even register many addresses that will be stored in the same bucket. These specifically created transactions will make the distributed key-storage store to behave like a single machine. This attack will make only one machine in the cluster to read and write data in the distributed storage rendering it slow. To mitigate this vector attack, the keys must be hashed with a pseudo-random, per-execution seed. By using this seed, the hash tables of the keys in the distribute storage will be fairly even distributed.

4.2.3.2 High Network latency due to Large Blocks Size

The maximum Block and Micro-blocks sizes should be capped at a certain size. Large block sizes will result in high network latency. Nodes can go offline for a few seconds, which means that blocks have to be downloaded from other peers. An attacker could also

try to do a Sybil attack by flooding the network with many fake nodes that offer slow download speeds. This will make the honest nodes to get many timeouts and they will not be able to know which are the honest nodes that offer the fastest download speeds. A solution to mitigate this is to split the blocks into smaller chunks and download them concurrently from multiple peers.

4.2.3 Implementation of a Horizontally Scalable Blockchain

The three-layer solution which was proposed to horizontally scale PandoraPay's blockchain requires that transactions be stateless. In this case, the transactions can be downloaded separately by different workers and can be verified for the validity of their zero-knowledge proofs. In the event that the transactions are valid, the state can be updated in the distributed key-value store. ACID properties for transactions are required in order to avoid special edge cases when the state is not present in a valid state across the cluster.

Go is a great choice for implementing a cluster of distributed machines. A docker image of the software that runs on each of the workers can be created. A cluster manager will have to start the workers and control them via a TCP/IP socket in order to make sure that the workers are synchronized and processing the same block. Each worker will download a micro-block of the current block and validate the transactions inside. If all workers agree that all of the micro-blocks are valid, the manager can ask the workers to start updating the distributed Key-Value store.

A global parameter for the number of workers in a cluster should be used to avoid uneven powerful nodes in the network. The λ parameter sets the number of workers that a full noe should be using. The value of λ can be changed through a hard fork from time to time or can be updated automatically based on the number of transactions that were included in the las last k blocks.

4.3 Decentralized Anonymous Voting

The PandoraPay blockchain could be used to ensure anonymity in voting systems. Many countries including Romania have constitutions that guarantee anonymous voting. For an application that needs complete anonymity and be trust-less, centralized solutions are not reliable enough. Some countries have replaced paper ballots with electronic devices. Users' votes could be potentially deanonymized by timing correlation and man-in-the-middle attacks when the vote is broadcasted to the central system that counts the votes. Voting requires some form of authentication and timing correlation is inevitable for most electronic voting. PandoraPay makes it possible for anonymous voting to be a reality in a free and democratic country.

The PandoraPay blockchain is a ready to be used technology that could facilitate anonymous voting. To guarantee privacy, users must open the wallet on their own devices and submit their address to the Electoral Registration Office (ERO). Before the voting begins, the ERO will airdrop exactly 1 coin (one unit) to every single citizen who registered. When voting begins, the ERO will accept transactions in the network. Now, users will be able to open the wallet on their devices and send 1 coin to the address of their choice.

The anonymity set for the user is 128, while the anonymity set for the receiver (candidate) is 128 for a Zether 256-ring size transaction in PandoraPay.

Decentralization of the network means that multiple entities can participate in the network and validate every single vote. Validity of the blockchain voting is guaranteed by the immutability of its properties.

The network is permission-less, meaning that regular citizens can run their own full nodes to validate the state of the ledger. This way, voting is ensured and validated by any interested citizen.

4.4 Decentralized On-Chain Messaging

We propose the application of con-chain end-to-end encrypted chat. The blockchain could be enhanced with special Zether scripts that allow storing on-chain meta-data for a decentralized chat application. Users would encrypt messages using ECIES and store them on the blockchain. Other users could scan the meta-data from these transactions and try to see if they can decrypt the messages using their secret key. This way two users could use the blockchain to communicate with each other securely and privately. The chat features could be integrated in the web wallet, too.

4.5 Decentralized On-Chain Bazaar

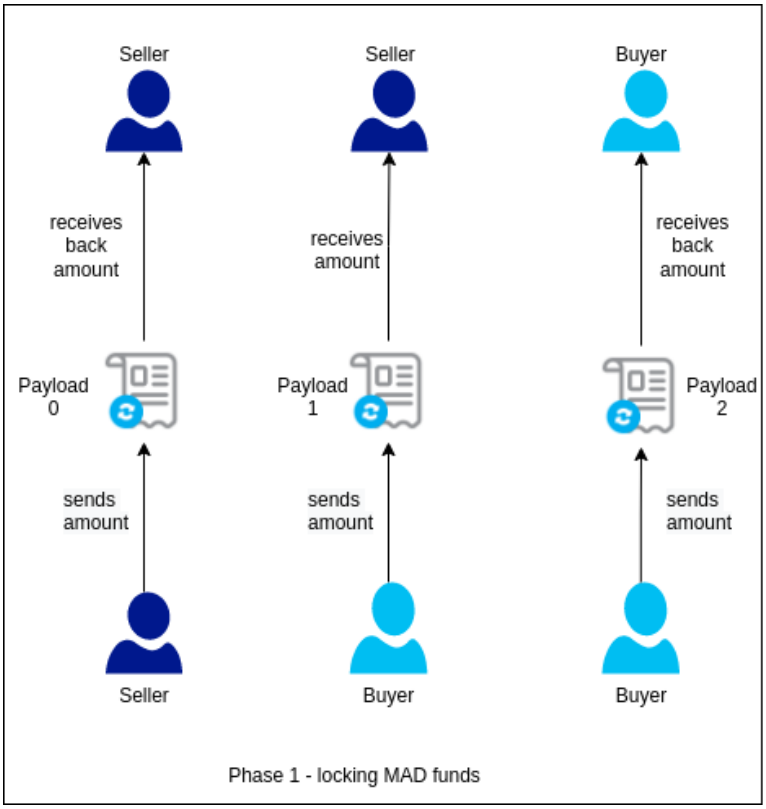
Another possible application of the blockchain technology is the creation of an on-chain decentralized peer-to-peer bazaar (marketplace). A user who wants to sell products could publish his products information on the blockchain using a Zether transaction. The items could be encoded and stored in the transaction's extra information. Other users can access the products that have been stored in the blockchain. A user who wants to buy a product will need to chat with the seller. Once the buyer and seller agree to the terms of a sale, the buyer can initiate a payment for the agreed-upon amount of coins.

The problem is that the seller might not deliver the product as promised, while the buyer has already transferred his coins. There must be a way to compel the seller to send the product and the buyer to confirm the payment. If the buyer doesn't unlock the payment, or the seller doesn't send the product, a dispute arises. It can be resolved using MAD (Mutual Assured Destruction) in which both have financial interests to solve the dispute. MAD forces both parties to act in their interests in order to resolve the dispute, or they will lose money. In this case, the seller will create a Zether transaction that locks an amount equal to the price of the product, while the buy will lock twice the amount he has to pay. The transaction will have three payloads. The seller will create a payload to

transfer the amount from his own address to another address owned by him. The buyer will create a payload transferring the amount to the seller address and another payload transferring to other addresses owned by him. The transaction should be completed only after both parties have created the three payloads.

If the transaction is included in the blockchain, the receiver subring (made of only positive encrypted ciphertexts) will not be credited. No receiver will be credited, rather their credits will be placed into an indefinite pending state.

When the dispute is resolved, both parties will sign a message requiring the previous ZetherTx to be unlocked. The message will be signed using Schnorr multisig. This message can be included in a special transaction that could even be free to be include in the blockchain. If either party fails to live up to their end of the bargain, both will lose out financially.



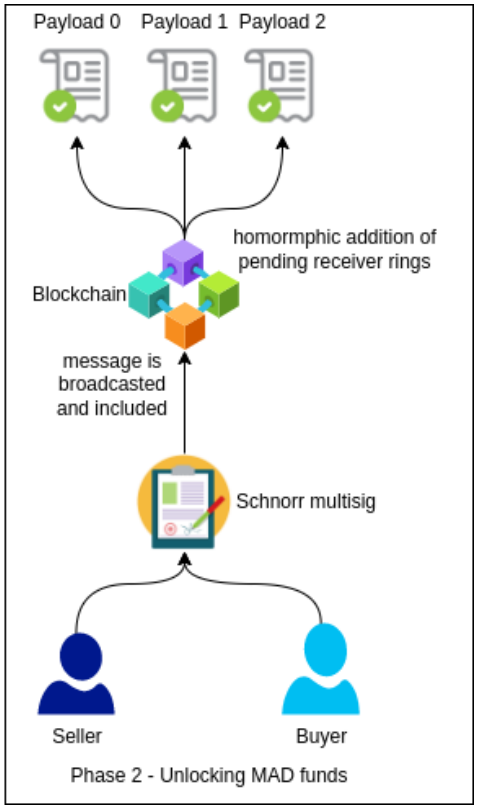


Figure 29. Diagram that shows the workflow of the MAD (Mutual Assured Destruction) contract. Both users (buyer and seller) will lock a certain number of coins to ensure that their interests are aligned in resolving any kind of disputes.

4.6 Smart Contracts

The ability to create and execute smart contracts natively on the blockchain with privacy-preserving transactions could provide significant benefits for both the blockchain and its users. We need first to import a Turing Complete Virtual Machine library in the codebase. The Virtual Machine should have a security audit and some adoption by other projects. Such libraries are Tengo, go-lua, goja, gypthon, etc. The smart contracts should get compiled to bytecode and the Virtual Machine should be able to execute it instruction by instruction.

Users could store the bytecode of a smart contract using in a transaction extra. Once the transaction is included in the blockchain, its bytecode will be stored in the distributed Key-Value storage. Other users should be able to call global functions the smart contract's bytecode with various arguments.

To prevent users from abusing smart contracts by creating infinite loops or very computationally expensive programs, the virtual machine should track the number of execute instructions. Every line of code executed should decrease a counter called Gas. When the counter reaches zero, the program should immediately terminate with an error. When paying the transaction fee, users will have to pay an extra fee for the Gas they want to use. The more gas used, the more instructions the virtual machine can execute.

Conclusions

We believe that a privacy-preserving blockchain that is horizontally scalable can get a world-wide adoption by millions of daily users. The Anonymous Zether cryptographic protocol solves many issues that all other UTXO based privacy models have, including the removal of the blockchain scanning for calculating the user's balance, completely prunable transactions history and state-less transactions to be verifiable using a horizontally scalable infrastructure.

This thesis shows a rigorous documentation of most of the current privacy-preserving protocols that can run on trustless and permission-less networks. A blockchain research prototype for the PandoraPay reference code was successfully implemented and tested on GitHub.

A new branch in the git repository was created to adapt the source code to use the ED25519 elliptic curve. This allowed to run a test-net for the WEBD2.0 network with public transactions only.

Using the Statistic tool from Intelij we have counted the number of lines of code written in this research project:

- 23,850 lines of code in Go
- 3,211 lines of code in JS
- 7,822 lines of code in Vue

The developed source code could be enhanced with many different applications and features, like atomic swaps, smart contracts, decentralized anonymous voting, a decentralized on-chain bazaar, censorship resistance networks, and decentralized finance (DeFi).

Annexes

Annex 1 – Appending blocks into the blockchain

```
func (chain *Blockchain) AddBlocks(blocksComplete []*block_complete.BlockComplete, calledByForging bool,
exceptSocketUUID advanced_connection_types.UUID) (kernelHash []byte, err error) {

  if err = chain.validateBlocks(blocksComplete); err != nil {
    return
  }

  //avoid processing the same function twice
  chain.mutex.Lock()
  defer chain.mutex.Unlock()

  chainData := chain.GetChainData()

  if calledByForging && blocksComplete[len(blocksComplete)-1].Height == chainData.Height-1 &&
chainData.ConsecutiveSelfForged > 0 {
    err = errors.New("Block was already forged by a different thread")
    return
  }

  gui.GUI.Info("Including blocks " + strconv.FormatUint(chainData.Height, 10) + " ... " +
strconv.FormatUint(chainData.Height+uint64(len(blocksComplete)), 10))

  //chain.RLock() is not required because it is guaranteed that no other thread is writing now in the chain
  var newChainData = &BlockchainData{
    helpers.CloneBytes(chainData.Hash),          //atomic copy
    helpers.CloneBytes(chainData.PrevHash),      //atomic copy
    helpers.CloneBytes(chainData.KernelHash),    //atomic copy
    helpers.CloneBytes(chainData.PrevKernelHash), //atomic copy
    chainData.Height,                    //atomic copy
    chainData.Timestamp,                 //atomic copy
    new(big.Int).Set(chainData.Target),        //atomic copy
    new(big.Int).Set(chainData.BigTotalDifficulty), //atomic copy
    chainData.TransactionsCount,             //atomic copy
    chainData.AccountsCount,                 //atomic copy
    chainData.AssetsCount,                   //atomic copy
    chainData.Supply,
    chainData.ConsecutiveSelfForged, //atomic copy
  }

  allTransactionsChanges := []*blockchain_types.BlockchainTransactionUpdate{}

  insertedBlocks := []*block_complete.BlockComplete{}

  //remove blocks which are different
```

```
  removedTxHashes := make(map[string][]byte)
  insertedTxs := make(map[string]*transaction.Transaction)

  var removedTxsList [][]byte               //ordered list
  var insertedTxsList []*transaction.Transaction //ordered list

  removedBlocksHeights := []uint64{}
  removedBlocksTransactionsCount := uint64(0)

  var dataStorage *data_storage.DataStorage

  err = func() (err error) {

    chain.mempool.SuspendProcessingCn <- struct{}{}

    err = store.StoreBlockchain.DB.Update(func(writer store_db_interface.StoreDBTransactionInterface) (err error)
{

      defer func() {
        if errReturned := recover(); errReturned != nil {
          err = errReturned.(error)
        }
      }()

      savedBlock := false

      dataStorage = data_storage.NewDataStorage(writer)

      //let's filter existing blocks
      for i := len(blocksComplete) - 1; i >= 0; i-- {

        blkComplete := blocksComplete[i]

        if blkComplete.Block.Height < newChainData.Height {
          var hash []byte
          if hash, err = chain.LoadBlockHash(writer, blkComplete.Block.Height); err != nil {
            return
          }
          if bytes.Equal(hash, blkComplete.Block.Bloom.Hash) {
            blocksComplete = blocksComplete[i+1:]
            break
          }
        }

      }

      if len(blocksComplete) == 0 {
        return errors.New("blocks are identical now")
      }

      firstBlockComplete := blocksComplete[0]
      if firstBlockComplete.Block.Height < newChainData.Height {
```

```
        index := newChainData.Height - 1
        for {

          removedBlocksHeights = append(removedBlocksHeights, 0)
          copy(removedBlocksHeights[1:], removedBlocksHeights)
          removedBlocksHeights[0] = index

          if allTransactionsChanges, err = chain.removeBlockComplete(writer, index, removedTxHashes,
allTransactionsChanges, dataStorage); err != nil {
            return
          }

          if index > firstBlockComplete.Block.Height {
            index -= 1
          } else {
            break
          }
        }

        if firstBlockComplete.Block.Height == 0 {
          gui.GUI.Info("chain.createGenesisBlockchainData called")
          newChainData = chain.createGenesisBlockchainData()
          removedBlocksTransactionsCount = 0
        } else {
          removedBlocksTransactionsCount = newChainData.TransactionsCount
          newChainData = &BlockchainData{}
          if err = newChainData.loadBlockchainInfo(writer, firstBlockComplete.Block.Height); err != nil {
            return
          }
        }

        if err = dataStorage.CommitChanges(); err != nil {
          return
        }

      }

      if blocksComplete[0].Block.Height != newChainData.Height {
        return errors.New("First block hash is not matching")
      }

      if !bytes.Equal(firstBlockComplete.Block.PrevHash, newChainData.Hash) {
        return fmt.Errorf("First block hash is not matching chain hash %d %s %s ", firstBlockComplete.Block.Height,
base64.StdEncoding.EncodeToString(firstBlockComplete.Bloom.Hash),
base64.StdEncoding.EncodeToString(newChainData.Hash))
      }

      if !bytes.Equal(firstBlockComplete.Block.PrevKernelHash, newChainData.KernelHash) {
        return errors.New("First block kernel hash is not matching chain prev kerneh lash")
      }
```

```
        err = func() (err error) {

          for _, blkComplete := range blocksComplete {

            //check block height
            if blkComplete.Block.Height != newChainData.Height {
              return errors.New("Block Height is not right!")
            }

            //check existance of a tx with payloads
            var foundStakingRewardTx *transaction.Transaction
            for index, tx := range blkComplete.Txs {
              if tx.Version == transaction_type.TX_ZETHER {
                txBase := tx.TransactionBaseInterface.(*transaction_zether.TransactionZether)
                if len(txBase.Payloads) == 2 && txBase.Payloads[0].PayloadScript ==
transaction_zether_payload_script.SCRIPT_STAKING && txBase.Payloads[1].PayloadScript ==
transaction_zether_payload_script.SCRIPT_STAKING_REWARD {
                  if foundStakingRewardTx != nil {
                    return errors.New("Multiple txs with staking & reward payloads")
                  }
                  foundStakingRewardTx = tx
                  if index != len(blkComplete.Txs)-1 {
                    return errors.New("Staking reward tx should be the last one")
                  }
                  continue
                }
                for _, payload := range txBase.Payloads {
                  if payload.PayloadScript == transaction_zether_payload_script.SCRIPT_STAKING ||
payload.PayloadScript == transaction_zether_payload_script.SCRIPT_STAKING_REWARD {
                    return errors.New("Block contains other staking/reward payloads")
                  }
                }
              }
            }

            // not staking and reward tx
            if foundStakingRewardTx == nil {
              return errors.New("Block is missing Staking and Reward Transaction")
            }

            //check blkComplete balance
            foundStakingRewardTxBase :=
foundStakingRewardTx.TransactionBaseInterface.(*transaction_zether.TransactionZether)
            if foundStakingRewardTxBase.Payloads[0].BurnValue <
config_stake.GetRequiredStake(blkComplete.Block.Height) {
              return errors.New("Staked amount is not enough!")
            }

            //verify staking amount
            if foundStakingRewardTxBase.Payloads[0].BurnValue != blkComplete.StakingAmount {
              return errors.New("Staked amount is different that the burn value")
            }
```

```
        if !bytes.Equal(foundStakingRewardTxBase.Payloads[0].Proof.Nonce(), blkComplete.StakingNonce) {
          return errors.New("Staked Proof Nonce is not matching with the one specified in the block")
        }

        //verify forger reward
        var reward, finalForgerReward uint64
        if reward, finalForgerReward, err = blockchain_types.ComputeBlockReward(blkComplete.Height,
blkComplete.Txs); err != nil {
          return
        }

        if
foundStakingRewardTxBase.Payloads[1].Extra.(*transaction_zether_payload_extra.TransactionZetherPayloadExtra
StakingReward).Reward > finalForgerReward {
          return fmt.Errorf("Payload Reward %d is bigger than it should be %d",
foundStakingRewardTxBase.Payloads[1].Extra.(*transaction_zether_payload_extra.TransactionZetherPayloadExtra
StakingReward).Reward, finalForgerReward)
        }

        //increase supply
        var ast *asset.Asset
        if ast, err = dataStorage.Asts.GetAsset(config_coins.NATIVE_ASSET_FULL); err != nil {
          return
        }

        if err = ast.AddNativeSupply(true, reward); err != nil {
          return
        }
        if err = dataStorage.Asts.Update(string(config_coins.NATIVE_ASSET_FULL), ast); err != nil {
          return
        }

        newChainData.Supply = ast.Supply

        if difficulty.CheckKernelHashBig(blkComplete.Block.Bloom.KernelHashStaked, newChainData.Target) !=
true {
          return errors.New("KernelHash Difficulty is not met")
        }

        if !bytes.Equal(blkComplete.Block.PrevHash, newChainData.Hash) {
          return errors.New("PrevHash doesn't match Genesis prevHash")
        }

        if !bytes.Equal(blkComplete.Block.PrevKernelHash, newChainData.KernelHash) {
          return errors.New("PrevHash doesn't match Genesis prevKernelHash")
        }

        if blkComplete.Block.Timestamp < newChainData.Timestamp {
          return errors.New("Timestamp has to be greater than the last timestmap")
        }
```

```
        if blkComplete.Block.Timestamp >
uint64(time.Now().UTC().Unix())+config.NETWORK_TIMESTAMP_DRIFT_MAX {
          return errors.New("Timestamp is too much into the future")
        }

        if err = blkComplete.IncludeBlockComplete(dataStorage); err != nil {
          return fmt.Errorf("Error including block %d into Blockchain: %s", blkComplete.Height, err.Error())
        }

        if err = dataStorage.ProcessPendingStakes(blkComplete.Height); err != nil {
          return errors.New("Error Processing Pending Stakes: " + err.Error())
        }

        //to detect if the savedBlock was done correctly
        savedBlock = false

        if allTransactionsChanges, err = chain.saveBlockComplete(writer, blkComplete,
newChainData.TransactionsCount, removedTxHashes, allTransactionsChanges, dataStorage); err != nil {
          return errors.New("Error saving block complete: " + err.Error())
        }

        if len(removedBlocksHeights) > 0 {
          removedBlocksHeights = removedBlocksHeights[1:]
        }

        newChainData.PrevHash = newChainData.Hash
        newChainData.Hash = blkComplete.Block.Bloom.Hash
        newChainData.PrevKernelHash = newChainData.KernelHash
        newChainData.KernelHash = blkComplete.Block.Bloom.KernelHash
        newChainData.Timestamp = blkComplete.Block.Timestamp

        difficultyBigInt := difficulty.ConvertTargetToDifficulty(newChainData.Target)
        newChainData.BigTotalDifficulty = new(big.Int).Add(newChainData.BigTotalDifficulty, difficultyBigInt)

        if newChainData.Target, err = newChainData.computeNextTargetBig(writer); err != nil {
          return
        }

        newChainData.Height += 1
        newChainData.TransactionsCount += uint64(len(blkComplete.Txs))
        insertedBlocks = append(insertedBlocks, blkComplete)

        newChainData.saveTotalDifficultyExtra(writer)

        newChainData.saveBlockchainHeight(writer)
        if err = newChainData.saveBlockchainInfo(writer); err != nil {
          return
        }

        savedBlock = true
      }
```

```
        return
      }()

      //recover, but in case the chain was correctly saved and the mewChainDifficulty is higher than
      //we should store it
      if savedBlock && chainData.BigTotalDifficulty.Cmp(newChainData.BigTotalDifficulty) < 0 {

        //let's recompute removedTxHashes
        removedTxHashes = make(map[string][]byte)
        for _, change := range allTransactionsChanges {
          if !change.Inserted {
            removedTxHashes[change.TxHashStr] = change.TxHash
          }
        }
        for _, change := range allTransactionsChanges {
          if change.Inserted {
            insertedTxs[change.TxHashStr] = change.Tx
            delete(removedTxHashes, change.TxHashStr)
          }
        }

        if calledByForging {
          newChainData.ConsecutiveSelfForged += 1
        } else {
          newChainData.ConsecutiveSelfForged = 0
        }

        if err = newChainData.saveBlockchain(writer); err != nil {
          panic("Error saving Blockchain " + err.Error())
        }

        if len(removedBlocksHeights) > 0 {

          //remove unused blocks
          for _, removedBlock := range removedBlocksHeights {
            if err = chain.deleteUnusedBlocksComplete(writer, removedBlock, dataStorage); err != nil {
              panic(err)
            }
          }

          //removing unused transactions
          if config.SEED_WALLET_NODES_INFO {
            removeUnusedTransactions(writer, newChainData.TransactionsCount,
removedBlocksTransactionsCount)
          }
        }

        //let's keep the order as well
        var removedCount, insertedCount int
        for _, change := range allTransactionsChanges {
          if !change.Inserted && removedTxHashes[change.TxHashStr] != nil && insertedTxs[change.TxHashStr] ==
nil {
```

```
            removedCount += 1
          }
          if change.Inserted && insertedTxs[change.TxHashStr] != nil && removedTxHashes[change.TxHashStr] == nil
{
            insertedCount += 1
          }
        }
        removedTxsList = make([][]byte, removedCount)
        insertedTxsList = make([]*transaction.Transaction, insertedCount)
        removedCount, insertedCount = 0, 0

        for _, change := range allTransactionsChanges {
          if !change.Inserted && removedTxHashes[change.TxHashStr] != nil && insertedTxs[change.TxHashStr] ==
nil {
            removedTxsList[removedCount] = writer.Get("tx:" + change.TxHashStr) //required because the garbage
collector sometimes it deletes the underlying buffers
            writer.Delete("tx:" + change.TxHashStr)
            writer.Delete("txHash:" + change.TxHashStr)
            writer.Delete("txBlock:" + change.TxHashStr)
            removedCount += 1
          }
          if change.Inserted && insertedTxs[change.TxHashStr] != nil && removedTxHashes[change.TxHashStr] == nil
{
            insertedTxsList[insertedCount] = change.Tx
            insertedCount += 1
          }
        }

        if config.SEED_WALLET_NODES_INFO {
          removeTxsInfo(writer, removedTxHashes)
        }

        if err = chain.saveBlockchainHashmaps(dataStorage); err != nil {
          panic(err)
        }

        newChainData.AssetsCount = dataStorage.Asts.Count
        newChainData.AccountsCount = dataStorage.Regs.Count + dataStorage.PlainAccs.Count

      } else if err == nil { //only rollback
        err = errors.New("Rollback")
      }

      if dataStorage != nil {
        dataStorage.SetTx(nil)
      }

      return
    })

    return
  }()
```

```
  if err == nil && len(insertedBlocks) == 0 {
    err = errors.New("No blocks were inserted")
  }

  if err == nil {
    kernelHash = newChainData.KernelHash
    chain.ChainData.Store(newChainData)
    chain.mempool.ContinueProcessingCn <- mempool.CONTINUE_PROCESSING_NO_ERROR
  } else {
    chain.mempool.ContinueProcessingCn <- mempool.CONTINUE_PROCESSING_ERROR
  }

  update := &BlockchainUpdate{
    err:           err,
    calledByForging:  calledByForging,
    exceptSocketUUID: exceptSocketUUID,
  }

  if err == nil {
    update.newChainData = newChainData
    update.dataStorage = dataStorage
    update.removedTxsList = removedTxsList
    update.removedTxHashes = removedTxHashes
    update.insertedTxs = insertedTxs
    update.insertedTxsList = insertedTxsList
    update.insertedBlocks = insertedBlocks
    update.allTransactionsChanges = allTransactionsChanges
  }

  chain.updatesQueue.updatesCn <- update

  return
}
```

Annex 2 – Forging Thread

```
func (thread *ForgingThread) stopForging() {
  thread.workersDestroyedCn <- struct{}{}
  for i := 0; i < len(thread.workers); i++ {
    close(thread.workers[i].workCn)
  }
}

func (thread *ForgingThread) startForging() {

  thread.workers = make([]*ForgingWorkerThread, thread.threads)

  forgingWorkerSolutionCn := make(chan *ForgingSolution)
  for i := 0; i < len(thread.workers); i++ {
    thread.workers[i] = createForgingWorkerThread(i, forgingWorkerSolutionCn, thread.addressBalanceDecryptor)
```

```
    recovery.SafeGo(thread.workers[i].forge)
  }
  thread.workersCreatedCn <- thread.workers

  recovery.SafeGo(func() {
    for {

      s := ""
      for i := 0; i < thread.threads; i++ {
        hashesPerSecond := atomic.SwapUint32(&thread.workers[i].hashes, 0)
        s += strconv.FormatUint(uint64(hashesPerSecond), 10) + " "
      }
      gui.GUI.InfoUpdate("Hashes/s", s)

      time.Sleep(time.Second)
    }
  })

  recovery.SafeGo(func() {
    var err error
    var newKernelHash []byte

    for {
      solution, ok := <-forgingWorkerSolutionCn
      if !ok {
        return
      }

      lastPrevKernelHash := thread.lastPrevKernelHash.Load()
      if lastPrevKernelHash != nil && solution.blkComplete.Height > 1 &&
!bytes.Equal(solution.blkComplete.PrevKernelHash, lastPrevKernelHash) {
        continue
      }

      if newKernelHash, err = thread.publishSolution(solution); err != nil {
        gui.GUI.Error(fmt.Errorf("Error publishing solution: %d error: %s ", solution.blkComplete.Height, err))
      } else {
        gui.GUI.Info(fmt.Errorf("Block was forged! %d ", solution.blkComplete.Height))
        thread.lastPrevKernelHash.Store(newKernelHash)
      }

    }
  })

  recovery.SafeGo(func() {
    for {
      newWork, ok := <-thread.nextBlockCreatedCn
      if !ok {
        return
      }

      thread.lastPrevKernelHash.Store(newWork.BlkComplete.PrevKernelHash)
```

```
        for i := 0; i < thread.threads; i++ {
          thread.workers[i].workCn <- newWork
        }

        gui.GUI.InfoUpdate("Hash Block", strconv.FormatUint(newWork.BlkHeight, 10))
      }
    })

}

func (thread *ForgingThread) publishSolution(solution *ForgingSolution) ([]byte, error) {

  newBlk := block_complete.CreateEmptyBlockComplete()
  if err := newBlk.Deserialize(helpers.NewBufferReader(solution.blkComplete.SerializeToBytes())); err != nil {
    return nil, err
  }

  newBlk.Block.StakingNonce = solution.stakingNonce
  newBlk.Block.Timestamp = solution.timestamp
  newBlk.Block.StakingAmount = solution.stakingAmount

  txs, _ := thread.mempool.GetNextTransactionsToInclude(newBlk.Block.PrevHash)

  txStakingReward, err := thread.createForgingTransactions(newBlk, solution.publicKey,
solution.decryptedStakingBalance, txs)
  if err != nil {
    return nil, err
  }

  newBlk.Txs = append(txs, txStakingReward)

  newBlk.Block.MerkleHash = newBlk.MerkleHash()

  newBlk.Bloom = nil
  if err = newBlk.BloomAll(); err != nil {
    return nil, err
  }

  //send message to blockchain
  result := make(chan *blockchain_types.BlockchainSolutionAnswer)
  thread.solutionCn <- &blockchain_types.BlockchainSolution{
    newBlk,
    result,
  }

  res := <-result
  return res.ChainKernelHash, res.Err
}
```

Annex 3 - Encryption Cipher

```
type EncryptionCipher struct {
  gcm cipher.AEAD
  sync.Mutex
}

func CreateEncryptionCipher(password string, salt []byte, time uint32) (*EncryptionCipher, error) {

  if len(salt) != 32 {
    return nil, errors.New("Salt must be 32 byte")
  }

  key := argon2.IDKey([]byte(password), salt, time, 32*1024, 4, 32)

  block, err := aes.NewCipher(key)
  if err != nil {
    return nil, err
  }

  gcm, err := cipher.NewGCM(block)
  if err != nil {
    return nil, err
  }

  nonce := make([]byte, gcm.NonceSize())
  if _, err = io.ReadFull(rand.Reader, nonce); err != nil {
    return nil, err
  }

  return &EncryptionCipher{gcm, sync.Mutex{}}, nil

}

func (encryption *EncryptionCipher) Encrypt(data []byte) ([]byte, error) {

  encryption.Lock()
  defer encryption.Unlock()

  nonce := make([]byte, encryption.gcm.NonceSize())
  if _, err := io.ReadFull(rand.Reader, nonce); err != nil {
    return nil, err
  }

  return encryption.gcm.Seal(nonce, nonce, data, nil), nil
}

func (encryption *EncryptionCipher) Decrypt(data []byte) ([]byte, error) {

  encryption.Lock()
  defer encryption.Unlock()

  nonceSize := encryption.gcm.NonceSize()
  nonce, ciphertext := data[:nonceSize], data[nonceSize:]

  out, err := encryption.gcm.Open(nil, nonce, ciphertext, nil)
```

```
  if err != nil {
    return nil, err
  }
  return out, nil
}
```

Annex 4 - Wallet Encryption

```
type WalletEncryption struct {
  wallet        *Wallet
  Encrypted     EncryptedVersion `json:"encrypted" msgpack:"encrypted"`
  Salt          []byte          `json:"salt" msgpack:"salt"`
  Difficulty    int             `json:"difficulty" msgpack:"difficulty"`
  password      string
  encryptionCipher *encryption.EncryptionCipher
}

func createEncryption(wallet *Wallet) *WalletEncryption {
  return &WalletEncryption{
    wallet:    wallet,
    Encrypted: ENCRYPTED_VERSION_PLAIN_TEXT,
  }
}

func (self *WalletEncryption) Encrypt(newPassword string, difficulty int) (err error) {
  self.wallet.Lock.Lock()
  defer self.wallet.Lock.Unlock()

  if !self.wallet.Loaded {
    return errors.New("Wallet was not loaded!")
  }

  if self.Encrypted != ENCRYPTED_VERSION_PLAIN_TEXT {
    return errors.New("Wallet is encrypted already! Remove the encryption first")
  }

  if difficulty <= 0 || difficulty > 10 {
    return errors.New("Difficulty must be in the interval [1,10]")
  }

  self.Encrypted = ENCRYPTED_VERSION_ENCRYPTION_ARGON2
  self.password = newPassword
  self.Salt = helpers.RandomBytes(32)
  self.Difficulty = difficulty

  if err = self.createEncryptionCipher(); err != nil {
    return
  }

  if err = self.wallet.saveWalletEntire(false); err != nil {
    return
  }

  globals.MainEvents.BroadcastEvent("wallet/encrypted", true)
  return
}
```

```
func (self *WalletEncryption) encryptData(input []byte) ([]byte, error) {
  if self.Encrypted == ENCRYPTED_VERSION_ENCRYPTION_ARGON2 {
    return self.encryptionCipher.Encrypt(input)
  }
  return input, nil
}

func (self *WalletEncryption) createEncryptionCipher() (err error) {
  if self.encryptionCipher, err = encryption.CreateEncryptionCipher(self.password, self.Salt, uint32(self.Difficulty)*30); err != nil {
    return
  }
  return
}

func (self *WalletEncryption) Decrypt(password string) (err error) {
  return self.wallet.loadWallet(password, false)
}

func (self *WalletEncryption) decryptData(input []byte) ([]byte, error) {
  if self.Encrypted == ENCRYPTED_VERSION_ENCRYPTION_ARGON2 {
    return self.encryptionCipher.Decrypt(input)
  }
  return input, nil
}

func (self *WalletEncryption) CheckPassword(password string, requirePassword bool) error {
  self.wallet.Lock.RLock()
  defer self.wallet.Lock.RUnlock()

  if !self.wallet.Loaded {
    return errors.New("Wallet was not loaded!")
  }

  if requirePassword {
    if self.Encrypted == ENCRYPTED_VERSION_PLAIN_TEXT {
      return errors.New("Wallet is not encrypted!")
    }
    if self.password == "" {
      return errors.New("Wallet password was not set!")
    }
  }

  if self.password != password {
    return errors.New("Password is not matching")
  }

  return nil
}

func (self *WalletEncryption) RemoveEncryption() (err error) {
  self.wallet.Lock.Lock()
  defer self.wallet.Lock.Unlock()

  if !self.wallet.Loaded {
    return errors.New("Wallet was not loaded!")
  }
  if self.Encrypted == ENCRYPTED_VERSION_PLAIN_TEXT {
```

```
    return errors.New("Wallet is not encrypted!")
  }

  self.Encrypted = ENCRYPTED_VERSION_PLAIN_TEXT
  self.password = ""
  self.Difficulty = 0

  if err = self.wallet.saveWalletEntire(false); err != nil {
    return
  }

  globals.MainEvents.BroadcastEvent("wallet/removed-encryption", true)
  return
}

func (self *WalletEncryption) Logout() (err error) {
  self.wallet.Lock.Lock()
  if !self.wallet.Loaded {
    self.wallet.Lock.Unlock()
    return
  }
  if self.Encrypted == ENCRYPTED_VERSION_PLAIN_TEXT {
    self.wallet.Lock.Unlock()
    return errors.New("Wallet is not encrypted!")
  }
  self.wallet.clearWallet()
  self.wallet.Lock.Unlock()

  if err = self.wallet.loadWallet("", true); err != nil {
    return nil
  }

  globals.MainEvents.BroadcastEvent("wallet/logged-out", true)
  return
}
```

Annex 6 - Mempool thread worker

```
//process the worker for transactions to prepare the transactions to the forger
func (worker *mempoolWorker) processing(
  newWorkCn <-chan *mempoolWork,
  suspendProcessingCn <-chan struct{},
  continueProcessingCn <-chan ContinueProcessingType,
  addTransactionCn <-chan *MempoolWorkerAddTx,
  insertTransactionsCn <-chan *MempoolWorkerInsertTxs,
  removeTransactionsCn <-chan *MempoolWorkerRemoveTxs,
  txs *MempoolTxs,
) {

  var work *mempoolWork
  var dataStorage *data_storage.DataStorage

  txsList := []*mempoolTx{}
  txsMap := make(map[string]*mempoolTx)
  listIndex := 0
```

```
includedTotalSize := uint64(0)
includedTxs := []*mempoolTx{}

resetNow := func(newWork *mempoolWork) {

  if newWork.chainHash != nil {
    dataStorage = nil
    work = newWork
    includedTotalSize = uint64(0)
    includedTxs = []*mempoolTx{}
    listIndex = 0
    if len(txsList) > 1 {
      sortTxs(txsList)
    }
  }
}

removeTxNow := func(tx *mempoolTx, txWasInserted bool, includedInBlockchainNotification bool) {

  delete(txsMap, tx.Tx.Bloom.HashStr)

  if txWasInserted {
    txs.deleteTx(tx.Tx.Bloom.HashStr)
    txs.deleted(tx, txWasInserted, includedInBlockchainNotification)

  }
}

removeTxs := func(data *MempoolWorkerRemoveTxs) {

  removedTxsMap := make(map[string]bool)
  for _, hash := range data.Txs {
    if hash != "" {
      if tx := txsMap[hash]; tx != nil {
        removedTxsMap[hash] = true
        removeTxNow(tx, true, true)
      }
    }
  }
  if len(removedTxsMap) > 0 {

    newLength := 0
    for _, tx := range txsList {
      if !removedTxsMap[tx.Tx.Bloom.HashStr] {
        newLength += 1
      }
    }

    newList := make([]*mempoolTx, newLength)
    c := 0
    index := 0
    for _, tx := range txsList {
      if !removedTxsMap[tx.Tx.Bloom.HashStr] {
        newList[c] = tx
        c += 1
      } else if index < listIndex && listIndex > 0 {
        listIndex--
        index--
```

```
        }
        index++
      }
      txsList = newList
    }

    data.Result <- len(removedTxsMap) > 0
  }

  insertTxs := func(data *MempoolWorkerInsertTxs) {
    result := false
    for _, tx := range data.Txs {
      if tx != nil && txsMap[tx.Tx.Bloom.HashStr] == nil {
        txsMap[tx.Tx.Bloom.HashStr] = tx
        txs.insertTx(tx)
        txs.inserted(tx)
        txsList = append(txsList, tx)
        result = true
      }
    }
    data.Result <- result
  }

  suspended := false
  for {

    select {
    case <-suspendProcessingCn:
      suspended = true
      continue
    case newWork := <-newWorkCn:
      resetNow(newWork)
    case data := <-removeTransactionsCn:
      removeTxs(data)
    case data := <-insertTransactionsCn:
      insertTxs(data)
    case continueProcessingType := <-continueProcessingCn:

      suspended = false

      switch continueProcessingType {
      case CONTINUE_PROCESSING_ERROR:
      case CONTINUE_PROCESSING_NO_ERROR:
        work = nil //it needs a new work
      case CONTINUE_PROCESSING_NO_ERROR_RESET:
        dataStorage = nil
        listIndex = 0
      }

    }

    if work == nil || suspended { //if no work was sent, just loop again
      continue
    }

    //let's check hf the work has been changed
    store.StoreBlockchain.DB.View(func(dbTx store_db_interface.StoreDBTransactionInterface) (err error) {
```

```
if dataStorage != nil {
  dataStorage.SetTx(dbTx)
}

var tx *mempoolTx
var newAddTx *MempoolWorkerAddTx

for {

  if dataStorage == nil {
    dataStorage = data_storage.NewDataStorage(dbTx)
  }

  tx = nil
  newAddTx = nil

  if listIndex == len(txsList) {
    select {
    case newWork := <-newWorkCn:
      resetNow(newWork)
    case <-suspendProcessingCn:
      suspended = true
      return
    case data := <-removeTransactionsCn:
      removeTxs(data)
    case data := <-insertTransactionsCn:
      insertTxs(data)
    case newAddTx = <-addTransactionCn:
      tx = newAddTx.Tx
      if txsMap[tx.Tx.Bloom.HashStr] != nil {
        if newAddTx.Result != nil {
          newAddTx.Result <- errors.New("Already found")
        }
        continue
      }
    }
  } else {
    select {
    case newWork := <-newWorkCn:
      resetNow(newWork)
    case <-suspendProcessingCn:
      suspended = true
      return
    case data := <-removeTransactionsCn:
      removeTxs(data)
    case data := <-insertTransactionsCn:
      insertTxs(data)
    default:
      tx = txsList[listIndex]
      listIndex += 1
    }

  }

  if tx == nil {
    continue
  }
```

```
var finalErr error
var exists bool

if exists = dbTx.Exists("txHash:" + string(tx.Tx.Bloom.HashStr)); exists {
  finalErr = errors.New("Tx is already included in blockchain")
}

if finalErr == nil {
  //was rejected by mempool nonce map
  finalErr = func() (err error) {

    defer func() {
      if errReturned := recover(); errReturned != nil {
        err = errReturned.(error)
      }
    }()

    if err = tx.Tx.IncludeTransaction(work.chainHeight, dataStorage); err != nil {
      dataStorage.Rollback()
    } else {

      if includedTotalSize+tx.Tx.Bloom.Size < config.BLOCK_MAX_SIZE {

        includedTotalSize += tx.Tx.Bloom.Size
        includedTxs = append(includedTxs, tx)

        atomic.StoreUint64(&work.result.totalSize, includedTotalSize)
        work.result.txs.Store(includedTxs)

        if err = dataStorage.CommitChanges(); err != nil {
          return
        }

      } else {
        dataStorage.Rollback()
      }

      if newAddTx != nil {
        listIndex += 1
        txsList = append(txsList, newAddTx.Tx)
        txsMap[tx.Tx.Bloom.HashStr] = newAddTx.Tx
        txs.insertTx(tx)
        txs.inserted(tx)
      }

    }

    return
  }()

  if finalErr != nil {
    if newAddTx == nil {
      //removing
      //this is done because it was inserted before
      txsList = slices.Delete(txsList, listIndex-1, listIndex)
      listIndex--
    }
    removeTxNow(tx, newAddTx == nil, exists)
```

```
            }

            if newAddTx != nil && newAddTx.Result != nil {
              newAddTx.Result <- finalErr
            }

          }
        }

      })

    }
}
```

References

[1] Kademlia - https://en.wikipedia.org/wiki/Kademlia .

[2] Petar Maymounkov, David Mazières - Kademlia: A Peer-to-peer Information System Based on the XOR Metric - https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

[3] Satoshi Nakamoto - Bitcoin: A Peer-to-Peer Electronic Cash System - https://bitcoin.org/bitcoin.pdf

[4] J.D. Bruce – The Mini-Blockchain Scheme - http://cryptochainuni.com/wp-content/uploads/The-Mini-Blockchain-Scheme.pdf

[5] Nicolas van Saberhagen – CryptoNote v 2.0 whitepaper - https://bytecoin.org/old/whitepaper.pdf

[6] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin – An Empirical Analysis of Traceability in the Monero Blockchain https://arxiv.org/pdf/1704.04299/

[7] Chainalysis Team - Introducing Investigations & Compliance Support for Privacy Coins Dash and Zcash - https://blog.chainalysis.com/reports/introducing-investigations-compliance-support-for-privacy-coins/

[8] Zcash - Parameter Generation - https://z.cash/technology/paramgen/

[9] Sean Bowe, Jack Grigg, and Daira Hopwood - Recursive Proof Composition without a Trusted Setup - https://eprint.iacr.org/2019/1021.pdf

[10] Ian Miers, Christina Garman, Matthew Green, Aviel D. Rubin - Zerocoin: Anonymous Distributed E-Cash from Bitcoin - https://cryptorating.eu/whitepapers/Zcoin/ZerocoinOakland.pdf

[11] Lovesh Harchandani - ZCoin's new anonymous payment system, Lelantus - https://medium.com/@loveshharchandani/zcoins-new-anonymous-payment-system-lelantus-23b27a450a9

[12] Tom Elvis Jedusor, MIMBLEWIMBLE – https://docs.beam.mw/Mimblewimble.pdf

[13] Ivan Bogatty - Breaking Mimblewimble's Privacy Model - https://medium.com/dragonfly-research/breaking-mimblewimble-privacy-model-84bcd67bfe52

[14] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh - Zether: Towards Privacy in a Smart Contract World - https://crypto.stanford.edu/~buenz/papers/zether.pdf

[15] Benjamin E. Diamond - MANY-OUT-OF-MANY PROOFS and applications to Anonymous Zether https://eprint.iacr.org/2020/293.pdf

[16] Chainalysis Team - Mapping the Universe of 460 Million Bitcoin Addresses - https://blog.chainalysis.com/reports/bitcoin-addresses/

[17] Kazakhstan man-in-the-middle attack - https://en.wikipedia.org/wiki/Kazakhstan_man-in-the-middle_attack

[18] Brian Curran - What is The Dandelion Protocol? Complete Beginner's Guide - https://blockonomi.com/dandelion-protocol/

[19] ECIES Hybrid Encryption Scheme https://cryptobook.nakov.com/asymmetric-key-ciphers/ecies-public-key-encryption

[20] Transactions Rate Per Second – https://www.blockchain.com/charts/transactions-per-second

[21] Badger vs LMDB vs BoltDB: Benchmarking key-value databases in Go - https://dgraph.io/blog/post/badger-lmdb-boltdb/

[22] There are now more than 1 billion unique bitcoin addresses https://blazetrends.com/there-are-now-more-than-1-billion-unique-bitcoin-addresses/