SoK: A Taxonomy of Cryptocurrency Wallets

Kostis Karantias

IOHK
kkarantias@gmail.com

Abstract. The primary function of a cryptocurrency is money transfer between individuals. The *wallet* is the software that facilitates such transfers. Wallets are nowadays ubiquitous in the cryptocurrency space and a cryptocurrency is usually supported by many wallets. Despite that, the functionality of wallets has never been formally defined. Additionally, the mechanisms employed by the many wallets in the wild remain hidden in their respective codebases.
In this work we provide the first definition of a cryptocurrency wallet, which we model as a client to a server, or set of servers. We provide a distinction of wallets in various categories, based on whether they work for transparent or private cryptocurrencies, what trust assumptions they require, their performance and their communication overhead. For each type of wallet we provide a description of its client and server protocols. Additionally, we explore *superlight wallets* and describe their difference to *superlight clients* that have appeared in recent literature. We demonstrate how new wallet protocols can be produced by combining concepts from existing protocols. Finally we evaluate the performance and security characteristics of all wallet protocols and compare them.

1 Introduction

Cryptocurrencies have seen a significant increase in usage in recent years. In 2008, Bitcoin was introduced to the world [48]. Along with the whitepaper, a piece of software implementing the cryptocurrency was released that contained a single user-facing application—the wallet. The wallet allowed users to interact with the cryptocurrency, to send it to others and receive it. For a while it seemed that money transfer between individuals was the sole purpose of a cryptocurrency. With the rise of applications such as Colored Coins [57] built on top of Bitcoin, and ultimately Ethereum [8,70], wallets had to fill more roles than just money transfer. However, looking at the majority of wallets in the wild today, money transfer between individuals appears to remain their central role.

Wallets on the surface seem relatively unchanged throughout the years. Their user interface is usually simple and intuitive, and nowadays they are ubuquitous. There exist desktop wallets, mobile wallets, and hardware wallets. With the plethora of cryptocurrencies coming to exist in the recent years, many solutions have been developed to support them.

Due to the multitude of supported platforms for wallets, and the requirements for efficiency and ease of use, wallet developers have devised different ways

of implementing them. Usability is one of the most important obstacles in the adoption of cryptocurrencies [41]. One of the main problems wallet developers have been faced with is the blockchain size [41,5]. Simple Payment Verification (SPV), originally mentioned in the Bitcoin whitepaper [48] has been frequently leveraged in order to build wallets which are light on storage and network communication. Some very popular wallets are based on it, but the nuances of each of these protocols are hard to find and do not appear in literature.

With the advent of Ethereum and its account model, new ways of designing wallets appeared that also have not made their appearance in literature. Finally, private cryptocurrencies such as ZCash and Monero have appeared in recent years, for which different wallets have been built. The techniques and challenges faced by these wallets differ from the ones already discussed, but also do not appear in literature.

Unfortunately until today no exposition of all proposed solutions exists. This has led to wallets in the market that maybe insecure or have varying assumptions that may seem unlikely.

Our contributions. In this work, we:

- Define the purpose of a cryptocurrency wallet: to create and broadcast valid transactions, to know the balance and to know the transaction history in Section 3.
- Provide detailed descriptions of how wallets in practice work for transparent cryptocurrencies in both the UTXO (Section 4) and account (Section 5) models.
- Describe existing solutions for wallets for private cryptocurrencies in the UTXO model in Section 6.
- Define for the first time the functionality of superlight clients, which are commonly conflated with superlight wallets in Section 7.
- Provide constructions for superlight wallets based on a superlight client for transparent (Section 7.1) and private (Section 7.2) cryptocurrencies.
- Compare all wallet protocols on their performance and security characteristics in Section 8.

2 Preliminaries

2.1 The UTXO Model

In the UTXO model a transaction consists of inputs and outputs. An output consists of an amount and a locking script (called `scriptPubKey` in Bitcoin). In Bitcoin the locking script is described in a stack-based language called Bitcoin Script. An input consists of an unlocking script (called `scriptSig` in Bitcoin) and an output pointer (also called an outpoint) which is a tuple of the form $\langle \texttt{txID}, \texttt{outputIndex} \rangle$. We say that an input *spends* an output, where `txID` refers to the transaction id that includes the output and `outputIndex` refers to the zero-based index of the output to be spent in that transaction.

The global state of UTXO based cryptocurrencies includes all transaction outputs, and all spent transaction outputs. UTXO stands for unspent transaction outputs, which be trivially computed from any state σ_{btc} as $\sigma_{\text{btc}}.\texttt{utxo} \triangleq \sigma_{\text{btc}}.\texttt{outputs} \setminus \sigma_{\text{btc}}.\texttt{spentOutputs}$.

An input is valid with regards to a state σ_{btc} (a) if the output it references exists (i.e. $\in \sigma_{\text{btc}}.\texttt{outputs}$) and is not already spent (i.e. $\notin \sigma_{\text{btc}}.\texttt{spentOutputs}$) and (b) if the unlocking script of the input "unlocks" the locking script of the output. In the case of Bitcoin, the unlocking script runs on an empty stack, and subsequently the locking script runs on the resulting stack of the unlocking script's execution. The input is considered to unlock the output if the locking script terminates successfully.

The transaction as a whole is valid if all inputs are valid and the sum of the values generated by its outputs is at most the sum of the values brought in by its inputs. The absolute value of the difference of values is called the *fee* and is used to incentivise miners in the network as we are going to see shortly. The state can be updated by a valid transaction as shown in Algorithm 1, where the new outputs are added and newly spent outputs are marked as such.

The most usual form of output is Pay-to-Public-Key-Hash (P2PKH) which, as its name implies, includes the public key hash of the recipient. For an input spending it to be valid, it needs to contain a signature of the corresponding secret key over the new transaction skeleton and the output to be spent.

Algorithm 1 The $\text{apply}_{\text{btc}}$ function given a state σ_{btc} and a transaction tx.

```
1: function apply_btc(σ_btc, tx)
2:     for o ∈ tx.outputs do
3:         σ_btc.outputs ∪= {o}
4:     end for
5:     for i ∈ tx.inputs do
6:         σ_btc.spentOutputs ∪= {i.output}
7:     end for
8:     return σ_btc
9: end function
```

2.2 The Account Model

In the account model, a transfer transaction consists of a sender, a receiver, an amount, a nonce, a fee and a signature. The sender and receiver are public keys. The fee acts as an additional miner reward as in the UTXO model.

Each address has its own state. The state of an address is defined as $\langle \text{balance}, \text{nonce} \rangle$. For a previously unused address all fields are assumed to be of zero value, specifically: $\sigma_{\text{eth}}[\texttt{address}] = \{\text{balance}: 0, \text{nonce}: 0\}$. With $\sigma_{\text{eth}}[\texttt{address}]$ we denote the state of the account with the specified address.

For a transfer transaction to be valid with respect to a state σ_{eth}, the following conditions need to hold.

– The signature must be valid and generated by the secret key of the sender over the rest of the transfer transaction.
– The nonce must be equal to $\sigma_{\text{eth}}[\texttt{sender}].\text{nonce} + 1$.
– The sender must have adequate balance.

$$\sigma_{\text{eth}}[\texttt{sender}].\text{balance} \geq \texttt{amount} + \texttt{fee}$$

We remark that the nonce is necessary in order to avoid replay attacks [16], for if it was absent some receiver would be able to repeat the same transaction ad infinitum until the sender account is completely drained. Note that a potential solution to this would be to maintain a set of processed transaction ids and add a requirement that the transaction to be evaluated for validity does not belong in that set. If the sender ever wishes to send the same amount to the same recipient, they will need to generate a new signature and thus the transaction will have a different transaction id. In theory this solution stands, however in practice due to the malleability of ECDSA a signature can be modified in a way that it changes and consequently the transaction id changes, while retaining its validity [62].

A valid transaction according to a state can be applied to it to generate a new state. We perform this state transformation with a function $\text{apply}_{\text{eth}}(\sigma_{\text{eth}}, tx)$ which debits the sender, credits the receiver and increments the sender's nonce. The full definition of $\text{apply}_{\text{eth}}$ is given in Algorithm 2.

Algorithm 2 The $\text{apply}_{\text{eth}}$ function given a σ_{eth} and a transaction.

```
1: function apply_eth(σ_eth, tx)
2:     σ_eth[tx.sender].nonce += 1
3:     σ_eth[tx.sender].balance -= tx.amount
4:     σ_eth[tx.receiver].balance += tx.amount
5:     return σ_eth
6: end function
```

2.3 The Blockchain

Transactions are usually arranged in batches and included in so-called *blocks*. Blocks are arranged in a structure which resembles a linked list, in which each block contains a pointer to its previous block in the form of its hash. The only exception to this rule is the first block of the list, which is called the *genesis* and does not include any such pointer. This linked list is called the *chain*, which we usually denote with $\mathcal{C}$. With $\mathcal{C}[i]$ we denote the i-th block of the chain, and we customarily denote the genesis block as $\mathsf{Gen} \triangleq \mathcal{C}[0]$.

We adopt the Python notation to refer to parts of the chain, which is illustrated in Table 1. We call $\mathcal{C}[:-k]$ the *stable* part of $\mathcal{C}$.

Whether in the UTXO or account model, we say that each block has a state. The state of a block $\mathcal{C}[i]$ is defined as the state of the previous block $\mathcal{C}[i-1]$

Table 1. The notation used throughout this work.

$\mathcal{C}$	the chain
$\mathcal{C}[i]$	the i-th block of the chain ($i \geq 0$)
Gen	the genesis block ($\mathcal{C}[0]$)
$\mathcal{C}[-i]$	the i-th block from the end of the chain ($i > 0$)
$\mathcal{C}[i:j]$	all chain blocks starting from $\mathcal{C}[i]$ up to but not including $\mathcal{C}[j]$
$\mathcal{C}[:j]$	all chain blocks from the genesis up to but not including $\mathcal{C}[j]$ (equivalent to $\mathcal{C}[0:j]$)
$\mathcal{C}[i:]$	all chain blocks from $\mathcal{C}[i]$ up to and including the last (equivalent to $\mathcal{C}[i:\lvert\mathcal{C}\rvert]$)
k	the stability parameter of the blockchain protocol

upon application of all transactions present in $\mathcal{C}[i]$. More formally, we define the state of a block as follows:

$$\text{state}(\mathcal{C}[i]) = \begin{cases} \text{apply}(\text{state}(\mathcal{C}[i-1]), \mathcal{C}[i].\texttt{txs}) & \text{if } i \geq 0 \\ \epsilon & \text{otherwise} \end{cases}$$

Application of a sequence of transactions occurs by applying each transaction and using the new state for the next application. This can be written more formally as follows

$$\begin{cases} \text{apply}(st, [tx_1, \ldots, tx_z]) = \text{apply}(\text{apply}(st, tx_1), [tx_2, \ldots, tx_z]) \\ \text{apply}(st, []) = st \end{cases}$$

A block b is split in two parts, the header and the body. The header is a fixed-size string, and the body is the place where all transactions reside. All the transactions in a block's body are committed to in the block's header, usually in the form of a Merkle Tree root over all of them. A block with its body is what we call a *full block*. We will use $\mathcal{C}[i]$ to refer to a block header or the full block at height i interchangeably, as well as $\mathcal{C}$ to refer to both the header chain and the full block chain.

Money generation happens with coinbase transactions. A coinbase transaction is a special transaction that provides miners (in Proof-of-Work based cryptocurrencies) or minters (in Proof-of-Stake based cryptocurrencies) with a reward. Each block may have a single coinbase transaction. For cryptocurrencies in the UTXO model the coinbase transaction contains no inputs and generates only as much value as the block reward per the consensus rules, plus any transaction fees. Coinbase transactions could also exist in account-based cryptocurrencies, for example having a `sender` field of all zeros. In practice Ethereum-based cryptocurrencies don't have them but instead directly include the miner's address inside the block header. For our purposes we will not consider the change of state due to miner rewards.

2.4 Network Actors

Full Nodes A *full node* maintains the full block chain, along with every transaction. Every transaction is verified for validity and a block that contains an invalid

transaction is considered invalid by the full node. A full node participates in a Peer-to-Peer (P2P) network. Other nodes may request blocks or transactions from it, and it serves them akin to a relay.

Miners A *miner* is a full node that also attempts to mine a new block on top of the best block chain. If a block is successfully mined, it is broadcasted to the P2P network of full nodes for them to consider adopting it.

2.5 Simple Payment Verification

Simple Payment Verification (SPV) first appeared in the Bitcoin whitepaper [48] as a way for nodes to figure out their transactions without the need to download and store full blocks. An SPV client works by obtaining the header-chain from other full nodes and validating it. Upon doing that, it can verify any transaction is included in one of the blocks by verifying a Merkle inclusion proof against a Merkle root in one of the headers. We will now formally define an SPV client.

Definition 1 (SPV client). *An* SPV client *is an ITM that knows a genesis block* Gen *and connects to a set of full nodes* $\mathcal{P}$*, at least one of which is honest. It may only request block headers at a range of heights. It obtains all header-chains* $\mathcal{C}_1, \ldots, \mathcal{C}_{|\mathcal{P}|}$*, keeping only those for which the chain structure holds (*$\forall 1 < i \leq |\mathcal{C}| : H(\mathcal{C}[i-1]) = \mathcal{C}[i].previd$*). It subsequently compares them and adopts the chain with the most difficulty.*

The assumption of at least one honest full node is necessary to avoid eclipse attacks [26,72].

We state some results pertaining to the adopted chain of an SPV client which have appeared previously in [5,3] and are more formally derivable from the works of [20,21,55].

Definition 2 (Honest majority assumption). *The adversarial mining power is at all times upper bounded by the half of the total mining power of the network.*

Theorem 1 (SPV security). *Under the honest majority assumption, the stable part of an SPV client's adopted chain* $\mathcal{C}[:-k]$ *is a prefix of the chain of an honest full node.*

Corollary 1. *Under the honest majority assumption, the underlying full chain of the stable part of an SPV client's adopted chain produces a valid state.*

3 Our Model

A cryptocurrency wallet facilitates the transfer of funds between individuals. The wallet contains the private keys that can be used to spend the user's cryptocurrency. With those private keys, it should be able to fulfill the following functions.

1. Present the user with their balance.
2. Allow the user create new valid transactions, given a description.
3. Present the user with their transaction history.

The lifecycle of the wallet is as follows.

Sync. First the user generates a new secret key or recovers from an existing secret key. The wallet then performs an initial synchronisation by using the network, in order to obtain everything necessary for it to be able to perform the aforementioned functions. We denote this synchronisation step as producing a wallet state $\mathsf{wstate} \leftarrow \text{Sync}_{\mathsf{Gen},\mathcal{S}}(pk)$, where Gen is the genesis block of the blockchain and $\mathcal{S}$ a set of servers the wallet is allowed to interact with. After this initial synchronisation is complete, the wallet keeps using the network to stay up to date with relevant events such as the user receiving a new transaction. The user may shut down the wallet and start it at some other point in time, when the wallet will attempt to catch-up to the network in order to be one again usable.

Fund. The user may wish to create a new transaction. They specify a *transaction description* to the wallet which includes the recipients and the amounts of cryptocurrency the users wishes to send to each one. To obtain the final valid transaction invokes a function $tx \leftarrow \text{Fund}(sk, \mathsf{wstate}, \mathsf{description})$ which makes use of the user's secret key sk, the wallet state wstate obtained from the Sync function and the provided transaction $\mathsf{description}$.

History. We model the transaction history as a function extracting it from the wallet state, namely $tx_1, \ldots, tx_y \leftarrow \text{History}(pk, \mathsf{wstate})$.

Balance. The wallet provides a function to determine the balance of the user based on the wallet state as $\mathsf{balance} \leftarrow \text{Balance}(pk, \mathsf{wstate})$.

Definition 3 (Wallet Protocol). *A* wallet protocol π_{wallet} *with respect to some server protocol* $\pi_{wallet\text{-}server}$ *that each server* $s \in \mathcal{S}$ *implements and a blockchain rooted in* Gen *is defined as a tuple of functions* $\langle \mathit{Sync}_{\mathsf{Gen},\mathcal{S}}, \mathit{Fund}, \mathit{History}, \mathit{Balance} \rangle$.

Seeds. Bitcoin wallets traditionally generate a different address for every transaction description to use for the tranasction's change output. This however means that the user needs to periodically backup an ever-increasing number of secret keys or risk some of their coins. A solution to this was presented in 2011 in the form of "hierarchical deterministic wallets" [40,71]. A single secret called the *seed* is used to deterministically generate all future secret keys of the wallet. It is thus necessary only to backup this seed for a user to be able to retain all their funds, no matter how long they will make use of the wallet for. It is possible for a master public key to be created that only gives access to the public keys that correspond to the private keys derivable from the secret seed. Whereas wallets in practice usually restore from a seed, we adopt the simplification that the wallet

is called to restore from a signature scheme secret key. Restoring from the seed can be thought of as invoking the Sync protocol many times, starting from the first derivable public key of the seed and continuing until we reach a public key that seems unused. Additionally, in protocols when the public key is directly provided as part of synchronisation, the master public key could be provided instead so that the server has access to all of the wallet's public keys directly.

4 Transparent UTXO-based Wallets

A transaction description specifies the desired outputs of a transaction and a fee. In the simplest case of sending some funds to a specific address, the transaction description is comprised of a single P2PKH output.

A wallet needs to be able to turn a transaction description into a valid transaction in a process called funding. To fund a transaction description, the wallet performs the following steps:

- Collect a set S of unspent transaction outputs spendable by the public key such that
$$\sum_{o \in S} o.\text{value} \geq \mathsf{value}$$
- Create a change output directed back to the public key[1] such that
$$\sum_{input} input.\text{value} - \sum_{output} output.\text{value} = \mathsf{fee}$$
- For each unspent transaction output in S, generate its unlocking script and use it as an input. In the typical case the unlocking script includes the signature of said transaction with the secret key of said output recipient.

The balance of a wallet can be obtained as the sum of the values of all unspent outputs destined to public key.

The transaction history of a wallet can be extracted from the set of all transactions implicating the public key. Specifically, inputs which are spent by the secret key are debits, and outputs which are directed to the public key are credits.

4.1 Full Node

A full node contains in its state the complete chain with full blocks, that include all of their transactions. In addition, the chain is assumed to be verified for validity.

Full construction. When presented with a public key, the full node wallet starts evaluating all transactions in every block from genesis up to the tip to detect which ones implicate the user's public key. This process is linear in the chain size, thus $\Theta(n)$.

[1] A new public key is normally generated through the seed, but we study a simplified model here.

Reduced functionality mode. A full node needs to hold at all times the state at the blockchain tip in order to be able to verify potential new blocks extending the tip. For UTXO-based cryptocurrencies this state is exactly the UTXO. When presented with a public key, the wallet instead of looking through all transactions in history to obtain the full user's history can locate all currently unspent outputs directed to the user by looking through the UTXO set. In this way the user's balance can be directly obtained as the sum of such outputs. However, the transaction history is not known but new transactions can be generated without a problem.

If the UTXO is stored as a list, performance is linear in its size. However, the UTXO can easily be indexed by address, which makes detection $\Theta(1)$.

In practice. Cryptocurrencies are defined by their full nodes so the implementation already exists, so they are the easiest to build a wallet on top of. Usually the wallet is included in the bundle of software that the full node ships with. This is the case with Bitcoin [48], Litecoin [38], Dogecoin [12], Ergo [15], and many other cryptocurrencies.

4.2 BIP-37 SPV

SPV was first described in the Bitcoin whitepaper [48] and proposed how transactions can be verified by a node without them holding the full block chain, but instead by only holding the header chain. The idea is as follows, a light node only requests and holds the header chain, that is only the header part of each block (which in Bitcoin is 80 bytes) and not the full block. When someone wishes to prove a transaction took place to an SPV node, they need to provide (a) the transaction (b) the id of the block that contains it and (c) a Merkle proof of inclusion to the Merkle root included in said block. The SPV node can check that the claimed block is part of its local best header chain and the Merkle proof provided is valid against the block's Merkle root.

For BIP-37 SPV, the server protocol supports the following special calls:

1. `filterload`(bloom-filter): sets the connection bloom filter to the one provided
2. `getblock`(height):
   (a) if the connection bloom filter is set, returns a partial block for $\mathcal{C}[h]$ with:
      i. the block header
      ii. only the transactions of the block with outputs that match against the bloom filter
      iii. a multi-element proof of inclusion in the block transaction tree for all the aforementioned transactions
   (b) otherwise: returns the full block $\mathcal{C}[h]$

Using this functionality of an SPV node as a building block and the server protocol, we examine how an SPV wallet according to BIP-37 [24] can be built. The SPV wallet connects with the full node network. There, it establishes connection with some server. Remember the aim of the wallet is to figure out all

transactions relevant to a specified public key pk. The wallet constructs a bloom filter that includes pk and sets it as the connection bloom filter via a `filterload` call. After that it starts sending `getblock` requests to the server, starting from the genesis block.

For every block the server returns, it scans each transaction for matches to the set bloom filter. To the wallet, the block header is sent along with the transactions. Finally for verification, a multi-element Merkle proof of inclusion (named Partial Merkle Tree in BIP-37) for all the matching transactions is also sent. The wallet repeats `getblock` calls until it reaches the blockchain tip. By that point and assuming an honest peer, the full transaction history is known, along with the balance, and new transactions can be successfully created. The process is illustrated in detail in Algorithm 3.

Algorithm 3 The BIP-37 client syncing protocol for a blockchain rooted in Gen, with access to a fixed set of servers $\mathcal{S}$ and a public key pk for which all transactions must be obtained.

function $\mathrm{Sync}_{\mathsf{Gen},\mathcal{S}}(pk)$
  $\mathcal{C} \leftarrow \mathsf{Gen}$
  for server $\in \mathcal{S}$ **do**
    $\hat{\mathcal{C}} \leftarrow$ server.`GetHeaderChain`(Gen)
    $\mathcal{C} \leftarrow \mathsf{maxvalid}(\mathcal{C}, \hat{\mathcal{C}})$
  end for
  ourTXs $\leftarrow \{\}$
  filter $\leftarrow$ bloom-filter($\{pk\}$)
  for server $\in \mathcal{S}$ **do**
    server.`filterload`(filter)
    for $blk \in \mathcal{C}$ **do**
      $\langle txs, \pi \rangle \leftarrow$ server.`getblock`(blk)
      if $\neg\mathsf{Ver\text{-}Multi}_{\mathsf{auth\text{-}set}}(blk.\texttt{merkleRoot}, txs, \pi)$ **then**
        break
      end if
      for $tx \in txs$ **do**
        if $relevant(tx, pk)$ **then**
          ourTXs $\cup=$ $\{tx\}$
        end if
      end for
    end for
  end for
  return ourTXs
end function

In practice. A well known implementation of BIP-37 SPV is included in the bitcoinj library [25], which is a full Bitcoin implementation in Java. Andreas Schildbach's famous Android Bitcoin wallet [58] is based on that library, as well as decentralized exchange Bisq [4]. Other applications that implement BIP-37

SPV wallets include BRD [6] and OpenBazaar [49]. Other libraries that implement BIP-37 include uspv [14] and its fork spvwallet [64] (available both for Bitcoin and Bitcoin Cash [52]).

Note that the protocol may seem wasteful, especially in case the header chain is already synced as it forces us to re-download the whole header chain. Additionally, it may seem wasteful that the peer needs to process every single transaction in history in order to service the peer. It has been shown that not only is it wasteful but can be an exploitable Denial-of-Service vector [65]. This lead to the reference implementation of Bitcoin, Bitcoin Core, to disable BIP-37 server functionality by default on full nodes [10]. These inefficiencies are mitigated in the next protocol we are going to discuss.

4.3 Electrum

Electrum [68] was the first alternative and so-called "light" wallet for Bitcoin after the reference implementation. The protocol it utilizes is as follows. First, the wallet connects to 10 servers chosen from a hardcoded list in the software. One of them is selected as primary at random. The servers speak an Electrum-specific protocol called Stratum which follows the design originally proposed in [60] and not the full node protocol. In Electrum, servers hold an address index over all historical transactions. In early versions [69] of the server the index was arranged in a Merkle Patricia Trie [56]. The most widely used server implementation [76] does not arranging the index in this way and simply relies on an underlying key-value store.

The wallet obtains all the header-chains advertised by its servers and verifies their Proof-of-Work. It then only keeps the heaviest valid chain for further processing. Subsequently it makes use of the `blockchain.scripthash.get_history` [2] API call to obtain all transactions concerning the address of the user. For each of these transactions it requires a Merkle proof of inclusion that is valid against some block in the heaviest chain. The full process of syncing is illustrated in Algorithm 4.

We remark that the Electrum protocol is more efficient than the previous SPV wallet solution. Unfortunately, this comes at the cost of directly revealing the user's addresses to the remote server. An Electrum server customarily maintains an address index for all historical transactions, making servicing the `get_history` very efficient, in contrast with the heavy work a full node has to do to service an SPV wallet.

4.4 Neutrino

Neutrino is a new proposal for more efficient light syncing on Bitcoin which aims to be an improvement over SPV.

[2] The full API offered by an Electrum server is shown in `https://electrumx.readthedocs.io/en/latest/protocol-methods.html`

Algorithm 4 The Electrum client syncing protocol for a blockchain rooted in Gen, with access to a fixed set of servers $\mathcal{S}$ and a public key pk for which all transactions must be obtained.

function $\mathrm{Sync}_{\mathsf{Gen},\mathcal{S}}(pk)$
  $\mathcal{C} \leftarrow \mathsf{Gen}$
  $\mathsf{primary} \xleftarrow{\$} \mathcal{S}$
  for $\mathsf{server} \in \mathcal{S}$ **do**
    $\hat{\mathcal{C}} \leftarrow \mathsf{server.GetHeaderChain}(\mathsf{Gen})$
    $\mathcal{C} \leftarrow \mathsf{maxvalid}(\mathcal{C}, \hat{\mathcal{C}})$
  end for
  $txs \leftarrow \mathsf{primary.GetTXs}(pk)$
  $\mathsf{ourTXs} \leftarrow \{\}$
  for $\langle tx, h, \pi \rangle \in txs$ **do**
    if $\neg \mathsf{Ver}_{\mathsf{auth\text{-}set}}(\mathcal{C}[h].\mathtt{merkleRoot}, tx, \pi)$ **then**
      continue
    end if
    $\mathsf{ourTXs} \cup\!= \{tx\}$
  end for
  return ourTXs
end function

The simplest of its variants works as follows. It requires a hard or soft fork that is yet to occur for Bitcoin. With this fork, every block header includes a commitment to a bytearray which represents information about the block's transactions. Specifically, the bytearray which is called the *filter* is a Golomb-Coded Set [23] which encodes the set of all output scripts and the scripts of the outputs the inputs spend (except `OP_RETURN`s for technical reasons). The full construction for the filter is described in [50].

With this filter in place the light wallet works as follows. Initially, it downloads all block headers as usual and verifies them. Then it proceeds to download the filter corresponding to each block. It then checks locally for every block if it contains transactions of interest by making use of the filter. If a block contains transactions of interest, all its transactions are requested without witness data (e.g. signatures and data belonging to unlocking scripts). Then from the transactions obtained the wallet only keeps the actual transactions of interest and discards the rest. The process is illustrated in Algorithm 5.

Deployment paths. We look at two ways of implementing Neutrino filters in an existing cryptocurrency like Bitcoin.

Forks. First, with a soft-fork or hard-fork a commitment to the filter can be included in each block. With a soft-fork the commitment is included in the coinbase transaction, similar to how the wtxid root is included for SegWit [39]. With a hard-fork the commitment is included directly in the block header as a new field. The soft-fork is also indirectly commited in the header due to the transaction Merkle Tree root in the header. The commitment can be verified by additionally

verifying a Merkle proof of inclusion for the coinbase transaction against the tree root in the header and checking that the coinbase transaction itself contains the filter commitment. The hard-fork implementation of a Neutrino client is illustrated in Algorithm 5.

Majority vote. Another way to utilize Neutrino filters without adjusting the block structure and thus eliminating the need for either a soft- or hard-fork is to ask all servers for the filter of each block and pick the one with the most votes as valid, if a majority of the server has voted on it. This requires us to harden our security assumptions about the set of servers the wallet is connected to. Whereas previously we required that at least one server is honest, now we require that the majority of servers is honest.

The filter chain. In [51] a construction is put forth which is secure under our standard assumption of at least one honest server in $\mathcal{S}$. The construction introduces the notion of a filter chain, which is a hash chain of block filters. More formally if $\text{filter}(\mathcal{C}[i])$ refers to the Neutrino filter of block $\mathcal{C}[i]$, the filter chain F for the block chain $\mathcal{C}$ is defined for $0 \leq i < |\mathcal{C}|$ as

$$F[i] = \begin{cases} H(F[i-1], H(\text{filter}(\mathcal{C}[i]))) & \text{if } i > 0 \\ H(\text{filter}(\mathsf{Gen})) & \text{otherwise} \end{cases}$$

where every element of the filter chain is called a *filter header*. Neutrino servers compute this chain based on their filters.

The construction on a high-level works as follows. After first syncing the header-chain $\mathcal{C}$, the wallet connects with many Neutrino servers and requests all their filter chains. If two servers disagree on their filter chains we wish to tell who is honest and who is dishonest. The wallet finds the first filter header they disagree on, and obtains the filters for the corresponding block from both servers, as well as the full block. It then checks if one of the provided filters is invalid against the full block, and if it is the server that sent it is dishonest and the competing filter chain wins. This competition takes place for all disagreements and is how the honest filter chain is decided. Finally, the full filters can be downloaded from any server and verified against the local filter chain. The rest of the protocol proceeds as normal.

The improvements over SPV are twofold:

Privacy. In an SPV wallet, the transactions of interest to the wallet are directly leaked to the peer, except with some relative deniability due to the bloom filter's false positive rate. Due to the low false positive rate of a bloom filter and the widely known transaction linking heuristics that can be applied [42,22] the peer can be almost certain which transactions belong to the same entity, breaking the pseudonymity of Bitcoin.

Server performance. Observe how in Neutrino the server does no special computation for each client in contrast with an SPV server. In Neutrino servers are essentially relays of information, which is much cheaper and makes it more appealing to operators to operate them.

Algorithm 5 The Neutrino client syncing protocol for a blockchain rooted in Gen, with access to a fixed set of servers $\mathcal{S}$ and a public key pk for which all transactions must be obtained. The deployment method is through a hard-fork.

```
1: function Sync_{Gen,S}(pk)
2:     C ← Gen
3:     for server ∈ S do
4:         Ĉ ← server.GetHeaderChain(Gen)
5:         C ← maxvalid(C, Ĉ)
6:     end for
7:     ourTXs ← {}
8:     for blk ∈ C do
9:         for server ∈ S do
10:            filter ← server.GetFilter(blk)
11:            if H(filter) ≠ blk.filter then
12:                continue
13:            end if
14:            if filter ∩ {pk} ≠ ∅ then
15:                fullblk ← server.GetFullBlock(blk)
16:                valid ← validate(fullblk)
17:                if fullblk.header ≠ blk ∨ ¬valid then
18:                    continue
19:                end if
20:                for tx ∈ fullblk do
21:                    if relevant(tx, pk) then
22:                        ourTXs ∪= {tx}
23:                    end if
24:                end for
25:            end if
26:        end for
27:    end for
28:    return ourTXs
29: end function
```

Client performance. Additionally, in contrast with SPV, if the wallet happens to hold the header-chain through some means, it can just obtain the filters without the need to redownload the already downloaded headers as is the case with SPV.

However these improvements come at the cost of bandwidth. For each block its corresponding filter must be downloaded to detect relevant transactions in that block. The filter size for a 1.4MB block is approximately 20KB [61]. This is a significant overhead compared to SPV, where no such filter needs to be downloaded.

In practice. Neutrino is currently implemented in the bchd full node [54], giving an easy option to any bchd full node to act as a Neutrino server if they so desire. The homonymous Neutrino wallet [53] for Android works for Bitcoin Cash and uses the filter-chain variant of the Neutrino protocol. The Lightning wallet [33]

makes use of their Lightning Lab's inhouse Neutrino library [34] that implements the filter-chain variant of the Neutrino protocol.

4.5 Explorer-based

Most wallets in practice are explorer based. An explorer based wallet does not connect to full nodes or peers that relay block headers, only to a blockchain explorer. It requests the user's transactions from the explorer, but does not verify that the transactions are all included in blocks in the best chain. From them the wallet determines the UTXO set and can thus fund transaction descriptions, compute the balance and show the transaction history. Since no transaction verification is taking place, the explorer is a trusted third party.

In practice. The most popular wallets implement explorer-based solutions including Yoroi [73], Exodus [18] and Coinbarn [43]. Additionally, the apps that utilize hardware wallets like Ledger [37] and Trezor [66] work in this manner.

5 Transparent Account-based Wallets

For simplicity and without loss of generality we assume an Ethereum-like cryptocurrency.

Because in the account model there is no need for change outputs, new addresses are not automatically derived by wallets. New addresses are derived and used only by the explicit request of the user. Traditionally account-based wallets only hold a single address and public key, and we will assume the same in this section for simplification.

A transaction description is comprised of the address of the recipient, an amount of cryptocurrency and a fee. In Ethereum this fee takes the form of a gas price [70]. To fund a transaction description the wallet performs the following steps:

1. Fill in a valid address corresponding to a public key that:
   (a) holds at least as much cryptocurrency in its balance than what will be spent in the transaction and
   (b) is derivable from the seed.
2. Fill in the valid nonce in order to make this transaction spendable.
3. Sign the transaction with the appropriate secret key.

The transaction can subsequently be broadcast and accepted by the network.

We remark that an account-based wallet may offer limited functionality. For example, having access to the state it is easy to know the nonce and balance of any address of interest, thus funding a transaction description is straightforward. State however does not allow us to obtain the transaction history.

5.1 Full Node

A full node has access to the complete state and the full chain (i.e. all blocks and all their transactions), after a sync. For each derivable address, its balance and nonce can be looked up directly from the state. To obtain the transaction history for a set of addresses, every tranaction from the genesis block up to the tip must be checked.

5.2 Explorer-based

An explorer-based wallet as in the UTXO model connects to a trusted server and requests the list of all transactions, which then becomes its wallet state and it uses to function. It has no notion of the honestly adopted chain or any chain at all. For the account-model the differences are minimal. In Ethereum the wallet may additionally connect to a trusted Ethereum node for getting the balance or transaction count (i.e. the nonce).

In practice. Metamask [44] is the most popular Ethereum wallet and is explorer-based. An explorer is used in order to obtain the transaction history, usually Etherscan [17].

5.3 SPV without history

It is very easy to conceive how an SPV wallet for Electrum would work without history capabilities. Observe that for spendability only the nonce for the wallet's address is necessary. The nonce along with the balance are included in the state trie of Ethereum, which is committed in the form of the trie root inside every block header. Thus a wallet would work as follows:

1. Perform a headers sync to obtain the blockchain tip.
2. Ask for the value of the address of the user in the state trie at the tip, along with a proof for the value.
3. Verify the that the value is indeed committed in the root found at the tip block header and extract the balance and nonce from it.

5.4 Electrum-like

We propose a solution based on Electrum for account-based cryptocurrencies. We augment the wallet server protocol that we demonstrated in Section 4.3 for accounts with one more function which we call `get_state`. `get_state`(pk, h) takes a public key pk and a block height h and returns the state $\langle \text{balance}, \text{nonce} \rangle$[3], along with an authenticated map proof against the state trie found in the block header at height h.

[3] For simplification we only mind the state for Externally Owned Accounts and we completely disregard smart contracts in our accounts model.

The operation of the wallet proceeds as follows. The wallet first completes a headers-based synchronisation obtaining the honestly adopted chain. Servers which did not provide the honestly adopted chain are disregarded for the next steps. It then picks a server at random and asks for all transactions using `get_history`. After verifying all the Merkle proofs for all transactions, it requests the state and a proof for the chain tip by invoking `get_state`$(pk, |\mathcal{C}| - 1)$. It verifies this proof against the block header of its local chain tip.

It then is able to reconcile the history obtained with the confirmed state of pk in order to ensure that no transactions have been omitted. First, the number of debits should be equal to the nonce found in the state, and secondly the balance should be equal to the sum of all credits minus the sum of all debits and their corresponding transaction fees. If any of these checks fail, the server lied about the history and another server is chosen for the process to repeat, until a server finally says the truth. By our standard assumption of at least one honest server this is bound to happen.

6 Privacy Wallets

In this section we will discuss wallets for ZCash [74,27] and Monero [47], two of the most prominent private cryptocurrencies. ZCash is based on the Zerocoin protocol [45] which is in the UTXO model. Monero is based on the Cryptonote [67] protocol, also in the UTXO model [2]. To our knowlege no private cryptocurrencies exist in practice with wallets, so naturally this section will only focus on wallets for private cryptocurrencies in the UTXO model.

In private cryptocurrencies transaction data such as sender, receiver and amounts are not publicly disclosed as is the case with their transparent counterparts. Instead, only the sender and receiver of the transaction can know this information. Additionally we have two kinds of keypairs: the *viewing key* and the *spending key*. An address is comprised of the public viewing and spending keys. The private information of a transaction can only be obtained by use of the private viewing key. In order to spend funds, the private spending key must be used.

We first examine the requirements for a ZCash wallet to be functional. To create a new transaction sending funds to some party, the wallet needs to know the unspent outputs belonging to the user. From these outputs the amount and some auxillary values must be discovered, in order to allow the wallet to create a nullifier for the commitment and spend it. In order to create a valid transaction with a valid Zero Knowledge proof the wallet also needs to know the full contents of the commitment tree, to prove that they are spending a commitment included in that tree.

6.1 Full Node

The full node wallet works in a straightforward manner. All blocks and transactions are assumed to be already downloaded and verified locally. Upon being

presented with a seed, the wallet evaluates every transaction from the genesis up to the tip for relevance. Outputs to the shielded address are decryptable by the secret key corresponding to that shielded address.

When evaluating an output, it may be directed to a shielded address of the user. In that case we store the commitment and private values of the output. When evaluating an input, it may be that the user has spent a previous commentment, in which case we mark the commitment as spent. Additionally for every input and output we store the transaction to display to the user.

Finally, we end up with a set of unspent commitments along with their private values. Given a transaction description the wallet can then spend any of those unspent commitments.

6.2 SPV

SPV nodes for private cryptocurrencies don't exist in practice due to the detection problem. Private cryptocurrencies are designed to make it hard to detect receivers of transactions. Thus a helpful server who has the chain that is able to detect transactions on account of a user is orthogonal to the design of the cryptocurrency.

A hypothetical SPV wallet however, could avoid performing verification of the chain contents (the transactions) and only perform verification of the header-chain. While this does not reduce the bandwidth requirements of an SPV wallet, it could significantly reduce the computation time for syncing. This is especially important in the case of ZCash where verifying the validity of a zero-knowledge proof of a transaction may take time in the order of milliseconds.

Compact Blocks. In ZIP-307 an optimization for bandwidth is proposed [63]. The optimization lies in noticing that since under the assumption of SPV security transactions don't need to be verified, transactions only need to hold the minimum amount of data such that they are detectable. By only keeping this data, transactions are compressed. Block headers remain unchanged. The new block structure including these compressed transactions is dubbed a compact block.

This is a backwards-compatible change to the ZCash network that does not require any consensus changes. Blocks are adapted to this compact block format by any interested server and are subsequently relayed to any clients that request them.

If a client detects an inbound or outgoing payment of interest, it can then request the full transaction from the server.

This compression of the transactions is lossy and means that the transactions inclusion in the claimed block header can't be verified via a Merkle proof in the same way it can be verified on transparent SPV. This is because the hash of the original transaction cannot be reproduced unless the full transaction is owned, which defeats the purpose. However, a similar verification can be performed as follows. Every shielded output references a new commitment. ZCash in its block header includes a Merkle Tree root of the existing commitment set after adding

all commitments the transactions of the block itself create. Thus even though transactions themselves cannot be verified as valid, shielded outputs can by verifying a Merkle inclusion proof of inclusion of the commitment in the Merkle Tree root of commitments. For the inputs no processing needs to be done. If an input reveals a note belonging to the user then this note can be marked as spent as the only person who would be able to reveal the note is the user itself and it can't be faked by any server. If an input reveals a note belonging to someone else the server has nothing to gain by tricking the wallet into thinking it is revealed in a block where it is not. Cases of double-spending by an adversarial server are completely thwarted by the output verification.

We remark that this is not an asymptotic improvement over the theoretical SPV solution. We posit that an asymptotic improvement over the theoretical SPV solution which preserves privacy (i.e. no private keys are revealed to the remote peer) cannot exist.

In practice. ZecWallet Lite [1] is a desktop wallet with a mobile companion app that utilizes the Compact Blocks proposal. On the backend it uses lightwalletd [75], which offers transforms full blocks to the compact format.

6.3 MyMonero

A final solution for private cryptocurrencies observed in practice comes from Monero. MyMonero [59] is a desktop and mobile wallet that operates with the help of a trusted server. The viewing private key is disclosed to the server so that it can detect transactions on the wallet's behalf. Any detected transactions are relayed to the wallet in order for the wallet to display and for the wallet to hold an up-to-date UTXO for the purposes of creating new transactions.

MyMonero works differently when the user creates a new seed and when the user recovers their wallet with an existing seed. When creating a new seed, the viewing private key is disclosed but because the server knows the key was just generated they can be certain that it has never received any transactions.

In the case of recovering from an existing seed there are two cases. Either the server already knows the corresponding viewing private key, in which case it can directly relay the transactions it knows, or the key is presented to the server for the first time. If the latter case holds, the server will take on the difficult task of looking through each transaction from the genesis up to the tip in order to detect transactions sent to the user. However, because of the computational difficulty of the task, it will request that the user pledges to send back a small reward for the server in the form of Monero before starting.

The server is a full-node that maintains the whole chain and its tip. Only when a client requests their transactions it scans the blocks starting from the last processed block for the requested viewing key up until the tip.

7 Superlight Wallets

We have seen that SPV allows clients to verify the best chain by only downloading all block headers while maintaining security. A body of research, starting with [46,19], tried to identify ways of adopting the best chain by downloading less than all block headers. We introduce the notion of a superlight client to more formally capture such solutions.

Definition 4 (Superlight client). *A* superlight client *is an ITM that after communicating with a set of parties P of which at least one is honest, can determine by using $o(n)$ of communication:*

1. *The tip of the honest chain.*
2. *The inclusion of some block in the honest chain.*

Non-Interactive Proofs of Proof-of-Work [30,31,7] (NIPoPoWs) were introduced that solve this problem, requiring clients to download only $\Theta(\text{polylog}\, n)$ block headers while maintaining security—an exponential improvement.

For our superlight client definition a parallel can drawn to the verifier in [31]. The verifier is non-interactive whereas we also allow interactive protocols for a superlight client. In NIPoPoWs a proof for the tip of the chain is called a suffix proof and a proof for the inclusion of some block in a chain is called an infix proof. A final difference is that in NIPoPoWs the proof size is fixed as $O(\text{polylog}\, n)$ whereas our definition is more general, allowing any sublinear proof size.

The leading directions for NIPoPoWs are the following. Superblock NIPoPoWs [31,28] are based on *superblocks*, blocks which have achieved much more proof-of-work than their nominal target. FlyClient NIPoPoWs [7] rely on random sampling of blocks.

A lot of attention has been given to superlight clients as a means to implement 1-way [29] and 2-way pegs [3,32]—ways to trustlessly transfer funds from one cryptocurrency to another.

Little attention has been paid until now to wallet constructions that make use of superlight clients. To this end we provide a first definition for wallets which are also superlight.

Definition 5 (Superlight wallet). *A wallet is called* superlight *if it can be usable in all 3 aspects (transaction description funding, balance and transaction history) by having a communication complexity of $o(n)$ (assuming a constant number of relevant transactions).*

7.1 An ideal construction

Suppose a hypothetical cryptocurrency where we can specify the format of the block header. We include a commitment to a map of the form $address \rightarrow [tx_1, \ldots, tx_k]$.

For the commitment any key-value commitment structure can be used such as Sparse Merkle Trees [11,36] or Merkle Patricia Tries which are already used for header commitments in Ethereum [70]. We require that the commitments are verified as part of the consensus protocol and that a block is rejected by honest full nodes if its block header commitment is bogus.

A superlight wallet can work as follows. It first fires up a superlight client according to some superlight client protocol and requests the blockchain tip. It then extracts the commitment from the block header. It subsequently queries a helpful node for the value of the map for an address of interest along with a proof that the value actually belongs to the committed map. Finally it verifies the proof and if successful can use this information to operate.

Performance. This construction succeeds in syncing by only downloading $o(n)$ block headers and $\Theta(y)$ transactions and proofs.

Theorem 2 (Security). *The ideal superlight wallet construction is secure with regards to a secure superlight client protocol.*

Proof (Proof Sketch). If the tip is chosen to k-buried then it belongs to the honest chain except with negligible probability in the security parameter. Thus the commitments obtained are valid, as honest nodes reject blocks with invalid header commitments. Finally the data obtained from the helpful node for the key of interest need to be consistent with the commitment due to the security of the authenticated map protocol.

Theorem 3 (Uncensorability). *An adversarial server cannot hide transactions from the superlight wallet.*

Proof. Assume an adversarial server that succeeds in hiding transactions from the superlight wallet. This means they produced for some y, map M and key k where $y \neq M[k]$ a proof π such that $\mathsf{Ver}_{\mathsf{auth\text{-}map}}(\mathsf{Commit}_{\mathsf{auth\text{-}map}}(M), \pi, k, y) = \mathsf{true}$, which directly conflicts with the security of the authenticatd map protocol.

Deployment paths. Although such helpful commitments have been proposed to aid in building superlight and secure wallets since as early as 2012 [56], such commitments do not exist in either Bitcoin or Ethereum today. We remark that such commitments could be incorporated as a non-contentious soft-fork.

7.2 Private cryptocurrencies

We now turn our attention to building superlight clients for private cryptocurrencies. ZCash recently announced the upcoming Heartwood hard-fork that will include FlyClient support [9] by means of adding a Merkle Mountain Range commitment to its block headers [35]. The addition is warranted as, according to the company in [9] "Flyclient enables interoperability efforts, cross-chain integration and superlight-client use cases".

It is interesting to note however that even though superlight clients are a welcome improvement, no straightforward light wallet solution has been proposed for a light client of a private cryptocurrency. For a potential solution, there exist two possible paths.

1. **Revealing the private viewing key.** The light wallet may work by sharing its private viewing key with a server. The server then needs to detect relevant transactions for the user. Unfortunately detecting transactions requires time linear to the number of transactions in the network. Each transaction is returned to the wallet, accompanied with a proof of inclusion in the wallet's adopted chain. The proof of inclusion of a transaction is comprised of:
   (a) the header of the block the transaction is included in
   (b) a Merkle inclusion proof against the Merkle root in that header
   (c) a Merkle Mountain Range inclusion proof for that block header against the MMR root in the tip header.

   This solution requires $O(m)$ computation for the server, and offers marginally better security guarantees than the solution of MyMonero from Section 6.3. Specifically, whereas in MyMonero the wallet is unaware of the honestly adopted chain and does not perform any verification of transations, here the it is guaranteed that the transactions the wallet receives are indeed included in the honestly adopted chain. This removes the advantage of a server to perform a double-spend attack against a vendor using the wallet.
2. **Not revealing the private viewing key.** The light wallet does not share any information with the server. We posit that in this case the wallet must receive all transactions, for if the server could know what transactions may be more relevant, identifying information would need to be revealed about these transactions. The so-called *transaction detection* [63] remains an open problem for private cryptocurrencies.

8 Comparison

We now provide a comparison between all the wallets presented. The notation used can be found in Table 3.

Our performance comparison is asymptotic in terms of the aforementioned variables and is broken down on the following aspects.

- **Communication**: The communication complexity between the wallet and server necessary to fully synchronise from scratch.
- **Client computation**: Any computation it is necessary for the wallet to perform in order to finish synchronisation and enter a usable state (for funding transaction descriptions, balance and transaction history).
- **Server lookups**: Computation the server needs to decide what data it needs to relay to the wallet. In the case of SPV this can be bloom filter checks, in the case of private wallets it can be trial-decryption with a viewing key.

Our security comparison focuses on the following aspects.

Table 2. A comparison of the cryptocurrency wallets in practice. Proposals with an asterisk* appear for the first time in this work. Server lookups only refers to the computation the server is required to do to determine what to serve to a wallet during the synchronisation process and does not include previous or continuous computation like maintaining an address index. Partial satisfaction of a criterion is denoted with ◐ and full satisfaction is denoted with ●.

						Server lookups	Open participation	Privacy	Chain check	Uncensorability
			Communication	Client computation						
Proposal	**TX Model**	**Privacy Model**	**Performance** $\Theta(\cdot)$			**Security**				
Full Node	UTXO	Transparent	$n+m$	$n+m$	1	●	●	●	●	
SPV	UTXO	Transparent	$n+y\lg\alpha$	$n+y\lg\alpha$	m	●	◐	●		
Electrum	UTXO	Transparent	$n+y\lg\alpha$	$n+y\lg\alpha$	1			●		
Neutrino	UTXO	Transparent	$n+y\alpha$	$n+y\alpha$	1	●	◐	●	●	
Explorer-based	Both	Transparent	y	1	1					
Full Node	Account	Transparent	$n+m$	$n+m$	1	●	●	●	●	
Electrum-like*	Account	Transparent	$n+y\lg\alpha+\lg m$	$n+y\lg\alpha+\lg m$	1			●	●	
Full Node	UTXO	Private	$n+m$	$n+m$	1	●	●	●	●	
SPV	UTXO	Private	$n+m$	$n+m$	1	●	●	●	●	
MyMonero	UTXO	Private	y	1	m					
Superlight*	Both	Transparent	$\operatorname{polylog} n+y+\lg m$	$\operatorname{polylog} n+y+\lg m$	1	●		●	●	
Superlight*	UTXO	Private	$\operatorname{polylog} n+y\lg\alpha$	$\operatorname{polylog} n+y\lg\alpha$	m	●		●		

Table 3. The notation used throughout our comparison.

n	number of all blocks
m	number of all transactions
y	number of relevant transactions (implicating the wallet user)
α	number of transactions per block

- **Open participation**: Whether participating as a wallet's server is possible for everyone.
- **Privacy**: Whether the wallet reveals information about the user's addresses to any server in order to synchronise.
- **Chain check**: We say that a wallet fulfills chain check whether the wallet will only accept transactions included in the best chain.
- **Uncensorability**: Whether the server is not able to lie by omitting transactions from the wallet.

A comparison of the wallets in tabular form is presented in Table 2.

8.1 Communication

The full node requires the most amount of communication, $n+m$ as all full blocks must be downloaded. This is the case for both UTXO and account models.

SPV implementations for transparent cryptocurrencies require $n+y \lg \alpha$ communication, which is broken down as (a) each block header (b) each relevant transaction and (c) a Merkle inclusion proof for each transaction. In the worst case each transaction belongs to a different block and no re-use of parts of the proof can take place thus we need y proofs of size $\lg \alpha$ each.

Electrum-based implementations for UTXO have the same communication complexity. In Neutrino, all block headers need to be downloaded as well. In addition, for every block including a relevant transaction it also needs to be downloaded in full. In the worst case each relevant transaction will be included in a different block yielding a communication complexity of $n + y\alpha$.

Our Electrum based construction for accounts requires $n + y \lg a + \lg m$ communication, with the $\lg m$ representating the state trie proof size. We assume that the state trie grows as $\Theta(m)$.

Explorer-based wallets and MyMonero are all querying a trusted-third party and obtain the relevant transactions directly, thus have a communication complexity of y.

SPV wallets for private cryptocurrencies that do not reveal the private viewing key to a server require that all full blocks be downloaded, resulting in $n + m$ communication complexity.

Our superlight wallets based on some NIPoPoW superlight client have a communication complexity of $\text{polylog}\, n + y + \lg m$, and our private superlight wallets have communication $\text{polylog}\, n + y \lg \alpha$.

8.2 Client computation

Full nodes need to very all blocks and transactions in history resulting in $n + m$ computation.

In SPV wallets, for each block header verification takes place and for each relevant transaction verification of its attached Merkle proof takes place, thus client computation is also $n + y \lg \alpha$. The same holds for Electrum-based wallets.

Our Electrum based construction for accounts requires $n + y \lg a + \lg m$ computation.

For Neutrino the client computation is $n + y\alpha$, as every header is verified and the full block including each transaction is also verified.

Explorer-based wallets base their security on a trusted third party and perform no verification of what they obtain from the server, thus yielding a complexity of 1. The same is true for privacy wallet MyMonero.

Our private superlight wallets based on NIPoPoWs verify NIPoPoW infix proofs and for each relevant transaction its inclusion in a block of the proof, resulting in a client computation of $\text{polylog}\, n + y \lg \alpha$. Our transparent superlight wallets only verify a single Merkle inclusion proof after syncing through the superlight client, resulting in $\text{polylog}\, n + y + \lg m$ computation.

8.3 Server lookups

A full node server (its peer) does not need to perform any lookup to evaluate what it needs to send to the client, yielding lookups of order 1.

BIP-37 SPV's weak point is in server lookups, as the server looks though every single transaction in history to evaluate if it matches the provided bloom filter or not, thus it has the worst server computation complexity of all proposals, $n + m$.

An Electrum server, simply relays all block headers, and because it maintains an index of addresses to transactions where it can just look up the client's advertised addresses, resulting in a constant number of lookups.

The strong point of Neutrino is server lookups compared to BIP-37 that it is designed to replace. The server simply acts as a relay of information so the number of lookups is constant.

Explorer-based wallet servers (i.e. explorers) also hold an index of addresses to their corresponding transactions as Electrum servers do. Similarly to electrum, server lookups are constant.

8.4 Open Participation

The full node network is P2P and fully open to participation. Additionally full nodes usually act as servers for BIP-37 SPV wallets, making them to open to participation.

Server addresses are hardcoded in the Electrum software so participation is limited.

Neutrino is planned to be integrated in the full node P2P protocol which makes participation open.

For explorer-based wallets, the server address is hardcoded in the software, this is not a P2P protocol and participation is not open.

8.5 Privacy

A full node has the best privacy as everything is downloaded thus no network peer can distinguish which transactions or blocks are of interest. The address of the wallet is never revealed to the network.

SPV wallets reveal a lot of information to the servers, even though the addresses are not directly revealed as shown in [22].

For private cryptocurrencies, SPV wallets that download full blocks but don't verify the transaction validity while relying on SPV security are fully private. However solutions like compact blocks which provide a constant factor optimization need to be treated carefully. The reason is that after the wallet detects the compressed transactions that are relevant to its secret key, it may request to get the full transactions for some reason (for example to examine the transaction's memo field in Zcash). This would lead to privacy loss. To mitigate this the suggestion in [63] is to ask for all transactions in the block the transaction is included in instead of the single transaction.

In Electrum, and explorer-based wallets addresses are directly provided to the servers for efficiency.

For Neutrino, addresses are not leaked directly, however full blocks are only received when they contain transactions of interest which could yield some information about the identity of the client to an adversary. In [51] the mitigation proposed is that full blocks be queried from servers at random. However in the face of a Sybil attack [13] the same loss of privacy ensues.

In MyMonero and our superlight client solution for private cryptocurrencies the private viewing key is directly provided to the server, leading to complete privacy loss.

8.6 Chain check

Motivation. If the wallet does not verify the transactions it receives, we show an easy attack that can be performed against its user. Assume the user of the wallet is a vendor that provides an item for some fee in Bitcoin. A transaction that looks like it directs funds to the vendor can very easily faked. A full node would not accept that transaction, as it spends inputs that are already spent or never existed. However, if the wallet performs no check that the transactions received are in the best chain, it cannot know that the transaction is fake. The attack works as follows. The server, who now is a customer of the vendor, presents the vendor's wallet with a fake transaction. The wallet shows the transaction but performs no check and considers it valid. The vendor is satisfied enough and ships the item, for which the customer never actually paid.

The full node by definition verifies that any transaction used is contained in the best chain.

BIP-37 SPV wallets and Electrum perform verification of Merkle proofs of inclusion against the longest header-chain provided. By SPV security we know that chain is actually the best chain a honest full node would adopt, so transactions are actually verified against inclusion in the best chain.

Neutrino also maintains the best chain due to SPV security. For blocks that contain transactions of interest the whole block is downloaded and verified against some header in the best header chain.

Explorer-based wallets do not have any notion of a chain locally and trust the transactions provided. Thus the server could provided transactions that are not part of the best chain and the wallet would have no way of knowing.

Our superlight wallet constructions synchronise the best chain due to NIPoPoW security. All transactions provided are checked against blocks in that chain.

8.7 Uncensorability

A full node cannot be censored as it obtains all transactions and holds the full blocks.

Unfortunately for most other solutions: SPV, Electrum (UTXO-based), explorer-based, MyMonero the server can very easily not include transactions and the wallet has no way of knowing.

For Neutrino however, as long as the filters are correct, transactions cannot be censored. If a filter matches the whole block is downloaded and verified.

For our Electrum construction in the account model, a transaction-omitting server can be easily identified.

Finally, for our ideal superlight wallet construction as proven in Theorem 3 no censorship of transactions can take place. For our superlight wallet construction for private cryptocurrencies no such commitment exists to prevent censorship and the server may lie by omitting transactions with no way to be detected.

Acknowledgements

The authors wish to thank Dionysis Zindros, Pyrros Chaidos and Andrianna Polydouri who read early versions of this paper and provided helpful suggestions.

References

1. adityapk00: Zecwallet-lite is z-addr first lightwallet for zcash (2019), `https://github.com/adityapk00/zecwallet-lite`
2. Alonso, K.M.: Zero to monero (2018), `https://www.getmonero.org/library/Zero-to-Monero-1-0-0.pdf`
3. Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., Wuille, P.: Enabling blockchain innovations with pegged sidechains (2014), `https://blockstream.com/sidechains.pdf`
4. Bisq developers: Bisq - A decentralized bitcoin exchange network (2017), `https://bisq.network/`
5. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In: 2015 IEEE Symposium on Security and Privacy. pp. 104–121. IEEE (2015)
6. BRD developers: Brd - bitcoin wallet (2013), `https://brd.com/`

7. Bünz, B., Kiffer, L., Luu, L., Zamani, M.: Flyclient: Super-light clients for cryptocurrencies. (2020)
8. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
9. Company, E.C.: Introducing heartwood (2020), https://electriccoin.co/blog/introducing-heartwood/
10. Corallo, M.: [bitcoin-dev] Bitcoin Core to disable Bloom-based Filtering by default (2019), https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2019-July/017145.html
11. Dahlberg, R., Pulls, T., Peeters, R.: Efficient sparse merkle trees. In: Nordic Conference on Secure IT Systems. pp. 199–215. Springer (2016)
12. Dogecoin developers: Dogecoin (2013), https://dogecoin.com/
13. Douceur, J.R.: The sybil attack. In: International Workshop on Peer-to-Peer Systems. pp. 251–260. Springer (2002)
14. Dryja, T.: uspv - micro-spv library (2016), https://github.com/mit-dci/lit/tree/master/uspv
15. Ergo developers: Ergo: A resilient platform for contractual money (2019), https://ergoplatform.org/docs/whitepaper.pdf
16. Ethereum developers: Ethereum glossary (2014), https://github.com/ethereum/wiki/wiki/Glossary
17. Etherscan developers: Etherscan (2016), https://etherscan.io/
18. Exodus developers: Exodus (2016), https://www.exodus.io/
19. Friedenbach, M.: Compact SPV proofs via block header commitments. Available at: https://sourceforge.net/p/bitcoin/mailman/message/32111357/ (2014), https://sourceforge.net/p/bitcoin/mailman/message/32111357/
20. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. Annual International Conference on the Theory and Applications of Cryptographic Techniques pp. 281–310 (2015)
21. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol with chains of variable difficulty. In: Annual International Cryptology Conference. pp. 291–323. Springer (2017)
22. Gervais, A., Capkun, S., Karame, G.O., Gruber, D.: On the privacy provisions of bloom filters in lightweight bitcoin clients. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 326–335 (2014)
23. Golomb, S.: Run-length encodings (corresp.). IEEE transactions on information theory **12**(3), 399–401 (1966)
24. Hearn, M., Corallo, M.: BIP: 37 — Connection Bloom filtering (2012), https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki
25. Hearn, M., et al.: bitcoinj - A library for working with Bitcoin (2013), https://bitcoinj.github.io/
26. Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on bitcoin's peer-to-peer network. In: USENIX Security Symposium. pp. 129–144 (2015)
27. Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash protocol specification (2016), https://raw.githubusercontent.com/zcash/zips/master/protocol/protocol.pdf
28. Karantias, K., Kiayias, A., Zindros, D.: Compact storage of superblocks for nipopow applications. In: The 1st International Conference on Mathematical Research for Blockchain Economy. Springer Nature (2019)
29. Karantias, K., Kiayias, A., Zindros, D.: Proof-of-burn. In: International Conference on Financial Cryptography and Data Security (2019)

30. Kiayias, A., Lamprou, N., Stouka, A.P.: Proofs of proofs of work with sublinear complexity. In: International Conference on Financial Cryptography and Data Security. pp. 61–78. Springer (2016)
31. Kiayias, A., Miller, A., Zindros, D.: Non-Interactive Proofs of Proof-of-Work. In: International Conference on Financial Cryptography and Data Security. Springer (2020)
32. Kiayias, A., Zindros, D.: Proof-of-work sidechains. In: International Conference on Financial Cryptography and Data Security. Springer, Springer (2019)
33. Labs, L.: Lightning: Fast bitcoin wallet (2019), https://play.google.com/store/apps/details?id=engineering.lightning.LightningMainnet
34. Labs, L.: Privacy-preserving bitcoin light client (2019), https://github.com/lightninglabs/neutrino
35. Lai, Y.T., Prestwich, J., Konstantopoulos, G.: [zip 221] flyclient - consensus-layer changes (2020), https://github.com/zcash/zips/blob/master/zip-0221.rst
36. Laurie, B., Kasper, E.: Revocation transparency. Google Research, September p. 33 (2012)
37. Ledger developers: Ledger live : Most trusted & secure crypto wallet (2018), https://www.ledger.com/ledger-live
38. Litecoin developers: Litecoin (2011), https://litecoin.org/
39. Lombrozo, E., Lau, J., Wuille, P.: BIP: 141 - Segregated Witness (Consensus layer) (2015), https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki
40. Maxwell, G.: Deterministic wallets (2011), https://bitcointalk.org/index.php?topic=19137.msg239768
41. Meiklejohn, S.: Top ten obstacles along distributed ledgers path to adoption. IEEE Security & Privacy **16**(4), 13–19 (2018)
42. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of bitcoins: characterizing payments among men with no names. In: Proceedings of the 2013 conference on Internet measurement conference. pp. 127–140 (2013)
43. Meshkov, D.: Coinbarn wallet (2019), https://github.com/coinbarn/coinbarn-extension
44. Metamask developers: Metamask (2016), https://metamask.io/
45. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous distributed e-cash from bitcoin. In: 2013 IEEE Symposium on Security and Privacy. pp. 397–411. IEEE (2013)
46. Miller, A.: The high-value-hash highway. bitcoin forum post (2012)
47. Monero developers: Monero (2014), https://www.getmonero.org/
48. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Available at: https://bitcoin.org/bitcoin.pdf (2008), https://bitcoin.org/bitcoin.pdf
49. OpenBazaar developers: Openbazaar (2014), https://openbazaar.org/
50. Osuntokun, O., Akselrod, A.: BIP: 158 — Compact Block Filters for Light Clients (2017), https://github.com/bitcoin/bips/blob/master/bip-0158.mediawiki
51. Osuntokun, O., Akselrod, A., Posen, J.: BIP: 157 — Client Side Block Filtering (2017), https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki
52. Pacia, C.: Bitcoin cash p2p spv wallet (2018), https://github.com/cpacia/BitcoinCash-Wallet
53. Pacia, C.: Neutrino wallet (2019), https://neutrino.cash/
54. Pacia, C., et al.: BCHD - The full node built for Bitcoin Cash developers (2019), https://bchd.cash/

55. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 643–673. Springer (2017)
56. Reiner, A.: Ultimate blockchain compression w/ trust-free lite nodes (2012 (accessed February 15, 2015)), https://web.archive.org/web/20150214114638/https://bitcointalk.org/index.php?topic=88208.0
57. Rosenfeld, M.: Overview of colored coins. White paper, bitcoil. co. il **41**, 94 (2012)
58. Schildbach, A.: Bitcoin wallet app for your android device (2014), https://github.com/bitcoin-wallet/bitcoin-wallet
59. Shapiro, P.: The js codebase for the mymonero desktop and web wallet apps (2018), https://github.com/mymonero/mymonero-app-js
60. slush: [Stratum] Overlay network protocol over Bitcoin (2011), https://bitcointalk.org/index.php?topic=55842.0
61. Song, J.: Neutrino Presentation from Breaking Bitcoin 2019 (2019), https://diyhpl.us/wiki/transcripts/breaking-bitcoin/2019/neutrino/
62. Stern, J., Pointcheval, D., Malone-Lee, J., Smart, N.P.: Flaws in applying proof methodologies to signature schemes. In: Annual International Cryptology Conference. pp. 93–110. Springer (2002)
63. Tankersley, G., Grigg, J., Green, M.: Light client protocol for payment detection (2018), https://github.com/zcash/zips/blob/996cdd9686eb68a50ad75f13b58ff7c3b9f4ee58/zip-XXX-light-payment-detection.rst
64. team, O.: P2p spv wallet/library in go used in openbazaar 2.0 (2016), https://github.com/OpenBazaar/spvwallet
65. Todd, P.: petertodd/bloom-io-attack (2016), https://github.com/petertodd/bloom-io-attack
66. Trezor developers: Trezor beta wallet (2020), https://beta-wallet.trezor.io/
67. Van Saberhagen, N.: Cryptonote v 2.0 (2013), https://cryptonote.org/whitepaper.pdf
68. Voegtlin, T.: Electrum (2011), https://electrum.org/
69. Voegtlin, T.: Electrum-server for the Electrum client (2012), https://github.com/spesmilo/electrum-server/
70. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)
71. Wuille, P.: BIP: 32 — Hierarchical Deterministic Wallets (2012), https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki
72. Wüst, K., Gervais, A.: Ethereum eclipse attacks. Tech. rep., ETH Zurich (2016)
73. Yoroi developers: Yoroi - light wallet for cardano (2018), https://yoroi-wallet.com/
74. Zcash developers: Zcash (2016), https://z.cash/
75. Zcash developers: Lightwalletd is a backend service that provides a bandwidth-efficient interface to the zcash blockchain (2019), https://github.com/zcash/lightwalletd/
76. Zeyde, R.: Electrum Server in Rust (2018), https://github.com/romanz/electrs