Scorex Tutorial

Amitabh Saxena

June-August, 2016

Chapter 1

Executive Summary

This paper describes the Scorex project and how it can be used to create blockchain protocols such as cryptocurrencies (Bitcoin). Scorex is a library written in Scala with loosely coupled components that can be used as the underlying framework for making applications using a blockchain (a type of decentralized consensus-based protocol). The intended audience is developers wanting to create or experiment with such applications. Some basic knowledge of cryptography, data structures and cryptocurrencies is required. Some programming background is also required to understand the code-snippets. For good explanation of cryptography primitives and protocols please refer to the foundational book of [4].

In order to understand Scorex, it is helpful to consider Bitcoin, Namecoin and Nxt as three distinct applications of blockchain. Scorex gives the underlying framework for developing any of the three apps (and several others) by writing a thin layer of code.

1.1 Organization

This document is organized as follows. In Chapter **??**, we describe basic building blocks of a blockchain system. In Chapter **??**, we describe how the blocks are implemented in the Scorex framework. We provide code snippets in Scala language. No prior knowledge of the Scala language is required.

Chapter 2

Introduction

If you have heard of blockchain then you have indirectly heard of Bitcoin, the decentralized peer-to-peer currency network with some fancy features such as the ability to transfer real value over virtual channels. This is because Bitcoin is the first widespread use of the blockchain, which is essentially a *decentralized tamper-resistant append-only database* of transactions (sometimes also referred to as a *ledger*). Let us elaborate on this in more detail in the following sections.

2.1 Blockchain and Boxes

Define F to be the set of mappings $\mathbb{Z} \times \mathbb{Z}^+ \mapsto \{-1, 0, 1\}$ over the integers. Each pair represents a 'box' of type ((public-key, coinID), amount)[1].

Each element from F maps the pairs of such integers (i.e., all boxes) to one of three states: $-1, 0, 1$, indicating that the corresponding box is respectively *Unused* (never used), *Closed* (contains funds) and *Opened* (funds have been used). A blockchain is an infinite sequence $f_0, f_1, f_2, \ldots \in F$ of functions from F applied iteratively on $\mathbb{Z}$ such that:

1. $f_i \neq f_{i+1}$ (there must be at least one state change in the boxes).
2. $\forall (x, y) : f_i(x, y) \leq f_{i+1}(x, y)$ (state cannot decrease).
3. $\forall (x, y) : f_i(x, y) \cdot f_{i+1}(x, y) = 0$ (state cannot increase > 1).

f_0 is called the *genesis mapping* or *genesis state*. Each mapping f_i represents a snapshot of the system at iteration i, which indicates the state of each box. The set $m_i = \{(x, y) | f_i(x, y) = 0\}$ is called the *minimal state* at iteration i.

A *block* b_i gives the deltas between two successive states (f_{i+1}, f_i) and is essentially defines a set of boxes to open and close. Formally, b_i is the set $\{(x, y) | f_{i+1}(x, y) \neq f_i(x, y)\}$. The transformation rules ensure that there is

[1] Each integer in the left of $\mathbb{Z} \times \mathbb{Z}^+$ is assumed to map to a unique pair of type (public-key, coinID), where coinID is some unique identifier for the coins. For now, it is sufficient to consider the left integer as some 'ownership identifier' of the amount on the right.

a unique block for each iteration. At any iteration i, a client needs only the current minimal state m_i and the next block b_i to compute the next minimal state. While this is enough to define a generic blockchain, Bitcoin has some additional rules:

1. For any block b_i,

$$\sum_{(x,y)\in b_i} y \cdot f_{i+1}(x,y) + \sum_{(x,y)\in b_i} y \cdot f_i(x,y) \leq r(i),$$

   for some *reward* function r.

   Note that $\sum_{(x,y)\in b_i} y \cdot f_i(x,y)$ is the negative of the sum of amount in all the boxes to be closed in this iteration and $\sum_{(x,y)\in b_i} y \cdot f_{i+1}(x,y)$ is the sum for the boxes to be opened. Thus, the LHS is the amount of bitcoins 'created' in this block minus the amount bitcoins 'destroyed'. The inequality says that the net amount of bitcoins created in a block cannot be more than the reward.

2. For each box $(x,y) \in b_i$, there is a *transaction* that acts as the *witness* of opening or closing that box. A transaction acts like a proof that a particular state change was authorized via some public key, which will be discussed in detail in the next chapter.

At any iteration i of the blockchain, older snapshots $(f_0, f_1, f_2, \ldots)$ are considered tamper-resistant while fresh copies $(\ldots, f_{i-2}, f_{i-1}, f_i)$ may be susceptible to deletion or tampering, depending on the computational power of the attacker. However, after the snapshot becomes sufficiently old, it can be considered as tamper-proof for all practical purposes. The rules of the protocol ensure that if all honest nodes start with the same initial state f_0 and apply the same blocks successively then they will have the same snapshots f_i at any iteration i. The task here is to ensure that the blocks are always applied in the right order. This is a consensus problem, and is solved in Bitcoin using a concept called *proof-of-work* – the solution to a hard puzzle. The idea is that not everyone gets to choose which block to apply to reach the next valid snapshot, but only by someone who has invested a large amount of computing power and provides the proof on a first-come-first-serve basis. Snapshots are connected – each block is linked to the previous one via a cryptographic hash so that if someone else provides a proof-of-work and the network accepts it, then all existing work for "winning" this block is invalidated and the contenders have to start again with the newly accepted block as the starting point. To incentivize nodes to expend work, each accepted snapshot comes with some reward in the form of tokens generated (additional states that benefit the node solving the puzzle).

The rules encoding state changes are hardwired into the peer-to-peer nodes so that once a majority of them are running the code and act honestly, we can be ensured that block updates follow the correct rules. Blockchain can tolerate a high number of corrupt nodes (those controlled by an attacker) – close to 40%.

Transactions: Transactions can also be seen as state changes authorized via private keys and validated by the corresponding public keys. The transaction can encode information pertaining to funds transfer (as in Bitcoin) or have additional information pertaining to the external world such as attaching a key-value pairs to a public key (as in Namecoin – a decentralized DNS). While Bitcoin seems to be the primary use of blockchain right now, other intriguing use cases such as Namecoin have also been presented. The most powerful of being programmable-blockchains. The idea is simple – while in Bitcoin, the rules encoding the state changes pertain only to spending of funds, nothing prevents us from coming up with more complex rule that can encode some business logic. One such example, already mentioned, is Namecoin, which additionally has rules for reserving doman names and for transferring them. In all cases, the rules will be written in a language that the node understands and accepts (the grammar is hardwired into the node).

The newer applications of blockchain are all based on how these rules can be written and how expressive the language is. For instance, Ethereum, another blockchain based protocol boasts of a Turing complete language in contrast to the tiny one that Bitcoin provides. The following summarizes the main proposals.

1. *Bitcoin:* Decentralized currency.

   Advantages: Decentralization of money, easy to store and spend.

   Challenges: Too much storage, wild price fluctuations, too much energy consumed, limited bandwidth of blocks, not suitable for micro-transactions.

2. *Namecoin:* Decentralized DNS (stores $key \rightarrow value$ mappings).

   Advantages: Decentralization of DNS, difficult to sensor or shut down.

   Challenges: Similar in design to Bitcoin; some similar challenges, low computing power so less attack resistant.

3. Ethereum: Smart contracts.

   Advantages: Enable trustless computing via decentralization.

   Challenges: Turing completeness causes complexity, may have security issues – example DAO attack.

Chapter 3

Overview of Bitcoin

Although the bitcoin protocol is quite complex, only a few basic concepts are necessary to understand the idea. These are: *transaction*, *input*, *output*, *reference*, *block* and *confirmation*. We describe them below. Firstly note that in bitcoin, funds are exchanged between *addresses* which are hashes of public keys[1]

3.1 Transaction

Roughly speaking, a transaction consists of a set of *inputs* (source of funds) and *outputs* (destination of funds).

Example: Suppose Alice is the owner of address A which received x bitcoins in a previous transaction. She wants to send $y \leq x$ bitcoins to Bob's address B. Alice constructs a transaction with A as the input and B as one of the outputs. She also inserts ref, the reference to the previous transaction's output where A received those x bitcoins. The entire amount x must be transferred from A. Alice sends y bitcoins to B, sets a transaction fee t and sends the remaining amount $z = x - y - t$ to her *change address* C, which is the other output. The change address is simply any address owned by Alice (possibly A). The message

$$\text{“}(ref\text{: remove } \text{฿}x \text{ from } A), (\text{put } \text{฿}y \text{ in } B), (\text{put } \text{฿}z \text{ in } C)\text{”}$$

is signed under A.

Notation: We will use the following notation:

- $X \overset{ref}{\rightarrow} x$ is the message “(ref: remove ฿x from X)”. This is an input.
- $X \leftarrow x$ is the message “put ฿x in X”. This is an output.

[1] We use the terms ‘address’ and ‘public key’ interchangeably. The meaning will be clear from the context.

- $\sigma_X(m)$ is signature on message m under public key X.

Alice's transaction is then $(m, \sigma_A(m))$, where $m = (A \overset{ref}{\rightarrow} x, B \leftarrow y, C \leftarrow z)$.

Transactions: The above scenario had a single input. In reality, a bitcoin transaction can have multiple inputs with no particular link between any source-destination pair. The entire transaction is signed under every input public key. The only requirement is that the sum of the funds at the inputs is greater than or equal to the sum of funds at the outputs. Any difference is considered a transaction fee. More formally, define m to be the message

$$M \overset{\text{def}}{=} (A_1 \overset{ref_1}{\rightarrow} x_1, A_2 \overset{ref_2}{\rightarrow} x_2, \ldots, A_n \overset{ref_n}{\rightarrow} x_n, B_1 \leftarrow y_1, B_2 \leftarrow y_2, \ldots, B_l \leftarrow y_l),$$

where: $(A_1, x_1, ref_1), (A_2, x_2, ref_2), \ldots, (A_n, x_n, ref_n)$ are n tuples each consisting of an address A_i, amount of funds x_i and a reference to a previous transaction where A_i received x_i bitcoins, and $(B_1, y_1), (B_2, y_2), \ldots, (B_l, y_l)$ are l pairs of addresses and amount of funds. A valid transaction tx is a tuple:

$$tx \overset{\text{def}}{=} (M, \sigma_{A_1}(M), \sigma_{A_2}(M), \ldots, \sigma_{A_n}(M)) \tag{3.1}$$

such that each signature $\sigma_{A_i}(M)$ verifies correctly and the following holds:

1. $\sum_{i=1}^{l} y_i \leq \sum_{i=1}^{n} x_i$
2. Each ref_i for $1 \leq i \leq n$ was never used in any prior transaction.

The ordering of the signatures in tx is determined from the ordering of messages inside M (which is fixed due to the signatures).

Referencing outputs: In future, when spending the funds from any of the outputs (say $B_i \leftarrow y_i$) of the above transaction, a reference $ref_{B_i \leftarrow y_i}$ to that output needs to be provided. Let tx be the string of Eqn. 3.1. Then

$$ref_{B_i \leftarrow y_i} \overset{\text{def}}{=} (Hash(tx), i)$$

Because ref is constructed from the hash of a previous transaction, it is guaranteed that two different transactions are distinct unless the outputs, input and ref are identical (a forbidden scenario). Due to this, it is also guaranteed (with high probability) that the refs generated by using hashes of two different transactions are also different. In fact, this is how bitcoin prevents double spending (see below). A ref can be used in a transaction at most once. Bitcoin clients maintain a list of unused refs to do this check.

Unspent outputs (and double-spends): An unspent output is essentially an unused reference, one that has never been used in any transaction. The protocol design guarantees that references to two different outputs will be distinct (see above). Each client maintains a set called 'unspent outputs'. Each output of every transaction is added to this set, and removed when is it used as a reference in another transaction. A transaction with a reference not in this list is considered a double spend and is not processed.

3.2 Processing Transactions

A new transaction is valid if all the references are unused. If so, the transaction is accepted as *valid* but *unconfirmed*, and is relayed on the network. The clients add each such transaction to a pool of unconfirmed transactions. Unconfirmed transactions can be double-spent. Here we describe the validation process in more detail. Recall that a transaction is equivalent to

$$tx \stackrel{\text{def}}{=} (M, \sigma_{A_1}(M), \sigma_{A_2}(M), \ldots, \sigma_{A_n}(M)),$$

where M is a message with the following semantics:

$$M \stackrel{\text{def}}{=} (A_1 \overset{ref_1}{\to} x_1, A_2 \overset{ref_2}{\to} x_2, \ldots, A_n \overset{ref_n}{\to} x_n, B_1 \leftarrow y_1, B_2 \leftarrow y_2, \ldots, B_l \leftarrow y_l),$$

each $\sigma_{A_i}(M)$ is a valid signature on M under A_i and the following holds:

1. $\sum_{i=1}^{l} y_i \leq \sum_{i=1}^{n} x_i$.
2. Each ref_i for $1 \leq i \leq n$ was never used in any prior transaction.

In reality, the inputs $A_1 \overset{ref_1}{\to} x_1$ are represented only using ref_i. The values A_i and x_i are obtained from a *UTXO database* that every client must maintain.[2] This database is a key-value store of type $ref_i \to (A_i, x_i)$. Note that in order to validate transactions and participate in the protocol, maintaining this UTXO database is necessary. Once a node has bootstrapped and synced, it need not store the entire blockchain. It can store just the UTXO database (plus a few recent blocks to handle rollbacks), and keep updating this database as new blocks are mined. Thus, even if a node is not storing blocks, it must still parse every new block to update its UTXO database. Also note that a node that does not store the blockchain cannot help other new nodes to bootstrap.

It is helpful to consider each (unused) ref_i above as a "closed box" with x_i inside, and the act of using ref_i in a transaction as "opening the box" and releasing x_i. A box can be opened at most once and the act of sending bitcoins to an address ($B_i \leftarrow y_i$) generates a new closed box with y_i inside.

The semantics of transactions are specified using some encoding and a DSL called *Script* (a stack-based language similar in design to Forth). A node validates transactions as follows.

1. For each input $A_i \overset{ref_i}{\to} x_i$ from $(A_1 \overset{ref_1}{\to} x_1, A_2 \overset{ref_2}{\to} x_2, \ldots, A_n \overset{ref_n}{\to} x_n)$
   (a) Load signature σ_i and public key A_i into stack.
   (b) Using ref_i, find box $A_i \leftarrow x_i$ from the database of unopened boxes. This is the output of a transaction where A_i received x_i bitcoins.

[2] The inputs additionally include the public key corresponding to address A_i. We can assume that this key is part of the signature.

(c) The loaded box contains the amount to be released as well as "unlocking instructions", a sequence of Script operations that verify the above signature on M. This is also loaded on the stack and evaluated. The box is opened if the output of the program is True (non-zero).

2. If all boxes are opened, then we create new boxes $B_i \leftarrow y_i$ as defined by the outputs, provided that the total amount released from boxes is more than or equal to the total amount in the newly created boxes.

The opened and created boxes are not immediately committed to the UTXO database. Rather, every node must wait for the network to "confirm" the changes implied by any given transaction. Transactions are confirmed in bulk such that all nodes quickly reach a consensus on which set of transactions to include in the next database update. To ensure consistency and fast consensus, not everyone gets to choose which transactions to commit but only those nodes who have a large amount of computing resources (called "miners").

3.3 Confirming Transactions

Miners are nodes that propose a set of transactions to commit along with a proof that they have put in a certain minimum amount of work (in the form of CPU cycles) after the last update. The network selects the first solution.

Roughly, the process of confirming transactions is as follows:

1. A bunch of unconfirmed transactions along with one reward transaction (known as the *coinbase transaction*) are combined into a 'block'.
2. Hash of the previous block h_{pr} is added to the block.
3. A nonce is added to the block.
4. Hash(b) of the final block b is computed.

If the output of the hash contains at least a specified number of leading zeros, the puzzle is solved, otherwise the miner tries with different nonces until the puzzle is solved or some other miner broadcasts the solution of a puzzle for a block referencing h_{pr}. A correct solution implies that the corresponding block is 'mined' and all transactions contained in it are confirmed.

Confirmations: The number of confirmations of a transaction are the number of blocks in the blockchain that have been accepted by the network since the block that includes the transaction. The possibility of double-spending a transaction decreases exponentially with the number of confirmations. The default client requires 6 confirmations for normal transactions and 100 confirmations for reward transactions before they can be spent.

Transaction pool management: Each client maintains a pool of unverified (but valid) transactions. An element is removed from this pool when that transaction gets included in a mined block. This ensures that even if a transaction is not included in an immediate block, it is kept in the pool until it gets mined. If a transaction is not confirmed within 72 hours then it forgotten.

3.4 Block Structure

A block consists of a variable-size *payload* containing the actual transactions in a Merkle tree structure and a fixed-size *header* describing the payload. The header is 80 bytes and contains:

1. The root hash of the Merkle tree of transactions.
2. The current block index and the previous block-header hash.
3. The nonce, the corresponding difficulty target and a timestamp.

Earlier we stated (for simplicity) that the PoW is computed as a hash of the entire block. However, this is not true. The PoW is computed only on the header and not the payload. This enables nodes to verify PoW using just header information, while the payload can be verified later via the root hash.

Example of header: The following is an example of a block header[3]:
`02000000b6ff0b1b1680a2862a30ca44d346d9e8910d334beb48ca0c00000000`
`000000009d10aa52ee949386ca9385695f04ede270dda20810decd12bc9b048a`
`aab314724d95a5430c31b18fe9f0864`

The different sections are identified by alternating underlines:

1. Block version: `02000000` (decodes to 2).
2. Hash of previous block's header: `b6ff0b1 ... 48ca0c0000000000000000`.
3. Merkle root hash of payload: `9d10aa52 ... decd12bc9b048aaab31471`.
4. Unix timestamp: `24d95a54` (decodes to 1415239972).
5. Difficulty target: `30c31b18` (decodes to $1bc330_{\texttt{hex}} \cdot 256^{18_{\text{hex}}-3}$).
6. Nonce: `fe9f0864`.

Computing the Merkle root: Merkle tree is a technique for authenticating small slices (few transactions) from a large chunk of data (entire payload) without having to authenticate the entire data (however, the entire data can be authenticated if needed). The transactions are first arranged in some order that satisfies the consensus rules given below. Their transaction hashes (TXIDs) are considered as the last row (leaves) of the tree that will be constructed. Starting

[3] https://bitcoin.org/en/developer-reference#block-headers

with the last row, each row is iteratively processed to get the previous (parent) row until the currently processing row has only one node, the Merkle root. If the currently processing row has two or more nodes, we first ensure that there are even number (say n) of them, by repeating the last node if necessary. Then we pair the nodes to form $n/2$ pairs. Each pair (L, R) is concatenated and its hash $\texttt{SHA256}(\texttt{SHA256}(L||R))$ forms the parent for the next iteration. This process is repeated until the root is reached.

Consensus rules: A client rejects block that do not follow the below rules:

1. The coinbase transaction's TXID is always placed first.
2. Any input within this block can spend an output which also appears in this block (assuming the spend is otherwise valid). However, the TXID corresponding to the output must be placed at some point before the TXID corresponding to the input. This ensures that any program parsing block chain transactions linearly will encounter each output before it is used as an input.
3. If a block only has a coinbase transaction, the coinbase TXID is used as the merkle root hash.

Payload: The first field of the payload defines the number of transactions. The rest of the payload contains the raw transactions concatenated in the same orders as in the Merkle tree.

3.5 Security and Privacy

At the heart of Bitcoin is the concept of the blockchain, a distributed global ledger of transactions that each node holds. The goal of the protcol is to ensure *eventual consistency*; if the network is completely synchronized then all nodes will have an identical copy of the blockchain. Thus, the primary goal of Bitcoin is to select the "valid" chain given more than one contenders.

Selecting a unique chain: It is conceivable that an attacker (or network failure) can cause two or more competing chains to be temporarily (a *soft-fork*) or permanently (a *hard-fork*) present in the system. The nodes should be able to quickly reject the "invalid" one. The protocol selects the chain with the highest cumulative difficulty, rather then the longest one.

Difficulty level adjustment: Difficulty is a measure of the hardness of the puzzle and can be quantified by x, the maximum possible number representable with that many leading zeros. A difficulty level of d implies that $d = 2^{256}/x$. Thus, the smaller x is, the larger the difficulty. The difficulty is adjusted every 2016 blocks based on the time it took to find the previous 2016 blocks. At the desired rate of one block each 10 minutes, 2016 blocks would take exactly two

weeks to find. If the previous 2016 blocks took more than two weeks to find, the difficulty is reduced. If they took less than two weeks, the difficulty is increased. The change in difficulty is in proportion to the amount of time over or under two weeks the previous 2016 blocks took to find.

Storage Requirements: As discussed earlier, at the minimum each node must store the unopened boxes in a UTXO database to ensure that the spender actually holds those bitcoins. Additionally, each node may store the entire blockchain starting from the genesis block to help other clients to bootstrap.

What can be pruned? If all the closed boxes generated in a particular block have been opened, that block is not needed for validation (we need blocks to validate/proof that our UTXO database is valid). Thus we can only store the headers for those blocks.[4]

Security Requirements: For security, we require the following: (1) The inability of an attacker to send bitcoins from addresses whose private key is not known, (2) The inability of an attacker to double-spend bitcoins or reverse a transaction, and (3) The inability of an attacker to prevent some valid transactions from confirming. The first requirement is satisfied if the underlying signature scheme is existentially unforgeable. The second and third requirements, formalized respectively as *persistence* and *liveness* in [2], can be achieved if the underlying proof-of-work (PoW) based consensus system satisfies the following two properties [2]:

1. *Common prefix:* If all honest participants remove the top (newer) k blocks from their chains for a large enough k then all of them will share a common prefix. In other words, the chains held by honest miners will either be identical or be contained in the others.

2. *Chain quality:* There is some minimal integer $\lambda \geq 1$ such that, if the combined computing power of honest parties is λ times that of the adversary, then a non-negligible amount of blocks generated by honest parties will make it into the chain.

If the difficulty level is sufficiently high and the network synchronization time (time between a new block being injected and reaching all participants) is short compared to average block generation period (10 minutes in Bitcoin) then the protocol offers high security. On the other hand, security is weakened if any of the following conditions hold [2]: (1) overall computing power is low, (2) blocks are generated too fast, or (3) network takes long time to synchronize.

[4]However, in the existing protocol, we cannot prove to another node that all generated boxes in a pruned block have been used up. A proposal of *block-level aggregation* of transactions could allow that.

Attacks on Implementation: Following are attacks specific to Bitcoin:

1. *Reused R-values:* The underlying signatures ECDSA can be broken if the same randomness is used in two different signatures [**?**]. Thus, implementations must take additional care to use true randomness or message-specific one (computed as a hash of the message).

2. *Centralization of mining:* If a majority of the mining power is concentrated in a few pools, then they can collude and attack bitcoin. Part of the reason for this threat is the susceptibility to ASIC mining [**?**].

3. *Denial of service attacks:* Certain attacks are based on miners forcing other miners to skip block validation by generating large blocks or ones that require expensive verification. Thus, they could send wrong data that will result in other miners to later lose their work.

4. *Malleability:* The signature encoding in Bitcoin is such that if a certain bits of the signature are toggled, the result is still a valid signature (this is due to the underlying ECDSA scheme). This allows miners to mine a transaction whose is different from the original, while keeping everything else (i.e., inputs/outputs) same. If a bitcoin service uses the transaction hash to monitor sent funds, then it could lose funds [**?**].

Privacy: The addresses serve as pseudonyms and provide some anonymity. However, bitcoin does not provide true anonymity because the inputs are linked to the previous outputs via a reference.

Chapter 4

A Blockchain System Design

4.1 Introduction

Consider Bitcoin as an example. Peers are holding money in form of algorithmically issued tokens. They do not trust each other, and do not to seek for a trusted mediator. Instead, they are running a Bitcoin protocol which builds a blockchain, a type of *append-only database* (or log). Older data in the blockchain can be considered *tamper-resistant* because it is protected using a consensus build using *proof-of-work* (a solution to a exponentially hard puzzle that can be verified efficiently, once known). The design of the protocol is such that the puzzle becomes harder as the data gets older. However, freshly added data (i.e., the last few versions of the log) are considered potentially unstable. Double spends are prevented using this database because the entire transfer history of any satoshi can be traced back to the time it was created. Thus, for example, if Alice had sent all her satoshis to Bob, she can't send anything after that and before receiving them from other party.

Using the above idea, we can define a blockchain as a *prefix-immutable append log of non-conflicting authenticated events in a decentralized peer-to-peer network*. Let us now elaborate on what this means.

Simply said, there are peers do not trust each other. There is no any trusted party, only a protocol peers need to follow (being effectively thrown away from the network otherwise). Peers are issuing authenticated (signed) events of some semantics. For example, they are sending out signed payments. Or they are registering $name \to value$ correspondences in a shared database (certificates, domains). "Prefix-immutable append log" means all the peers following the protocol are agree on immutable prefix of events append log. That is, if we cut a suffix of some length from the log a peer holds, for each peer, same-size prefixes will be the same, with overwhelming probability. Events in the ordered prefix-immutable log must be non-conflicting in order to have flawless history.

Consider Bitcoin as an example. Peers are holding money in form of algorithmically issued tokens. They do not trust each other, and do not to seek for

a trusted mediator. Instead, they are running a Bitcoin protocol which builds prefix-immutable append log containing token transfers. Last few versions of the log are considered potentially unstable, but before them the history is considered as irreversible. The payments history is flawless, so, for example, if Alice had sent all her tokens to Bob, she can't send anything after that and before receiving tokens from other party.

4.1.1 Security of Bitcoin

Like any multiparty protocol, Bitcoin needs *correctness* ('valid transactions' should go through) and *soundness* ('invalid transactions' must be blocked). Correctness is defined in terms of passive adversaries, who behave according to protocol and do not attempt deviate.

4.2 Cryptography

At the minimum, Bitcoin requires the following cryptographic primitives:

1. One-way hash functions: A one-way function is easy to compute in the forward direction but difficult to reverse. A hash function maps arbitrary sized strings to fixed size ones. We require the hash to be *collision resistance* (hard to find two arbitrary preimages with same hash) and *preimage resistance* (given hash, find any preimage). A good hash function can be modeled using a *random oracle*, where there is no better way to break it than by brute force. By breaking, we imply finding a preimage satisfying certain predicate on the output.

2. Signature scheme: A signature scheme is a type of public-key cryptography (i.e. based on two keys), where a private key is used to generate a *signature* on a message and the public key is used to verify the signature. Only knowledge of private key and the message allows creating a valid signature. The owner of a public key also cannot later deny having created the signature. We require that the attacker cannot generate valid signatures on any arbitrary message even using the signing oracle on any other arbitrary messages. This is called UF-CMA (unforgeability under adaptive chosen message attack).

   We actually require a stronger property called *strong unforgeability* (SUF-CMA). This requires that an attacker having access to the signing oracle for arbitrary messages cannot generate a *new* signature on a message whose signature has already been queried. Unfortunately, the signatures used in Bitcoin do not satisfy SUF-CMA (they do satisfy UF-CMA though). This has led to the *malleability* problem in Bitcoin.

4.3 Transactional Layer

In this section we define a generalized view of transactional semantics of a blockchain system. The two foundational concepts here is a *state* and a *transaction*.

4.3.1 Minimal State

Consider a transaction arrived at a node. The node is doing following on receiving it:

1. Checks whether a transaction is valid
2. Apply it if so

Intuitively, there are some stateless checks, e.g. whether a signature for a transaction is valid, whether amount of tokens to transfer is non-negative, but also there are stateful ones. For example, if Alice is sending tokens to Bob, a node must be sure Alice has enough funds in order to make a payment. Or, if Alice is registering a domain, a node must be sure it is not taken yet.

So a node needs to store some state in order to validate incoming transactions. And there is some *minimal state* representation enough to validate an arbitrary transaction while removing any element from the representation eliminating this property. So all the nodes share this minimal state but a node could also store some additional information.

By applying a transaction a minimal state is being modified. It should be impossible to apply a transaction already processed.

For many reasons almost all cryptocurrencies of today are packing transactions into *blocks*. We can think about a block as of *atomic batch state update*.

[TODO: Block header - tx part]

We can state some axioms here.

Axiom 1. *There is some initial state hard-coded into each node. Further we name it genesis state.*

Axiom 2. *Validation and application of a transactions(and possibly an additional metadata) are deterministic procedures. All the honest nodes follow the same rules.*

Proposition 1. *If the same sequence of blocks is applied to the genesis state for two different nodes, then the resulting minimal states will be the same.*

Proof. Consider the nodes have the same minimal state and trying to apply the same block to it. By the Axiom 2, they will have the same minimal state as result, as verification and application procedures are deterministic. By the Axiom 1, genesis state is the same for all the nodes. By induction, result of sequential applying of the blocks results in the same minimal states for all the nodes. □

Further we will use both the terms “minimal state” and “state” interchangeably.

4.3.2 Bitcoin

In Bitcoin a transaction contains multiple *inputs* and *outputs*. Inputs are connected with outputs of transactions previously applied to a state, and the connected outputs must not be spent yet. That is, the outputs to be connected by the inputs of the transactions do not have connections from transactions previously applied to the state. Thus an output could be spent as whole only and so we can consider a set of unspent outputs as a minimal state.

How to spend an output? In Bitcoin it contains a script in a stack-based language. Input also contains a script. Then an input could spend an output if a combined script made of inputs' and then outputs' could be executed and results in non-zero top stack item.

[TODO: example]

4.3.3 Boxes, Propositions and Proofs

Abstracting the Bitcoin-like model, a minimal state could be represented as a set of *closed boxes* of size n_S. Each box has a value associated with it. Say, a transaction opens n_k boxes and also creates n_b new closed boxes, then the resulting state set has the size of $n_S - n_k + n_b$ after applying the transaction to it.

How to open a box? We can protect a box with a script in Bitcoin language. Or we can put a public key into closed box and then it is possible to open it with a proof of private key knowledge, a signature (we will consider details further). To describe these approaches as well as many others possible in a general way, we say a box is protected by a *proposition* of some kind, and in order to open it, a *proof* of the same kind must be provided. There are some tricky details we will discuss further.

A box can has some additional to a value data inside. For example, it can contain a domain record or a certificate. Anyway, box contents matters for every full node until it is closed.

4.3.4 Namecoin

Namecoin is a descendant of Bitcoin which in addition to token transfers, introduce $name \rightarrow value$ storage. In general, values could be arbitrary, but there are few standard namespaces with predefined semantics, for domains and identities.

We do not specify Namecoin design precisely below, but some Namecoin-like design.

Consider a transaction contains a box with *name_register* command specifying a $name \rightarrow value$ correspondence. Such a box has zero value and associated with a public key pk. It is demanded to pay some fee in order to put such a box into state. The box lives in the state for some period of time, then it is considered as expired and could be thrown away from a minimal state. It is possible to renew or transfer ownership to a different public key by publishing a *name_update* box replacing an original one in the state.

This design has a critical flow. A block generator could refuse to include *name_register* command into a block and put its own value for the same name. This is an example of *frontrunning attack*, when an original transaction is suppressed by another one issued by an attacker. In order to avoid frontrunning attacks, Namecoin has *name_new* command to announce the intention to register a name by providing its hash value.

4.3.5 Nxt and Ethereum

Having a Bitcoin wallet, you can be goggled by complexity of boxes and propositions in form of stack-based scripts. Why not to have just accounts and token transfers between them instead?

Actually some of Bitcoin successors walked this path. For example, Nxt has a dedicated notion of accounts. An account is associated with its public key. A transaction transfers tokens from one account to another and needed to be signed by the sender. For such a system stateful verification needs for a minimal state in form of table holding a correspondence between accounts and their balances.

With such a simple minimal state design we have a problem though. Let's describe it with an example. Alice has 50 tokens at some moment of time. She issues a signed transaction to pay 5 tokens to Bob. A node can validate the transaction and found it valid, and so applicable. After the application Alice has 45 tokens. But how to prevent second application of the transaction? Our minimal state representation seems to be flawed.

Ethereum solves the problem by modifying minimal state representation adding "nonce" value to it. That is, minimal state is not about (public key $\rightarrow$ balance) correspondence anymore, but (public key $\rightarrow$ (nonce, balance)). Transaction contains nonce value $txnonce$ as well, and transaction is valid and so applicable only if $txnonce = nonce + 1$. By application, $nonce := txnonce$.

Unlike Bitcoin, Ethereum sets strict order of transactions issued by an account. In Bitcoin, transactions could be applied in any order, if they are spending non-overlapping sets of outputs, and input of one transaction does not spend an output of another. In Ethereum, order of transaction is set by nonce values.

4.3.6 Transactional Metadata

Assume we have a set of objects serializable to a set of unique byte arrays. We want to *authenticate* these binary representations in an efficient. That is, we want to calculate a fixed-sized value for a whole set such as a single bit change always results in a change of the *authenticating* (or *root*) value, and the value is collision-resistant, so it is impossible(with non-negligible probability) to generate different set resulting in the same root value.

[TODO: Merkle tree / authenticated data structures explanation]

Along with transactions, we can put some aggregated data about them. For example. in Bitcoin's block a root hash of a Merkle tree for the transactions in block is put into the block. That way it is possible for nodes in a network

to exchange not full blocks but *blockheaders*. A *blockheader* is a block without its transactions. By including transactions root hash into the blockheader it is possible to have it spread around a network and be sure it is impossible to show transactions set other that was included.

[TODO: Bitcoin example]

4.3.7 Transactional Layer Generalization

After the examples, let's summarize what we have in common in all the observable cryptocurrencies.

1. **A Proposition And A Proof.** In an every imaginable blockchain we have objects to be protected by secret owners. To achieve the property of being protected we introduce a proposition, and an object could be modified or destroyed only by presenting a proof satisfying a proposition. There are a lot of possible instantiations, e.g. Bitcoin scripts or digital signatures.

2. **Box Structure.** There is a minimal element of a replicated state we are calling a *box*. A box is protected by a proposition. It is possible to modify it or destroy it only by showing a proof satisfying a proposition.

3. **Minimal State.** Minimal state is a most compact structure giving an ability to verify a transaction against it. Minimal state is about a set of boxes.

4. **Transaction And Transactional Language.** Transaction is a smallest possible atomic state modifier. A transaction is to be verified against a state in a deterministic fashion (so given a state and a transaction, two nodes will always give the same validation result whether *true* or *false*). If a transaction is valid against a state it could modify the state. Validation and application rules are individual (Ethereum even brings quasi Turing completeness here).

5. **Block.** All the blockchain systems are storing transactions in full blocks. Most of them also have some authenticating value for the set of block transactions included into block thus it is possible to use block headers instead of full blocks in many scenarios in order to reduce a load.

[TODO: wallet section?]

4.4 Consensus

We have proven(in the proposition [?]) that if the same sequence of blocks carrying transactions is applied to the same genesis state for two node then they will have the same state. It is exactly what do we want to achieve, but how to have the same sequence of blocks for all the nodes?

In the first place, we can achieve this only for nodes willing to achieve this by following some protocol strictly. We refer to such nodes as to *honest* nodes, and to the protocol as to *consensus protocol*. If nodes are not following the protocol we call them *byzantine* nodes. A byzantine node could be malicious, but also it could be not able to follow the consensus protocol because of software bugs, problems with connectivity, misleading information sent from outer world etc.

[validity, agreement, termination]

Computer Science studies consensus protocols since early 1980s. A lot of interesting results were generated in this field. For example, it is impossible to achieve consensus using a deterministic procedure for a set of nodes if they are exchanging messages asynchronously and a single process could fail(Fischer-Lynch-Paterson theorem [1]).

Consensus in open networks, so with unknown number of participants, is pretty new and very hard question.

1. Validity
2. Agreement
3. Termination

For a blockchain consensus protocols, we can state following properties:

1. Consistency(or Prefix immutability) - for two honest nodes the probability to have different prefixes after cutting last k blocks should go down with k and be negligible after some value. The good option is to have the probability going down exponentially with k.
2. Chain Quality - a party having $x\%$ of voting power should produce no more than $(\alpha\cdot)x\%$ blocks in a long run, where α is constant.
3. Chain Growth - over time blockchain should always grow. No one is interested in a structure with possibility to stuck.

4.4.1 Proof-of-Work

Proof-of-Work consensus protocol introduced in the foundational paper of Bitcoin [6] is in the core of Bitcoin, Ethereum as well as many other cryptocurrencies. The basic idea of the protocol is to force miners to iterate over output of some function with a small probability of success per iteration. A successful result is giving a right to generate a block. The probability is adjusted automatically via *difficulty* parameter D.

In case of Bitcoin, the function is just a hash function, but what is about its input?

We want to make blocks immutable after creation. For that, we are applying hash function for all the block contents.

[TODO: Merkle tree]

We also want for a block to refer to a previous block. So we include a hash of a previous block, in this case it in order to replace a block with another one it is needed to replace all its descendants also. As some amount of work is needed to generate each block in order to replace a chain suffix of length l, amount of work proportional to l is needed.

With Proof-of-Work it is impossible to make a false claim on successful block generation. Such a claim is easily verifiable, as calling hash function is very cheap. Thus a Proof-of-Work works as a protection against Sybil attacks [].

4.4.2 Proof-of-Stake

There are some disadvantages of Proof-of-Work schemes. A lot of resources to be spent. During early days a Proof-of-Work currency could be destroyed by a miner already having a lot of computational resources working for another blockchain.

Can we have a protection against Sybil attacks without spending computational resources? The simplest way to achieve this is to use cryptocurrency tokens as anti-Sybil tools. That is, a probability to generate a block is proportional to a stake a node holds.

4.4.3 History

Previously we talked about a blockchain. But in all the global networks collisions are possible (to prevent them we need to have a global lock or synchronous rounds and then some kind of leader election). So in a network a blocktree lives.

[TODO: a blocktree picture]

In most cases the fact of blocktree existence in omitted. A node is storing a blockchain. A blockchain has some score(e.g. a chain length, but this is a totally insecure scoring function, see [9] for details). If better chain is declared in network, a node is throwing away blocks until common block and then apply better suffix. In this case a node sees a blocktree only during switching from one branch of it to another. There are some proposals to explicitly use a blocktree. For example, in GHOST scoring function [8] a chain with heaviest tree wins.

4.4.4 Consensus Layer Generalization

4.4.5 Full Node View

Full node is a node which holds at least some state enough to check whether an arbitrary transaction is valid against it and so applicable to it or not. We have defined such a state, *minimal state*, above. Nodes are also storing a history from which the minimal state has been rendered. In addition to the state modifiers log and the minimal state, a fullnode also contains two more entities. *Memory pool* contains transactions not yet included into blocks (and there is no guarantee of inclusion for them). *Vault* contains some node-specific information a node is

extracting from the log. For example, it could contain values encoded in some or all OP_RETURN instructions, or all the transactions for specific addresses. The well-known example of vault is *wallet* which contains private keys as well as transaction associated with their public images.

With the four entities being defined we can explicitly state a node view type now: $node\_view = \langle history, minimal\_state, vault, memory\_pool \rangle$.

The quadruple could be modified by applying whether an offchain transaction or a block coming from a local side(user issuing a transaction to a local node, mining software generating a block) or from remote peer. We can state some rules of the node view modification even for the most abstract definition given:

- an offchain transaction modifies vault and memory pool. Atomicity in this update is not critical.
- for a persistent node view modifier atomicity for an update is strictly needed! If history is producing rollback sude-effect, other parts must handle it properly before applying an update. This sounds trivial, but in fact many implementation are spending years fighting with bugs related to inconsistency and read-when-update issues.

[TODO: pseudocode? or Scala code in Scorex counter-part?]

4.5 Peer-to-Peer Network

Blockchain is maintained in a peer-to-peer network. For simplicity we are starting with a network where all the nodes [...]

4.6 Incentives

4.7 Complications And Alternative designs

In additions to the systems described in this chapter as well as many other systems used around, there are many designs existing only on paper. In this section we quickly observe some proposals. The goal of Scorex is to help to get them from papers to prototype implementations.

The various alternative designs are proposed to address some of the problems with Bitcoin explained below:

1. Storage scalability: In Bitcoin, every node must store the entire blockchain (currently over 10 GB). This is a bottleneck as the blockchain becomes older. Proposals have been presented that allow pruning of the blockchain under certain assumptions.
2. Memory usage: The UTXO set is something that needs to be kept in memory for fast validation. This set is currently several hundred MBs.

3. Rational behavior: If nodes behave rationally, they will not store the blockchain. Rather they will store only the UTXOs they are interested in. In the long run, this could lead to *tragedy of the commons*, where no one has a complete copy of the blockchain.

4. Privacy: While Bitcoin uses pseudo-random looking addresses for privacy, some information is inherently leaked. For instance, we can know all the addresses that a given satoshi has traveled to since its creating. Additionally, we can often infer that certain addresses belong to the same entity.

5. Delay in confirmations: Bitcoin has an average confirmation time of 10 minutes. However, if the size of the unconfirmed transactions is higher than the maximum network throughput (1 MB/10 minutes), then a transaction can remain unconfirmed for tens of hours. Additionally, since PoW is a randomized process, there is no guarantee that a block will be mined soon, even if the transaction fee is very high.

6. Transaction throughput: As mentioned in the previous point, the maximum block size is 1 MB. This roughly translates to about an average of 144 MB of data added to the blockchain per day if all blocks are full. Assuming a transaction size of the about 224 bytes (the minimum), we get a maximum of 28086 transactions/hour, which may become a bottleneck.

7. Validation-less Mining: Due to latency in network and the large size of blocks, many miners and mining pools do not validate headers. In other words, they start finding the solution using the Merkle root of transaction hashes without validating the batch of transaction themselves (which could be around 1 MB currently). Often this results in a split, where some clients create a Merkle root incorrectly (either deliberately or by accident) such as the softfork due to BIP66 in July 2015[1], where certain miners were creating invalid blocks and others were building a chain on top of those.

8. Fully prunable outputs: In Bitcoin, certain outputs cannot be provably spent (such as those with value 0 – those created using OP_RETURN[2]) or sent to an output that cannot be provably spent[3]. Such unspent outputs serve no meaningful purpose to Bitcoin nodes and can be prunes from their UTXO set.

4.7.1 SPV

SPV (Simple Payment Verification) is an alternative to Satoshi's Bitcoin protocol and operates on top of the same blockchain. It is designed to enable lightweight clients that can sync in minutes instead of days and store only a fraction of the information (few MBs) compared to a full node (several GBs).

[1] https://en.bitcoin.it/wiki/Softfork#2015_BIP66_Blockchain_Fork
[2] https://en.bitcoin.it/wiki/OP_RETURN
[3] Example: https://blockchain.info/address/1CounterpartyXXXXXXXXXXXXXXXUWLpVr

In this method, a node only verifies headers from the genesis block and validates only the last few full blocks (enough for rollback). The key idea in SPV is that the security of PoWs is not measured by block height (the number of blocks since the genesis block) but rather the depth (the number of blocks mined since a particular payment has been confirmed). This allows clients be able to verify payments quickly. However, the security of SPV has not been formally proven.

Implementations of SPV nodes use a protocol extension called *bloom filters*, described in BIP 37. They use headers for blocks prior to their wallet's birth timestamp, and request filtered blocks for the rest by sending a bloom filter to its peers. The peers then send the relevant transactions for blocks along with the Merkle paths.[4]

4.7.2 Rollerchain

Rollerchain (RC) is a proposal aimed to (1) reduce storage and (2) enforce nodes to store blocks and prevent tragedy of the commons. RC stores data only for verifying recent k iterations for some security parameter k. This is similar to SPV mode, where the nodes additionally store the last k blocks. However, rational nodes in RC are *enforced* to store the entire sequence of blocks for a certain depth $\leq k$ determined solely from the public key. To create a successful PoW, nodes must additionally prove storage of one or more snapshots determined via a hash of their public key and the current block index. Since the index changes with each block, there is a *sliding window* of snapshots from which a node must store a few. The index of stored snapshots also slides with the window and nodes must recompute this at every new block. Consequently, nodes are implicitly forced to store all the blocks till the depth of their earliest snapshot. In RC, the block header contains a Merkle root hash of the minimal state that will be obtained after the block is applied. Additionally, the Merkle root hash of the transactions and a Proof of Storage of snapshots is also present in the header.

[Not very clear how the Proof of storage works – Alex, please add some text?]

4.7.3 GHOST

The GHOST protocol is designed for blockchains with very fast block times. As the network becomes less synchronized and blocks are generated fast, a significant amount of PoW will not make it into the main chain due to orphans. This leads to reduced security. The GHOST proposal makes use of the additional orphaned blocks to strengthen the network instead of rejecting them. Thus, two different branches will be compared not just by length but how 'heavy' they are in terms of PoW. A GHOST protocol has various mechanisms for rewarding orphans. Etherium is one example that follows the GHOST strategy.

[4] http://bitcoin.stackexchange.com/a/11721/2075

4.7.4 Bitcoin-NG

Bitcoin-NG (or simply NG) was proposed to solve two main issues in Bitcoin:

1. *Reduce confirmation time:* Current Bitcoin transactions require an average of 10 minutes for the first confirmation. In NG, the first confirmation is very fast (within 30 seconds; roughly the time to cross the network).

2. *Increase transaction processing bandwidth:* Furthermore, much more transactions can be inserted into the blockchain between two successive PoWs. This is only limited by bandwidth and processing constraints of miners.

While Bitcoin is retrospective based (i.e., the PoW captures transactions created *before* the PoW was formed), NG is forward-looking in that the PoW enables a miner (aka *leader* in NG) to confirm transactions in the future.

The main idea in NG is to remove the link between transaction confirmations and the *leader selection*. A leader is essentially someone who's PoW is accepted. In NG, the leader is still selected based on PoW but the block selecting the leader (the *key-block*) does not contain transactions. Rather the currently selected leader (i.e., the public key) has the sole power to decide the transactions for the blockchain in several *micro-blocks* until the next leader is selected. The leader adds the transaction without much delay, ensuring that transactions get confirmed (with one PoW) very quickly. Furthermore, there is no limit to the number of micro-blocks that can be inserted, ensuring that the throughput of the network is high. We summarize the main features of NG:

1. Two types of blocks: *key blocks* that contain the PoW, a reference to the previous block and a coinbase (reward) transaction but no other transactions. The key block additionally gives the miner the ability to create several *micro-blocks* containing one or more transactions until the next PoW (i.e., key-block is found). Miners only compete for key-blocks.

2. The fees is split into a 40-60 ratio, where 40% goes to the owner of the current PoW and the rest to the next PoW.

4.7.5 ByzCoin

ByzCoin builds on the ideas of NG of separating the confirmations and leader selection. There are several differences from NG discussed below:

1. Two separate blockchains: The key blocks and micro-blocks are part of two different chains. A key-block links to the previous key-block (the *main chain*) and the micro-blocks form a *secondary chain* and also link to the corresponding key-block in the main chain.

2. Different process for mining micro-blocks: While key-blocks are mined in a similar fashion to NG, the micro-blocks are mined using a the Practical Byzantine Fault Tolerant (PBFT) protocol [?] run by the leader. The group of members who participate (the *replicas*) are selected from a sliding

window of miners who contributed a PoW in the last time-slice (say 24 hours). The amount voting power held by a miner is proportional to the amount of blocks contributed in that window.

3. Interactive consensus protocol: The replicas perform an interactive protocol using PBFT and combined signatures to obtain consensus on the micro-blocks to insert.

The key difference with NG is that while a transaction is instantly confirmed in NG, it is not permanent in the case of dishonest miners because there is no control over how a miner generates micro-blocks. ByzCoin adds a level of consensus even for micro-blocks so the chance of those rolling back is negligible compared to NG.

4.8 ZeroCash

ZeroCash (ZC) is a blockchain based protocol designed for privacy. Since the data in the Bitcoin blockchain is public, true anonymity is not present.

ZeroCoin: ZC can be considered an extension of ZeroCoin (ZCn), an earlier proposal. In order to describe ZC, we will first describe ZCn.

Assume that all coins to be obfuscated are of a single denomination, say 1 BTC. ZCn has its own currency called Z, such that 1 Z = 1 BTC. There is a pool of obfuscated Z coins called *Zero-Pool*. Alice wants to anonymize her Bitcoin represented by serial number c_A. In order to do that, she begins by exchanging c_A with an equivalent Z coin as follows. She generates a secret serial number s_A and another secret r_A. Then she creates a commitment z_A to s_A using r_A as randomness. That is, $z_A = \text{COMM}_{r_A}(s_A)$, which is the Z coin Alice will use in exchange for c_A. All such exchanged coins are automatically added to the Zero-Pool. The commitment has a computationally binding and perfectly hiding property so no knowledge of s_A or r_A is leaked from z_A. Later on Alice will take back her z_A using a transaction t_A, which is then automatically converted to 1 BTC. For anonymity, an adversary should not be able to link t_A to z_A. This is achieved using Non-interactive Zero-Knowledge Proofs (NIZKs).[5]

Alice creates a spend by revealing s_A (but not r_A) and creates a NIZK proof of the NP statement "I know r such that $z = \text{COMM}_r(s_A)$ and $z \in$ Zero-Pool".

[5] A NIZK proof is understood as follows. Let N be an instance of a hard problem, such as finding the factors of a large integer. Let P be a witness to this problem that Alice knows. There are three requirements of NIZK proofs: (1) Given a **random** string R, Alice can construct a short proof $\pi = \text{PROOF}(P, R)$ such that $\text{VERIFY}(\pi, R, N) = \textit{true}$ iff Alice 'really knows P'. The concept of 'knowing' is captured as follows: (2) If Alice can be fooled into using a **maliciously crafted** R, then π will reveal P with a high probability. (3) Additionally, Alice can construct pairs (π', R') such that $\text{VERIFY}(\pi', R', N) = \textit{true}$ without knowing any witness P as long as **she selects** R'. Thus, the generation of R is of outmost importance because if the verifier selects R, the secret is leaked and if the prover selects R then the proof is not convincing. If given the transcript of a proof, we cannot distinguish between the cases (1) and (3), then the protocol is zero knowledge, intuitively because because given some (π, R) pair, there is no way of knowing if it was generated using (1) or (3).

The process of creating a commitment and revealing only one input (but never opening them completely) may seem counter-intuitive, but this in fact makes the statement provable by efficient NIZKs, because the circuit for computing the commitment can be efficiently coded into an NP language. The spend could have been done even without revealing s_A. However, this is needed for the next property – security – so that Alice should not be able to spend z_A twice. This is achieved by keeping track of spent s_As in a public ledger, kept by each client.

Due to this ZCn has two scalability issues: (1) The Zero-Pool always grows as we cannot determine which of the Z coins have been spent (for anonymity). (2) The spent set always grows as we need to keep track of double spends. Unlike Bitcoin, which uses spends only for bootstrapping and stores only unspent outputs, ZCn must store both the spent and unspent outputs. The unspent outputs is simply the Zero-Pool.

ZC is an extension of ZCn with the following differences:

1. In ZC, the coins are minted in the protocol itself rather than exchanged with Bitcoins. This allows people to transact directly in ZC rather than using it as an add-on to Bitcoin or another currency. In other words, the spent coins also generate Z coins.

2. The statement "I know r such that $z = \text{COMM}_r(s_A)$ and $z \in$ Zero-Pool" is replaced by the NP statement "I know r such that $z = \text{COMM}_r(s_A)$ and z is a leaf node of a Merkle tree with root hash M", where M is the root hash of the Merkle tree of Z coins in the Zero-Pool. This makes the statements much shorter. The value s_A is revealed in order to keep prevent double spends as in ZCn.

3. ZC allows coins of arbitrary denominations instead of 1 BTC as in ZCn. This is done by introducing a *pour* operation that uses a bunch of Z coins and creates new ones without revealing the amount in either the destroyed or created one. The only thing revealed is that the sum of inputs is $\leq$ sum of outputs. This is done by appropriately modifying the Z coins and proving the corresponding NP statement using zkSNARKs.

4. ZC uses a variant of NIZKs called zkSNARKS (succinct non-interactive arguments of knowledge). A zkSNARK is essentially a NIZK with the proof and verification-time O(1), and secure against only a computationally bound adversary.

However,

4.9 Conclusion

4.10 Further Reading

Proof-of-Work and blockchain were introduced in the foundational Bitcoin whitepaper [6].

Overview of Bitcoin P2P layer along with description of possible Eclipse attacks(partly fixed to the moment) against it could be found in [3]

Chapter 5

Scorex

5.1 Introduction

Scorex is a modular blockchain core framework. It supports definitions given in the previous chapter in form of Scala code. What do you need to do in order to build something on Scorex is to provide implementations for all the abstract interfaces (possibly reusing code previously written for our projects).

5.1.1 Scala Language

We will describe some concepts of Scala language in sections started with "*". If you already a Scala developer you can miss the sections. Experience with programming languages (say, Java, C++, OCaml or Rust) is needed as we will explain Scala features used in code snippets provided very quickly. For a good introduction into the Scala language, please refer to [7].

Scala is functional, modular and also object-oriented language. There are several reasons to choose this language:

1. Scala runs on the JVM which allows it to be cross-platform.
2. Scala inter-operates seamlessly with Java.
3. Scala is fully functional and consequently allows compact and more readable code.
4. Scala has powerful constructs for concurrency.
5. Scala has powerful type system. Scorex is using typing features of the language extensively.

5.1.2 Propositions and Proofs

In the first place, we are getting into a mechanism to protect binary objects, e.g. transaction outputs, from non-permissioned access. We protect an object

with a *proposition*. Then in order to make an action with an object it is needed to provide a *proof* for its proposition.

Proposition is a very abstract concept. The only property we require from it is to be serialized into bytes. In form of Scala code it is described as:

```
trait Proposition extends BytesSerializable
```

In most cases, a proposition claims proof of knowledge of a secret to be provided in a non-interactive zero-knowledge form. For example, in most of popular signature schemes a digital signature is a non-interactive zero-knowledge proof of a private key knowledge. Scorex has a basic entity for that

```
trait ProofOfKnowledgeProposition[S <: Secret] extends Proposition
```

A proof is an object which could satisfy a proposition given an additional input, namely a *message*(e.g. transaction bytes). As well as a proposition, a proof could be serialized into bytes.

```
trait Proof[P <: Proposition] extends BytesSerializable {
  def isValid(proposition: P, message: Array[Byte]): Boolean
  ...
}
```

For a *ProofOfKnowledgeProposition* corresponding abstract proof is provided:

```
trait ProofOfKnowledge[S <: Secret, P <: ProofOfKnowledgeProposition[S]]
        extends Proof[P]
```

5.1.3 Box

A box is a minimal state element. An unspent output in Bitcoin is a box. An account in certain state in Nxt or Ethereum is also a box. Basically, a box is about some amount of tokens associated with it and also some proposition which protects a box from being spent by anyone but a party (or parties) knowing how to satisfy the proposition. Thus the basic abstraction is as follows:

```
trait Box[P <: Proposition] extends BytesSerializable {
  type Amount = Long

  val value: Amount
  val proposition: P

  val id: Array[Byte]
}
```

5.1.4 A Transaction and Minimal State

As it was shown in the Chapter 1[TODO: link], a transaction and a minimal state could be defined via each other: a transaction is a state modifier, which

also could be valid or not against a state, and applicable if and only if it is valid. A minimal state is a data structure which deterministically defines whether an arbitrary transaction is valid and so applicable to it or not.

We are defining functional interface for minimal state below:

So a minimal state knows its current version, can apply a block, and also rollback to a previous version. To validate a transaction, it just calls *validate* function of a transaction and checks its result.

5.1.5 Wallet

[TODO:]

5.1.6 Memory Pool

5.1.7

5.1.8 Node View Holder

5.1.9 Transactional Module

5.2 Consensus Layer

5.2.1 Block

5.2.2 History and Blockchain

5.3 Network Layer

Network layer in Scorex is simpler than in Bitcoin or Nxt.

5.3.1 Peer Discovery

5.3.2 Broadcasting Strategies

5.3.3 View Synchronizing

5.4 Ready Modules

There are few modules already implemented.

5.4.1 Proof-of-Stake

Proof-of-Stake module contains implementations of two Proof-of-Stake consensus algorithms.

5.4.2 Simple Transactions Module

Simplest Transactions Module contains an implementation of transactional module with only one kind of transactions, just tokens transfers from one public key to another.

5.4.3 Permacoin Implementation Module

The module contains an implementation of Permacoin consensus protocol [5].

5.5 Conclusion

Bibliography

[1] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[2] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. *The Bitcoin Backbone Protocol: Analysis and Applications*, pages 281–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[3] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoinâĂŹs peer-to-peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, 2015.

[4] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[5] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 475–490. IEEE, 2014.

[6] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[7] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[8] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[9] StackExchange. Strongest vs longest chain and orphaned blocks.