

# Scorex Tutorial

Alexander Chepurnoy

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 [5].

In order to understand Scorex, it is helpful to consider Bitcoin, Namecoin and Ethereum 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. *Bitcoin*: Decentralized currency.  
*Purpose*: Decentralization of money, easy to store and spend.
2. *Namecoin*: Decentralized DNS (stores *key*  $\rightarrow$  *value* mappings).  
*Purpose*: Decentralization of DNS, difficult to censor or shut down.
3. *Ethereum*: Smart contracts.  
*Purpose*: Enable trustless computing via decentralization.

### 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

# Overview of Bitcoin

The article “How the Bitcoin protocol actually works” by Michael Nielsen [8] discusses various problems that led to the creation of Bitcoin. The original whitepaper by Satoshi Nakamoto [7] provides a good description on the concepts and details can be found in the book “Mastering Bitcoin” [1].

Here we attempt to give a high-level overview of Bitcoin that is sufficient to understand and implement it in Scorex. The terms we need to understand are: *transaction*, *input*, *output*, *reference*, *block* and *confirmation*. Firstly note that funds are exchanged between *addresses* which are hashes of public keys. In this document, we often use the terms ‘address’ and ‘public key’ interchangeably (the meaning will be clear from the context). However, it should always be kept in mind that addresses are one-way hashes of public keys.

### 2.1 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  $ref_{A \leftarrow x}$ , 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

“(ref<sub>A←x</sub>, remove  $\mathbb{B}x$  from  $A$ ), (put  $\mathbb{B}y$  in  $B$ ), (put  $\mathbb{B}z$  in  $C$ )”

together with its signature under  $A$  is considered a ‘transaction’.

**Notation:** We will use the following shorthand notation:

- $ref_{X \leftarrow x}$  is the message “( $ref_{X \leftarrow x}$ , remove  $\mathbb{B}x$  from  $X$ )”. This is an input.
- $X \leftarrow x$  is the message “put  $\mathbb{B}x$  in  $X$ ”. This is an output.
- $\sigma_X(m)$  is signature on message  $m$  under public key  $X$ .

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

**Transactions:** A transaction can have several inputs and outputs. The only restriction is that the total amount of funds destroyed (via the inputs) must be  $\geq$  the total amount of funds created (via the outputs). Any difference is considered a transaction fee. More formally, define  $m$  to be the message

$$M \stackrel{\text{def}}{=} (ref_{A_1 \leftarrow x_1}, ref_{A_2 \leftarrow x_2}, \dots, ref_{A_n \leftarrow x_n}, B_1 \leftarrow y_1, B_2 \leftarrow y_2, \dots, B_l \leftarrow y_l),$$

where  $ref_{A_i \leftarrow x_i}$  are references to  $n$  previous transaction outputs where  $A_i$  received  $x_i$  bitcoins, and  $(B_i, y_i)$  are  $l$  pairs of type (addresses, amount). A valid transaction  $tx$  is a tuple:

$$tx \stackrel{\text{def}}{=} (M, \sigma_{A_1}(M), \sigma_{A_2}(M), \dots, \sigma_{A_n}(M)) \quad (2.1)$$

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  $ref_{A_i \leftarrow x_i}$  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. 2.1. Then

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

Because  $refs$  are constructed are hashes of previous transactions, 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 (called UTXO in Bitcoin) 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 it is 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.

## 2.2 Validating 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

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

where  $M$  is of type:

$$M \stackrel{\text{def}}{=} (ref_{A_1 \leftarrow x_1}, ref_{A_2 \leftarrow x_2}, \dots, ref_{A_n \leftarrow x_n}, B_1 \leftarrow y_1, B_2 \leftarrow y_2, \dots, B_l \leftarrow y_l).$$

The values  $A_i$  and  $x_i$  are obtained from a *UTXO database* that every client must maintain.<sup>1</sup> This database is a key-value store of type  $ref_{A_i \leftarrow x_i} \rightarrow (A_i, x_i)$ . 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. Observe that a node that does not store blocks cannot help new nodes to bootstrap.

It is helpful to consider each (unused)  $ref_{A_i \leftarrow x_i}$  above as a “closed box” with  $x_i$  inside such that the box can only be opened via the private key corresponding to  $A_i$ . The act of using  $ref_{A_i \leftarrow x_i}$  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,  $ref_{B_i \leftarrow y_i}$  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).

The output of a transaction contains an *Output Script* which is equivalent to “partial unlocking instructions”, a sequence of operations that when combined with another script will result in the Satoshi being unlocked. The remaining part of the unlocking instructions are provided when that output is later spent via a *different* transaction. This is called the *Input Script* because the output being spent becomes the input of the new transaction.

---

<sup>1</sup>The inputs additionally include the public key corresponding to address  $A_i$ . We can assume that this key is part of the signature.

The input script contains the signature and public key, while the output script contains instruction to verify the signature for the address holding the Satoshi's. The (signature, public-key, address) triplet is considered a valid if the combined script evaluates to non-zero (True).

A node validates a transaction by validating every input as follows:

1. For each input  $ref_{A_i \leftarrow x_i}$  from  $(ref_{A_1 \leftarrow x_1}, ref_{A_2 \leftarrow x_2}, \dots, ref_{A_n \leftarrow x_n})$ 
  - (a) Load signature  $\sigma_i$  and public key corresponding to  $A_i$  into stack (public key is in the signature). This is the Input Script.
  - (b) Using the key  $ref_{A_i \leftarrow x_i}$  find the value  $(A_i, x_i)$  from the UTXO database.
  - (c) The loaded box contains the amount to be released as well as the output-script (that verifies the above signature on  $M$ ). The input-output script is evaluated and the box is opened if the result is True (non-zero), otherwise an error is thrown.
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”).

## 2.3 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 works as follows. First the miner constructs a ‘block’  $b$ , a sequence of bytes containing the following data:

1. A bunch of unconfirmed transactions along with one reward transaction (known as the *coinbase transaction*).
2. Hash of the previous block  $h_{pr}$  and a nonce.

Finally,  $h = \text{Hash}(b)$  is computed. If  $h$  contains at least a specified number of leading zeros, the block  $b$  is considered *mined* and broadcast to the network, otherwise the miner tries with different nonces until a mined block is found or another mined block containing  $h_{pr}$  is received from elsewhere. All transactions in a mined block are considered *confirmed*.

**Confirmations:** The number of confirmations of a transaction are the number of mined blocks that have been accepted by the network after the block that included that transaction. In other words, it is the *depth* of that transaction in the blockchain. 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 is forgotten.

**Proof of Work:** Recall that we consider a block to be mined only if its hash contained at least a ‘specified’ number of leading zeros. Assuming that the hash function behaves like a random oracle, the probability of a random image of  $k$  bits having  $x$  leading zeros is about  $2^{-x}$  and the only way to obtain such a value is to try hashing around  $2^x$  different blocks.

This is called *Proof of Work* (PoW) because it is a witness that someone spent around  $2^x$  CPU cycles. The sole purpose of PoW is to secure the blockchain against tampering. Security in our sense implies the *persistence* of the blockchain, i.e., the inability to an attacker to fool a node into accepting a truncate or forked copy of the ‘true’ blockchain (i.e., with different set of transactions that either don’t include the real ones or double spend them). Bitcoin uses the following measure for difficulty: if the leading number of zeros are  $x$  then  $difficulty = 2^x$  and  $target = 2^{k-x}$ . If two competing blockchain are presented, honest nodes select the ones with the highest *cumulative difficulty*. That is, one where the total CPU cycles used are highest.

**Difficulty Adjustment:** We mentioned that the binary representation of the hash of a block must have at least a ‘specified’ number  $x$  of leading zeros. This is formalized as:  $\text{Hash}(b) \leq \text{target} (= 2^{k-x})$ . The current target changes in accordance with how fast or how slow the blocks are accepted by the network during a time frame of 2 weeks. At this rate, there should be exactly 2016 blocks generated in two weeks. If the average over 2 weeks is more or less than 1 block every 10 minutes, the protocol will adjust the current target and attempt to bring it back. However, if the hashing power of the network increases sub-exponentially (or faster), then difficulty may never catch up and the average block time will always be less than 10 minutes (as is happening with Bitcoin).

**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.<sup>2</sup> Additionally, there can be entries in UTXO that are provably unspendable (see Section ??). Even these can be pruned.

## 2.4 Block Structure

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

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

**Example of header:** The following is an example of a block header<sup>4</sup>:  
02000000b6ff0b1b1680a2862a30ca44d346d9e8910d334beb48ca0c00000000  
000000009d10aa52ee949386ca9385695f04ede270dda20810decd12bc9b048a  
aab3147124d95a5430c31b18fe9f0864

The different sections are identified by alternating underlines:

1. Block version: 02000000 (decodes to 2).
2. Hash of previous block's header: b6ff0b1 ... 48ca0c000000000000000000.
3. Merkle root hash of payload: 9d10aa52 ... decd12bc9b048aaab31471.
4. Unix timestamp: 24d95a54 (decodes to 1415239972).
5. Difficulty target: 30c31b18 (decodes to  $1bc330_{\text{hex}} \cdot 256^{18_{\text{hex}}-3}$ ).
6. Nonce: fe9f0864.

---

<sup>2</sup>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.

<sup>3</sup>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).

<sup>4</sup><https://bitcoin.org/en/developer-reference#block-headers>



**Computing the Merkle root:** 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 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  $\text{SHA256}(\text{SHA256}(L||R))$  forms the parent for the next iteration. This process is repeated until the root is reached.

**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.

**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.

## 2.5 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 protocol 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 than the longest one.

**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 [3], can be achieved if the underlying proof-of-work (PoW) based consensus system satisfies the following two properties [3]:

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  $\lambda$  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 [3]: (1) overall computing power is low, (2) blocks are generated too fast, or (3) network takes long time to synchronize.

**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 3

# A Blockchain System Design

### 3.1 Introduction

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 elaborate what this means.

Consider Bitcoin as an example, where a group of distrusting peers hold money in form of cryptographic tokens and can transfer them between themselves without any trusted mediator. The last part (“decentralized peer-to-peer network”) implies that these peers follow a protocol specified (being effectively thrown away from the network otherwise). Peers issue “authenticated events” (signed messages of some semantics). For example, they can send out signed payments or they can register a *name*  $\rightarrow$  *value* correspondence in a shared database (certificates, domains). Bitcoin keeps records of transfers by building a blockchain, a type of *append-only database* (or “append log”). If this append-log is “prefix-immutable” then all honest peers following the protocol will agree on an immutable prefix of events in the append log. That is, if we cut a suffix of some length from the log of every honest peer, then with overwhelming probability, the remaining entries in these logs will either be identical or be a proper sub-sequence of another honest node’s log. “Non conflicting” implies that the log should not contain any sequence of events that violate protocol constraints.

Prefix immutability implies that older data in the append-log can be considered *tamper-resistant*. The immutability is achieved using a consensus method called *proof-of-work* (a solution to an exponentially hard puzzle that can be verified efficiently). 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.

Non-conflicting events in Bitcoin require that the same Satoshi cannot be spent twice (Satoshis are considered immutable; a transaction destroys old Satoshis and creates new ones). Thus, for example, if Alice has sent all her

Satoshis to Bob, she can't send anything after that until she receives more of them from elsewhere. Such double spends are prevented using the prefix-immutable database because the entire transfer history of every Satoshi (the smallest unit of Bitcoin) can be traced back to the time it was created.

## 3.2 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.

## 3.3 Transactional Layer

In this section we define a generalized view of transactional semantics of a blockchain system.

### 3.3.1 Minimal State

Consider what a node does on receiving a transaction:

1. Checks whether the transaction is valid.
2. Apply the transaction if if so.

Intuitively, a node does some stateless checks (e.g., whether a signature is valid or the amount is non-negative) and some stateful ones (if Alice indeed has enough funds to send to Bob or if a registered domain has not been taken yet).

Thus, a node needs to store *some* state in order to validate incoming transactions. Additionally, there is a unique *minimal state* containing the smallest set of states a node must store to validate arbitrary transactions. All nodes store this (common) minimal state but a node could also store some additional information.

After applying a transaction, the minimal state is modified such that is impossible to apply the same transaction again at a later time.

Almost all cryptocurrencies of today are packing transactions into *blocks*, which we can think of as *atomic batch state updates*.

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 transaction (plus any metadata) is a deterministic process and all the honest nodes follow the same rules.*

**Proposition 1.** *Two nodes applying the same sequence of blocks to the genesis state will obtain identical minimal states.*

We will use the terms “minimal state” and “state” interchangeably.

**Example:** In Bitcoin, a transaction contains multiple *inputs* and *outputs*. Inputs are essentially the outputs of an older transaction previously applied to a state. In order to be valid, these older outputs should not have appeared as the input of any other transaction. One inefficient way to validate this would be to check every transaction in the blockchain appearing after that older output. A more efficient way would be to store the set of unspent older outputs and remove from the set once they appear as the input in a transaction. Thus, we can consider the set of unspent outputs (UTXOs) as the minimal state.

### 3.3.2 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.

How can we open a box? We can use a Script-like language as in Bitcoin, or we simply have public-keys instead of addresses and signatures instead of scripts. To describe these approaches as well as many others, 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. Intuitively, a proposition is an NP statement and a proof, its witness.

A box can have some additional data inside. For example, it can contain a domain record or a certificate. The contents of closed boxes are relevant to every node. However, nodes can forget about boxes that have been opened.

### 3.3.3 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. The following discussion assumes some Namecoin-like design.

Consider a transaction that contains a box with *name-register* command specifying a  $name \rightarrow value$  correspondence. Such a box has zero value and is associated with a public key  $pk$ . A fee may be required to insert this a box into the minimal state. The box lives in the minimal state for some period of time after which it is considered expired and discarded. It is possible to renew or transfer ownership to a different public key by publishing a *name-update* box, which replaces the original box in the minimal state.

This basic design has a critical flow. If a miner processing a name-register box finds that particular name interesting or worth snatching, nothing prevents him from grabbing that name before it goes into the blockchain. The miner could simply refuse to include original box and put its own one for the same name. This is an example of a *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 first registering a cryptographic commitment to the name (without revealing the name). That way, a miner cannot know if the domain name intended to be registered is interesting or not. Later on the name-register command simply opens the commitment. The same attack cannot be applied because the public key of the name-register and name-new must match.

### 3.3.4 Nxt and Ethereum

After understanding Bitcoin, you might wonder why Bitcoin has a Script-based structure instead of hard-wired semantics that validate public-key and signatures. In fact, why bother with address at all when we could simply map public keys to accounts and transactions to Satoshi's transferred between the accounts?

In fact, some of Bitcoin successors have adopted this approach. 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 needs to be signed by the sender. For such a system, the minimal state is a table holding a correspondence between accounts and their balances.

This simple example, however, has a critical flaw. Suppose Alice has 50 tokens at some time. She issues a signed transaction to pay 5 tokens to Bob. A node will find this valid, and therefore, applicable. After the application Alice has 45 tokens. However, Bob or anyone else can now *replay* the same transaction

again and again till Alice’s account is empty. Our minimal state representation seems to be flawed.

Ethereum solves the problem by modifying minimal state representation with a “nonce” value added to every account. That is, the minimal state is no longer just a (public key  $\rightarrow$  balance) correspondence, but a (public key  $\rightarrow$  (nonce, balance)) one. Each transaction contains some nonce value  $txnonce$  describing which box to open. A transaction is valid only if  $txnonce = nonce + 1$ . After applying the transaction the old box is destroyed and a new box with the increment nonce and update balance is created. Thus, even Ethereum has the idea of immutable boxes.

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

### 3.3.5 Transactional Metadata

Suppose we have a sequence  $s$  of objects serializable to byte arrays and want to *authenticate* their concatenated binary representation efficiently. That is, we want to calculate a fixed-sized value (the *root*) from  $s$  such that even a single bit change in the representation always results in a change of the root. Furthermore, we require *collision-resistant*. That is, it must be computationally hard to generate different sequences resulting in the same root. Any datastructure that supports this is called an *authenticated data structure*, an example of which is Merkle trees.

Along with transactions, we could put some aggregated data about them. For example, in Bitcoin, the root hash of the Merkle tree of transactions of that block is also put into the block. That way it is possible for nodes to exchange only *block-headers* and not the full blocks. A *block-header* is a block without its transactions. By including transaction root hash into the block-header, it is possible to have it spread around a network and be sure that it is impossible to show transactions other than what were included when the root was computed.

### 3.3.6 Transactional Layer Generalization

Let us summarize the common features of various cryptocurrencies.

1. **A Proposition and A Proof.** In an every imaginable blockchain we have objects to be protected by secret owners. The property of ‘being protected’ is mapped to a proposition (an NP statement), and objects can be modified or destroyed only by presenting the corresponding proof of the proposition (a witness of the NP statement). There are several instantiations; Bitcoin scripts or digital signatures.
2. **Box Structure.** There is a atomic element in the minimal state which we call a *box* and captures the objects of the above definition. A box is



protected by a proposition. It is possible to modify it or destroy it by showing a proof satisfying the proposition.

3. **Minimal State.** Minimal state is a most compact structure giving an ability to verify a transaction. Minimal state is a set of closed boxes.
4. **Transaction And Transactional Language.** Transaction is the smallest possible atomic state modifier. A transaction will be verified against the minimal state in a deterministic fashion (so given a state and a transaction, two nodes will always output the same result). If a transaction is valid against a state it could modify the state. Validation and application rules are blockchain specific (Ethereum even brings quasi Turing completeness here).
5. **Block.** Most blockchain systems don't update the minimal state arbitrarily. Rather they do it in batches by packing several transactions into a block and then applying the block. Most blocks also have some authenticating value for the transactions included therein. Thus, it is possible to use block-headers instead of full blocks in many scenarios.

### 3.4 Consensus

We know (from proposition 1) that if the same sequence of blocks is applied to the same genesis state by two nodes then they will arrive at the same minimal state. This is exactly what we want to achieve. So how we ensure that the same sequence of blocks are applied by each node? This is the consensus problem.

Firstly note that we can only hope to achieve this for nodes who follow the same protocol strictly. We refer to such nodes as *honest* and the method to obtain consensus as the *consensus protocol*. Nodes not following the protocol are called *byzantine* nodes and could be doing so if they are corrupted by an attacker or due to software bugs.

Computer scientists have studied consensus protocols since early 1980s. A lot of interesting results were obtained. 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 [2]). Consensus in open networks with unknown number of participants is pretty new and very hard question.

For 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 exponentially with  $k$ .
2. Chain Quality - a party having  $x\%$  of voting power should produce no more than  $\alpha x\%$  blocks in a long run, where  $\alpha$  is constant  $\geq 1$ .

3. Chain Growth - over time, the blockchain should always grow. Thus, as long there is an honest majority in the network (in terms of computing power, not the number of nodes), a transaction sent by a honest node should eventually make it to the blockchain.

### 3.4.1 Proof-of-Work

Proof-of-Work (PoW) as a consensus protocol was used for the first time in Bitcoin, as described in Nakamoto's seminal paper [7]. However, the concept of PoW came before bitcoin in the form of Hashcash [?]. PoW forms the core of Bitcoin, Ethereum and many other cryptocurrencies. The basic idea here is to force miners to iterate over 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. The input to the hash is the root of the Merkle tree of transaction hashes as an authenticator of the transactions included. Additionally, we also want a block to refer to a previous block, so we include the hash of the previous block. This ensures that in order to replace a particular block the attacker has to replace all its descendants. As some amount of work is needed to generate each block, the amount of work needed to defeat an honest chain increases exponentially with depth.

With PoW, it is impossible to make a false claim of successful block generation. Such a claim is easily falsifiable, as calling the hash function is very cheap. Thus, PoW also offers protection against Sybil attacks [].

### 3.4.2 Proof-of-Stake

PoW has some disadvantages. Firstly, a lot of resources are needed to obtain a sufficient guarantee of security. Secondly, during early days, a PoW currency can be destroyed by an attacker possessing sufficient computing power.

The question then is: can we have protection against Sybil attacks without spending computational resources? The simplest way to achieve this is to use cryptocurrency tokens themselves as anti-Sybil tools. That is, the probability to generate a block is proportional to the amount of cryptocurrency a node holds. This is called Proof-of-Stake (PoS).

### 3.4.3 Transaction History

Earlier we talked about a blockchain. However, in all the global networks, collisions are possible (i.e., two blocks referencing the same parent).<sup>1</sup> So in reality, it is a blocktree that exists.

In most cases, the existence of blocktree is ignored and a node stores only one valid chain. For multiple contenders, each blockchain has some score (e.g.,

---

<sup>1</sup>To prevent this, we would need to have a global lock or synchronous rounds and then some kind of leader election.

cumulative difficulty). If a chain with higher score is found, the older is discarded. In these cases, a node sees a blocktree only during switching from one branch of it to another.

One might ask that instead of discarding the lower score chains, why don't we store them as additional PoW enhancing the persistence of some common ancestor from the main chain. There are some proposals that consider this and explicitly use a blocktree. For example, in GHOST scoring function [10] a chain with heaviest tree wins.

#### 3.4.4 Full Node View

A full node is a node which holds at least some state that is 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 also store the history from which the minimal states have been generated. In addition, a full node contains two more entities: (1) *Memory pool* containing transactions not yet included into blocks (and there is no guarantee of inclusion for them), and (2) *Vault* containing some node-specific information 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 a *wallet* which contains private keys as well as transaction associated with them.

With these four entities defined, we can explicitly state a node view type now:  $\text{node-view} = \langle \text{history}, \text{minimal-state}, \text{vault}, \text{memory-pool} \rangle$ .

The quadruple could be modified by applying an offchain transaction or a block coming from local code or from a remote peer. We can state some rules of 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 of updates is strictly needed. If history is producing rollback side-effect, other parts must handle it properly before applying an update. This sounds trivial, but in fact many years have been spent uncovering bugs related to inconsistency and read-when-update issues.

### 3.5 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 current UTXO set (SPV nodes go even further and store only those UTXOs that 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<sup>2</sup>, 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<sup>3</sup>) or sent to an output that cannot be provably spent<sup>4</sup>. Such unspent outputs serve no meaningful purpose to Bitcoin nodes and can be pruned from their UTXO set.

---

<sup>2</sup>[https://en.bitcoin.it/wiki/Softfork#2015\\_BIP66\\_Blockchain\\_Fork](https://en.bitcoin.it/wiki/Softfork#2015_BIP66_Blockchain_Fork)

<sup>3</sup>[https://en.bitcoin.it/wiki/OP\\_RETURN](https://en.bitcoin.it/wiki/OP_RETURN)

<sup>4</sup>Example: <https://blockchain.info/address/1CounterpartyXXXXXXXXXXXXXXXXXXXXXXXXXXXXUWLpVr>

### 3.5.1 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). In this method, a node only verifies headers from the genesis block onwards 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.<sup>5</sup>

### 3.5.2 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 (i.e., minimal states) 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?]

### 3.5.3 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

---

<sup>5</sup><http://bitcoin.stackexchange.com/a/11721/2075>

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. Ethereum is one example that follows the GHOST strategy.

### 3.5.4 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 whose 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 micro-blocks 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 are split into a 40-60 ratio, where 40% goes to the owner of the current PoW and the rest to the next PoW. This is to dissuade attacks where miners try to fork the blockchain for stealing the previous block’s transaction fee.

### 3.5.5 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.

### 3.6 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. 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 (say Z), such that  $1\text{ Z} = 1\text{ BTC}$ . 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 is computationally binding and perfectly hiding 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).<sup>6</sup>

<sup>6</sup>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  $\text{VERIFY}(\pi, R, N) = \text{true}$  iff Alice

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  $z \in \text{Zero-Pool}$ ”. 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  $s_A$ s 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  $z \in \text{Zero-Pool}$ ” 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 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 ones. 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.

---

‘really knows  $P$ ’. The concept of ‘knowing’ is captured as follows: (2) If Alice can be fooled into using a **maliciously crafted**  $R$ , then  $\pi$  will reveal  $P$  with a high probability. (3) Additionally, Alice can construct pairs  $(\pi', R')$  such that  $\text{VERIFY}(\pi', R', N) = \text{true}$  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  $(\pi, R)$  pair, there is no way of knowing if it was generated using (1) or (3).



### 3.7 Conclusion

# Chapter 4

## Scorex

### 4.1 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 implement all the abstract interfaces, or just some of them and use ready modules for missing parts.

#### 4.1.1 \*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 [9].

Scala is functional, modular and also object-oriented language. Such a mix of concepts means different developers can follow very different styles.

There are several reasons for choosing Scala:

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.

[TODO: quick description]

### 4.2 Transactional Layer

Transactional layers describes blockchain semantics. That is, what is a minimal state enough to validate incoming transactions, what is transaction and how

processing of it affects the state, how transactions are protected from being spent by non-allowed party, how wallet is organized etc

#### 4.2.1 \*Scala: Traits and Type Parameters

The basic piece in a sufficiently large Scala codebase is a *trait*. Trait is an abstract(so non-instantiable) interface describing functions and values a concrete implementation(class or object) must provide to its users. We can also think about trait as of type and also a parametrized module. A trait could be parametrized not only by values, but also by types.

Quick example: [TODO: example]

#### 4.2.2 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 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 ByteSerializable
```

In most useful scenarios a proposition should be addressable. That is, it could be addressed by some identifier, or identifiers.

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 ByteSerializable {  
  def isValid(proposition: P, message: Array[Byte]): Boolean  
  ...  
}
```

#### 4.2.3 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 value(how many system tokens are associated with it), and some proposition which protects a box from being spent by anyone but a party(or parties) knowing how to satisfy the proposition.

#### 4.2.4 \*Scala: Sum Types and Try

#### 4.2.5 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.

#### **4.2.6 Wallet**

[TODO:]

#### **4.2.7 Memory Pool**

#### **4.2.8**

#### **4.2.9 Node View Holder**

#### **4.2.10 Transactional Module**

### **4.3 Consensus Layer**

#### **4.3.1 Block**

#### **4.3.2 History and Blockchain**

### **4.4 Network Layer**

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

#### **4.4.1 Peer Discovery**

#### **4.4.2 Broadcasting Strategies**

#### **4.4.3 View Synchronizing**

### **4.5 Ready Modules**

There are few modules already implemented.

#### **4.5.1 Proof-of-Stake**

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

#### **4.5.2 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.

#### **4.5.3 Permacoin Implementation Module**

The module contains an implementation of Permacoin consensus protocol [6].

### **4.6 Conclusion**

# Bibliography

- [1] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc.", 2014.
- [2] 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.
- [3] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. *The Bitcoin Backbone Protocol: Analysis and Applications*, pages 281–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 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] Michael Nielsen. How the bitcoin protocol actually works.
- [8] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.
- [9] 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.