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 [3].

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.

[TODO: AC: we have more complex division now, fix the list below?] The main components of Scorex are:

1. Mechanism for creating a genesis block and subsequent blocks using some transferable tokens (transactions).
2. Mechanism for forming consensus as to when a block is accepted in the blockchain (proof-of-work, proof-of-stake, etc).
3. Mechanism for defining arbitrary rules as to when a token transfer is valid (Bitcoin UTXOs, NXT accounts, Namecoin domains).

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 as a Finite-State Machine

Formally, a blockchain represents a finite-state machine with state changes encoded in blocks and a copy of a blockchain representing a snapshot of this state machine. There are certain rules that must be followed to encode a valid state change. These are encoded in transactions. Additionally, older snapshots are considered tamper-resistant while fresh copies 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 transactions ensure that if all honest nodes start with the same initial state and apply the same transactions then they will have the same snapshots at any time. The task here is to ensure that the transactions 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 set of transactions 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 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, Etherium, 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. Etherium: 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:

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

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 $A_1 \overset{ref_1}{\rightarrow} x_1$ 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 $ref_i \rightarrow (A_i, x_i)$. 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. 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 $A_i \overset{ref_i}{\rightarrow} x_i$ from $(A_1 \overset{ref_1}{\rightarrow} x_1, A_2 \overset{ref_2}{\rightarrow} x_2, \ldots, A_n \overset{ref_n}{\rightarrow} x_n)$
   (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.

A miner confirms new transactions 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 Security and Privacy

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

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

We additionally have some Bitcoin specific attacks:

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.

3.5 Scalability

Before discussing scalability, we describe block construction in more detail. 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 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 and the corresponding difficulty level.
4. 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.

Why do we need to store? As discussed earlier, at the minimum each node must store the unopened boxes in a UTXO database. 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.[3]

[...]

[3]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.

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 $name \rightarrow value$ 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

[TODO: public key cryptography]
[TODO: hash functions]

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 [5] 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 [8] 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 [7] a chain with heaviest tree wins.

4.4.4 Consensus Layer Generalization

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

fully prunable outputs ZCash

4.8 Conclusion

4.9 Further Reading

Proof-of-Work and blockchain were introduced in the foundational Bitcoin whitepaper [5].

Overview of Bitcoin P2P layer along with description of possible Eclipse attacks(partly fixed to the moment) against it could be found in [2]

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

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 [6].

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 readible code.
4. Scala has powerful constructs for concurrency.

[TODO: quick description]

5.2 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

5.2.1 *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]

5.2.2 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 BytesSerializable
```

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

5.2.3 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.

```
trait Box[P <: Proposition] extends BytesSerializable {
  val proposition: P

  val value: Long

  val id: Array[Byte]
```

```
}
```

5.2.4 *Scala: Sum Types and Try

5.2.5 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.

```
abstract class Transaction[P <: Proposition, TX <: Transaction[P, TX]] extends E
  def validate(state: MinimalState[P, TX]): Try[Unit]
  def changes(state: MinimalState[P, TX]): Try[StateChanges[P]]

  ...
}
```

We are defining functional interface for minimal state below:

```
trait MinimalState[P <: Proposition, TX <: Transaction[P, TX]] {
  def version: Int

  def isValid(tx: TX): Boolean = tx.validate(this).isSuccess

  def processBlock(block: Block[P, _, _]): Try[Unit]
  def rollbackTo(height: Int): Try[Unit]

  def closedBox(boxId: Array[Byte]): Option[Box[P]]

  ...
}
```

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.

5.2.6 Wallet

[TODO:]

5.2.7 Memory Pool

```
trait UnconfirmedTransactionsDatabase[TX <: Transaction[_, TX], TData <: Transac
  def putIfNew(tx: TX): Boolean
```

```
  def all(): Seq[TX]

  def getById(id: Array[Byte]): Option[TX]

  def packUnconfirmed(): TData

  def clearFromUnconfirmed(data: TData): Unit

  def onNewOffchainTransaction(transaction: TX): Unit

  def remove(tx: TX)

}
```

5.2.8 Transactional Block Data

```
trait TransactionalData[TX <: Transaction[_, TX]] extends BytesSerializable with
  val mbTransactions: Option[Traversable[TX]]

  val headerOnly = mbTransactions.isDefined
  ....
}
```

5.2.9 Transactional Module

5.3 Consensus Layer

5.3.1 Block

```
class Block[P <: Proposition, TData <: TransactionalData[_ <: Transaction[P, _]]
```

5.3.2 History and Blockchain

5.4 Network Layer

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

5.4.1 Peer Discovery

5.4.2 Broadcasting Strategies

5.4.3 View Synchronizing

5.5 Ready Modules

There are few modules already implemented.

5.5.1 Proof-of-Stake

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

5.5.2 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.

5.5.3 Permacoin Implementation Module

The module contains an implementation of Permacoin consensus protocol [4].

5.6 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] 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.

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

[4] 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.

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

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

[7] 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.

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