The Blockchain and the Scorex Tutorial

Alexander Chepurnoy and an anonymous contributor

Jun-Dec, 2016

Chapter 1

Executive Summary

This paper describes the Scorex project and how it can be used to implement blockchain protocols (such as Bitcoin). Scorex is a framework written in Scala with loosely coupled components. The intended audience is developers willing to create or experiment with such blockchain cores. 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 [19].

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

1. *Bitcoin:* Decentralized currency [24].

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

2. *Namecoin:* Decentralized DNS[1] (stores *key* $\rightarrow$ *value* mappings). *Purpose:* Decentralization of DNS, difficult to sensor or shut down.

3. Ethereum: Smart contracts. *Purpose:* Enable trustless computing via decentralization.

1.1 Organization

This document is organized as follows. In Chapter 2, we give an introduction to Bitcoin. In Chapter 3, we describe basic building blocks of a blockchain system. In Chapter 4, 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.

[1] http://namecoin.info/

Chapter 2

Overview of Bitcoin

Here we attempt to give a high-level overview of Bitcoin. Some of the terms we will discuss are: *transaction*, *input*, *output*, *reference*, *block* and *confirmation*. We will revisit these terms in next chapters providing more abstract definitions.

In most cases of Bitcoin, funds are exchanged between *addresses* which are hashes of public keys. We will sometimes use the terms 'address' and 'public key' interchangeably. However, it should be remembered that addresses are one-way hashes of public keys.

2.1 Transaction

A transaction is a string sent over the peer-to-peer network that results in funds being transferred from one or more addresses to other addresses. A transaction consists of *inputs* (source of funds) and *outputs* (destination of funds). The smallest unit of Bitcoin is Satoshi (1 Bitcoin = 10^8 Satoshis), which is what we will use in our text.

Suppose an address A owned by Alice has x Satoshis. Alice wants to send $y \leq x$ Satoshis to address B owned by Bob. This can be done by creating a transaction with A as the input and B as one of the outputs. The transaction additionally has information for other nodes to quickly validate that A indeed has x Satoshis to spend. This is done by adding a reference to some previous transaction's output where A actually received those x Satoshis. Let us call this $ref_{A \leftarrow x}$. The outputs are constructed as follows: The first output sends y Satoshis to B. Alice then sets a transaction fee t and adds a second output sending the remaining $z = x - y - t$ Satoshis to her *change address* C. The change address is simply any address owned by Alice and could also be A.

The message "($ref_{A \leftarrow x}$, take x from A), (put y in B), (put z in C)" together with its signature under A is considered a 'transaction'.

Let us define the following notation:

- $ref_{X \leftarrow x}$ is the message "($ref_{X \leftarrow x}$, take x from X)". This is an input.

- $X \leftarrow x$ is the message "put 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 $m = (ref_{A \leftarrow x}, B \leftarrow y, C \leftarrow z)$.

A transaction can have several inputs and outputs. The only restriction is that the total amount of funds destroyed (via the inputs) must be greater than or equal to the total amount of funds created (via the outputs). Any difference is considered a *transaction fee*.[1] More formally:

Definition 1. *Define m to be the message*

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

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:

$$tx \stackrel{def}{=} (M, \sigma_{A_1}(M), \sigma_{A_2}(M), \ldots, \sigma_{A_n}(M)) \tag{2.1}$$

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.*

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:

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

The ordering of the signatures in tx is determined from the ordering of messages inside M (which is fixed due to the signatures). Thus each i uniquely defines one output.

To prevent double spending, Bitcoin requires that a reference should be used in a transaction at most once. This is enforced as follows. An unused reference is also called an unspent output (UTXO for short). Each client maintains a set of UTXOs. 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 not processed.

[1] Transaction fee is collected by someone who confirms the transaction and is used both as an incentive for participation and a deterrent for spam attacks.

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 their (private) 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 of type:

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

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 $ref_{A_i \leftarrow x_i} \rightarrow (A_i, x_i)$. Note that in order to validate transactions and participate in the protocol, maintaining this UTXO database is necessary. Once a node has bootstrapped and downloaded some length of the blockchain, it can quietly discard those downloaded blocks once its UTXO database resulting from those blocks has been generated (it could store a few recent blocks to handle rollbacks). It can keep updating this database as new blocks are received. 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 be 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 Satoshis 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.

The input script contains the signature and public key, while the output script contains instruction to verify the signature for the address holding the Satoshis. 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 $(ref_{A_1 \leftarrow x_1}, ref_{A_2 \leftarrow x_2}, \ldots, ref_{A_n \leftarrow x_n})$

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

(a) Load signature σ_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 batches 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, only someone who has invested a large amount of CPU cycles get to select which set of transactions to commit in the next batch. This node is called a *leader* or *miner* and is selected at each update.

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.

A miner confirms transactions as follows. First the miner constructs a block b as a sequence of bytes containing the following data:

1. A number of unconfirmed transactions along with one reward transaction (known as the *coinbase transaction*).

2. Hash value 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 forgotten.

Proof of Work: Recall that we consider a block to be mined only if its hash value 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 2^x CPU cycles on average. 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 truncated 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 $\mathit{difficulty} = 2^x$ and $\mathit{target} = 2^{k-x}$. If two competing blockchain are presented, honest nodes select the ones with the highest *cumulative difficulty* (one where the summation of difficulties is the largest).

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 it was observed in Bitcoin during periods of exponential hashrate growth).

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.

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. 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[3] 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[4]:
02000000b6ff0b1b1680a2862a30ca44d346d9e8910d334beb48ca0c0000000000000000009d10aa52ee949386ca9385695f04ede270dda20810decd12bc9b048aaab314724d95a5430c31b18fe9f0864

The different sections are identified by alternating underlines:

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

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 adding a null element if necessary. Then we pair the nodes to form $n/2$ pairs. Each pair (L, R) is concatenated and its hash $\texttt{SHA256}(\texttt{SHA256}(L||R))$ forms the parent for the next iteration. This process is repeated until the root is reached.

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

[3]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).

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

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 then 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 [16], can be achieved if the underlying proof-of-work (PoW) based consensus system satisfies the following two properties [16]:

1. *Common prefix:* If all honest participants remove the top (newer) k blocks from their chains for a large enough k then all of them will share a common prefix. In other words, the chains held by honest miners will either be identical or be contained in the others.
2. *Chain quality:* There is some minimal integer $\lambda \geq 1$ such that, if the combined computing power of honest parties is λ times that of the adversary, then a non-negligible amount of blocks generated by honest parties will make it into the chain.

If the difficulty level is sufficiently high and the network synchronization time (time between a new block being injected and reaching all participants) is short compared to average block generation period (10 minutes in Bitcoin) then the protocol offers high security. On the other hand, security is weakened if any of the following conditions hold [16]: (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 [7]. 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 [33].

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 funds being sent, then it could lose funds [11].

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.

2.6 Further Reading

The article "How the Bitcoin protocol actually works" by Michael Nielsen [25] discusses various problems that led to the creation of Bitcoin. The original whitepaper by Satoshi Nakamoto [24] provides a good description on the concepts and details can be found in the book "Mastering Bitcoin" [1].

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 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) without any trusted arbiter. Peers issue "authenticated events" (signed messages of some semantics). For example, they can send out signed payments or they can register a $name \rightarrow value$ pair in a shared database (e.g. in order to register a domain). 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 a 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

asc9 notes : will be improving this section as per Mario's comments

Below are the cryptographic primitives needed to construct blockchain systems described in this chapter.

1. *One-Way Hash (OWH) functions:* A *One-Way Function* is easy to compute in the forward direction but difficult in the reverse. A *Hash Function* maps arbitrary sized strings to fixed size ones. Thus, a One-Way Hash (OWH) function efficiently maps arbitrary size strings to fixed sizes and is hard to reverse. Let there be an efficiently computable mappng $H : \{0,1\}^* \mapsto \{0,1\}^k$. We say H is a OWF if it is hard for any Poly-time attacker to compute two distinct values x, y such that $H(x) = H(y)$. This property, *collision resistance*, also implies *pre-image resistance*, the inability of an attacker to compute any pre-image x given $H(x)$.

2. *Signature scheme:* A signature schemes consists of three algorithms:

   (a) KEYGEN(k): This takes as input a security parameter k and outputs a (public-key, private-key) pair (pk, sk).

   (b) SIGH(sk, m): This takes as input a private-key sk and a message m. It outputs a signature s.

   (c) VERIFY(pk, m, s): This takes as input a public-key pk, a message m and a signature s. It outputs True or False.

   Correctness requires that for all (pk, sk) pairs output from KEYGEN and for all m, it must hold that VERIFY(pk, m, SIGN(sk, m)) = True.

   Security requires that this is the only way to generate valid signature. Formally, we define the security using a game where we give the attacker access to SIGN oracle for some private key sk and require him to generate a valid signature (one that outputs True on VERIFY with pk) on a message that was never queried to this oracle. This is called UF-CMA (unforgeability under adaptive chosen message attack) [29].

   We actually require a stronger property called *strong unforgeability* (SUF-CMA) [6]. This requires that an attacker having access to a SIGN 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. *Commitments:* Some proposals (such as ZeroCash [27]) use commitments, which allow a user, say Alice, to commit to a message m without revealing it. Instead she sends a commitment $c = \text{COMM}(m, r)$, where r is a random value. Later on, she can 'open' her commitment by revealing (m, r). A verifier then computes $\text{VERIFY}(c, m, r)$ and outputs either True or False. For correctness, we require that for all (r, m), it must hold that $\text{VERIFY}(\text{COMM}(m, r), m, r) = \text{True}$.

   Commitments can be either: (1) *computationally binding* (so that Alice can cheat and present a different m' if she has a large (but impractical) amount of computing power) and *perfectly hiding* (the commitment reveals no information about m even to someone with infinite computing power), or: (2) *perfectly binding* (Alice cannot cheat even if she has infinite computing power) and *computationally hiding* (someone with very large but finite computing power can learn m) opening [10]. A commitment scheme cannot be both perfectly hiding and perfectly binding.

4. *Non-Interactive Zero-Knowledge (NIZK) Proofs:* Let N be an instance of any NP 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 (of knowledge): (1) Given a **random** string R, Alice can construct a short proof $\pi = \text{PROOF}(P, R)$ such that $\text{VERIFY}(\pi, R, N) = \textit{true}$ iff Alice 'really knows P'. The concept of 'knowing' is captured as follows: (2) If Alice can be fooled into using a **maliciously crafted** R, then π will reveal P with a high probability. (3) Additionally, Alice can construct pairs (π', R') such that $\text{VERIFY}(\pi', R', N) = \textit{true}$ without knowing any witness P as long as **she selects** R'. Thus, the generation of R is of outmost importance because if the verifier selects R, the secret is leaked and if the prover selects R then the proof is not convincing. If given the transcript of a proof, we cannot distinguish between the cases (1) and (3), then the protocol is zero knowledge, intuitively because given some (π, R) pair, there is no way of knowing if it was generated using (1) or (3). An example of a NIZK proof is given in Appendix A.

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 is 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 state elements 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 a 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.

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 flaw. 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. An empirical analysis of Namecoin is given in [18].

3.3.4 Nxt and Ethereum

Bitcoin has the concept of immutable UTXOs and a Script-based structure for validating public-key and signatures. We can also conceive of a simpler model model where the ledger maps public keys to their balances and transactions to Satoshis transferred between the accounts. We can also have hard-wired semantics that validate public-key and signatures instead of Script code.

Nxt and Ethereum are two examples of such designs, which have use the term *account* to define a 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 explanation, 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 (Nxt solution is different and not covered in this document). 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 incremented nonce and update balance is created. Thus, even Etherium 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-resistance*. That is, it must be computationally hard to generate different sequences resulting in the same root. Any data structure that supports this is called an *authenticated data structure*, an example of which is a Merkle tree.

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

Alex notes : Sounds unfinished, we need to say this is just about transactions and state.

Let us summarize the common features of various blockchains. At the core is a *minimal state* that encodes some shared information between nodes. The minimal state is modified via *transactions*, which contain instructions for modifying the minimal state. This idea is broken down into several components as follows:

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, hardware crashes etc.

Computer scientists have been studying consensus protocols extensively 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 [15]). Randomized consensus in open networks with unknown number of participants is a hard problem.

For blockchain consensus protocols, we can state following properties [16]:

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 $\alpha \geq 1$ is a function of x.

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 [24]. However, the concept of PoW came before bitcoin in the form of Hashcash [2]. 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 a header comprising of a nonce, the current difficulty target, the root of the Merkle tree of transaction hashes (as an authenticator of the transactions included), and 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 where an attacker creates a large number of fake identities [12].

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) [20, 21].

3.4.3 Proof-of-Burn

Proof-of-burn is a method for distributed consensus and an alternative to PoW

and PoS. It can also be used for bootstrapping one cryptocurrency off of another. The idea is that miners should show proof that they destroyed some coins - that is, sent them to a verifiably unspendable address [32]. This is expensive from their individual point of view, just like proof of work; but it consumes no resources other than the burned underlying asset. However, to date, all proof of burn cryptocurrencies work by burning PoW-mined cryptocurrencies, so the underlying mechanism remains PoW.[1]

3.4.4 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).[2] 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., 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 [31] a chain with heaviest tree wins.

3.4.5 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: node-view = ⟨history, minimal-state, vault, memory-pool⟩.

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.

[1] https://en.bitcoin.it/wiki/Proof_of_burn

[2] To prevent this, we would need to have a global lock or synchronous rounds and then some kind of leader election.

- 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 about 1.5GB.

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 the complete blockchain. In fact, all rational nodes follow SPV.

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[3], 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[4]) or sent to an output that cannot be provably spent[5]. Such unspent outputs serve no meaningful purpose to Bitcoin nodes and can be pruned from their UTXO set.

3.5.1 SPV

SPV (Simple Payment Verification) was a protocol proposed in the original Bitcoin paper as an alternative to the full-node protocol [24]. 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 a unique blockchain is identified not by its height (the number of blocks before it since the genesis block) but rather its depth (the number of blocks mined since a client booted up).

Implementations of SPV nodes use a protocol extension called *Bloom filters*, described in Bitcoin Improvement Proposal (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.[6]

3.5.2 Rollerchain

Rollerchain [9] (RC) is a proposal aimed to (1) reduce storage and (2) incentivizing nodes to store some number of last blocks and also state snapshots (minimal states in a standard representation). There are two main parameters of the scheme, n and k. The parameter n specifies number of blocks (and also state snapshots) a network aims to store collectively (so a sufficiently large network stores last n full blocks and also state snapshots), while each miner is storing k state snapshots. To create a successful PoW, nodes must present proofs for the k state snapshots determined via a hash of their public key and the current

[3] https://en.bitcoin.it/wiki/Softfork#2015_BIP66_Blockchain_Fork
[4] https://en.bitcoin.it/wiki/OP_RETURN
[5] Example: https://blockchain.info/address/1CounterpartyXXXXXXXXXXXXXXXUWLpVr
[6] http://bitcoin.stackexchange.com/a/11721/2075

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.

3.5.3 GHOST

The GHOST (Greedy Heaviest-Observed Sub-Tree) protocol is designed for blockchains with very fast block times [30]. As the network becomes less synchronized and blocks are generated fast, a significant amount of PoW will not make it into the main chain due to orphans. This leads to reduced security. The GHOST proposal makes use of the additional orphaned blocks to strengthen the network instead of rejecting them. Thus, two different branches will be compared not just by length but how 'heavy' they are in terms of PoW. A GHOST protocol has various mechanisms for rewarding orphans. Etherium is one example that follows the GHOST strategy.

3.5.4 Bitcoin-NG

Bitcoin-NG [13] (or simply NG) was proposed to solve two main issues in Bitcoin:

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

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

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

The main idea in NG is to remove the link between transaction confirmations and the *leader selection*. A leader is essentially someone who's PoW is accepted. In NG, the leader is still selected based on PoW but the block selecting the leader (the *key-block*) does not contain transactions. Rather the currently selected leader (i.e., the public key) has the sole power to decide the transactions for the blockchain in several *micro-blocks* until the next leader is selected. The leader adds 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 is 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 [22] 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 [8] 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.

3.5.6 Composite Signatures

Composite signatures is an approach proposed in [28] for enhancing anonymity (an earlier version of the paper calls it (One-Way Aggregate Signatures (OWAS)). Composite signatures are based on another primitive called Aggregate Signatures [5]. In this, several individual signatures can be combined into one short object – the aggregate signature – that can be verified against all the (public-key, message) pairs of the individual signatures and the verification yields true if and only if all the individual (public-key, message, signature) triplets verified

correctly before aggregation. This itself has the advantage of reducing the sizes of very large transactions by compacting the signature.

Aggregate signatures, however, can be tweaked so that the aggregation process becomes one-way; given just the aggregate signature, it is very hard to extract the constituents. This gives someone the ability to aggregate two unrelated signatures so that no one can later separate them. Composite signatures use this idea to combine a signed input (which releases Satoshis from an address) with an unrelated output (which consumes those Satoshis) such that at the time of signing, the holder of the Satoshis may not even know who the recipient is. For instance, the sender could simply sign a input releasing the Satoshis and hand it over to some (possibly unknown) receiver over a private channel without specifying in the signature where the Satoshis should go. This is equivalent to handing out a "blank cheque". The receiver can then add an arbitrary signature sending those Satoshis to some address. This can be further combined with more such transactions over a private network before sending the large transaction to the public network. There can be various mechanisms of doing the private computation (such as sending it to an aggregator).

3.6 ZeroCash

ZeroCash (ZCash) [27] is a blockchain based protocol designed for privacy. Since the data in the Bitcoin blockchain is public, true anonymity is not present. ZCash can be considered an extension of ZeroCoin (ZCoin) [23], an earlier proposal. In order to describe ZCash, we will first describe ZCoin.

Assume that all coins to be obfuscated are of a single denomination, say 1 BTC. ZCoin has its own currency (say Z), such that 1 Z = 1 BTC. There is a pool of obfuscated Z coins called *Zero-Pool*. Alice wants to anonymize her Bitcoin represented by serial number c_A. In order to do that, she begins by exchanging c_A with an equivalent Z coin as follows. She generates a secret serial number s_A and another secret r_A. Then she creates a commitment z_A to s_A using r_A as randomness. That is, $z_A = \text{COMM}_{r_A}(s_A)$, which is the Z coin Alice will use in exchange for c_A. All such exchanged coins are automatically added to the Zero-Pool. The commitment 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).

Alice creates a spend by revealing s_A (but not r_A) and creates a NIZK proof of the NP statement "I know r such that $z = \text{COMM}_r(s_A)$ and $z \in$ Zero-Pool". The process of creating a commitment and revealing only one input (but never opening them completely) may seem counter-intuitive, but this in fact makes the statement provable by efficient NIZKs, because the circuit for computing the commitment can be efficiently coded into an NP language. The spend could have been done even without revealing s_A. However, this is needed for the next property – security – so that Alice should not be able to spend z_A twice. This

is achieved by keeping track of spent s_As in a public ledger, kept by each client.

ZCash is an extension of ZCoin with the following differences:

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

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

3. ZCash uses a variant of NIZKs called zkSNARKs (succinct non-interactive arguments of knowledge) [4, 3]. A zkSNARK is essentially a NIZK with the proof size and verification-time O(1), and secure against only a computationally bound adversary. Note that the zkSNARKs used in the ZeroCash paper [27] require pre-computation.

4. ZCash allows coins of arbitrary denominations instead of 1 BTC as in ZCoin. 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.

Scalability: We can see that both ZCoin and ZCash have 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, ZCoin must store both the spent and unspent outputs. The unspent outputs is simply the Zero-Pool.

3.7 Conclusion

In this chapter we discussed various issues and concepts of a blockchain system such as Bitcoin. We also briefly described various Bitcoin-like systems that can be implemented with Scorex. We summarize this chapter below:

Scorex generalizes the concept of the UTXO Set in Bitcoin to a *Minimal State*, the minimum amount of information needed by any node to validate transactions. Each Bitcoin UTXO is equivalent a locked *Box* that can be opened with a key. The lock can be considered a *proposition* and the key a *proof*. The proof in Bitcoin is simply the signature under the input public key and the proposition is a statement that validates the signature.

Some of the alternate blockchain designs we described were:

1. *Consensus models:* Proof-of-Work, Proof-of-Stake, Proof-of-Burn.

2. *Chain selection:* Longest chain, highest cumulative difficulty, heaviest branch of a blocktree (GHOST).

3. *Application:* Currency (Bitcoin), Domain names (Namecoin), Smart contracts (Ethereum).

4. *Fast confirmations and larger throughput:* Bitcoin-NG, ByzCoin.

5. *Lightweight clients and reduced storage:* SVP clients, Rollerchain.

6. *Anonymity:* ZeroCash, ZeroCoin, Composite Signatures.

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 top of Scorex is to provide implementations for all the abstract interfaces (possibly reusing code previously written).

Scorex is intended to build a full-node implementation, though other security models are possible (e.g. SPV).

4.1.1 Scala Language

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

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

For a good introduction into the Scala language, please refer to [26].

4.2 A Node View And Its Modification

The goal of a fullnode node is to get the same *minimal state* (see Section 3.3.1 for details) eventually for a version h as other honest nodes and also obtain some

specific information for the node users from the history and the state. To accomplish the goal, as described in the section 3.4.5, the node has a local view of the world comprising of four entities: $node\_view = \langle history, minimal\_state, vault, memory\_pool \rangle$. In Scorex we assume there is a dedicated reactive component holding a view node with a direct modification of it from outside prohibited.

This component, a *node view holder*, modifies a node view it holds on getting whether an offchain object, so an offchain transaction, or an on-chain object (a block, or key-block/micro-block in Bitcoin-NG, a blockheader/fullblock/state-snapshot in RollerChain etc.) we call *persistent node view modifier*. Type signatures of a trait and its sub-components are as follows:

```
trait NodeViewHolder[P <: Proposition, TX <: Transaction[P],
        PMOD <: PersistentNodeViewModifier[P, TX]] extends Actor {

  type SI <: SyncInfo
  type HIS <: History[P, TX, PMOD, SI, HIS]
  type MS <: MinimalState[P, _, TX, PMOD, MS]
  type VL <: Vault[P, TX, PMOD, VL]
  type MP <: MemoryPool[TX, MP]

  type NodeView = (HIS, MS, VL, MP)
```

In the snippet above we can think about types and traits as of modules, where each module is possibly parametrized with other modules (including self, this pattern is called *F-bounded polymorphism []* and being using widely in the Scorex). A concrete parameter has to be derived from an upper-bound abstract interface (for example, concrete minimal state representation has to be derived from the *MinimalState* trait).

4.2.1 Node View Modifiers

A node view could be modified by applying a *node view modifier* to it. The assumption we made in the design is that there are two kinds of node modifiers. One is *offchain transaction* which is not a part of history log and so to be processed by a memory pool and a wallet only. Another kind is about a hierarchy of *persistent node view modifiers*. In the simplest Bitcoin design the hierarchy is just about a Proof-of-Work block. In Rollerchain the hierarchy of log elements is about three kinds of objects: a blockheader, a state snapshot and a fullblock; with different validation and application rules.

It is supposed that all the modifiers have identifiers of the same length (to be set via *application.conf* file with default value of 32 bytes). We argue that the assumption is reasonable as usually identifier is an output of a single hash function.

Implemented functionalities are getting an identifier and also getting a binary representation of a modifier and each kind of modifier should provide its identifier also:

```
trait NodeViewModifier extends BytesSerializable with JsonSerializable {
  ...

  val modifierTypeId: ModifierTypeId

  def id: ModifierId

  lazy val bytes: Array[Byte] = ...
}
```

We assume log elements in the persistent node view modifiers hierarchy should also refer to a parent. A persistent modifier also can carry transactions (but that is optional):

```
trait PersistentNodeViewModifier[P <: Proposition, TX <: Transaction[P]] extends

  def parentId: ModifierId

  // with Dotty is would be Seq[TX] | Nothing
  def transactions: Option[Seq[TX]]
}
```

4.3 The State Design

4.3.1 A Minimal State

A minimal state is a data structure which deterministically defines whether an arbitrary transaction is valid and so applicable to it or not. See section 3.3.1 for details.

A signature of the *MinimalState* is

```
trait MinimalState[
    P <: Proposition,
    BX <: Box[P],
    TX <: Transaction[P],
    M <: PersistentNodeViewModifier[P, TX],
    MS <: MinimalState[P, BX, TX, M, MS]
] extends NodeViewComponent{
  self: MS =>
  ...
```

Per our definition, we define methods to validate a transaction:

```
  def validate(transaction: TX): Try[Unit]

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

The trait also has methods to get a closed box (see section 4.3.3), to apply a persistent modifier and to roll back to a previous version:

```
  def closedBox(boxId: Array[Byte]): Option[BX]
  def applyModifier(mod: M): Try[MS]
  def rollbackTo(version: VersionTag): Try[MS]
```

4.3.2 Propositions and Proofs

Blockchain can be thought of a mechanism to control access to some protected objects (such as transaction outputs). An object is protected by a *proposition* and a *proof* for that proposition is needed in order to modify the object.

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

```
trait Proposition extends BytesSerializable
```

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

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

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

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

For a *ProofOfKnowledgeProposition* corresponding abstract proof is provided:

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

4.3.3 Box

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

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

  val value: Amount
  val proposition: P

  val id: Array[Byte]
}
```

4.3.4 A History

At the very basic level, Scorex does not have a notion of a blockchain. This is done in order to support blocktrees and other non-linear structures. So Scorex has an abstract concept of *history* defined by the trait *History*.

```
trait History[P <: Proposition,
        TX <: Transaction[P],
        PM <: PersistentNodeViewModifier[P, TX],
        SI <: SyncInfo,
        HT <: History[P, TX, PM, SI, HT]] extends NodeViewComponent {
                def isEmpty: Boolean

        type ModifierIds = Seq[(ModifierTypeId, ModifierId)]

        def applicable(modifier: PM): Boolean =
                openSurfaceIds().exists(_ sameElements modifier.parentId)

        def modifierById(modifierId: ModifierId): Option[PM]

        def append(modifier: PM): Try[(HT, Option[RollbackTo[PM]])]

        def openSurfaceIds(): Seq[ModifierId]

        def continuationIds(from: ModifierIds, size: Int): Option[ModifierIds]

        def syncInfo(answer: Boolean): SI

        def compare(other: SI): HistoryComparisonResult.Value
  }
```

The trait contains no any mention of block or a blockchain. In principle, it supports non-linear structures and many object types to make the log.

A node continuosly tries to get alternative (and possibly better) histories from the network around and also feed nodes with less developed history with locally stored modifiers. To provide an information on how much local history is developed, there is an abstract type *SyncInfo* mentioned in the type signature of the *History* trait. We assume that *SyncInfo* at least provides information about *starting points*, persistent modifier identifiers a node sends in order to help other node to understand its synchronization status. Usually it is about last block identifiers.

```
trait SyncInfo extends BytesSerializable {
  def answer: Boolean
  def startingPoints: Seq[(NodeViewModifier.ModifierTypeId, NodeViewModifier.ModifierId
}
```

Function *compare* is checking a *SyncInfo* instance got from another party and returns one of *HistoryComparisonResult* values provided below:

```
object HistoryComparisonResult extends Enumeration {
  val Equal = Value(1)
  val Younger = Value(2)
  val Older = Value(3)
```

```
  val Nonsense = Value(4)
}
```

If another peer is holding the same history, then *compare* should return *Equal*. If other peer's history is more developed than local then *compare* returns *Older*, if less developed - *Younger*. If from *SyncInfo* it is not possible to get other peer's hystory status, *compare* should return *Nonsense*.

Function *openSurfaceIds* returns last block if a history is linear, otherwise it returns last blocks from a blocktree etc. That is why it is returning a sequence of modifier identifiers (Seq[ModifierId]). The function *continuationIds* returns modifiers after given. Again, if the history is linear, *from* parameter is about just 1-element sequence (we can not easily enforce that with the Scala type system, so an implementation should check that in runtime), otherwise, from could be about elements of different chains. The function returns modifiers after provided if found, otherwise *None*.

Function *append* is trying to extend the history with a modifier given. No any attempt could be successful (for example, if history does not contain modifier's parent, or modifier is not valid, append must aborts). There is an optional side-effect of an application to be passed to other components of the node. That is a rollback, which provides information about a last modifier id *to* after the rollback (and before the application), sequence of modifier thrown away and also a sequence of blocks to be applied (after *to*):

```
case class RollbackTo[PM <: PersistentNodeViewModifier[_, _]](to: ModifierId, thrown: S
```

4.3.5 A Vault

A *vault* is a storage for node-specific information. Usually this is about wallet functionality. The information is to be obtained via scanning whether an offchain transaction or a persistent modifier. A rollback is surely possible. Thus the interface of a vault is as follows.

```
trait Vault[P <: Proposition,
            TX <: Transaction[P],
    PMOD <: PersistentNodeViewModifier[P, TX],
    V <: Vault[P, TX, PMOD, V]] extends NodeViewComponent {
  self: V =>

  type VersionTag = NodeViewModifier.ModifierId

  def scanOffchain(tx: TX): V

  def scanOffchain(txs: Seq[TX]): V

  def scanPersistent(modifier: PMOD): V

  def rollback(to: VersionTag): Try[V]
}
```

Scorex also contains *Wallet* interface derived from the *Vault*, parametrized with types of secrets and their public images also.

```
trait Wallet[P <: Proposition,
        TX <: Transaction[P],
        PMOD <: PersistentNodeViewModifier[P,TX],
        W <: Wallet[P, TX, PMOD, W]] extends Vault[P, TX, PMOD, W] {
  self: W =>

  type S <: Secret
  type PI <: ProofOfKnowledgeProposition[S]

  def generateNewSecret(): W

  def historyTransactions: Seq[WalletTransaction[P, TX]]

  def boxes(): Seq[WalletBox[P, _ <: Box[P]]]

  def publicKeys: Set[PI]

  def secrets: Set[S]

  def secretByPublicImage(publicImage: PI): Option[S]
}
```

4.3.6 A Memory Pool

A memory pool holds transaction not included into blocks yet so unconfirmed. We omit its functions here (as they are pretty trivial) and provide only the type signature of the trait.

```
trait MemoryPool[TX <: Transaction[_], M <: MemoryPool[TX, M]]
```

4.3.7 Block

It is possible to develop block-less system, the only requirement is to have a history which contains instances of persistence node view modifiers. Nevertheless, the basic interface for a block is provided as it is needed for almost all of imaginable application. In addition to *identifier* and *parent identifier* fields inherited from *PersistentNodeViewModifier*, a *Block* generic interface contains also a version (we assume block format could change with time) and timestamp.

```
trait Block[P <: Proposition, TX <: Transaction[P]]
  extends PersistentNodeViewModifier[P, TX] with JsonSerializable {

  def version: Version

  def timestamp: Timestamp

  ...
}
```

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

4.5 Examples

There are few examples included into the distribution.

4.5.1 Proof-of-Stake

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

4.5.2 Hybrid Proof-of-Work / Proof-of-Stake implementation

Bibliography

[1] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies.* " O'Reilly Media, Inc.", 2014.

[2] Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

[3] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90–108. Springer, 2013.

[4] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. Cryptology ePrint Archive, Report 2013/879, 2013.

[5] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 416–432. Springer, 2003.

[6] Dan Boneh, Emily Shen, and Brent Waters. Strongly unforgeable signatures based on computational diffie-hellman. In *International Workshop on Public Key Cryptography*, pages 229–240. Springer, 2006.

[7] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. Cryptology ePrint Archive, Report 2013/734, 2013.

[8] Miguel Castro et al. Practical byzantine fault tolerance.

[9] Alexander Chepurnoy, Mario Larangeira, and Alexander Ojiganov. A prunable blockchain consensus protocol based on non-interactive proofs of past states retrievability. *CoRR*, abs/1603.07926, 2016.

[10] Ivan Damgård. Commitment schemes and zero-knowledge protocols. In *Lectures on Data Security*, pages 63–86. Springer, 1999.

[11] Christian Decker and Roger Wattenhofer. Bitcoin transaction malleability and mtgox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.

[12] John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[13] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, 2016.

[14] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[15] 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.

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

[17] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for np. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 339–358. Springer, 2006.

[18] Harry Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of namecoin and lessons for decentralized namespace design. 2015.

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

[20] Aggelos Kiayias, Ioannis Konstantinou, Alexander Russell, Bernardo David, and Roman Oliynykov. A provably secure proof-of-stake blockchain protocol, 2016.

[21] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[22] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Poster: Bitcoin meets collective signing.

[23] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 397–411. IEEE, 2013.

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

[25] Michael Nielsen. How the bitcoin protocol actually works.

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

[27] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[28] Amitabh Saxena, Janardan Misra, and Aritra Dhar. Increasing anonymity in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 122–139. Springer, 2014.

[29] Bruce Schneier. One-way hash functions. *Applied Cryptography, Second Edition, 20th Anniversary Edition*, pages 429–459, 1996.

[30] Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. Cryptology ePrint Archive, Report 2013/881, 2013.

[31] 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.

[32] I Stewart. Proof of burn. bitcoin. i t, 2012.

[33] Michael Bedford Taylor. Bitcoin and the age of bespoke silicon. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 16:1–16:10, Piscataway, NJ, USA, 2013. IEEE Press.

A. NIZK Proofs

This section gives an example of a *Non-Interactive Zero-Knowledge* (NIZK) proof (of knowledge). We illustrate this by first giving an example of an *Interactive Zero-Knowledge Proof* of Knowledge and then convert this to a non-interactive proof.

An Interactive Proof: Assume that n is a public composite integer that is hard to factor. Let $y \in \mathbb{Z}_n^*$ be a public value. In the following 3-step protocol, a prover P proves to a verifier V, the *knowledge* of some secret x such that $y \equiv x^2 \pmod{n}$. Note that without the factors of n, not only is it hard to find any such x, but also it is hard to decide if x even exists. In other words, it is hard to decide if y is a *quadratic residue* modulo n.[1]

1. P generates $r \xleftarrow{R} \mathbb{Z}_n^*$ and sends $s = r^2 \bmod n$ to V.
2. V randomly selects a bit $b \xleftarrow{R} \{0, 1\}$ and sends b to P.
3. P sends $z = x^b r \bmod n$ to V, who accepts if $z^2 \equiv y^b s \pmod{n}$

In essense, depending on the bit b the prover is required to either reveal a square root of s or a square root of sy (but never both, since that would reveal x). First observe that the protocol is a *proof of knowledge* because if P can be "reset" (i.e., be forced to reuse randomness) then V can extract x by resetting P and giving it a different b in the second run of the protocol.

To see that it is zero knowledge, first note that if P indeed knows x then P can respond in the third step such that V accepts 100% of the time. On the other hand, if P does not know any such x than P can attempt to guess b and select s differently (i.e., $s = r^2/y^b$) and still make V accept in the third step (by setting $z = r$) as long as the guess was correct. Since the chance of correctly guessing b is 50%, therefore, a cheating P can make V accept only 50% of the time. Given the protocol transcript where V accepts, there is no way to decide if P indeed knew x because P still has 50% chance of cheating without knowing x. Hence the protocol is zero-knowledge.

We know that the probability of a cheating P succeeding in one run of the protocol is $1/2$. This implies that for m *parallel runs*, the probability of a cheating P succeeding is $1/2^m$, which can be made arbitrarily small by increasing m.

A Parallel Interactive Proof: We give the m-parallel round protocol below.

1. P generates $r_1, r_2, \ldots, r_m \xleftarrow{R} \mathbb{Z}_n^*$ and sends $s_i = r_i^2 \bmod n$ to V for $1 \leq i \leq m$.
2. V randomly selects m bits $b_i \xleftarrow{R} \{0, 1\}$ for $1 \leq i \leq m$ and sends all b_is to P.

[1] We could envision a different scenario where P proves to V that y is indeed a quadratic residue modulo n without necessarily proving that P 'knows' a square root of y. This is called a *proof of membership*. A proof of knowledge is stronger in that it additionally allows V to extract the square root of y if P's state can be reset to use the same randomness.

3. P computes $z_i = x^{b_i} r_i \bmod n$ and sends all z_is to V (for $1 \leq i \leq m$). V accepts if $z_i^2 \equiv y^{b_i} s_i \pmod{n}$ (for $1 \leq i \leq m$).

A Non-Interactive Proof: We will use the Fiat-Shamir heurestic [14] to convert the above to a NIZK Proof. In this method, V's challenge in the second step is obtained by applying a one-way hash function to the value sent to V in Step 1. This approach assumes that the hash function can be modeled using a random oracle.

P generates r_is and sets $s_i = r_i^2 \bmod n$. P then computes $h = H(r_1, r_2, \ldots, r_m)$ for a one-way hash function H that outputs m bits. P then emulates Step 2 by setting b_i as the i^{th} bit of h. Finally P emulates Step 3 and the entire transcript of the three emulated steps forms the NIZK proof.

NIZK proofs for NP: The above proof is for a particular NP language, the set of quadratic residues modulo n. Since this is not believed to be NP-Complete, we cannot use this to construct NIZK proofs for arbitrary NP statements. However, similar techniques can be used for *circuit satisfiability* (SAT), a problem known to be NP-Complete. One construction based on pairings is given in [17]. We create a NIZK proof for an arbitrary NP statement s by first reducing it to t, an instance of SAT. Then we give a NIZK proof of t, which is equivalent to a NIZK proof of s. Note that such reductions, although polynomial time, are inefficient in practice. Hence, we often use specialized NIZK proofs (such as the ones given above).