Cryptolosophers - Fagprojekt

s164412, Christian Hjelmslund
s161765, Andreas Bundgaard
s154407, Noah Sturis
s164413, Emir Sircic

Supervisor: Elmar Wolfgang Tischhauser

June 15 2018

Contents

1 Introduction

(Emir Sircic)
On 31 October 2008 a paper titled *Bitcoin: A Peer-to-Peer Electronic Cash System* was released by author Satoshi Nakamoto describing a trust-less system for electronic transactions and later in early 2009 the Bitcoin network went live. The cryptocurrency known as Bitcoin had in 2012 an estimated worth of around 5$, though it has only been in the last couple of years that the interest and popularity of Bitcoin and blockchain in general tremendously spiked, where each Bitcoin reached a peak of roughly 20.000$ market value. This extreme growth of value and rise in popularity set out a wave of alternative blockchain concepts. Other concepts, such as Ethereum and IBM's Hyperledger have been around for years, but also benefited from the increased awareness. It seemed this "new" technology was the next big thing. The craze was so high at times that companies, with no affiliation to the technology whatsoever, began to re-brand themselves or in some other way simply incorporate the word "blockchain" in their name where the result was a significant rise in market value.

The aim of this report is to gain a fundamental understanding of what blockchains are and how they work, both in theory and in practice. The cryptographic principles used in the technology will be briefly explained before moving on to the key parts of blockchains, such as accounts, transactions, the blocks themselves and various consensus mechanisms. Furthermore the concept of *Smart Contracts* will be explained, what they are and how they get executed, before presenting our own smart contract and how it was developed. The practical and to some extent the theoretical part of this report, will focus on the Ethereum platform on which the smart contract was developed and deployed. Together with our supervisor Elmar Wolfgang Tischhauser the following success criteria for the project was set:

- Develop a solid theoretical and practical understanding of how various blockchains work. Which security properties they have, how they resolve consensus and how they execute smart contracts (if they do)
- Setup of a custom blockchain
- Demonstrate a smart contract on Ethereum or Hyperledger
- Demonstrate an attack on Ethereum or Hyperledger

Furthermore some of the directions the project took will be discussed along with software engineering principles used in the process of developing a smart contract. Tools and frameworks will also be presented and evaluated along with a discussion and conclusion of the project as a whole. Under each (sub)section title it will be present who wrote the given section.

2 Principles of Blockchains

(Christian Hjelmslund)
The following section gives a brief introduction to what a blockchain conceptually is. Afterwards, detailed subsections which describe what the main components a blockchain consists of will be presented.

A blockchain is a data structure, consisting of blocks that are linked together. The blocks are linked together using cryptographic hash functions, which means that each block points to the previous block in the chain. The blocks themselves contain data fields such as a timestamp (when was the block attached to the chain), the hash of the previous block and transaction data. However, the fields vary depending on the specific blockchain. Blockchains provide a lot of features, with two of the most powerful being the resistance toward data tampering and transactional transparency.

The resistance to data tampering is provided by consensus mechanisms and the underlying cryptographic hash functions. If someone wants to change the structure of an earlier appended block, it will be rejected. This will be explained more in-depth later in the paper.

As for transparency, everyone can see every transaction which has taken place from the first block (also called the genesis block), to the last. One can monitor which accounts were involved in a particular transaction, and eventually trace the sequence of transactions down to a particular transaction, which is very powerful, and one of the reasons blockchain provides solutions to real world problems. In order to explore the blockchain technology, it's helpful to look at the underlying cryptographic concepts. The following segment is largely based on Bitcoins implementation of these, as the specifics differ from blockchain to blockchain.

2.1 Cryptographic concepts

2.1.1 Secure hashing algorithm, SHA-256

(Andreas Bundgaard)
Cryptographic hash functions are integral to all cryptography, but are central to cryptocurrencies. SHA-256 is the successor to SHA-1 from 2002, and is widely used. To hash something means to map a message of variable size to data of fixed size, in SHA-256s case 256 bits. The result of any message hashed becomes a unique, pseudo-random output. Describing key features needed for cryptocurrency, such as compression and 2nd preimage attacks will be presented later in the paper.

2.1.2 Merkle-damgaard construction

(Andreas Bundgaard)
For iterated hash functions such as SHA-256, there exists a method which shows that if you can find a collision (two messages of data hashing to the same output) on one of its parts, the whole function is vulnerable. The other way around holds true as well, if a part of the function is sufficiently random, a function consisting of a multiple of such parts is pseudo random, hence the iterative nature. Thus instead of trying to compress large messages, one can split it up, usually in 512 or 1024 bits, and then hash these individually to 256 bits.

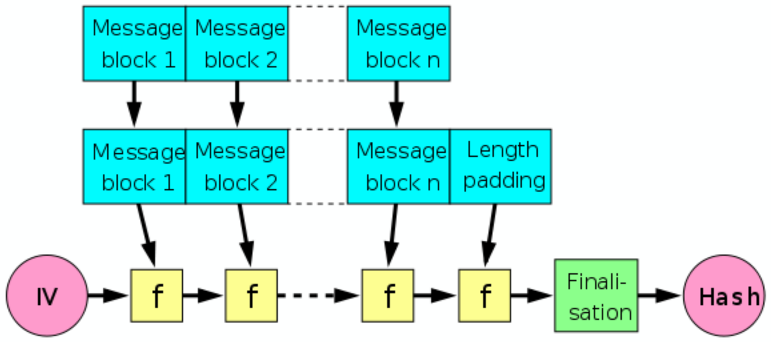


Figure 1: From https://en.wikipedia.org/wiki/Merkle-Damgard_construction

The concept is seen on Figure 1. The message is broken down into parts of proper size and fed to the compression function. The functions take two parameters, the previous function called chaining variable (or a start value IV, for the first run) and a message of appropriate size. In case a message doesn't divide into proper sizes, padding is added at the end. The padding is important, as it's a potential point of weakness. Generally it's a 1 followed by a number of zero's. The padding has shown to be collision resistant as well. For SHA-256 the maximum message length is $2^{64}-1$, which in practice allows for compression of most messages.

2.1.3 2nd preimage attack

(Andreas Bundgaard)
Concepts such as brute-force and 2nd preimage attacks will be briefly touched upon, as they resemble the process known as mining. In brute-force attacks the question becomes, can one guess every bit of a string correctly by trying all possible combinations, such that we end up with the desired hash? For an ideal hash function let $x \in \{0,1\}^* \bigwedge y \in \{0,1\}^n$. Let $H(x)$ describe a hash function, thus formally:

$$Pr(H(x)) = y = 1/2 * 1/2 * ... = 2^{-n}$$

Meaning that a hash function is perfectly random, and the chance of guessing a hash is 2^{-n}. Furthermore, nothing is learned about the hashing function by computing it with new inputs.
A 2nd preimage attack involves some information about the hashing algorithm. In a preimage attack, only the hash output itself is known, in a 2nd preimage attack, both the input and the resulting hash is known. With this information one is trying to find another input that hashes to the given hash. It's formalized as such:

$$\text{given } x \text{ find } x' \neq x \text{ such that } H(x') = H(x)$$

The probability of finding a solution is related to brute-forcing. As seen earlier 2^{-n} was the probability of being right once. Trying multiple times, all independently, nets the probability:

$$1-(1-2^{-n})^q$$

This is the chance of only successes with q attempts. The reason for bringing this up is the similarity to mining. When miners mine, they're looking for a hash less than a set difficulty. Often this is a specified integer value. They have knowledge of the hash function and the input used. In fact they know that inputs have to be the previous blocks hash, transactions gathered and verified, and a nonce, which is an integer counting every attempt. Thus mining is akin to a 2nd preimage attack, where instead of looking for a specific hash, one is looking for a hash less than a set difficulty, with known parameters included.

2.1.4 Merkle trees

(Andreas Bundgaard)
Entire transactions are hashed with SHA-256 twice, and then used as the identification or name of the transaction. Merkle trees are binary tree data structures that uses hashes as a means of verification and to save storage. The leaves contain whatever data needs to be stored - and in the case of Bitcoin, the identification of different transactions. Every node above a leaf is then the two leaves below it, concatenated and hashed twice, to form it's own unique identifier. In case there's an odd number of transactions, the last transaction is duplicated. By recursively repeating the process one ends up with a single hash at the root of the tree, called a merkle root. The merkle root has the property of being immutable, meaning that any change in any of the data in the leaves completely changes the root hash. Only a few nodes are needed to verify a transaction, and conversely it's easy to travel down a branch to find the error introduced. The merkle trees allow for effective storage of data, while still knowing if something is attempted altered. When enough blocks have been added to the chain, a merkle root of the transactions can be kept instead of keeping all transactions of a block. This slims down a block considerably.

2.2 Transactions

(Andreas Bundgaard)
Following the previous sections, this section will describe transactions in a simple form, loosely based

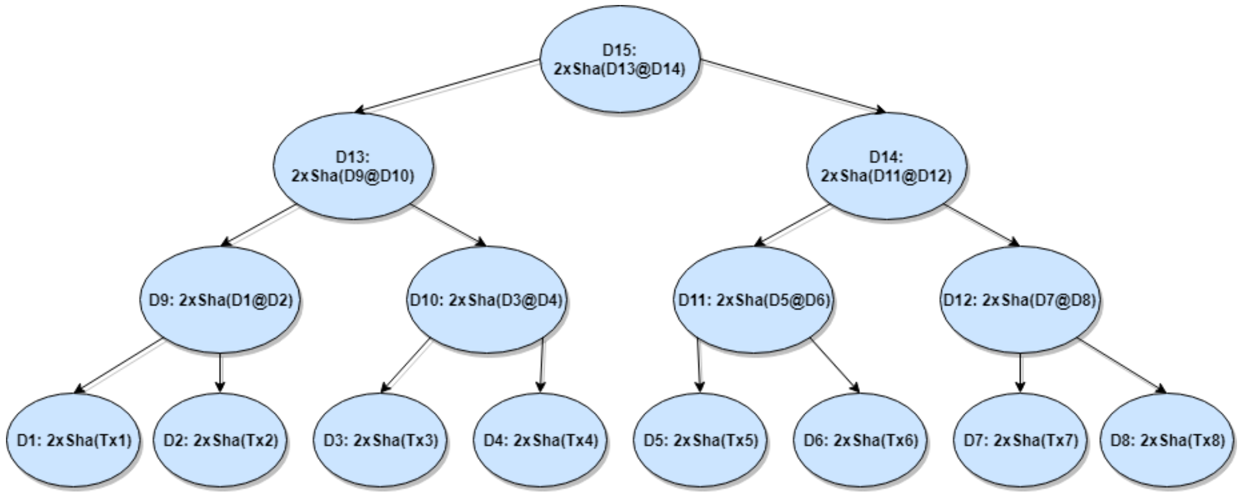


Figure 2: Simple Merkle tree

upon Bitcoin[1]. Technologies such as Ethereum's network will allow for much more complex transactions described later.

A transaction is conceptually different than everyday transactions, as it is a ledger system being updated, rather than currency changing hands. A transactions general format inside a block looks something like this:

Field	***Description***	***Size***
Version no.	Currently 1	4 bytes
Input counter	Positive integer counter	1-9 bytes
List of inputs	The inputs for the current transaction	Variable length pending on the number of Tx*
Output counter	Positive integer counter	1-9 bytes
List of outputs	The outputs for the current transaction	Variable length
Lock_time	Block height(size) or timestamp when transaction is final, with some constraints	4 bytes

Table 1: Overall format of a transaction

As seen a transaction (from here on Tx) is divided into two subgroups, a list of Tx input (Txin) and a list of Tx output (Txout) of arbitrary length. Generally there can be many Txin's and Txout's in a single transaction. The following properties characterizes Txin:

- Previous transaction hash, double-hashed by SHA-256, 32 bytes
- Previous Txout-index, organizing the Txout to be used, 4 bytes
- Txin-script length, a positive integer, 1-9 bytes
- Txin-script / scriptSig, script, many bytes
- sequence no., normally 0xFFFFFFFF, 4 bytes

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

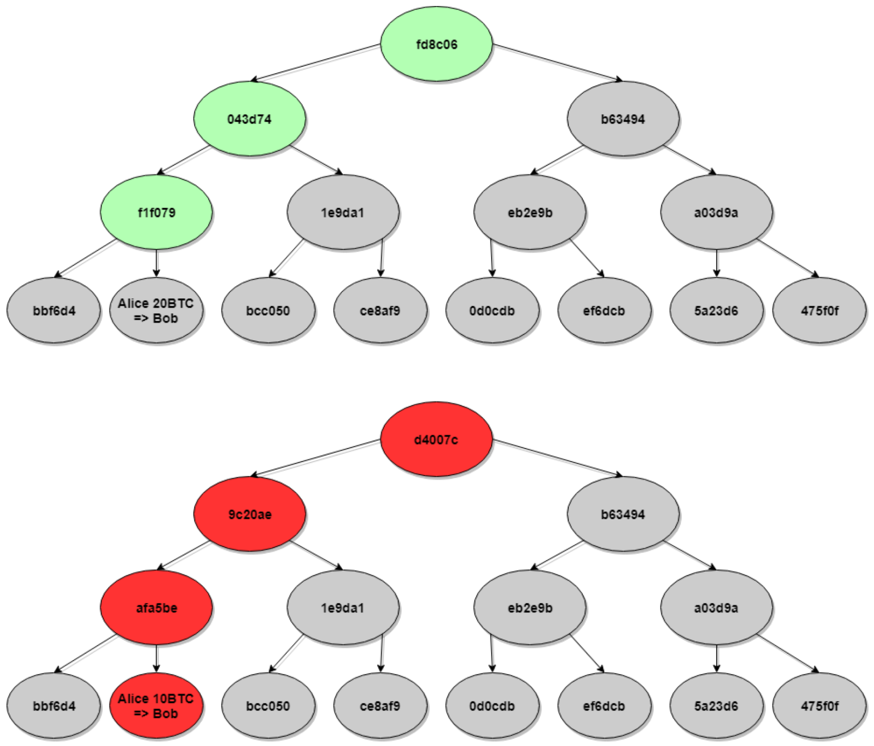


Figure 3: Merkle trees with verification

The first field of a Txin is thus the double hash of the current transaction message, which is also used as the name for the whole transaction. This also means that Txout doesn't have the current transaction name. It's not only at block level that verification exists, but also at inter-transactional level.

The second field is the Txout index used, and thus linked together with Txins. Txins must point to a Txout claiming the coins. Script is a language that is a Forth-like, simple, stack based and not Turing complete. It's all opcode, or opcode-like functions and commands, that can change the parameters of what's needed to claim a certain transaction. In short it's a list of instructions describing how the next person wanting to claim them gains access to them. For the simple transaction of sending Bitcoins to an address, there are two typical attributes the spender must provide:

1. a public key that when hashed gives the address in question
2. a signature to prove ownership of the private key to the public key just provided

Finally there's the sequence number that's only being used in scenarios where time and timelocks are a factor(not often used).

For Txout there are three similar fields:

- A value of Satoshis($BTC/10^8$) to be transfered
- Txout-script length, a positive integer, 1-9 bytes
- Txout-script / scriptSig, script, many bytes

The first one is simply the value of Satoshi to be transfered. 10^8 of these make one Bitcoin. The reason for this conversion is that systems like Bitcoin and many other cryptocurrencies can't operate with floating point numbers, as it complicates calculations. To circumvent fractions, the lowest possible value of a coin is many orders of magnitude lower than the official coin.

The rest of the Txout is a script like the one in Txin. Again, this decides who can access the coins. It is entirely possible at this stage, to allow anyone to spend them:

```
scriptPubKey: (empty)
scriptSig: OP_TRUE
```

These two pieces of script allows for no public key, and lets the person who executes it to send it to any account. This is especially useful in cases where one wants miners to mine the particular transaction, as extra compensation. This could be useful if there are liquidity concerns.

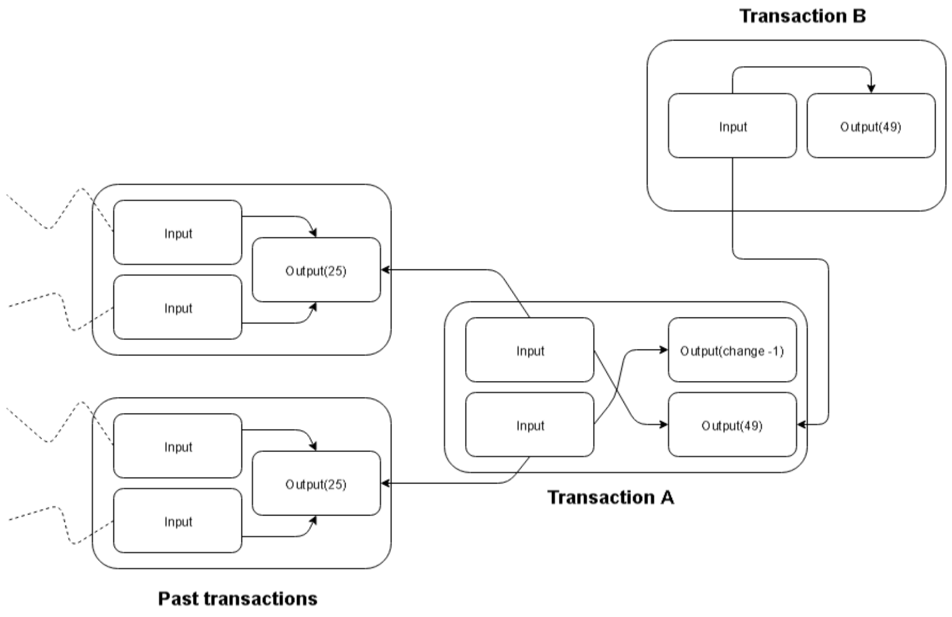


Figure 4: Linked transactions

In figure 4 it is seen how a series of different transaction could happen. The bigger square encompassing the smaller ones are the transactions, while the smaller squares are the Txin or Txout. The arrows are references, seen in the Txin. Transaction A is dependable on the output from past transactions, and A's input transactions point to these. Both of them verify that there are 50 coins to be spent in total, and prove that A(whoever is making the transaction) meets the conditions. In transaction A there's then 49 coins to be used in another transaction, transaction B. Often a "change" output is added to the Txout list, as a way to send money to one self, otherwise the difference would go to miners. In this case, 49 is send for another to use, 1 is kept. In B, the output is available, but hasn't been claimed yet. This shows the nature of Bitcoin and many other cryptocurrencies, and how they differ from normal currencies. Ownership is defined by permissions rather than physical storage. These permissions and proof of ownership are all integrated and stored in transactions. All inputs must be accounted for, before one can use them in an output transaction. Additionally in a transaction, all coins are used, which is showed in transaction A. Whatever the inputs add up to, the output must add up to as well, which is why the remaining 1 BTC is kept.

2.3 Blocks

(Andreas Bundgaard)
The above examples on transactions directly allow to explain the concepts of blocks. Blocks are essentially a collection of some or all current transactions. All transactions get collected in a list and saved in the block. A blocks structure looks as such:

Table 2: Block structure

Field	***Description***	***Size***
Magic no.	Unix file format identification	4 bytes
Blocksize	Number of bytes making up the block	4 bytes
Blockheader	Made up by several items	80 bytes
Transaction counter	Positive integer counter	1-9 bytes
List of transactions	A list of recent or current transactions	Variable length

Not all that surprising except for the block header. The block header contains information relevant for the hashing process explained earlier. For Bitcoin the specific hash function is called hashcash. A block header consists of:

- Version no. of the block, 4 bytes
- 256-bit hash of the previous blockheader, 32 bytes
- 256 bit hash merkle root of the transactions, 32 bytes
- Timestamp
- Difficulty(target), a number that is used in mining
- Nonce, a counter that keeps track of hashes attempted

Right away it can be seen that the principle of immutability is prevalent; included in the header is the hash of the previous header. This is exactly what makes the blocks chain/link together. Every block header includes the hash of the previous one, which makes them unique and dependable with respect to each other. Changing the hash of the previous block changes the hash of the current one and all following blocks. This is also the mechanism used to verify a block.

The next field is the aforementioned merkle root. Every time a transaction is added to the block, the root changes, akin to what was seen when altering a transaction. In Bitcoin, when enough blocks has been appended, immutability comes by dropping all other fields in the block except the block header. The header contains all instrumental information, but no longer information about specific transactions, only the nodes in the merkle tree. This is done to save space on the network.

A timestamp is self explanatory, but notably in smart contracts of other coins, the timestamp is used for time-sensitive applications. Both difficulty and nonce is used in mining, which was explained earlier. The difficulty is the numerical target miners aim to hit lower than when mining. The nonce is the number of attempts they used in the process.

2.4 Blockchains

(Christian Hjelmslund)
Permissioned vs. Permissionless
There are a lot of different blockchains, which can be used for various kinds of use cases. The blockchain that suits a given use case the best, really depends on what kind of application you want to create. In

this section permissioned and permissionless blockchains will be explained.

Permissioned
A permissioned blockchain differs the most from the usual well-known decentralized (e.g. permisionless) blockchains such as Ethereum, Bitcoin etc. The essence of a permissioned blockchain, is that every user on the blockchain has to be approved by a central authority. This means that there are more similarities to a traditional database and a permissioned blockchain, than a decentralized permissionless blockchain such as Bitcoin. One of the most known permissioned blockchains is Hyperledger Fabric, which is developed by IBM. Typical use cases for a blockchain like Hyperledger, would be where there needs to be some kind of central entity. It could be a government, which creates a blockchain for citizen data storage, and for a user to grant permission, it would need to hold citizenship. It is also called a blockchain platform for enterprises[2], which stresses that the whole decentralization part is undermined in Hyperledger.

Permissionless
A permisionless blockhain is the blockchain which most of the well-known cryptocurrencies uses. It is usually managed by a peer-to-peer network, and everyone can be a part of this network by running their own node. This means it is decentralized and not run by a central entity. Every peer has a locally stored copy of the whole blockchain, and when a new block is to be appended to the blockchain, every peer adheres to the blockchain protocol and validates that the new block doesn't violate the protocol. That protocol is refered to as a consensus mechanism.

2.5 Consensus Mechanisms

(Noah Sturis)
A central property of blockchain systems is that no single entity controls the content of the data. However, how does the system resolve a differing of opinions as to which data is valid and which is not? This is known as the Byzantine Generals Problem, where multiple generals have to coordinate a siege on a city through messengers, but some portion of the generals can be sending conflicting messages to the others. The result of this is known as a Byzantine Fault, where if more than 1/3 of the generals are malicious, they can confound the remaining generals and trust in the system breaks down.
The same issue is present in blockchains (known as chain selection or the fork choice rule) with various mechanisms used to establish consensus and prevent Byzantine Faults. Additionally, there are the questions as to who gets to produce the next block, who gets a reward for doing so and when these blocks are produced.

2.5.1 Proof of Work

(Noah Sturis)
The general idea behind the Proof of Work (PoW) consensus mechanism is to have the system's peers solve the above-outlined time-consuming, non-trivial (but not impossible) cryptographic puzzle before being able to append a block to the blockchain. When the puzzle is solved, the miner gets rewarded an amount of the cryptocurrency for their work along with transaction fees, if there are transactions in the block. The process of verifying these blocks is a simple and quick task, since the hard-to-find pieces of the puzzle have already been computed, however changing a transaction or anything else in the block requires recomputing the block (and any blocks following it).

The cryptographic puzzle needed to be solved to find/append a new block in a Proof of Work blockchain is to compute a hash value (using SHA-256) that is lower than a set target. This target is represented as a 256-bit value with a set number of leading zeroes where each leading zero raises the difficulty

[2]https://hyperledger-fabric.readthedocs.io/en/latest/

exponentially. The difficulty directly translates to how many tries on average it takes to compute the next block hash and this difficulty is adjusted dynamically to maintain a hashrate of 10 minutes per block (in the case of Bitcoin). The maximum target theoretically (easiest difficulty) would be $2^{256} - 1$, but this has the consequence that every hash value would be valid, which would allow thousands of blocks to be generated per second. The maximum target in practice is approximately 2^{224} and the current difficulty is determined by the ratio of the maximum target and the current target. Because of the pseudo-randomness of SHA-256 this mechanism ensures peers must spend some amount of CPU-time to mine the next block, which in other words translates to some amount of work. Furthermore, Proof of Work incentivises the miners to keep working on the longest chain, as working on shorter chains quickly becomes computationally inefficient. In a proof of work system, more than 50% of all miners need to be malicious to cause a Byzantine Fault - known as a 51% attack.

2.5.2 Proof of Stake

(Noah Sturis)
Instead of requiring raw computing power to mine blocks, a Proof of Stake consensus protocol requires the miner to reserve ("stake") a portion of their cryptocurrency behind a certain chain, while validating blocks. Whichever chain has the largest accumulated stake behind it is the one the network deems valid. To be able to produce blocks, stakers are to be chosen at random as the block leader, with the probability being proportional to total amount staked.

2.5.3 Other types of consensus mechanisms

Proof of Elapsed Time
(Noah Sturis)
Proof of Elapsed time works by having each participant wait for a random amount of time and the participant that finishes waiting first wins the lottery and is chosen as the authority for the new block. For this to work there needs to be certainty that the winner actually chose a random time to wait and that they waited the specified amount of time. Otherwise one could just choose a short time to win or not fully wait for the specified amount of time. To accomplish this, Intel developed a special CPU instruction set called Intel Software Guard Extensions (SGX) that allows for trusted code be run in a protected environment, which ensures that the lottery regarding the authority on the new block is fair.

Byzantine Fault Tolerance
(Noah Sturis)
Hyperledger Fabric resolves consensus with a simple Byzantine Fault Tolerance, which means if more than 1/3 of the nodes in the network are malicious, the system breaks down. As it is a permissioned blockchain, the chance of malicious actors is minimized, but it is still less fault tolerant than a Proof of Work-based system.

3 Ethereum

3.1 Smart Contracts

(Christian Hjelmslund)

When you read the term ”smart contract”, the first thing to pop into your mind, might not correspond to the reality of what a smart contract really represents. A smart contract is in reality a small program, written in a simple programming language (such as Solidity), which is deployed on a blockchain. The smart contracts’ properties vary depending on which blockchain the developer intends to deploy the contract on. To be on the same page, a small demonstration of a concrete smart contract is shown on Figure 5 (the understanding of what the contract does is irrelevant for now)[3].

```
contract Puzzle{
  address public owner;
  bool public locked;
  uint public reward;
  bytes32 public diff;
  bytes public solution;

  function Puzzle() //constructor{
    owner = msg.sender;
    reward = msg.value;
    locked = false;
    diff = bytes32(11111); //pre-defined difficulty
  }

  function(){ //main code, runs at every invocation
    if (msg.sender == owner){ //update reward
      if (locked)
        throw;
      owner.send(reward);
      reward = msg.value;
    }
    else
      if (msg.data.length > 0){ //submit a solution
        if (locked) throw;
        if (sha256(msg.data) < diff){
          msg.sender.send(reward); //send reward
          solution = msg.data;
          locked = true;
        }}}}
```

Figure 5: A small demonstration of the Puzzle smart contract

Comparing with Bitcoin, if one is to ignore the consensus mechanisms (e.g. we assume that the transaction is valid etc.), only used transaction in the Bitcoin blockchain (roughly speaking) is:

$$\textit{send 10 btc from } A \rightarrow B, \textit{if } A \textit{ has a balance of 10 btc or more} \tag{1}$$

$$\textit{else return error.} \tag{2}$$

A smart contract is much more complex, and opens up for endless opportunities of when to trigger (invoke) a transaction, and in that way makes it ”smart”. Once a smart contract is validated and deployed on the blockchain, it is immutable. In practice that means, once it is deployed on the network, if you’ve made a mistake in the contract, it is irreversible and the code cannot be changed. Hence making it a ”contract”. For the rest of this section, the paper will focus on smart contracts in the Ethereum network, with the following focus points:

- accounts
- transactions
- messages
- gas
- fetching data outside the blockchain

3.1.1 Accounts

(Christian Hjelmslund)

There are two types of accounts in the Ethereum network. There are externally owned accounts (EOA)

[3] The code is an example from: https://eprint.iacr.org/2016/633.pdf

and contract accounts. EOA's are a public/private key combination where the hashed public key make up the account's address and this address is stored in the blockchain with a balance. No other code is attached to an EOA; it is just an 20-byte address, where one can send a message to/from. It is still very much part of the blockchain though, because if Alice wants to send a message to Bob with 20 Ether, Alice creates a signed transaction, which will need to be validated and accepted by the miners on the blockchain, and that's how everyone can see that Bob now has received 20 Ether from Alice. Alice signs this transaction with her private key, because the network needs to know that she transferred 20 Ether to Bob. The EOA's are not unique to the Ethereum network, and is in reality the same that the Bitcoin blockchain offers. The contract accounts although are another story.

The contract accounts are deployed on the blockchain, and as shown on Figure 5 it's basically a piece of code. It is up to the developer of the contract, whether or not the contract has any Ether at the creation of the contract. The contract is stored permanently on the blockchain. If Alice wants to interact with it (invoke it), she would submit a transaction, where the address of the account she wants to interact with, is the address of the contract code.

There's a general scheme a contract account has to follow. It has some variables initialized at the top such as owner, it has a constructor etc. When the contract is deployed on the blockchain the constructor is called, and the variables are set as wanted by the owner. It also allows for the definition of `function()` methods. When somebody invokes the contract, the specific `function()` called is the part of the code which is run.

3.1.2 Transactions

(Christian Hjelmslund)
Sometimes it can be a bit hard to distinguish the difference between messages and transactions, but there are some key elements. A transaction can be seen as a package containing some data, and the package is initialized by an externally owned account. If Alice sends 10 Ether to Bob, she does that by initializing a transaction. A transaction contains some different fields as the recipient, signature, and amount of Ether to transfer, but that's the same as other blockchains. A transaction also contains an optional data field, `startgas` and `gasprice`. The optional data-field is optional because if Alice wants to send 10 Ether to Bob, there's no need to provide any data for that kind of transaction. On the other hand, if Alice wants to invoke a smart contract, she can choose to send some kind of data along, if the smart contract needs it. The `startgas`, `gasprice` and `gas` in general will be explored later on.

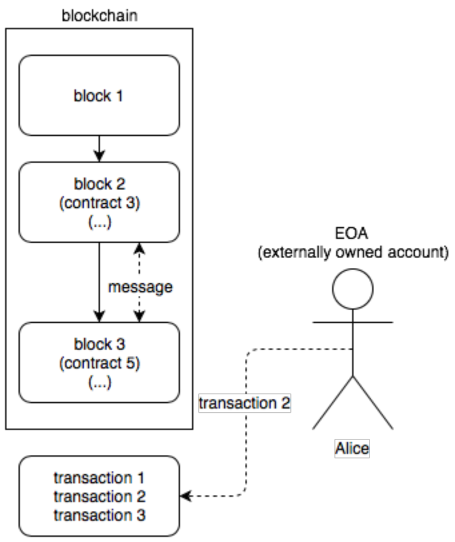
3.1.3 Messages

(Christian Hjelmslund)
Contrary to the transaction, a message is an object that only exists within the Ethereum network. If contract A wants to communicate with contract B, it does so by sending a message and not a transaction. The message is quite similar to the transaction, but the difference is that a transaction is executed by an EOA (Alice) and a message is called from a contract account. They do essentially behave the same way, because when a contract receives a message, its code will be executed. A message contains the following values;

- sender
- recipient
- Ether amount
- data
- startgas

which is why it can be hard to distinguish the transactions and messages. Figure 6 is a very simplified example, but essentially shows what the difference between a transaction and message is.

Figure 6: A simplified example of the difference between message and transaction



Alice (EOA) orders a transaction, and if the miners approve it, it will be attached to a block, which will be appended to the blockchain. A message is if contract 5 is invoked and executes a `CALL` opcode, which creates a message, and in this case sends it to a contract in block2, which will be invoked and react according to its code.

3.1.4 Gas

(Christian Hjelmslund)
There are a few problems with smart contracts which have to be dealt with. What if someone creates a contract, with a bug in it's code and what if a contract is invoked, and there happens to be an infinite loop? Or if some outside actor tries to creates x-times transactions, just to overflow the network? Ethereums solution to that problem is gas.

Every single computational step costs x-amount of gas (a certain amount of Ether), which is why one has to specify in a transaction the maximum amount of gas wanted to spend to execute the transaction. There are some impediments though, because one might ask what happens with the gas you spend? The miners are introduced now, because they are the ones carrying out the computations and therefore are rewarded with the gas. When Alice wants to create a transaction, she specifies the maximum amount of gas she wants to spend, and if the transaction costs more than her predefined limit, the rest of the function cannot be executed, the transaction will fail and she won't get back her Ether. It's her own fault that she miscalculated the amount of Ether to use, not the miners, which will still get the reward. Guards against this exist in many user-facing applications to help against accidently miscalculating (by pre-calculating what a transaction is likely to cost). However this is an effective measure against spam transactions, as the one paying for the transactions is the one invoking them, therefore aligning the incentives of wanting the invoked transaction to have value to the invoker.

To get back to the point as to why gas was introduced. If some outside actor with malicious intent is on the blockchain and wants to overflow the network by filling it with heaps of transactions for example, the actor also has to pay for it! Eventually the actor's Ether will run out, which is why gas is useful as a way to prevent denial-of-service attack. This is also why the problem with the infinite loop isn't really a problem for the network, because at some point the contract which was deployed with a certain amount of Ether will eventually run out, and thereby throw an exception instead of blocking the entire network.

3.1.5 Fetching data outside the blockchain

(Christian Hjelmslund)
As explained in the previous sections, the majority of the nodes have to confirm that the transactions are valid by running the contract itself, and checking if the conditions are fulfilled. The groundbreaking difference between the Ethereum blockchain and Bitcoin is the smart contracts, but let's say Alice created a contract, which says that if she manages to create a website with 10.000 unique hits, she'll transfer 10 Ether to Bob. The question here is, who is to determine if she had 10.000 unique hits and how will all the mining nodes decide this (which is off-chain, and on the web).

This is when the term oracle needs to be introduced. An oracle is basically a third party, which provides counsel and useful information, which both Alice and Bob can agree upon leaving the decision to that oracle.
There are different kind of oracles implemented today, which operates on-chain or off-chain, but the focus in this report is on `oraclize.it`, which is an on-chain smart contract. When implementing a smart contract, it needs to inherit Oraclize, and the contract is provided with the URL in which you want Oraclize to fetch the data from. In Alice's example, she would probably encode a Google Analytics URL, and inherit Oraclize in the smart contract. When Alice wants to check whether she had 10.000 unique hits, her contract would send a request to an Oraclize contract.

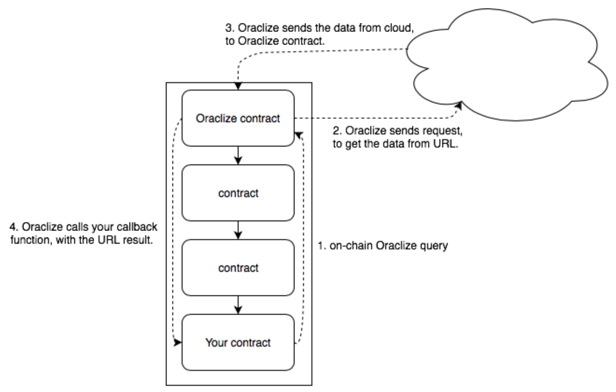

3. Oraclize sends the data from cloud, to Oraclize contract.
Oraclize contract
2. Oraclize sends request, to get the data from URL.
contract
4. Oraclize calls your callback function, with the URL result.
1. on-chain Oraclize query
contract
Your contract

Figure 7: Life cycle of an oraclize query

Oraclize would receive that request, do an query off-chain and then create a transaction, which calls a callback function implemented in Alice's smart contract, with the result of the API call (see Figure 7). This means that none of the miners have to do any external calls, because Oraclize fetches the data off-chain and sends a transaction to Alice, with the data. This of course falls apart if Oraclize is untrustworthy, but they claim that they can provide a cryptographic proof which proves that they've gathered the data, from the exact URL at the specified time, implemented in the given smart contract.

3.2 Ethereum Blockchain and Mining

(Emir Sircic)
The structure of the Ethereum blockchain is similar in many ways to Bitcoin, by the fact that they make use of mostly the same cryptographic proofs to ensure security and validity of the chain. To better understand the differences between the two, first and foremost Ethereum's state transition function will be explained in the following section, before moving on to the validation process of blocks.

3.2.1 State Transition Function

(Emir Sircic)
Inside a block on the Ethereum chain, information such as the root hash, timestamp, a transaction list etc. resides, and along with those the most recent *state* is also stored. This state in Ethereum describes the information exchanged between accounts (both EOA's and contracts), the data exchanged between these and the balance changes of accounts. In other words the state reflects the actions of the transactions that occurred between two blocks on the chain as depicted on figure 8. This is a key principle of the Ethereum platform since it can be described as a "transaction state machine". The

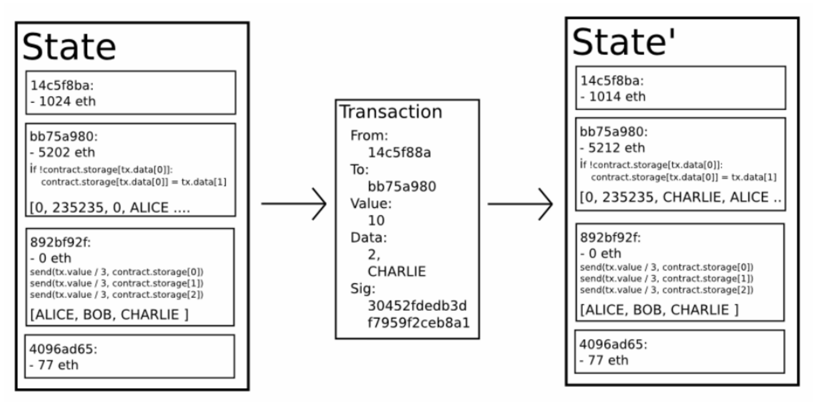

State
14c5f8ba:
- 1024 eth
bb75a980:
- 5202 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, 0, ALICE ....
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth
Transaction
From:
14c5f88a
To:
bb75a980
Value:
10
Data:
2,
CHARLIE
Sig:
30452fdedb3d
f7959f2ceb8a1
State'
14c5f8ba:
- 1014 eth
bb75a980:
- 5212 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, CHARLIE, ALICE ..
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth

Figure 8: The Ethereum states and state transitions

state is stored in a block as a trie (Patricia tree [4]) to ensure fast and compact storage. Moving on to the *state transition function* as described in the official Ethereum white paper[5]:

1. Check if the transaction is well-formed (ie. has the right number of values), the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.

2. Calculate the transaction fee as STARTGAS * GASPRICE, and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.

3. Initialize GAS = STARTGAS, and take off a certain quantity of gas per byte to pay for the bytes in the transaction.

4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.

5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.

6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

Interesting things to note about the function is whether the receiving end of a transaction is a contract or an EOA. If the receiver is an EOA the amount of gas spent (or transaction fee) would be equal to the GASPRICE multiplied with the length of the transaction in bytes. When a transaction invokes

[4]Practical algorithm to retrieve information coded in alphanumeric trie

[5]`https://github.com/ethereum/wiki/wiki/White-Paper#ethereum`

a contract on the other hand, the contracts code is then executed where each computation consumes some of the gas for reasons described earlier. Another important thing to notice is item 4 in the above list, in regards to transferring Ether from the sender's account to the receiving account. If the receiving account at the time has not been created, a new account will be created and receive the transferred amount of Ether. Since the state transition function creates this new account its private/public key-pair is not owned by anyone and the transferred Ether could potentially be locked completely.
When invoking transactions one should therefore confirm the receiving address is indeed the intended receiver to circumvent possible loss of Ether.

3.2.2 Mining and Block Validation Process

(Emir Sircic)
As described in the previous section, the blocks in Ethereum mainly contain the transaction list and the most recent state, but the question is how new blocks are appended to the chain and by who? The answer for that question is: the miners! A miner on the Ethereum network is a node which has a full copy of the entire blockchain locally, is synchronized with the network and is offering their CPU (or GPU) power to solve time-consuming cryptographic puzzles to find the next block. Since the Ethereum platform is meant to be a trustless immutable network, where anyone on a global scale is able to participate, there has to be some sort of agreement between all nodes in how and when new blocks can be appended and this is exactly why Ethereum uses the "Proof of Work" (PoW) consensus mechanism. This section will delve into the actual mining process on the Ethereum platform, how miners find the next block and what computations must be made.

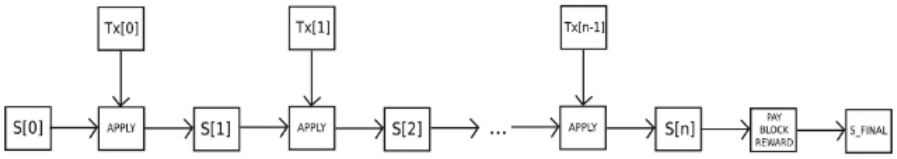


Figure 9: `https://github.com/ethereum/wiki/wiki/White-Paper#blockchain-and-mining`

The general idea behind mining on a PoW network as Ethereum, is that finding the next block should be hard and take time, while validating that block and the work done to find it should be trivial and very fast to do. This is to prevent malicious nodes controlling the network and further promote the immutability of the data. To delve even deeper, here are the actual contents of a block in Ethereum[6]:

- **Parent hash:** a hash of the previous block header.
- **Uncle (or Ommers - gender neutral) hash: Also known as an "orphaned" block. A current block A's uncle would have the same block number as A's parent, but not considered to be part of the validated chain.**
- **Reward address or beneficiary:** An address to which the reward for mining the block is paid.
- **State root:** A hash of the root node of a state trie (as described in the previous section)
- **Transaction root:** A hash of the root node of a transaction trie.
- **Receipt root:** Same as transaction root but where the trie contains receipts of the transactions.
- **Logs Bloom:** A "Bloom filter". A way to re-create/compute transaction events if necessary (the logs themselves are not stored in the block to save space)

[6]`http://gavwood.com/paper.pdf`

- **Difficulty:** A difficulty (simply a value) that can be computed from the previous blocks difficulty and time stamp, used to control the rate of new blocks.
- **Block number:** The current index of the block
- **Gas limit:** A value corresponding the max amount of gas that can be used per block
- **Gas used:** A value indicating the total amount of gas used for all transactions in this block
- **Time stamp:** Using Unix's time() to denote when the block was found
- **Extra data:** 32 or fewer bytes of data relevant to this block
- **Mix hash:** A hash that is used together with the nonce to prove enough computations has been made.
- **Nonce:** A value indicating the number of tries to find the next block.

All of the entries above hashed together forms the block header, which again can be used to quickly verify the block as valid by all nodes. The process of validating blocks is defined in the Ethereum white paper[7] as follows:

1. Check if the previous block referenced exists and is valid.
2. Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes into the future
3. Check that the block number, difficulty, transaction root, uncle root and gas limit (various low-level Ethereum-specific concepts) are valid.
4. Check that the proof of work on the block is valid.
5. Let S[0] be the state at the end of the previous block.
6. Let TX be the block's transaction list, with n transactions. For all i in 0...n-1, set S[i+1] = APPLY(S[i],TX[i]). If any applications returns an error, or if the total gas consumed in the block up until this point exceeds the GASLIMIT, return an error.
7. Let S_FINAL be S[n], but adding the block reward paid to the miner.
8. Check if the Merkle tree root of the state S_FINAL is equal to the final state root provided in the block header. If it is, the block is valid; otherwise, it is not valid.

The main part of the validation process is the state transition function APPLY(S[i], TX[i]) as described in the previous section, where all the transactions from the transaction list are applied to the state. As pointed out in the white paper, the entire state appears to be stored in every block, but because of the data structure used to store the state (Merkle trie and Patricia trie), it is only necessary to store the state once and then create references to the data with pointers. Another thing to notice about the validation process is item 7 in the above list, because in the final state transition function, the miner adds the block reward to himself, which is an amount of Ether (around 3.5 Ether at the time this report is written[8]) that is simply generated out of "thin air". This reward is to provide incentive for miners to stay as "honest" nodes and keep the entire network running. Ethereum currently also rewards miners who find "stale" or orphaned blocks (referred to as uncle blocks) that do not become part of the main chain per se, but this protocol (GHOST Protocol) is out of scope for this report. Another interesting question is when contract code gets executed, which is the subject of the following section.

[7]`https://github.com/ethereum/wiki/wiki/White-Paper#blockchain-and-mining`
[8]`https://bitinfocharts.com/ethereum/`

3.2.3 Code Execution

(Emir Sircic)
When a smart contract, written in a "high-level" programming language as Solidity, needs to be deployed it must first be compiled to a "low-level" bytecode language known as "Ethereum Virtual Machine code" (EVM code). The EVM code basically boils down to a series of bytes where each byte represents some operation (or op-code), or in other words it is a stack based machine. The EVM code has access to three types of memory space which are the stack, memory and storage as defined in the Ethereum white paper [9]:

- **Stack:** A list-in-first-out container to which values can be pushed and popped
- **Memory:** An infinitely expandable byte array
- **Storage:** The contract's long-term storage, a key/value store. Unlike stack and memory, which reset after computation ends, storage persists for the long term.

Other than the three types of space presented above the EVM also has access to the block header data, which contains the world state (which includes all accounts and their balances and storage) and the transaction lists, which is necessary since code execution occurs in the state transition function. The way EVM code gets executed is practically an infinite loop that runs the specified operations at the current program counter (pc) until either no more code is present, a STOP/RETURN operation is reached or ultimately if an error occurs. The state of the code execution is represented by the tuple `(block_state, transaction, message, code, memory, stack, pc, gas)` [10], where the entries in this tuple are topics that have already been covered in this report. Upon execution the program counter determines which operation should be used and all the different operations affect the tuple in different ways. All available operations will not be described here but are all listed on the Ethereum Yellow Paper[11]. Another thing to note here is the `gas` which is the provided amount of gas, by the transaction that invoked the contract, to be spent during execution. In most operations the program counter gets incremented by 1 and the gas decremented by 1 and should the gas reach 0 before the end of the EVM code, an Out of Gas exception is thrown and the state is reverted.

4 BlockCycle

4.1 An introduction to BlockCycle

(Christian Hjelmslund)
BlockCycle is a decentralized bicycle insurance, which is created on the Ethereum platform. Why is a decentralized bicycle insurance based on the Ethereum network, a good use of blockchain technology? One could ask, what kind of value does an insurance company provide for each user? It provides trust and a fair distribution of risk among all users of the insurance company. But there is a lack of transparency and automation in the insurance industry, which could be put to an end with the introduction of insurance applications based on blockchains. Blockchains could be argued to minimize the risk of fraudulent data, which combined with smart contracts would reduce the time drastically, from when a user issued a claim to the company, to when the user actually gets a refund. The process would be automatic and the smart contracts, would do the necessary computation to check whether a user is entitled to a refund or not.

In the case of BlockCycle, there would ideally be an oracle call incorporated in the contract, which would call the police's database over stolen bikes. Hence making the data trustworthy, but that hasn't

[9] `https://github.com/ethereum/wiki/wiki/White-Paper#code-execution`
[10] `https://github.com/ethereum/wiki/wiki/White-Paper#code-execution`
[11] `http://gavwood.com/paper.pdf`

been implemented. Instead an EC2 instance with a public facing API is used to simulate the police database over stolen bikes. It is worth mentioning, that if a user reports a bicycle stolen to the police, and it is not stolen, the data would in fact be fraudulent, but that cannot be taken into account. The question of the perfect oracle will also be touched on later.

There are three core features BlockCycle has: The possibility of buying an insurance, renewing an insurance and of course claiming an insurance when the bicycle is stolen. The optimization of the core features has been a priority during the project, and they are the backbone of the smart contract. An area of the contract that has been prioritized low, is the financial one. Much of the underlying logic is simple; the rate is arbitrarily chosen, based on an input value, same with the depreciation rate when renewing an insurance. There's no evaluation of the reserve and no payback functionality. All these areas can be developed further on, to properly reflect the original idea.

4.2 Developing BlockCycle

(Christian Hjelmslund)
A lot of tools have been used in the process of developing BlockCycle such as Truffle, Remix and Ethereum Wallet. The tools will be introduced and briefly explained individually. Throughout the project, traditional software engineering practices has also been utilized, to ensure the relevance and quality of BlockCycle.

An agile software approach has been used, to ensure continuous improvement and flexible responses to changes. Since blockchain and smart contract development is a relatively new technology, it was necessary to have such an approach, since a lot of the technology isn't fully functional yet and in general a lot of problems occurred during the evolution of the project. It also ensured that the smart contract was (in theory) delivery-ready during the implementation of latest features, though not perfect, and since smart contracts are immutable upon deployment on a blockchain, that part of agile software development doesn't suit a project like this very well. There are ways around this, by deploying another smart contract that acts as an interface for various versions of the BlockCycle contract, but this somewhat undermines the principle of immutability, as a malicious actor with rights to update the contract, could change the code to their benefit. Deploying a well-tested smart contract and guaranteeing its immutability increases trust in the contract, as there's no way to change the behaviour. This also leaves no option to fix errors discovered post deployment.

4.2.1 User Stories

(Christian Hjelmslund)
User stories are an easy way to describe the core features a program needs, in the perspective of the user. In this case, for BlockCycle the following user stories have been created, and the contracts core features ensures that a user can in fact interact with the contract as stated below:

1. as a user, I want to insure my bike, so that I can get economic compensation if my bike is stolen.
2. as a user, I want the process of claiming compensation to be efficient, so that I receive my compensation as fast as possible.
3. as a user, I want transparency, so that I can check for myself that the compensation I am given is a fair compensation.
4. as a user, I want the data to be trustworthy, so that I can ensure that I'm getting a fair treatment.
5. as a user, I want an automated insurance system, such that the cost of getting insured is minimal.
6. as a user, I want to renew my bike insurance, so that I wouldn't need to buy another insurance, when my current insurance expires.

The user stories also show why an insurance based on the Ethereum Network is a good idea, because story 2, 3 and 4 can easily be achieved by creating a smart contract on the Ethereum Network. The compensation is received in the time span from when you claim the insurance, to when the block is mined. Transparency and trustworthiness is provided by being able to follow the transaction and that the data is verified by all peers of the network.

4.2.2 Detailed use cases

(Christian Hjelmslund)
Since BlockCycle is a decentralized application, every actor who uses BlockCycle is just a regular user. There is no hierarchy, and every user is equal, hence there will only be one kind of actor. In relation to that, there are only three core features in BlockCycle which are `buyInsurance()`,`renewInsurance()` and `claimInsurace()`. This means that a use case diagram seemed otiose and therefore only three detailed use cases which cover the above mentioned features, will be presented. The described scenarios in each of the detailed use cases are based on user interaction only, so scenarios where for example Oraclize, the Ethereum Network isn't working, or any other non-user interaction based issues occurs, are not taken into account.

---

1. Buy Insurance

Name: Buy Insurance
Description: A user buys a new bicycle insurance
Actor: User
Preconditions:

- The user has an Ethereum account.
- The user has enough Ether to pay for the insurance.

Main scenario:

- **1.** The user sends a transaction containing ether to the contract.
- **2.** The contract saves the user's insured amount along the hashed serial number in the contract.

Alternative scenario:

- **1a.** The user doesn't provide enough Ether (see note)
    1. The contract terminates and the user's gas is burnt.
- **1b.** The user's bike is already stolen.
    1. User gets an error message. The contract terminates and the user's gas is burnt.
- **1c.** The user's bike is already insured
    1. User gets an error message. The contract terminates and the user's gas is burnt.

Note: enough Ether is when $msg.value \geq \frac{insuredamount}{20} + oraclizeFee$

---

2. Renew Insurance

Name: Renew Insurance
Description: A user renews a cycle insurance
Actor: User
Preconditions:

- The user has an Ethereum account.
- The user has enough Ether to pay for the insurance.

Main scenario:

- **1.** The user sends a transaction containing Ether to the contract.
- **2.** The contract renews the users insurance, and stores the new time stamp in the database.

Alternative scenario:

- **1a.** The user doesn't provide enough Ether (see note)
  1. The contract terminates and the user's gas is burnt.
- **1b.** The user's bike is already stolen.
  1. User gets an error message. The contract terminates and the user's gas is burnt.
- **1c.** The user isn't the owner of the bike he/she tries to renew.
  1. User gets an error message. The contract terminates and the user's gas is burnt.
- **1d.** The user tries to renew an insurance on a bike, which has never been insured before.
  1. User gets an error message. The contract terminates and the user's gas is burnt.

Note: enough Ether is when $msg.value \geq (\frac{insuredamount}{20} \times \frac{9}{10}) + oraclizeFee$

---

3. Claim Insurance

Name: Claim Insurance
Description: A user claims an insurance
Actor: User
Preconditions:

- The user has an Ethereum account.
- The user has enough Ether to pay for the insurance.

Main scenario:

- **1.** The user sends a transaction containing ether to the contract.
- **2.** The contract transfers the users insured amount to the user.

Alternative scenario:

- **1a.** The user doesn't provide enough Ether (see note)
  1. The contract terminates and the user's gas is burnt.
- **1b.** The user's bike is **not** stolen.
  1. User gets an error message. The contract terminates and the user's gas is burnt.
- **1c.** The user isn't the owner of the bike he/she tries to claim insurance for.
  1. User gets an error message. The contract terminates and the user's gas is burnt.
- **1d.** The insurance is expired.
  1. User gets an error message. The contract terminates and the user's gas is burnt.

Note: enough Ether is when $msg.value \geq oraclizeFee$

4.2.3 Unit Testing

(Noah Sturis)
To ensure the behaviour of the smart contract matches with the outlined use cases, tests were constructed for the main scenario and each of the alternative scenarios for all three use cases.
Testing smart contracts is tricky, as certain functions are private and therefore not accessible to anything other than the smart contract. Further complicating testing is the fact that Oraclize is used to verify whether or not the bicycle is stolen, which first sends a request and then at some point in the future, when the API-call is executed and the result is recieved, the Oraclize contract calls the BlockCycle `__callback` function, which is not callable (for security reasons) by any other contract than the Oraclize contract. Theoretically it would be possible to alter the visibility and security checks when unit-testing and then altering them back when deploying, but that is difficult to manage in a secure and efficient way.
Instead a public view function `seeInsurance` was implemented, which shows the insured amount and the expiration date of a given serial number, if the address calling the function is the same as the owner of the insured bike. This allows for testing of `buyInsurance` and `renewInsurance`, as it's possible to check whether or not a bike is insured and for how much before and after the functions are called. To handle the Oraclize query, a `sleep` function was implemented, which pauses the test for a specified amount of time before continuing. This allows enough time for the callback-function to be called and for the BlockCycle contract to execute according to the result.

4.2.4 Security Considerations

(Noah Sturis)
In a smart contract, security concerns are very valid and before deployment of a contract, every security issue should have been dealt with. When the contract is deployed, there is no way to fix those issues, hence one might loose a lot of ether. For BlockCycle three main security issues were encountered.

In `claimInsurance` it was discovered that the smart contract transfers money before actually deleting the insurance from the contract. In practice it seems like there shouldn't be any issue with that, because of the contract will be executing the `delete bikes[_serialNo]` in the end of the method. But in theory if someone manages to send two transactions, running in parallel, the `user.transfer(Bikes [_serialNo].insuredAmount)` could be executed twice, hence the malicious user would receive twice the amount. Good practice would be to delete the insurance, before actually transferring the ether, which was implemented in the latest revision.

Another security consideration in general is dealing with information that exists outside the blockchain. It's relatively simple to guarantee the behaviour of smart contracts, when all the conditions relied upon in the smart contract exist within the blockchain, however to determine outside conditions and oracle, such as Oraclize, needs to be implemented. You can choose to pay 5 times the amount than a regular Oraclize query, and Oraclize then provides a cryptographic proof that the data received is actually valid at that point in time.

In extension of this, it is possible that either the Police API itself is compromised through a hack or that an attacker is able to fool the Oraclize API-call with a man-in-the-middle DNS attack, or something similar, which would point the Oraclize API-call to a different web-server than the intended one without realizing it. A way of defending against this would be to encrypt a token within the API-call with Oraclize and have the web-server require the token - along with the smart contract's callback function requiring a signed message from the verified API-server.

Finally, since the value of the bicycle is decided upon by the person buying the insurance, a relatively straight forward attack would be to insure a bicycle for a the total amount of Ether on the smart contract, fraudulently report the bike stolen and claim the insurance policy - clearing the entire smart contract of all Ether.

4.3 Tools used for development

Truffle
(Noah Sturis)
Truffle is a framework used for Ethereum smart contract and decentralized application development. It can handle the compilation of smart contracts along with their deployment to an Ethereum blockchain. The key feature in Truffle used for this project was their integrated contract testing, which allowed for automated unit-testing to ensure that the smart contract performs according to the use cases outlined. Along with Truffle, the tool `ganache-cli` was used to simulate an Ethereum blockchain on a local machine. This allows for the same accounts to be generated every time, which eases testing and development.

Remix
(Christian Hjelmslund)
Remix is an online Solidity compiler, which also works as a kind of test network. The smart contract can be developed in Remix, and after the compilation, one can deploy the contract on a fictive Ethereum network, stating how much Ether the contract is initialized with etc. In the same instance it also grants an account, in which you have a pre-defined amount of Ether, such that a developer can test the different implemented functions, by invoking them and sending along x-amount of ether. It is very useful to test functionality during the development process, and it saves you the time of deploying the contract on a real test network such as Rinkeby every time.
Only the developer can interact with the smart contract when it is deployed on Remix, so when bigger functions are implemented and need to be tested, it is recommended to deploy it on a test network as Rinkeby, or a local private test network, because that simulates the real Ethereum network, and multiple peers can interact with the contract at the same time.

Ethereum Wallet
(Christian Hjelmslund)
Ethereum Wallet is an application in which you can write, deploy and interact with smart contracts in the Ethereum network. It is mainly for that purpose this project has used Ethereum Wallet, because it provides a nice GUI platform, in which the different functions can be tried out. As mentioned earlier in the paper, every user has a complete copy of the whole Ethereum ledger, which means that to use Ethereum Wallet the whole blockchain has to be downloaded and stored on your device, and at this particular moment the size of Ethereum is over 60 gb. Recently it has been made possible to use an experimental light client version of the Ethereum Wallet, which uses significantly less storage space.
Fortunately Ethereum Wallet can be launched on different test networks, and BlockCycle has been deployed on the Rinkeby test network, which is only around 10 gb. The smaller size isn't the most important thing about a test network, but the fact that every developer gets free ether, making it identical to when the contract is deployed on the main network, minus the fact that nothing has to be paid to deploy the contract, which is ideal for testing.

Geth

(Christian Hjelmslund)
Geth is standing for Go ethereum, and is the implementation of an Ethereum node in the Go programming language. The node is initialized through a terminal, and upon initialization Geth will connect to a blockchain depending on the settings.
After the node is setup and running, from another terminal the command `geth attach` can be run, and from there various commands can be run such as creating accounts, inspecting blocks and deploying contracts. Ethereum Wallet also makes use of Geth and apart from that Geth can be used to

initiate mining.

5 Work process

(Andreas Bundgaard)
This sections contains a chronological view of our collaborated effort. From the beginning, we knew that the project would be more research based, as no formal cryptography related projects were presented through the course. Thus we organized the project through Elmar Wolfgang Tischhauser, and expressed our interest in a cryptocurrency project. Due to the conservative nature of cryptography, not much active research at DTU is being done in blockchain technology, but never the less Elmar agreed to supervise us.

Together with him, we found the initial goals for the project; we were to setup, work with, and understand IBM's HyperLedger Fabric. Later on we could write a smart contract, and if time permitted, setup attacks on our contract, or the network itself. However, as discussed earlier in the report we found that HyperLedger Fabric didn't really suit our idea of a decentralized insurance. Furthermore, HyperLedger Fabric was complex, and allowed for a ton of configuration, much of which is much more suitable for a large supply chain network, than a small test network. These factors, along with Fabric being a permissioned network, made us switch to Ethereum, which was decentralized and had available test networks if we needed them.

Thus after a handful of weeks of research into how blockchains worked and HyperLedger Fabric in particular, we switched frameworks. We divided the workload into further blockchain research along with smart contract development, and setting up a private Ethereum blockchain. It was possible to get several EC2-instances to talk to each other and Geth was used to deploy and interface with the private blockchain. As can be seen in the folder `private_chain` it is relatively trivial to do so. The file `mygenesis.json` specifies Genesis Block, which defines the initial qualities of the network, like the mining difficulty along with the identifying chain ID (necessary for others to specify if they want to connect). Note that it is possible to change some of these parameters later on if there is consensus to do so and some, like the difficulty, can be automatically determined based on the mining power available. The difficulty on the main Ethereum network is dynamically adjusted so one block is produced by the network on average every 15 seconds. On the private implementation this was set at its lowest possible value, which allowed blocks to be generated almost instantaneously, even with the relatively low computing-power of the EC2 instances.

The private network allowed us to test and deploy a basic smart contract in a fast and efficient way. Other considerations such as anonymity and protection against tampering/fiddling is also gained on a private network. Since we were only us on the network, in order to run anything, we had to be the ones mining as well, which helped understand the concept of mining.

As BlockCycle developed, we saw that we needed to include an oracle. On a private network we would have to develop the oracle ourselves, from scratch. This task seemed unnecessary, and instead we opted to try and join one of the many public Ethereum test-networks. On these, there exist oracle contracts with the exact same interface as oracles on the live network. Yet again, some were tasked with finding a proper network, and getting us setup while the others continued development. We ended up both on the Ropsten and Rinkeby networks. From this point on, we all started developing on core features of the contract. As it came together, we assigned one person to start unit testing the application. Again this required a private network with a bridge to the oracle service, which had to be setup, so the test process could be automated.

Finally we started to look at our own(and others) contract for potential security flaws. Due to time constraints and the detour we took into Hyperledger Fabric, the goal of hacking our own smart contract was just out of reach. Notably we found different types of attacks that perhaps could be used, and while they were attempted, we weren't successful in breaking our smart contract.

6 Discussion

6.1 Choice of Blockchain

(Noah Sturis)
One thing that was explored in the project, was setting up a custom blockchain. Initially work began with Hyperledger Fabric, as that was the initial focus of the project, but as we gained more knowledge about the technology, the appreciation of Hyperledger Fabric diminished. Fabric is a permissioned blockchain with its strong suit being allowing private transactions to be made through private channels, and ledgers to be different with different degrees of shared information. This only works because of its permissioned nature, which requires a central actor (called the Certificate Authority) that authenticates and allows actors to participate in the system. The result is that trust is placed in a centralized entity and what initially was enticing about blockchain technology to us - collaboration in the absence of trust - was undermined.

When looked at through the lens of the BlockCycle concept, Hyperledger Fabric seemed pointless, as if a centralized authority was required to grant users access to the network, a traditional database structure could just as well have been used. After successfully setting up a private Hyperledger Fabric blockchain, we decided to abandon that aspect of the project and from there the focus shifted towards Ethereum.
Like Hyperledger Fabric, Ethereum also allows for smart contracts, but more significantly it is a permissionless blockchain, circumventing the problem of having a central authority deciding who gets to participate. It also has a large developer community, which means not only does Ethereum have good official documentation, but the community has already answered a bunch of questions we were bound to have ourselves. We even had opportunities to talk with developers directly, when using their interfaces.

6.2 Consensus Mechanisms

(Noah Sturis)
Ethereum's consensus mechanism is currently the same as Bitcoins: Proof of Work, but they're working on transitioning to a Proof of Stake/Proof of Work hybrid. There is general concern about the nature of Proof of Work's energy consumption and on-chain scalability, but choosing a pure Proof of Stake algorithm also leads to a host of issues. In Proof of Work actual resources (energy) is consumed in the race to find the nonce which leads to a valid block. Changing history in a Proof of Work system additionally requires recomputing all blocks following it, which makes it exponentially more difficult to change things the more blocks are ahead of it.

With a Proof of Stake system, it turns out consensus itself can be difficult to achieve, since the same stake can be applied to multiple competing chains, increasing the chance for a reward with no penalty for choosing the wrong chain. If a penalty for changing ones mind is introduced, it has to outweigh the value of the reward for staking on multiple chains, however this itself has the effect of not wanting to change ones mind, as the penalty for doing so is larger than the potential reward. An illustration of this can be found in the below figure[12].

[12] https://blog.bitmex.com/complete-guide-to-proof-of-stake-ethereums-latest-proposal-vitalik-buterin-interview/

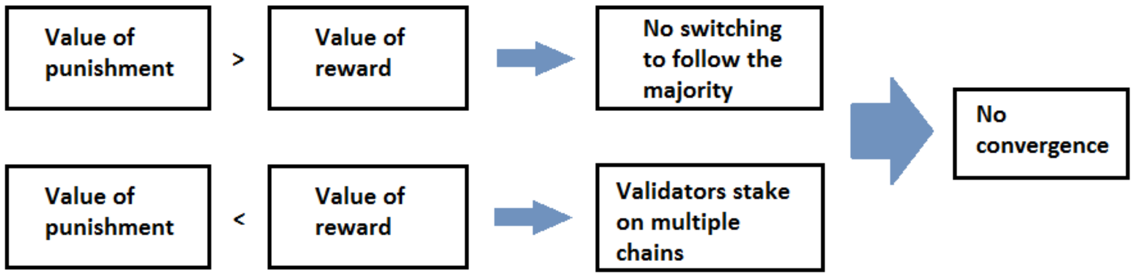


Figure 10: Incentives in a Proof of Stake-system

As opposed to Proof of Work-based chains, the stake is linked to the chain itself and nothing in the real world, which allows for the use on two conflicting chains. This is known as the "Nothing at stake" problem, which is commonly viewed as one of the major issues with Proof of Stake.

6.3 Blockchain with oracle difficulties

(Andreas Bundgaard)
The articles "Do you need a blockchain" and "Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough?" tries to answer the question, is blockchain really worth the hype? After working with the technology, we feel we're somewhat equipped to try and answer. First and foremost as the one article puts it *"...chasing decentralization for the sake of itself..."*[13] is a definitely a concern. However there are some real gains to be made, particularly reducing transaction cost, and to handle claims more efficiently. However that efficiency is dependable on reliable communication to off chain oracles. Naturally this begs the question, is there such a thing as a perfect oracle? The oracle problem is often mentioned, and isn't trivial to answer. We're using the police database to verify whether bikes are stolen or not, but even this comes with limitations. You could easily report your bike stolen, claim the insurance but still have the bike! We aware of this problem, but a solution on a larger scale is needed.

7 Conclusion

(Emir Sircic)
To finish up this report it would be suitable to go through the criteria of success established in the introduction:

- Develop a solid theoretical and practical understanding of how various blockchains work. Which security properties they have, how they resolve consensus and how they execute smart contracts (if they do)
- Setup of a custom blockchain

[13]Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough

- Demonstrate a smart contract on Ethereum or Hyperledger
- Demonstrate an attack on Ethereum or Hyperledger

Most of the criteria have been covered and fulfilled. The majority of this report revolves around understanding the blockchain technology and subjects such as what blockchains are, how they work and how actors on the chain agree on consensus mechanisms have all been covered. Additionally an explanation of how some blockchains includes properties such as smart contracts and these gets executed is present as well, which sums up the first criteria. After gaining this fundamental foundation of the technology a private blockchain was spun up, which was used in the development process of writing a smart contract that also sped up the process considerably since there was no need to synchronize with a large network to test basic functionality. The private network also granted access to construct unit tests on the contract which is a major beneficial property when developing smart contracts in general. The work of writing a smart contract has been covered and a functional and tested contract has been deployed on one of the official test networks on Ethereum, which fulfills the second and third criteria. Unfortunately due to not being able to receive supervision at the end of the project timeline the last criteria has not been met: to demonstrate an attack on Ethereum or Hyperledger. Though some forms of possible security flaws on the smart contract has briefly been discussed, it was not possible to confirm if these security flaws are present or if they are possible at all. All in all the project is deemed a succes and a solid theoretical and practical understanding of the technology has definately been achieved.

References

[1] Lars Ramkilde Knudsen, *Cryptology, how to crack it*, Polyteknisk forlag

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli, *A survey of attacks on Ethereum smart contracts*, https://eprint.iacr.org/2016/1007.pdf

[3] Valentina Gatteschi, Fabrizio Lamberti, Claudio Demartini, Chiara Pranteda and Víctor Santamaría, *Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough?*, www.mdpi.com/1999-5903/10/2/20/pdf

[4] Karl Wüst, Arthur Gervais, *Do you need a Blockchain?*, https://eprint.iacr.org/2017/375.pdf

[5] Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, https://bitcoin.org/bitcoin.pdf

[6] Ethereum whitepaper, https://github.com/ethereum/wiki/wiki/White-Paper

[7] DR. GAVIN WOOD, *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*, https://ethereum.github.io/yellowpaper/paper.pdf

[8] Hyperledger presentation, https://cachin.com/cc/talks/20170106-blockchain-rwc.pdf

[9] Hyperledger Fabric documentation, https://hyperledger-fabric.readthedocs.io/en/release-1.1/ Bitcoin official wiki, https://en.bitcoin.it/

[10] Ethereum stack exchange, https://ethereum.stackexchange.com/

[11] Proof of elapsed time, https://medium.com/kokster/understanding-hyperledger-sawtooth-proof-of-elapsed-time-e0c303577ec1