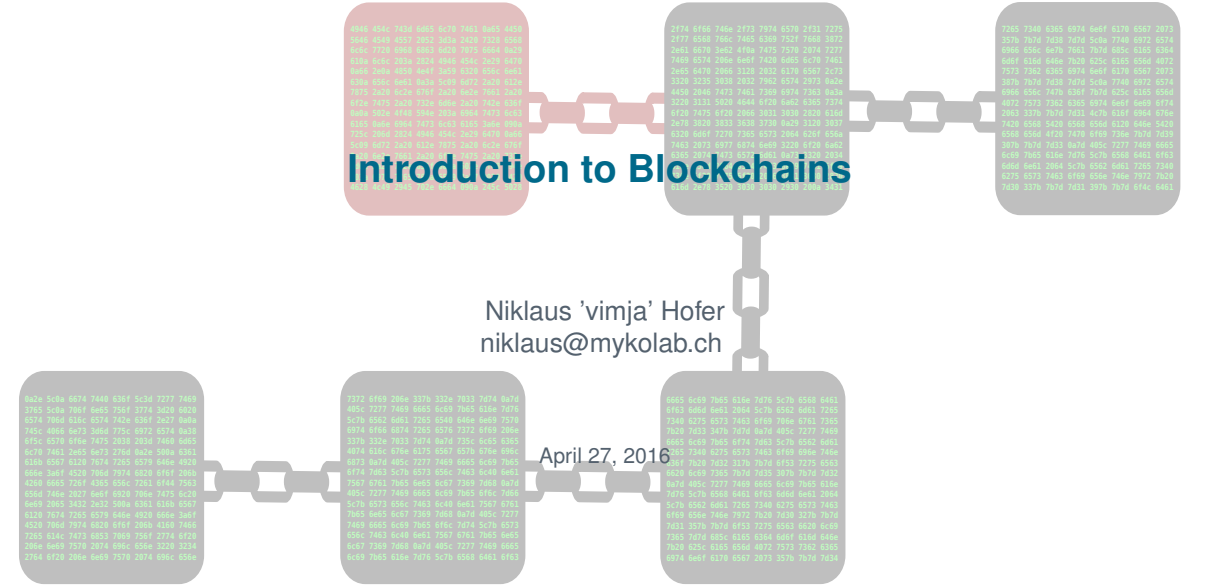


Introduction to Blockchains

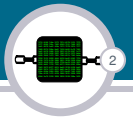
Niklaus 'vimja' Hofer
niklaus@mykolab.ch

April 27, 2016

/me

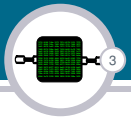
- IT student at BFH
  - Specialisation in Infosec
  - I have been working with Blockchain Technology for 1.5 years now
  - Writing my Bachelor thesis on the analysis of the Bitcoin Blockchain
- Member of the LugBE
- Founding member of the Chaostreff Bern

Disclaimer



I am not associated with any Blockchain related company or organisation

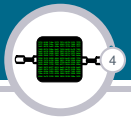
Content I



Content

With this presentation, I want to tell you not only how Blockchains work, but also why they work the way they do and why the ideas and concepts are robust enough.

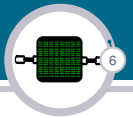
Content II



Pruning client

Section 1

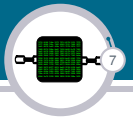
State transition systems



Subsection 1

General concepts

What is a state-transition system



A State-transition-system consists of two elements:

- State
- Transition

A transition brings the system from one state to the next one

$$State_A \overset{transition}{\rightarrow} State_B$$

This works over and over again

$$State_A \overset{transition_\alpha}{\rightarrow} State_B \overset{transition_\beta}{\rightarrow} State_C \overset{transition_\gamma}{\rightarrow} State_D \overset{transition_\delta}{\rightarrow} \ldots$$

State-transition applied to financial systems

We can map virtually any financial system to a state-transition system:

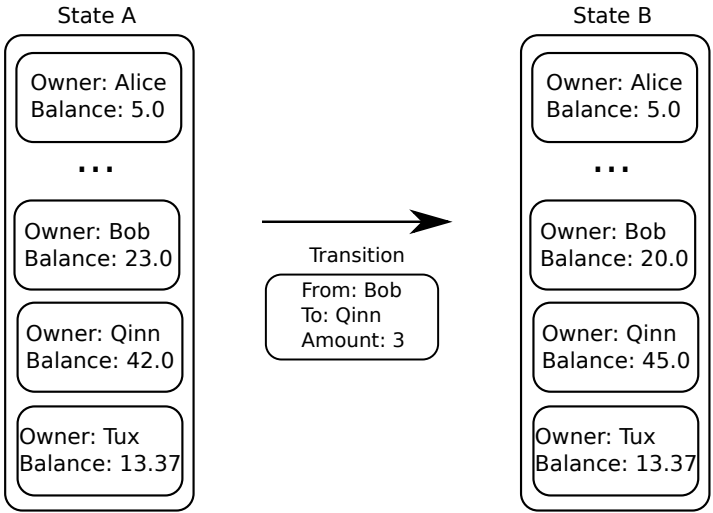
- Banks (especially electronic payments)
- Cash

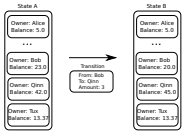
The mapping looks something like this:

State	Collection of all accounts
Account	Owner and associated amount
Transition	Transaction (Moving value from one account to another)

(Also works for other systems of ownership. Especially estate.)

Showcase


State A
Owner: Alice
Balance: 5.0
...
Owner: Bob
Balance: 23.0
Owner: Qinn
Balance: 42.0
Owner: Tux
Balance: 13.37
Transition
From: Bob
To: Qinn
Amount: 3
State B
Owner: Alice
Balance: 5.0
...
Owner: Bob
Balance: 20.0
Owner: Qinn
Balance: 45.0
Owner: Tux
Balance: 13.37



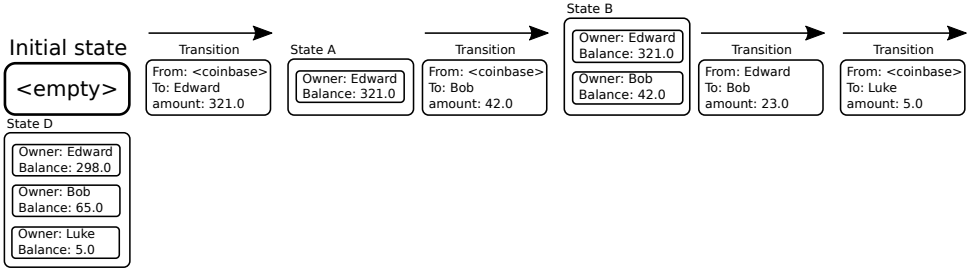
Here we have an example of a (monetary) base-transition system. "State A" is the collection of all accounts. The transition is a transaction that transfers "3" from Bob to Qinn. As a result of this transaction, the system transitions from "State A" into "State B".

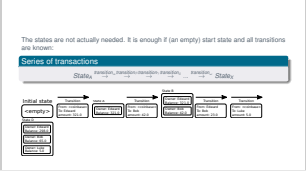
State-transition without states

The states are not actually needed. It is enough if (an empty) start state and all transitions are known:

Series of transactions

$$State_A \xrightarrow{transition_\alpha} \xrightarrow{transition_\beta} \xrightarrow{transition_\gamma} \xrightarrow{transition_\delta} \ldots \xrightarrow{transition_\omega} State_X$$





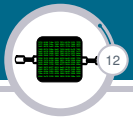
Having an (initial) state, we can apply transitions to form new states. When we have a number of transitions to apply, we actually don't have to remember each of the states created by those transitions. Instead we can apply them all to get the final state directly.

Coinbase, I should probably mention, is the source of the money. This is how new value is added to the system. In the case of the Swiss Financial system, the coinbase would be something like new money printed by the Schweizer Nationalbank.

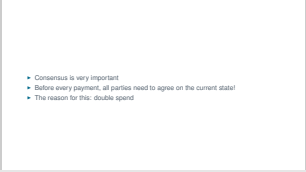
Subsection 2

Double-spend

Consensus

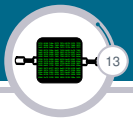


- Consensus is very important
- Before every payment, all parties need to agree on the current state!
- The reason for this: double spend



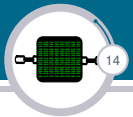
Consensus: All parties of the system agree on the state of the system. I will now show how the system can be exploited if the parties can't agree on a state.

Double spend



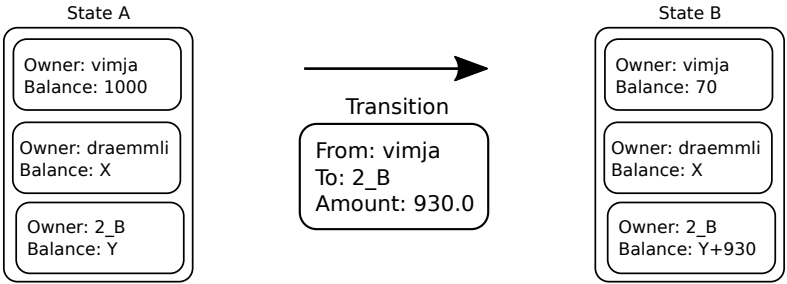
- To spend the same money twice
- Obviously malicious
- Well known attack in the Bitcoin world

Double spend money example



- I have CHF 1000.-
- 2_B has for sale a cool bicycle for CHF 930.-
- draemmli has for sale an old computer for CHF 750.-
- I want to buy both!

Example: 2_B's view

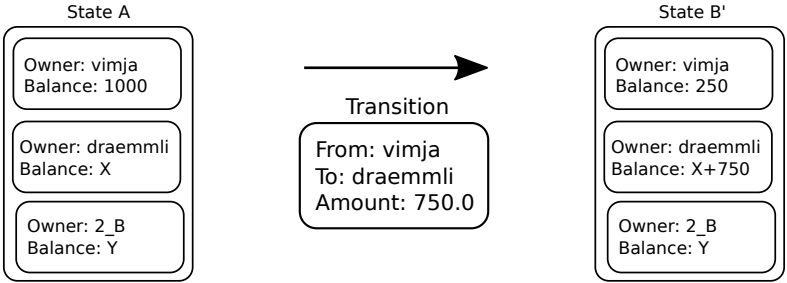


2_B has gotten the money. He now gives me the bicycle.

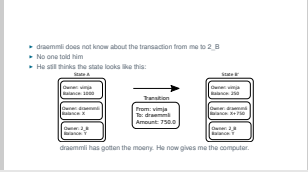
I make a transaction, to transfer the 930 to 2_B. This results in a new state of the system. In this new state, 2_B has received the money from me. Since he has received the money, he hands over the bicycle.

Example: draemmli's view

- draemmli does not know about the transaction from me to 2_B
- No one told him
- He still thinks the state looks like this:

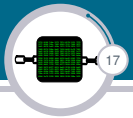


draemmli has gotten the moeny. He now gives me the computer.



Let's imagine draemmli never learned about the previous transaction I made to 2_B. draemmli still thinks I have 1000 on my account. He will happily accept my transaction of 750 onto his account and once he's received the money, I get the computer.

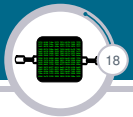
Example: problem



- We now have a problem: I just spent the same money twice
- 2_B and draemmli do not agree on the state
- Their views on the state of the system are incompatible
- This breaks the system:
  - Possibility to spend infinite amounts of money
  - People can't trade with each other

Before we look at a possible solution, let's consider another example.

Double spend estate example I



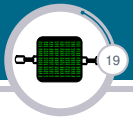
I'm selling an estate at the Lake of Thun:

- At 14 pm I meet with Markus
- I sell him the estate, he gets a receipt
- At 15 pm I meet with draemmli
- I sell him the estate, he gets a receipt

I just double spent my estate:

- I got the money twice
- They both have a receipt
- They can't both own the same estate!

Double spend estate example II

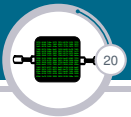


The estate world has a simple solution for this:

- Registry of deeds (Grundbuchamt)
- A central authority
- That central authority controls the state of the system
- Every time an estate is sold (a transition is made), it has to be done via the registry of deeds
- For each transition, the central authority performs certain checks to make sure the transition is compatible with the current state and either accepts or rejects it

That way, everybody can agree on a certain state and that state is always valid.

Solutions for monetary systems



Requirements:

- All parties need to agree on the current state
- All parties need to agree on whether a transition is valid

Currently, there are two ways of achieving this:

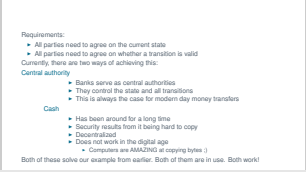
Central authority

- Banks serve as central authorities
- They control the state and all transitions
- This is always the case for modern day money transfers

Cash

- Has been around for a long time
- Security results from it being hard to copy
- Decentralized
- Does not work in the digital age
  - Computers are AMAZING at copying bytes ;)

Both of these solve our example from earlier. Both of them are in use. Both work!



The same solution that worked for the estate system, a central authority controlling the state, can also work for monetary systems. Here banks control the state of the system.

Cash on the other hand does not need a central authority to control the state. Instead the state is secured by physics - I can only spend a coin once and it's hard to copy the coin.

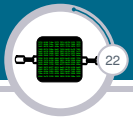
Section 2

Concepts of a Blockchain

Section 2
Concepts of a Blockchain

- I will introduce three important concepts of a Blockchain
- After each concept, we will look at how far we've come, what works, what does not work yet
- Then we will add yet another concept to fix some of the problems
- Only by adding the third concept (POW) we will get a proper Blockchain

A third solution



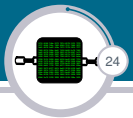
Blockchains can serve as a third solution for the same problem. It is an internet age solution. It provides these (amazing) properties:

- No central authority
- Parties do not need to trust each other
- Parties need no information on who is participating
- ... not even any information on how many others are participating
- And still they can all agree on a state

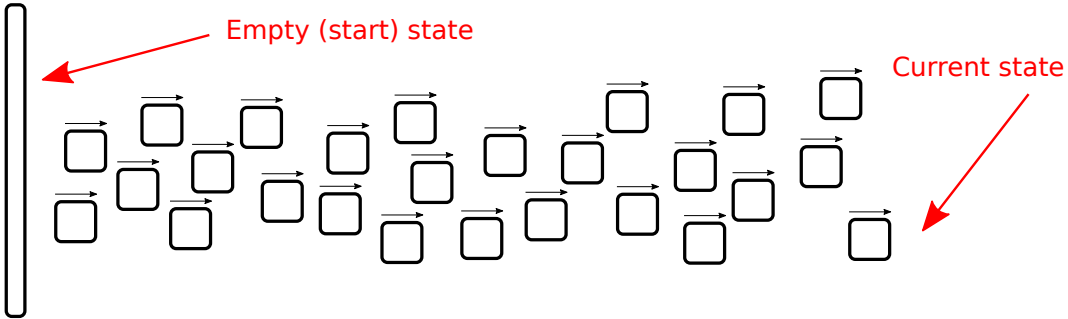
Subsection 1

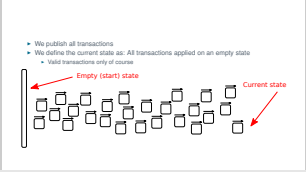
Concept 1: Public ledger

Idea



- We publish all transactions
- We define the current state as: All transactions applied on an empty state
  - Valid transactions only of course

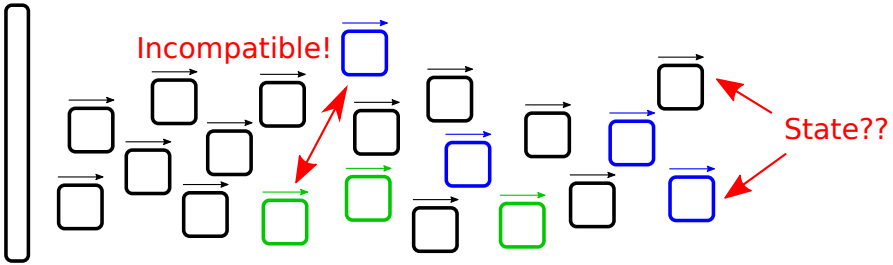


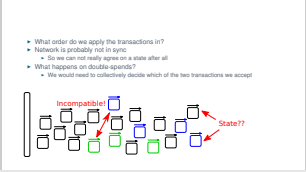


By applying all transactions to the initial state, we get the current state of the system.

Problems

- What order do we apply the transactions in?
- Network is probably not in sync
  - So we can not really agree on a state after all
- What happens on double-spends?
  - We would need to collectively decide which of the two transactions we accept



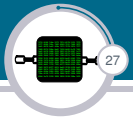


We can't actually decide on a state. There are too many problems. So clearly, this concept on it's own is useless. It does not work at all. So we need to add the second concept...

Subsection 2

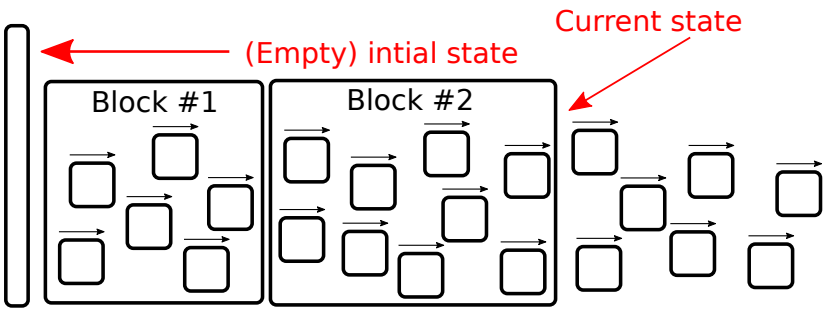
Concept 2: Blocks

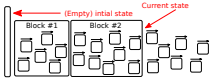
Blocks



- We group the transactions into blocks
- Blocks depend on one-another
- Blocks have to meet certain criteria:
  - All transactions within the block must be valid
  - Transactions within the block have to be compatible with one another
  - Transactions within the block have to be compatible with transactions in earlier blocks
- Anyone can form a new block at any time
- We define the current state as: All transactions within the longest branch of blocks applied on an empty state
  - Transactions only become part of the state once they are inside a block

Blocks (visualized)



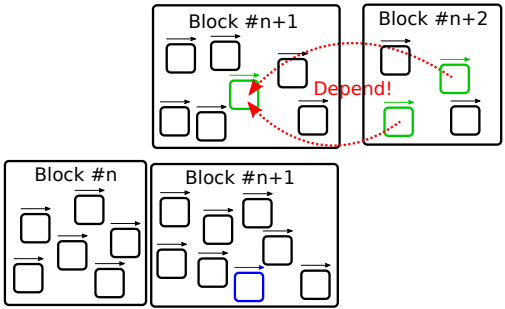


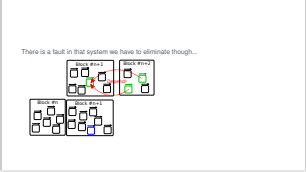
Loose transactions are not part of the state. The state is at the end of the last block. Only when a new block containing the transactions is formed, will they become part of the state.

In case a new block is created but does still not include a certain transaction, that specific transaction will not be part of the state. Only when a transaction is included in a block is it part of the state.

Block chain

There is a fault in that system we have to eliminate though...



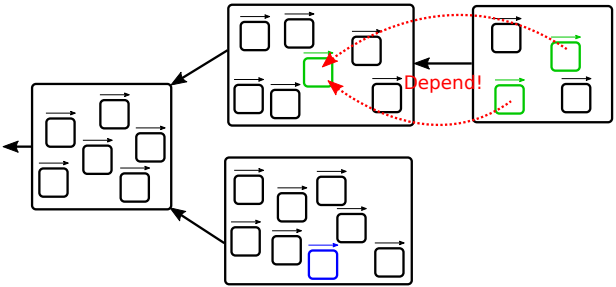


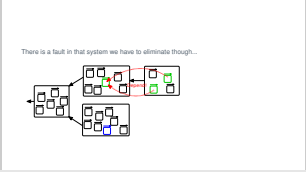
So far, we have numbered the blocks to determine their order. But if there are two blocks that both have the same number but different content, a problem appears. In the image above it's not immediately clear which of the blocks "n+1" the block "n+2" depends on. Only when we look inside the block "n+2" it becomes clear that it contains transactions (the green ones) that depend on the original green transaction. This is tedious and much more complicated examples are easily imagined.

So numbering the blocks is not a good solution. We need a way to better express the dependencies of blocks.

Block chain

There is a fault in that system we have to eliminate though...

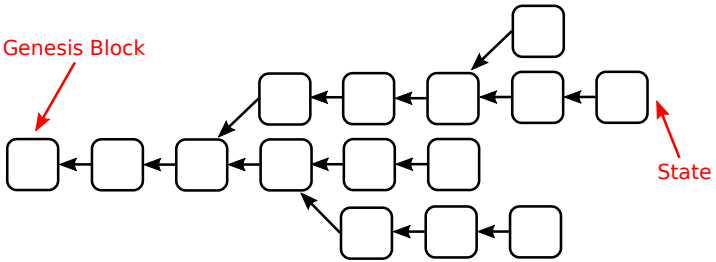


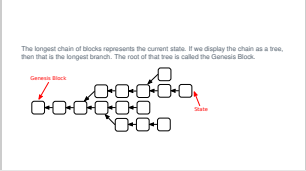


As a solution, each block points to it's previous block/references the previous block. This reference is distinct and thus it is now possible to tell exactly what block a block depends on.
Each block can only depend on exactly one prior block.
For the programmers: This is a bit like a one way linked list.

Longest branch (longest chain)

The longest chain of blocks represents the current state. If we display the chain as a tree, then that is the longest branch. The root of that tree is called the Genesis Block.



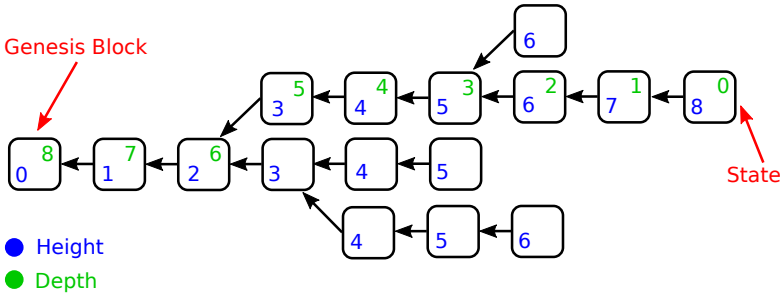


If suddenly, another branch were to become longer, then that branch would then be the state of the system.

Block height and depth

Height identifier

Depth expression of security



Even though we don't use numbers of blocks anymore to determine their dependencies, numbering them can still be useful. It allows for easy referencing of specific blocks. There are two important ways of doing this:

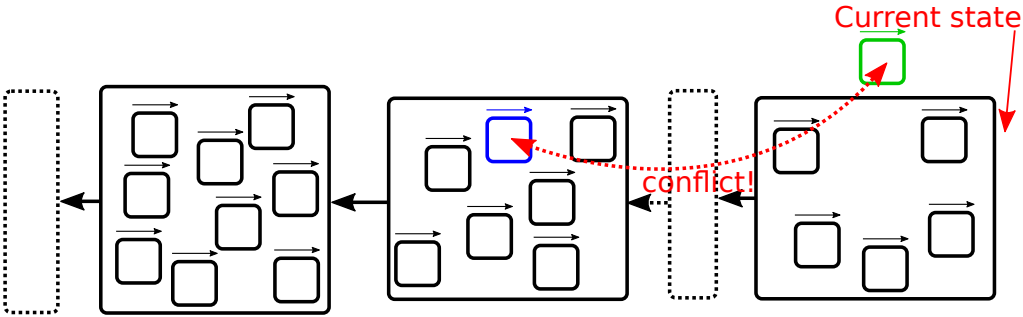
Height The block height expresses the block's distance from the Genesis Block. The height of a block is always the same and does not change when new blocks are added to the system. Every block in the system has a height.

Depth The depth of a block expresses the block's distance from the end of the branch. When a new block is appended to the branch, the depth of every block in the branch is increased by one. The depth is usually only expressed for blocks of the longest branch.

Later on when we introduce POW, the depth will become an important expression of security.

Solving double spend

Following these rules might fix double spend...



Remember: Transactions within the block have to be compatible with transactions in earlier blocks.

Having introduced the second concept, blocks, we would hope that this were to solve the double spend problem. Here is how:

- I make the transaction to 2_B
- Eventually that transaction will be included in a block, it is now part of the state, 2_B has received the money and he hands me over the bicycle
- (further blocks might be added to the system, represented by the dotted block)
- I make a transaction to draemmli
- However, it is in conflict with the transaction to 2_B I made earlier
- Because this new transaction is in conflict with a transaction already on the chain, it will not be included in the chain. A block containing it could be created, but it would be an invalid block and thus not become part of the state.
- Since the transaction never becomes part of the state, draemmli never receives the money and thus never gives me the computer

Unfortunately, this is not the end of the story. Double spend is still possible...

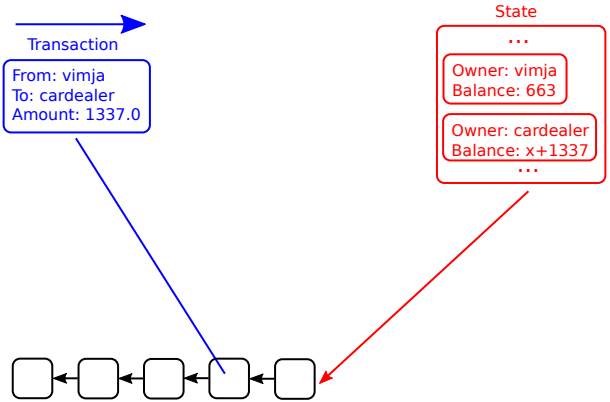
Subsection 3

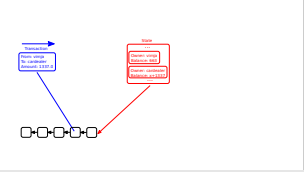
Double spend on Blockchains

Subsection 3
Double spend on Blockchains

The double spend attacks covered in this chapter are the same ones potentially used against real Blockchains. In the next chapter we will discuss how actual Blockchains protect against those.

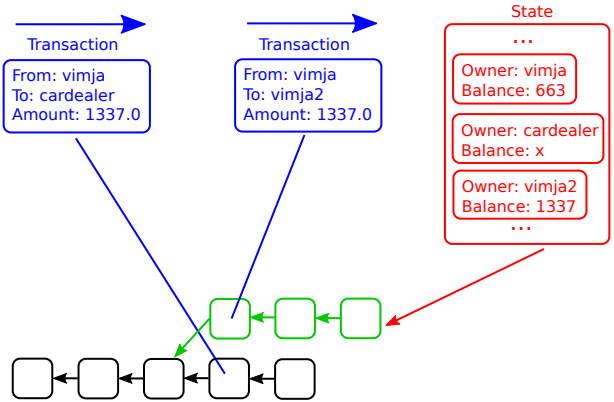
Classic double spend

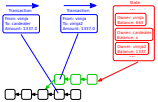




Since these attacks are a little bit more difficult, I'll be stealing something bigger this time. How about a Tesla (car).
I go to the car dealer. Then create a new transaction, transfering the money to the car dealer. A short time later, someone will create a block containing this transaction. So the transaction becomes part of the state, the car dealer has received the money and I get the car. I then drive away with the car. Potentially, further blocks are created.

Classic double spend

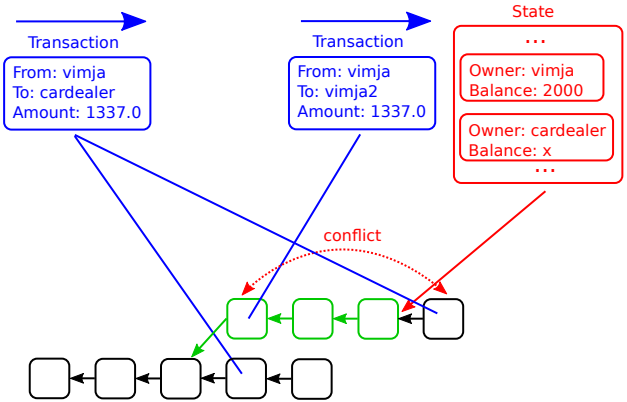




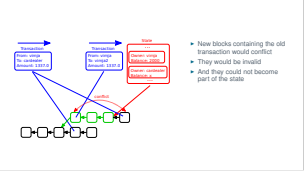
Now I create another transaction, also transfering the same money, but not to the car dealer. Instead, this transaction transfers the money onto a second account that also belongs to me. Then I create a block containing this transaction and quickly add a couple more blocks, until my branch is longer than every other branch in the system.

Finally, I publish that branch. In this new branch, the transaction transfering the money to my second account is part of the state, but the transaction transfering money to the car dealer is not. Since this new branch is the longest chain it is the new state of the system. In this state, the car dealer has never received the money. I have successfully stolen the car!

Explanation

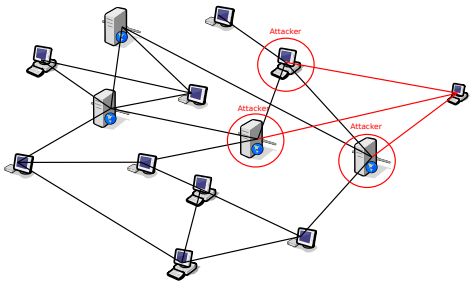


- New blocks containing the old transaction would conflict
- They would be invalid
- And they could not become part of the state



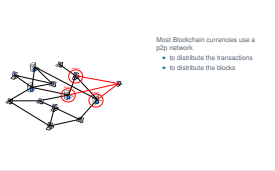
By creating a new transaction (the one transfering the money onto the vimja2 account) that is conflicting with the earlier transaction (the one transfering the money to the car dealer) and putting the new one into my chain/branch, I prevent the first transaction from ever becoming part of the same branch again. A block containing one of them will be in conflict with block containing the other, thus they can't become part of the same branch.

p2p networking attack setup



Most Blockchain currencies use a p2p network

- to distribute the transactions
- to distribute the blocks



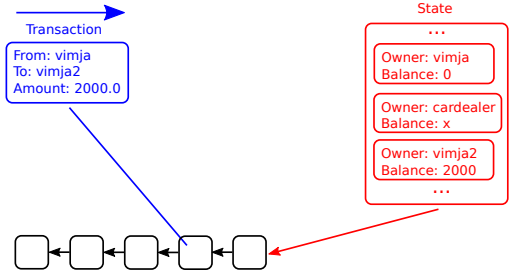
An attacker that controls our connection to the p2p network used to distribute transactions (either by controlling our network connection or by controlling the peers we connect to) has a lot of power. Such an attacker can:

- Control our view of the network
- Present to us blocks and transactions he does not present to the rest of the network
- Hide from us blocks and transactions the rest of the network sees

This can be used to perform a slightly more complicated and sophisticated form of double spend attack.

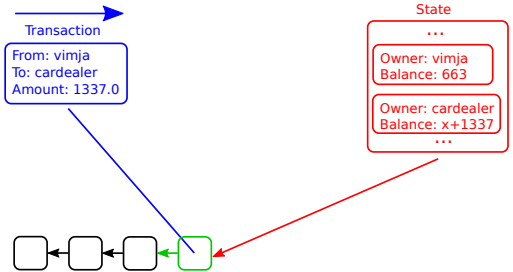
Double spend network attack

Public network view:

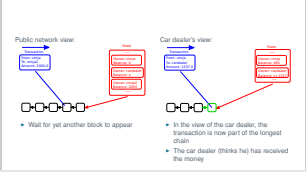


- Wait for yet another block to appear

Car dealer's view:



- In the view of the car dealer, the transaction is now part of the longest chain
- The car dealer (thinks he) has received the money



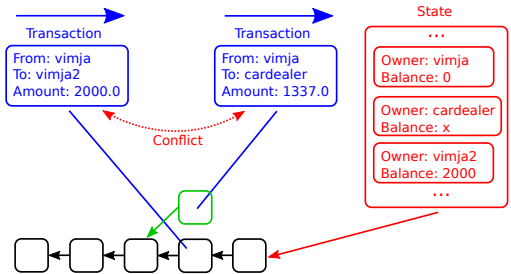
I, the attacker, sit between the network and the car dealer and control which messages go where. Here are the steps I take on each side of the barrier:

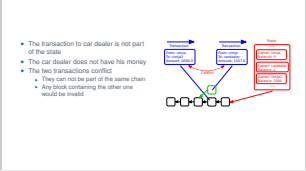
- Create a transaction transfering the money to my second account
- Hide this transaction from the car dealer
- Wait for this transaction to be included in a block
- Wait for yet another block to be formed
- Hide both those blocks from the car dealer

- Create a transaction transfering the money to the car dealer
- Present this transaction to the car dealer, but not to the network
- Create a block containing this transaction
- Present this block to the car dealer, but not to the network

Double spend network attack

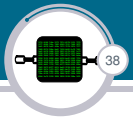
- The transaction to car dealer is not part of the state
- The car dealer does not have his money
- The two transactions conflict
  - They can not be part of the same chain
  - Any block containing the other one would be invalid



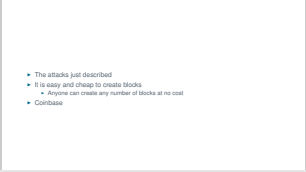


Eventually, I stop the attack. Now the car dealer becomes a part of the network again. The car dealer and the rest of the network synchronize their blocks and transactions.
The car dealer's branch is the shorter one. But in the other branch, which is the state of the system, he has never received the money.
Again, I have created a conflicting transaction to prevent the one to the car dealer from ever becoming part of the state again.

Remaining problems



- The attacks just described
- It is easy and cheap to create blocks
  - Anyone can create any number of blocks at no cost
- Coinbase



With the last two attacks, which were both successful, I have demonstrated that our current system is not good enough. Clearly just the two concepts introduced so far are not enough to form a strong system. So we have to introduce yet another concept.
By doing so, we complete the Blockchain. This is why there is a new section at this point.

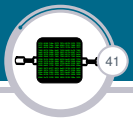
Section 3

Blockchain

Subsection 1

POW and mining

Concept 3: POW I



- Problem: Creating blocks is easy
- Solution: Make creating blocks hard
- To create a new block, one has to solve a problem
  - The solution gets included in the block
  - It's called a Proof of Work (POW)
- We call this: Mining

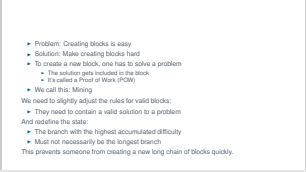
We need to slightly adjust the rules for valid blocks:

- They need to contain a valid solution to a problem

And redefine the state:

- The branch with the highest accumulated difficulty
- Must not necessarily be the longest branch

This prevents someone from creating a new long chain of blocks quickly.



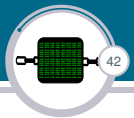
The problem needs to depend on the input. The input, of course, are the transactions that are part of the block. For a different set of transactions, the solution will need to be different as well.

This is to prevent someone from pre computing a large set of solutions and then using it to suddenly create a long branch. If the problem depends on the input, work on solving the problem can only start once the exact transactions making it into the block are known.

The process of finding the solution is called mining.

The solution to the problem is attached to the block and published. Everyone receiving the block can easily check if the solution is a correct one.

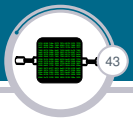
Concept 3: POW II



The problem has to meet certain criteria. It needs to:

- be Hard! Solvable only by brute force
- be adaptable in difficulty (to the amount of power available to the network)
- depend on the input (the block in question)
  - prevent pre compute attacks

Reward

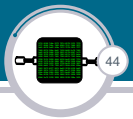


- There needs to be an incentive for creating new blocks
- We have no way of creating new money yet
- Where does it come from (we start with an empty state)

We solve both problems at once:

- The miner of a block gets brand new money
- Done via a coinbase transaction
  - Transfers money from nowhere to the miner's account
- Thus the term mining
- Often called a reward
- Additional idea: Transaction fees

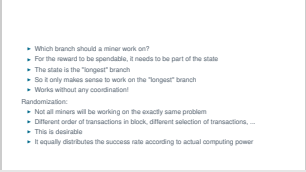
Mining I



- Which branch should a miner work on?
- For the reward to be spendable, it needs to be part of the state
- The state is the "longest" branch
- So it only makes sense to work on the "longest" branch
- Works without any coordination!

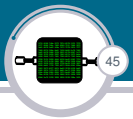
Randomization:

- Not all miners will be working on the exactly same problem
- Different order of transactions in block, different selection of transactions, ...
- This is desirable
- It equally distributes the success rate according to actual computing power



Randomization: If all miners were to work on the exact same problem, that would be a problem: They would all be brute forcing the problem. So the one with the fastest computer would always find the solution first. In practice this is not a problem, because they will not all be working on the same problem: The coinbase transaction, for one, will be different for each miner. And also the order of transactions might be different. So since the inputs are different and the problem depends on the input, they are working on different problems. This works very well. In reality, a miner that has 10% of the network's computational power, will be the first to finish work on a block in about 10% of the cases.

Mining II



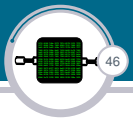
This is where the 50% attack comes from:

- All miners work with their combined power on the longest branch
- An attacker attempting to create a longer branch needs to produce blocks faster than all the other miners combined
- For that he needs more power than all the other miners combined
- He needs more than 50% of the network's power

Block creation:

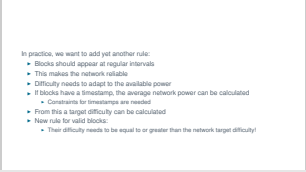
- Whilst a block is created, new transactions pile up
- As soon as a block is done, all miners start working on the next block
- It will contain all the transaction that showed up whilst the last one was created

Network difficulty



In practice, we want to add yet another rule:

- Blocks should appear at regular intervals
- This makes the network reliable
- Difficulty needs to adapt to the available power
- If blocks have a timestamp, the average network power can be calculated
  - Constraints for timestamps are needed
- From this a target difficulty can be calculated
- New rule for valid blocks:
  - Their difficulty needs to be equal to or greater than the network target difficulty!



We want blocks to appear at regular intervals (how long these intervals are, is of lesser importance). That way, when we create a new transaction, we can estimate how long it will take for the transaction to be included in a new block and thus become part of the state.
So we define a difficulty target. When the network's computational power changes, we adapt the difficulty accordingly. That way, the time between blocks stays the same.
Blocks that include a solution to a problem which is not hard enough are invalid.
This makes the distinction between the longest branch and the branch with the highest accumulated difficulty almost completely obsolete. In almost every case, the longest branch and the one with the highest accumulated difficulty will be the same.

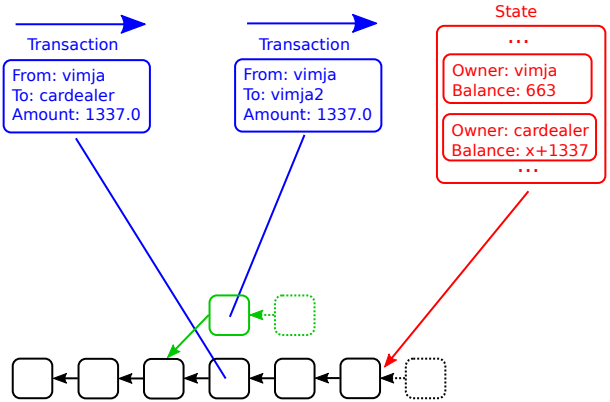
Subsection 2

Solving double spend

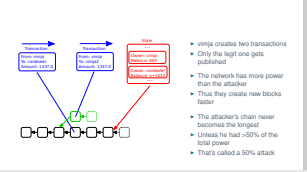
Subsection 2
Solving double spend

Let's revisit the double spend attacks from earlier. The two attacks will be the same as in the chapter before, but now we have POW to protect from the attacks. Let's see how this goes...

50% attack



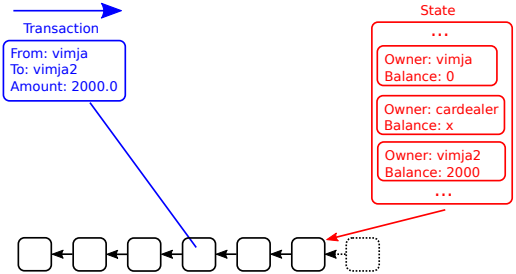
- vimja creates two transactions
- Only the legit one gets published
- The network has more power than the attacker
- Thus they create new blocks faster
- The attacker's chain never becomes the longest
- Unless he had >50% of the total power
- That's called a 50% attack



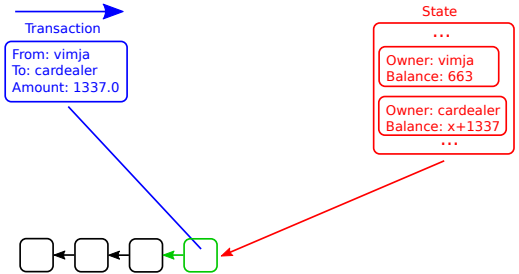
Now the miners are constantly working on new blocks. This is represented by the dotted blocks.
The situation is largely the same as it was for the same attack in the last chapter. However, this time, the attacker can't easily create a longer branch. He needs to mine each block. This is a lot of work.
Since the attacker's mining power is less than the combined power of all the other miners in the network, the attacker creates new blocks at a slower rate. Thus, the attacker's branch never becomes the longest one and never becomes the state of the system.
The attack has successfully been prevented!
The only way the attack could succeed is if the attacker had more mining power than all the other miners combined. That is to say, the attacker would need more than 50% of the network's mining power. Thus the term 50% attack.

Network attack

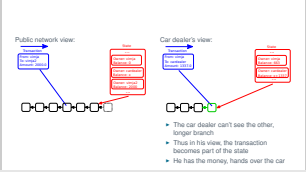
Public network view:



Car dealer's view:



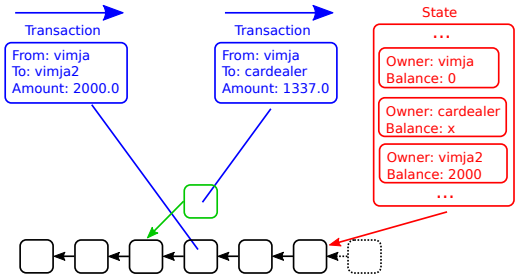
- The car dealer can't see the other, longer branch
- Thus in his view, the transaction becomes part of the state
- He has the money, hands over the car



This attack too, is largely the same as in the previous chapter.
The attacker now has to put in effort to create the block he presents to the car dealer. Since the rest of the miners are faster at creating new blocks than the attacker is, the branch on the public network is the longer one.
However, since the car dealer can't see the network's branch, he accepts the attacker's block as the longest chain. So the car dealer thinks he has the money and hands over the car.

Network attack

- The transaction to car dealer is not part of the state
- The car dealer does not have his money



Again, when the attack stops and the car dealer synchronizes with the rest of the network, it becomes clear that the car dealer's branch is the shorter one. So this kind of attack still works despite POW.

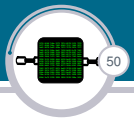
However, on the following slides, I will explain why such an attack is not worth the effort.

Short discussion: 10 minutes, the block time used by Bitcoin, is a long time. If you buy, say, an ice cream, you don't want to wait 10 minutes for your transaction to be processed. Because of this, some people will accept transactions as soon as they appear on the network - given of course, that the transaction is valid and not in conflict with any other transaction that is already part of the chain.

This will work fine in most cases, and if the ice cream seller gets tricked once or twice a week, he can probably still run his business. However, this practice is NOT SECURE AT ALL. Transactions should only be accepted as valid once they are included in a block.

This very network attack demonstrates why. If the car dealer were to accept transactions that are not yet in any block, all the attacker would have to do was to send a transaction to the car dealer. He would not have to put in the time to mine an extra block and thus the mechanisms described on the following slides would not work.

What happened



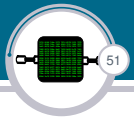
- The attack still worked

But:

- It now costs the attacker effort and time
- The attacker needs resources
- He can use these resources to
  - Mine on the network
  - Execute the attack
- But only ever one at a time
- So while he is doing the attack, the attacker can’t be mining
- And this is expensive

Just how expensive exactly?

The costs of the attack



- Attacker power: $20\% = \frac{1}{5}$ of the network
- Network avg. block time: 10min
- Block reward: 25 Btc

The attacker needs to create one block:

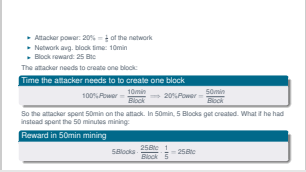
Time the attacker needs to to create one block

$$100\%Power = \frac{10min}{Block} \implies 20\%Power = \frac{50min}{Block}$$

So the attacker spent 50min on the attack. In 50min, 5 Blocks get created. What if he had instead spent the 50 minutes mining:

Reward in 50min mining

$$5Blocks \cdot \frac{25Btc}{Block} \cdot \frac{1}{5} = 25Btc$$

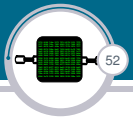


The attacker can either use the power to perform the attack, or to do legitimate mining in the main network. The attack, in this example, would take 50 minutes. Spending 50 minutes performing the attack, means that, for 50 minutes, the attacker can't be mining.

But in that time, if the attacker were mining instead of attacking, he would mine (on average) one block and earn 25 Btc.

So the attacker has to decide: Should he perform the attack, or should he earn 25 Btc. Obviously, he will go for the more profitable choice. Performing the attack then, is only worth it, if the attacker can steal more than 25 Btc in these 50 minutes.

Protecting from the attack

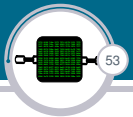


- Instead of doing the attack, the attacker could have made 25 Btc mining
- The attack was only profitable, if the gain was >25 Btc
- In every other case, spending the time mining would have been more profitable

Attacker's power

The power of the attacker does not matter. An attacker with more power needs less time, but he looses more money per time.

Waiting



Waiting for 3 Blocks, 20% attacker power

$$20\%Power \implies \frac{50min}{Block} \implies \frac{150min}{3Block}$$

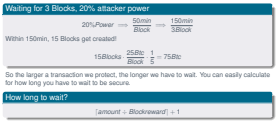
Within 150min, 15 Blocks get created!

$$15Blocks \cdot \frac{25Btc}{Block} \cdot \frac{1}{5} = 75Btc$$

So the larger a transaction we protect, the longer we have to wait. You can easily calculate for how long you have to wait to be secure.

How long to wait?

$$\lceil amount \div Blockreward \rceil + 1$$



To protect ourselves from an attacker, we don't hand out the goods immediately. Instead, once the transaction has been included in a block, and thus become part of the state, we wait until some more blocks have been added to the same branch.

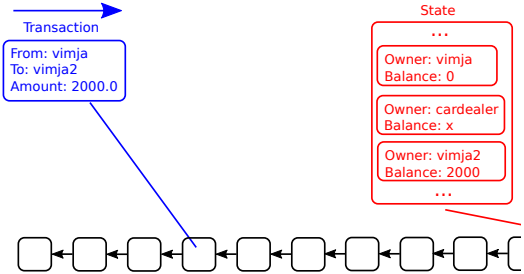
If we were under attack, the attacker would have to create those additional blocks as well. It would cost him even more time to do so. All this time the attacker spends on creating the additional blocks he can't spend mining. Thus it costs him money. The longer we wait, the more expensive an attack becomes.

We can calculate how expensive the attack gets. So we can calculate how long we have to wait (depending on how large the transaction we receive is) for an attack to not be profitable.

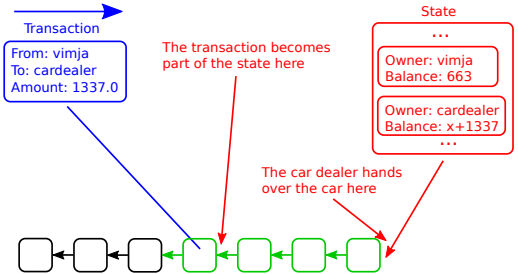
If we do this, then an attack is never ever worth it. It is then more profitable for the attacker to NOT perform the attack! So even though the attacker could attack us, he doesn't want to!

The car dealer's protection

Public network view:



Car dealer's view:



If the car dealer, after the transaction has been included in a block (and thus has become part of the state), waits for another few blocks to be added on top of that, then the attacker has to create all these additional blocks. This will take the attacker a long time. During this time, the attacker can't use the computational power to mine on the main network. So he has to do without the block rewards he could get during that time.

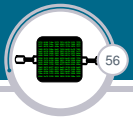
If the car dealer waits for enough blocks, the attack will become so expensive that using the power to mine instead is more profitable.

Obviously, three (as shown above) is not the correct number of blocks to wait. The correct number of blocks to wait depends on the price of the car and the block reward.

Subsection 3

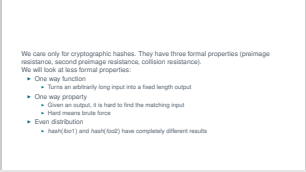
Implementing POW

A word on Hahes



We care only for cryptographic hashes. They have three formal properties (preimage resistance, second preimage resistance, collision resistance).
We will look at less formal properties:

- One way function
  - Turns an arbitrarily long input into a fixed length output
- One way property
  - Given an output, it is hard to find the matching input
  - Hard means brute force
- Even distribution
  - *hash*(*foo*1) and *hash*(*foo*2) have completely different results



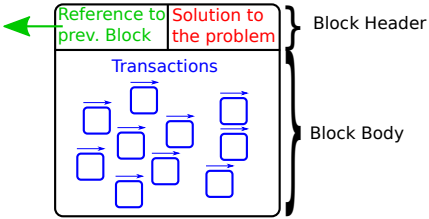
I will be talking a lot about hashes in the coming slides. In Blockchains, they are often used to solve a variety of problems. This slide is for those who do not yet know what a hash is.

Block Header

At the minimum, a block contains:

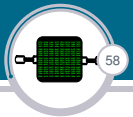
- Transactions
- Reference to the previous block
- Solution for the challenge

The latter two need to be stored somewhere. We call that the block header.



People rarely talk about block bodies though.

How it works with hashes I



Easy for the reference to the previous block:

- The previous block's hash
- If the block changed (even just 1 Bit) the hash would change
- Virtually impossible to fake (see slide on hashes)
- Called the previous-block-hash or prev_blk_hash

We can also use hashes for the problem that needs solving:

Problem/Challenge Create a block which's hash starts with *n* Bits of value 0

Solution A nonce, stored in the solution field of the block header

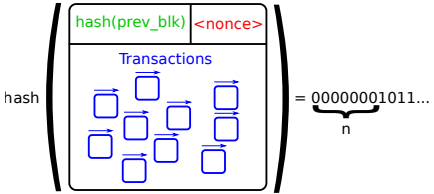
mining Iterate (brute force) through the nonce, until the block's hash starts with *n* 0s

Difficulty *n* represents the difficulty. The more leading 0s are required, the harder it is

Difficulty manipulation Changing *n* changes the difficulty. The difficulty is non-linear

A block references the previous Block via the previous block's hash. This is called the previous block hash, or prev_blk_hash for short.
Modern Blockchain designs use POW algorithms that are more complicated than hashing. The algorithms used today are all memory-hard, that is to say, to solve them, one needs a lot of memory (RAM). This makes building dedicated mining hardware (so called ASICs) expensive.
ASICs can be bad for the security of a Blockchain/POW based system. I will not discuss the reasons for this here though.

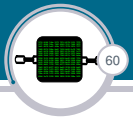
How it works with hashes II



A few notes:

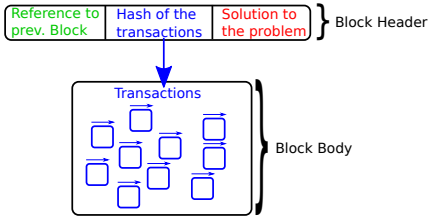
- This was often done in older Blockchains
- Newer Blockchains use other things
  - bcrypt, scrypt, ...
  - Usually things that are memory hard
  - However, on a high level view, it works similar to hashes

Decoupling the header from the content I

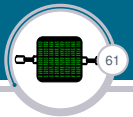


It can make sense to decouple the block header from the body. We do it like this:

- Create a hash of all the transactions
- Add this to the block header

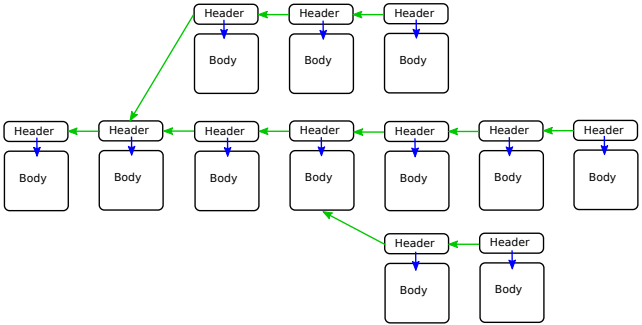


Decoupling the header from the content II

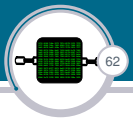


This does a number of things:

- The prev_blk_hash is now actually the hash of the previous block's header
- So each header references the previous header
- Each header also references its block body, that is the transactions



Decoupling the header from the content III



What does this mean for security?

- The transactions are all still just as secure
  - Changing a single transaction would change the body's hash
- It's now possible to store the bodies and headers separately
- The chain can be built from the headers only
  - Creating a new header is as hard as creating a new block
  - Creating a fake header is as hard as creating a fake block (it's impossible)
- The bodies can be pulled from anywhere and then checked against the headers

We will see how this can be used to create amazing functionality later on.

Section 4

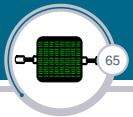
The Bitcoin Blockchain

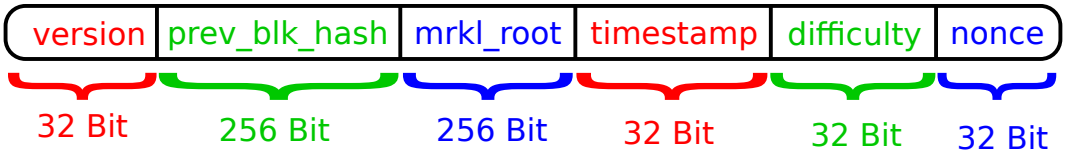
I talked a lot about generic Blockchain designs and concepts. To get a better idea of how these concepts are implemented, we will now take a look at the Bitcoin Blockchain.

Subsection 1

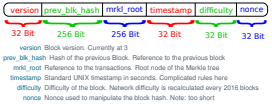
The structure of a block

Header



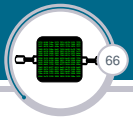


version	Block version. Currently at 3
prev_blk_hash	Hash of the previous Block. Reference to the previous block
mrkl_root	Reference to the transactions. Root node of the Merkle tree
timestamp	Standard UNIX timestamp in seconds. Complicated rules here
difficulty	Difficulty of the block. Network difficulty is recalculated every 2016 blocks
nonce	Nonce used to manipulate the block hash. Note: too short



The image above shows the header of a Bitcoin Block.
Note that the nonce is only 32 bit long. By now, this is too short for the POW.
So additionally, the coinbase transaction gets modified, to modify the mrkl_root. More details on that will follow on a later slide.
Rules for block Timestamp:
"A timestamp is accepted as valid if it is greater than the median timestamp of previous 11 blocks, and less than the network-adjusted time + 2 hours. 'Network-adjusted time' is the median of the timestamps returned by all nodes connected to you." - The Bitcoin Wiki: `https://en.bitcoin.it/w/index.php?title=Block_timestamp&oldid=51392`
2016 Blocks equals 14 days.

Hashes and POW

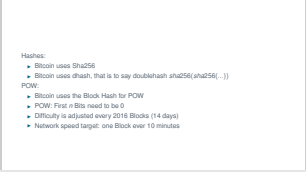


Hashes:

- Bitcoin uses Sha256
- Bitcoin uses dhash, that is to say doublehash $sha256(sha256(...))$

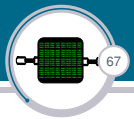
POW:

- Bitcoin uses the Block Hash for POW
- POW: First n Bits need to be 0
- Difficulty is adjusted every 2016 Blocks (14 days)
- Network speed target: one Block ever 10 minutes



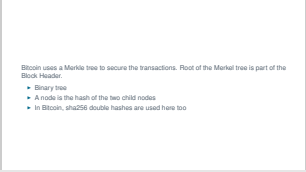
Every Bitcoin client compares the actual time it took to generate these blocks with the two week goal and modifies the target by the percentage difference. Each node calculates the network difficulty independently from all other nodes. It is then, important that they all get the same result and thus that they all use the same algorithm.

Merkle tree



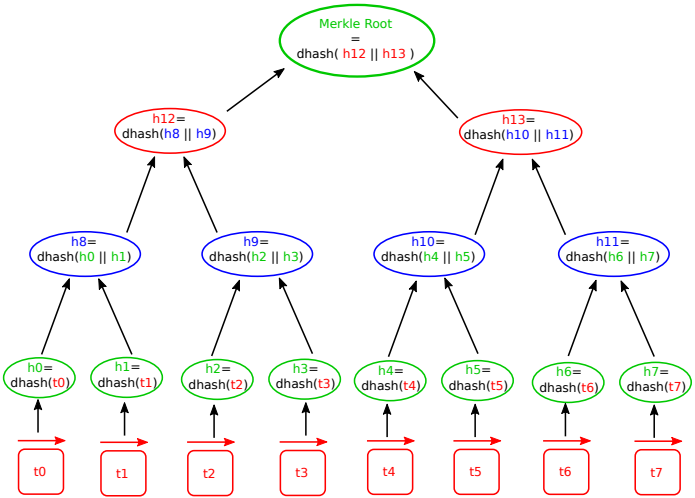
Bitcoin uses a Merkle tree to secure the transactions. Root of the Merkel tree is part of the Block Header.

- Binary tree
- A node is the hash of the two child nodes
- In Bitcoin, sha256 double hashes are used here too

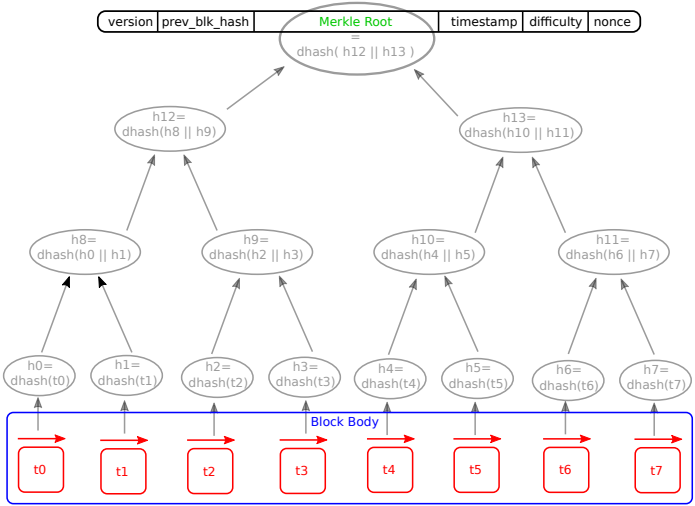


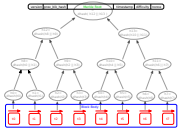
We have already seen the term mrkl_root in the image of the Bitcoin Block Header. It stands for Merkle root and means the root node of a Merkle tree. The Merkle tree is used to secure the transactions and to finally form one hash that represents all transactions and can be put into the block header. The Merkle tree, as used by Bitcoin, is a binary tree. Every node, except the leave nodes, is formed by creating the doublehash of the concatenation of the two child nodes.

Merkle tree visualization



Merkle tree visualization



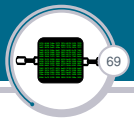


The Merkle Root is put into the Block Header. The collection of transactions become the block body. The rest of the Merkle tree is then no longer needed. If someone validates the block, they retrieve both the header and the body, then create the Merkle tree from the body they retrieved and compare the Merkle root to the value from the block header. If the two values match, everything is in order

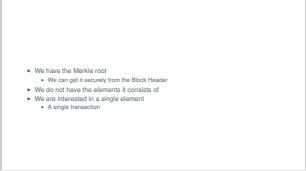
It's important that the order of blocks in the body is preserved, otherwise the result of forming the Merkle tree would be a different one.

All transactions are secure. If one would be changed or replaced, the hash of that transaction would change, which would be reflected in the node above, which in turn would change that node's parent and so on until finally, the Merkle root would be a different one.

Merkle branch

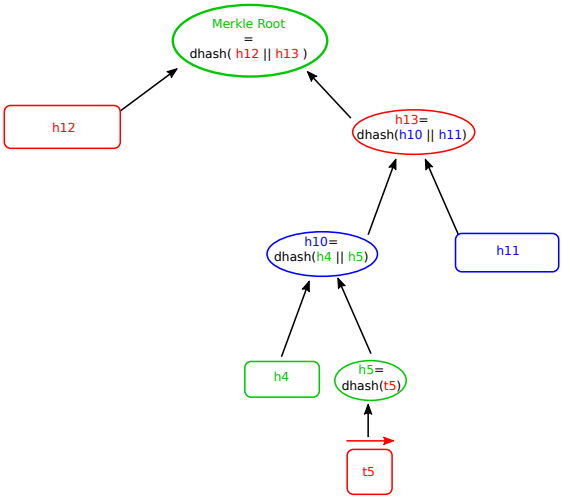


- We have the Merkle root
  - We can get it securely from the Block Header
- We do not have the elements it consists of
- We are interested in a single element
  - A single transaction



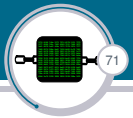
The Merkle tree allows us to secure all the transactions. Even more interesting though is another feature - the Merkle branch. The Merkle tree allows for a single branch to be extracted and then stored in a space efficient yet safe way.

Merkle branch visualization



To form the Merkle branch, only the values in rectangular shapes are required, the values in the ellipses can then be calculated.
In this example, where we want to check Transaction t5, we need to download only t5 and the hashes h4, h11 and h12.
From these four values, we can form the Merkle branch and get the Merkle root.

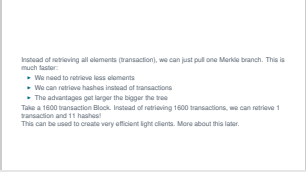
Merkle branch advantages



Instead of retrieving all elements (transaction), we can just pull one Merkle branch. This is much faster:

- We need to retrieve less elements
- We can retrieve hashes instead of transactions
- The advantages get larger the bigger the tree

Take a 1600 transaction Block. Instead of retrieving 1600 transactions, we can retrieve 1 transaction and 11 hashes!
This can be used to create very efficient light clients. More about this later.

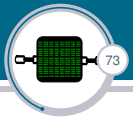


So to prove that a specific transaction is actually part of a block, the Merkle branch for this specific transaction can be retrieved. The Merkle branch will, especially for very large blocks, be smaller than the entire body.
Only the specific transaction plus log_2(number of transactions in the block) hashes need to be downloaded to form and verify the Merkle branch for a transaction.
Downloading and verifying the Merkle branch is just as secure as downloading the entire Block body.

Subsection 2

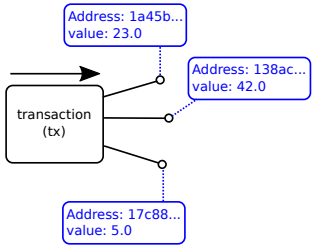
The structure of transactions

Transaction (Outputs)



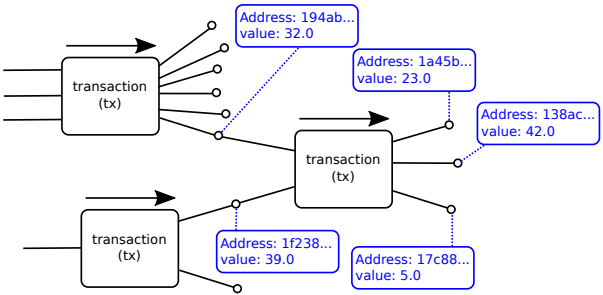
Bitcoin uses an Unspent Transaction Output design:

- A transaction has outputs
  - An output has an associated address and value
  - The address represents the recipient. Think of it as an account number
  - The value gets transfered to that address
  - There can be one to many outputs



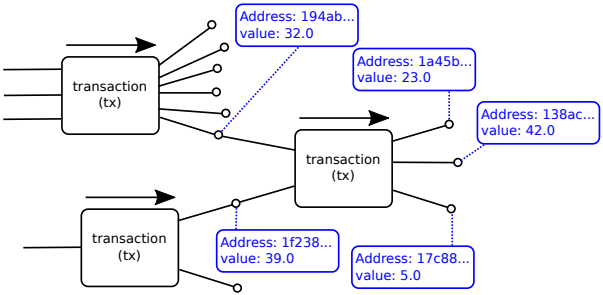
Transaction (Inputs)

- A transaction has inputs
  - This is where the money comes into the transaction
  - Each input refers an output of a previous transaction, it spends that output
  - An output can only be spent once and only by exactly one input
  - The input spends the entire amount of the referenced output
  - There are one to many inputs

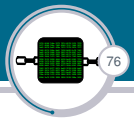


Transaction (...)

- An output can only be spent by the owner of the address
  - (the holder of the account)
- The total value of all inputs needs to be $\geq$ than the value of all outputs
  - The difference between input amount and output amount is the transaction fee



utxo



- An output that is yet to be spent is called an unspent transaction output, utxo
- The balance of an address (think account) is determined by the utxos for that account
- The balance equals the total value of all utxo for that address

Remember our definition of the state of the system from earlier:

- Collection of all accounts, which in turn have a balance

Definition

The state of the Bitcoin system is the collection of all utxo

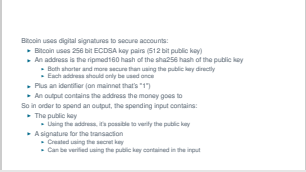
Address

Bitcoin uses digital signatures to secure accounts:

- Bitcoin uses 256 bit ECDSA key pairs (512 bit public key)
- An address is the ripmed160 hash of the sha256 hash of the public key
  - Both shorter and more secure than using the public key directly
  - Each address should only be used once
- Plus an identifier (on mainnet that's "1")
- An output contains the address the money goes to

So in order to spend an output, the spending input contains:

- The public key
  - Using the address, it's possible to verify the public key
- A signature for the transaction
  - Created using the secret key
  - Can be verified using the public key contained in the input

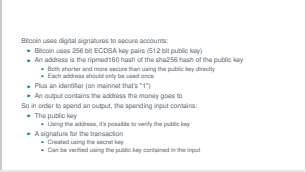

Bitcoin uses digital signatures to secure accounts:

- Bitcoin uses 256 bit ECDSA key pairs (512 bit public key)
- An address is the ripmed160 hash of the sha256 hash of the public key
  - Both shorter and more secure than using the public key directly
  - Each address should only be used once
- Plus an identifier (on mainnet that's "1")
- An output contains the address the money goes to

So in order to spend an output, the spending input contains:

- The public key
  - Using the address, it's possible to verify the public key
- A signature for the transaction
  - Created using the secret key
  - Can be verified using the public key contained in the input

It's called a crypto currency, but so far we haven't used a whole lot of cryptography (other than the cryptographic secure hashes).
Bitcoin (as do other crypto currencies) uses public-key cryptography to secure transactions. Transactions get signed with the secret key. The hash of the public key is used as and address to which money can be transfered.
There are different types of addresses (not discussed here) and there are several networks. To make it possible to distinguish the different types of addresses and the different networks they are part of, a set of prefixes is defined. The "normal" Bitcoin addresses (Pay 2 pubkey addresses) have the Prefix "1".


Bitcoin uses digital signatures to secure accounts:
- Bitcoin uses 256 bit ECDSA key pairs (512 bit public key)
- An address is the ripmed160 hash of the sha256 hash of the public key
  - Both shorter and more secure than using the public key directly
  - Each address should only be used once
- Plus an identifier (on mainnet that's "1")
- An output contains the address the money goes to

So in order to spend an output, the spending input contains:
- The public key
  - Using the address, it's possible to verify the public key
- A signature for the transaction
  - Created using the secret key
  - Can be verified using the public key contained in the input

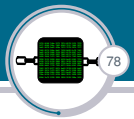
Publishing the hash of the public key rather than the public key itself is a question of security: As soon as the public key is actually public, someone can start attacking it (trying to find the matching secret key). If only the hash of the public key is published, such an attack is not feasible.

However, in order to spend an output, one needs to publish the matching public key, so that other people can verify the signature. From that moment on, the key pair can potentially be attacked. This is no danger for the output we just spent, but it is a potential danger for other outputs that point to the same address.

Because of this, a Bitcoin address should only ever be used once.

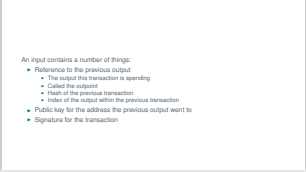
So for each transaction you receive, a new key pair has to be created. But managing (potentially) hundreds of key pairs could become very tedious. To circumvent this problem, Bitcoin clients/wallets can use key derivation to securely derive a (virtually) infinite number of new key pairs from a single secret.

What an input looks like



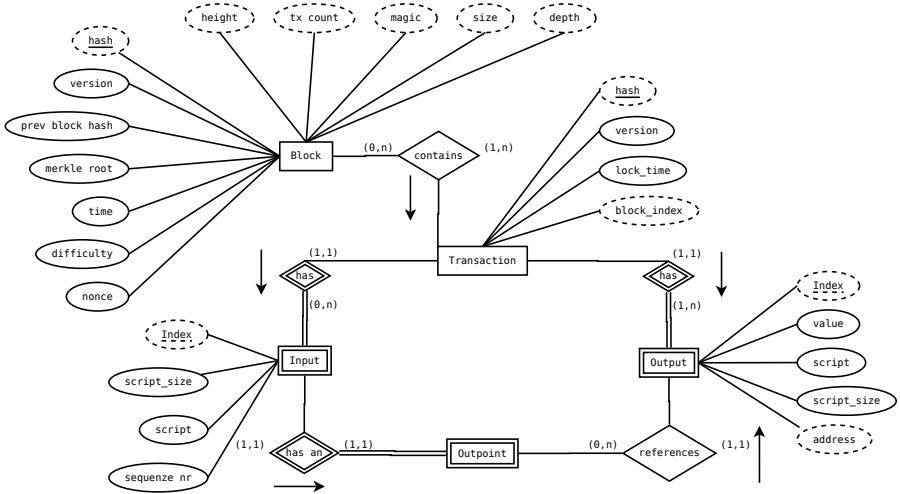
An input contains a number of things:

- Reference to the previous output
  - The output this transaction is spending
  - Called the outpoint
  - Hash of the previous transaction
  - Index of the output within the previous transaction
- Public key for the address the previous output went to
- Signature for the transaction

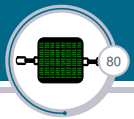


Inputs must be valid. That is to say, they need to correctly reference an utxo (previously unspent transaction output) and they need to contain proof that they have been created by the holder of the keys for that previous output. Outputs on the other hand can contain pretty much just anything. Because of this they get used for other things, such as storing information in the Blockchain.

ER diagram


height
tx count
magic
size
depth
hash
version
prev block hash
merkle root
time
difficulty
nonce
Block
(0,n)
contains
(1,n)
hash
version
lock_time
block_index
Transaction
(1,1)
has
(0,n)
(1,1)
has
(1,n)
Index
value
script
script_size
address
Index
Input
Output
script_size
script
sequenze nr
(1,1)
has an
(1,1)
Outpoint
(0,n)
references
(1,1)

Input and output amounts

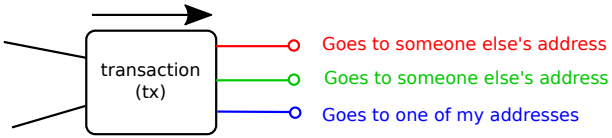


To create a new transaction, you:

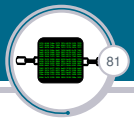
- Figure out how much you want to transfer
- Chose some utxo(s) to use as inputs

However, the value of the utxo(s) will rarely exactly match the sum you want to spend:

- So some of the money needs to return to you
- Easy: Just add another output that sends some of the money to one of your addresses
- Proper Bitcoin clients will create a new address for that
  - This also makes these outputs indistinguishable from other outputs



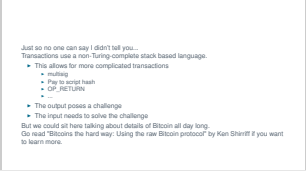
Well actually...



Just so no one can say I didn't tell you...
Transactions use a non-Turing-complete stack based language.

- This allows for more complicated transactions
  - multisig
  - Pay to script hash
  - OP_RETURN
  - ...
- The output poses a challenge
- The input needs to solve the challenge

But we could sit here talking about details of Bitcoin all day long.
Go read "Bitcoins the hard way: Using the raw Bitcoin protocol" by Ken Shirriff if you want to learn more.



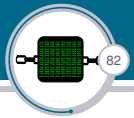
Actually, Bitcoin transactions are a lot more complex than that. Each input and output contain a script which is written in a language developed for this purpose. The scenarios I described above still hold: They are the most used sort of transaction in the Bitcoin network. They can be implemented in very few commands of the scripting language.

However, the scripting language also allows for much more complicated scripts (though not arbitrarily complicated ones since the language is not Turing complete).

This however, is outside the scope of this talk.

"Bitcoins the hard way: Using the raw Bitcoin protocol" by Ken Shirriff - `http://www.righto.com/2014/02/bitcoins-hard-way-using-raw-bitcoin.html`

Coinbase: Creating money



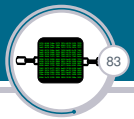
The miner of a new block gets a reward. The reward is the sum of:

- Fixed reward/ newly generated money
  - Started at 50 Btc per Block
  - Halves every 210000 Blocks (ca. 4 years)
- The transaction fees of all transactions in the block

To do this, the miner creates an additional transaction:

- So called coinbase-transaction
- First (left most) transaction in the Block

Coinbase transaction



The coinbase transaction has one input:

- Called the coinbase
- It does NOT refer a previous output
- Instead it contains arbitrary data

A number of things are to be found in most coinbases:

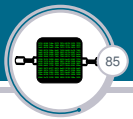
- BIP34 Block Height
- ExtraNonce for mining
- Other data

The output of a coinbase transaction is like all the other outputs in Bitcoin. The input however, is different. A coinbase transaction always has exactly one input. This input, unlike every other input in the Blockchain, does not reference a previous output. Instead, it contains arbitrary data.
The input of a coinbase transaction is called a coinbase.
I have already mentioned that the nonce in the block header is not large enough for mining. So miners change the data in the coinbase. This changes the Merkle root and thus the block header. That way, the 32Bit size limit of the Block header's nonce can be circumvented. Because of this, the random value added to the coinbase for this purpose is often called the ExtraNonce.
Miners also often put their name into the coinbase.

Section 5

Light clients

Bitcoin wallets



A wallet should:

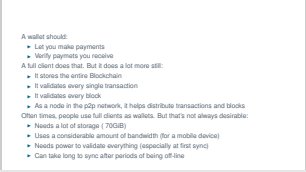
- Let you make payments
- Verify paymets you receive

A full client does that. But it does a lot more still:

- It stores the entire Blockchain
- It validates every single transaction
- It validates every block
- As a node in the p2p network, it helps distribute transactions and blocks

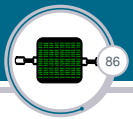
Often times, people use full clients as wallets. But that's not always desirable:

- Needs a lot of storage (70GiB)
- Uses a considerable amount of bandwidth (for a mobile device)
- Needs power to validate everything (especially at first sync)
- Can take long to sync after periods of being off-line



Wallet applications are used to make and receive payments in Bitcoin.
Full nodes are the nodes powering the p2p network. They have a lot of features and most of them can be used as a wallet application. But that's not always desirable. Especially on mobile devices where bandwidth and battery runtime are concerns, the high resource usage of a full node is a no go.
So people need wallet software that's less demanding.

Light clients

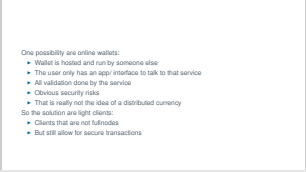


One possibility are online wallets:

- Wallet is hosted and run by someone else
- The user only has an app/ interface to talk to that service
- All validation done by the service
- Obvious security risks
- That is really not the idea of a distributed currency

So the solution are light clients:

- Clients that are not fullnodes
- But still allow for secure transactions

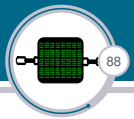


There are cloud/online wallet services that can be used. They come with a web fronted or even dedicated wallet apps. However, the keys used to manage the wallet are stored online on the servers of the service provider. This is an anti-feature of crypto wallets, where the user is supposed to be in control of their money. So this is not the solution we're looking for.
The better solution to the problem are light clients. These are Bitcoin clients that don't need to download, stored and/or validate the entire Blockchain but still provide wallet functionality.

Subsection 1

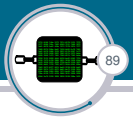
Simplified Payment Verification

Intro to SPV



- SPV stands for Simplified Payment Verification
- These clients do NOT validate the entire Blockchain
- They rely solely on the difficulty of creating blocks
- They are still almost equally secure
  - We have shown in the network attack scenario, that faking blocks is not worth it
  - Even if, the same attack could also work against a full node!

Working of an SPV client



- Connect to the network
- Download all block headers
  - $88\text{Byte} \cdot 410000\text{blocks} \div 2^{20} \approx 32\text{MB}$
- Form the chain
  - This is just as secure as forming the chain from entire blocks

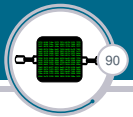
To create new transactions, we have to know our utxos:

- Request utxo for all our keys together with the matching Merkle trees
  - This is just as secure as downloading each entire transaction

When we retrieve money:

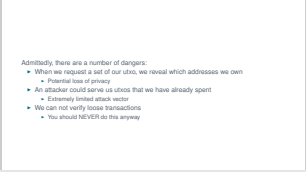
- Wait until the transaction has been put into a block
- Download the block header
- Request a Merkle-Branch proving the transaction is part of that block
- Wait for more blocks to be formed to confirm the security

Dangers of using SPV Clients

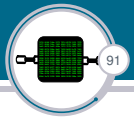


Admittedly, there are a number of dangers:

- When we request a set of our utxo, we reveal which addresses we own
  - Potential loss of privacy
- An attacker could serve us utxos that we have already spent
  - Extremely limited attack vector
- We can not verify loose transactions
  - You should NEVER do this anyway



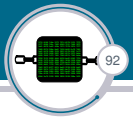
A SPV client can't validate loose transactions. However, loose transactions should never be trusted anyway. In the network attack scenario I showed earlier, trusting loose transactions would be wrong, even if a full client is used.



Subsection 2

Pruning client

On the size of the Blockchain



- The chain, as of now, is approx 70 GiB in size
- It grows at max 1MiB/10 min

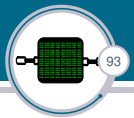
But let's say it grew for 8MiB/ 10 min:

$$365 \cdot 24 \cdot 6 \div 2^{10} = 411\text{GiB/year}$$

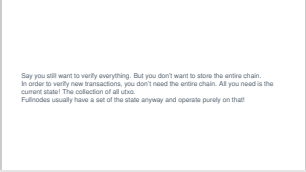
You can get 4TB disks for less than CHF 130.-. It will last you 9 years (even at this insane growth rate). That's less than CHF 15 per year.

You might want a validating light client. SPV clients can't be used for that. Before we look at that, I want to say a word on Blockchain size. The Bitcoin Blockchain is currently 70 GiB in size. This sounds like a lot on first glance, but I think it's not a real concern for anyone willing to run a full node.
At the moment, the Blockchain grows by only 1MiB per 10 minutes. But let's say it were to grow at 8MiB per 10 minutes. Even at that rate, you can get a hard disk that will serve you for 9 years for under CHF 130.-. That's less than CHF 15.- per Year, which should be acceptable for most people.

The idea

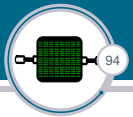


Say you still want to verify everything. But you don’t want to store the entire chain.
In order to verify new transactions, you don’t need the entire chain. All you need is the current state! The collection of all utxo.
Fullnodes usually have a set of the state anyway and operate purely on that!



Even fullnodes that have a copy of the entire Blockchain rarely even access that chain. Instead they have a separate copy of the state, a set of all utxo, they work on. This is sufficient to verify new transactions and blocks.
The full Blockchain is only used on rare occasions, such as when a backtrack is necessary because of orphan blocks.

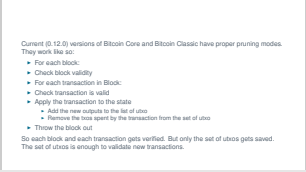
How to get there



Current (0.12.0) versions of Bitcoin Core and Bitcoin Classic have proper pruning modes. They work like so:

- For each block:
- Check block validity
- For each transaction in Block:
- Check transaction is valid
- Apply the transaction to the state
  - Add the new outputs to the list of utxo
  - Remove the txos spent by the transaction from the set of utxo
- Throw the block out

So each block and each transaction gets verified. But only the set of utxos gets saved. The set of utxos is enough to validate new transactions.

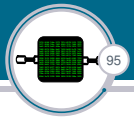

Current (0.12.0) versions of Bitcoin Core and Bitcoin Classic have proper pruning modes. They work like so:

- For each block:
- Check block validity
- For each transaction in Block:
- Check transaction is valid
- Apply the transaction to the state
  - Add the new outputs to the list of utxo
  - Remove the txos spent by the transaction from the set of utxo
- Throw the block out

So each block and each transaction gets verified. But only the set of utxos gets saved. The set of utxos is enough to validate new transactions.

It's advisable to still keep the last few blocks. This is needed when orphan blocks are detected. In that case, one needs to unapply the transactions from the orphan blocks from the state. For that, these transactions need to be known, so the block needs to be available.

Other functions of pruning clients

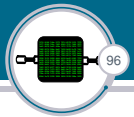


- They can be used for mining
  - Since they can validate every loose transaction, they can create new blocks of valid transactions
- Help distribute transactions and blocks
  - Ability to detect and block invalid ones

But actually, the whole network could be made from pruning nodes:

- No one stores the entire chain
- Every node stores a random set of blocks
- That way, all blocks are still in the network
- No one needs the storage to hold the entire chain

Problems of pruning nodes

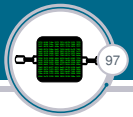


Pruning nodes do not work on low bandwidth devices:

- They still need to download all data
- For initialisation, they need to download the ENTIRE chain
- They need to validate the entire chain

That is just not feasible for mobile devices!

Home made fix



Let's say we have a trusted fullnode (my PC at home).
On that node I do:

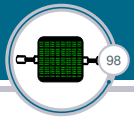
- Take a set of the current state
- Create a hash of that set
- Create a new OP_RETURN transaction with that hash

On the pruning node I do:

- Download all block headers
- Request the transaction I created on the fullnode with a Merkle tree
- Extract the hash
- Request a set of all utxos
- Verify that set against the hash retrieved above

That way, a pruning node can be initialised without downloading the entire chain.

Problems with that solution



This is a rather ugly solution for a number of reasons:

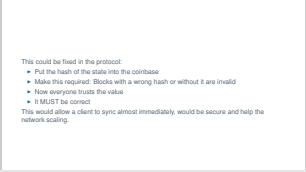
- I need a fullnode
- I need to make a transaction
- I will be the only one trusting that piece of information
- Custom code and custom clients needed

Pruning on steroids

This could be fixed in the protocol:

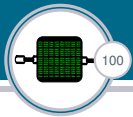
- Put the hash of the state into the coinbase
- Make this required: Blocks with a wrong hash or without it are invalid
- Now everyone trusts the value
- It MUST be correct

This would allow a client to sync almost immediately, would be secure and help the network scaling.

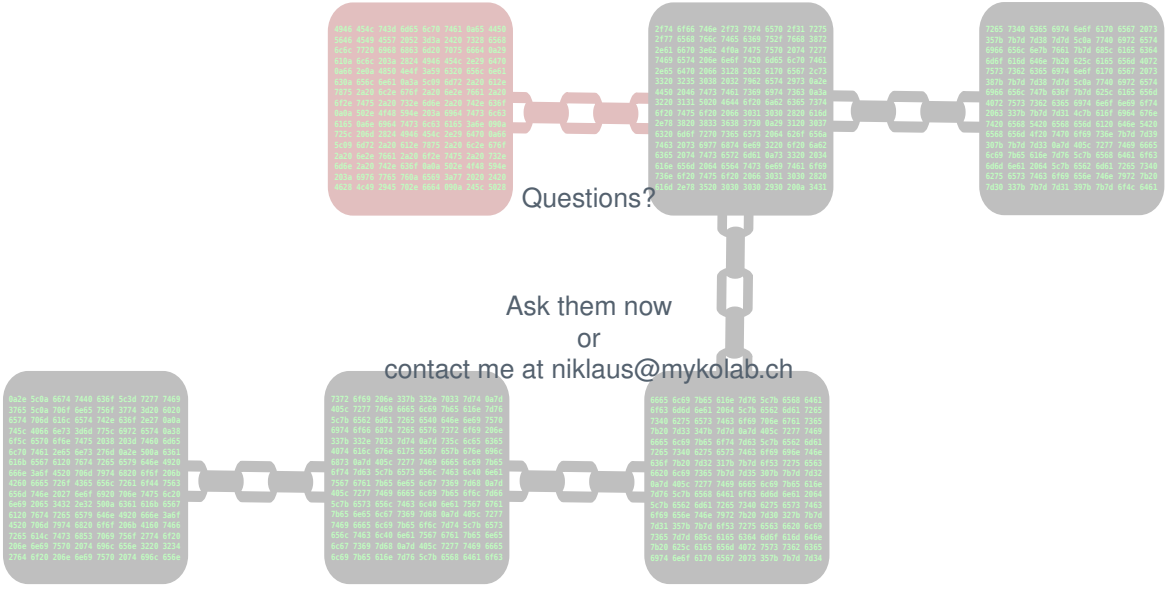


Possibly the best way of building pruning nodes, would be to include the necessary information in the Blockchain. The hash of the state (set of utxos) could be put into the coinbase. If you made this a requirement (every valid block has to contain this hash in the coinbase and the hash needs to be correct, otherwise the block is not valid), then everyone could trust the value. It would be easy, save and quick to initialize a new pruning node.

Copyright



- This presentation is licensed under a Creative Commons Attribution 4.0 International License `http://creativecommons.org/licenses/by/4.0/`
- In the creation of this presentation, I used the Feather Beamer Theme by "Lilyana Vankova" which is released under the GPLv3 license.
  - As required by the GPLv3, I make the exact sources of the theme, as used by me, including all modifications I made, available to you. You can download them from `https://gitlab.honet.ch/vimja/beamer_template` (commit 1776ad90).


Questions?
Ask them now
or
contact me at niklaus@mykolab.ch