DAT650 Blockchain Technology - Course Script

Leander Jehl

October 16, 2019

Chapter 1

Data structures

1.1 Hash Function

Definition 1. A (cryptographic) **hash function** H ($H(x) \mapsto y$) is a *deterministic* function that maps inputs (x) of arbitrary size to "small" outputs (y), e.g. 256 bit numbers.

Note 1. The idea for a hash function is that the result should "look" random. However the function is *deterministic*, i.e. the same input always results in the same output.

Note 2. A mental image for a hash function is the following:

Imagine a gnome that lives in a closed box. When you give a new peace of text to the gnome, he writes down the text, roles some dice and gives you a random number for your text.

When given the next peace of text, the gnome will look up if he has seen this text before. If yes, he will return the number from before. If no, he roles the dice and gives you a new number.

Example 1. *Example for hash functions are*

MD5 *not secure*

RIPEMD-160 *outputs 160 bits - not secure*

SHA-1 *not secure*

SHA-2 *actually SHA-256 and SHA-512 give 256 and 512 bit output.*

SHA-3 *standard from 2015*

On unix terminal (also mac) compute SHA-256 of a file

```
shasum -a 256 file.txt
```

or

```
printf "hello world" | shasum -a 256
```

Hashes here are given in hexadecimal numbers.

Security summary at
`https://en.wikipedia.org/wiki/Hash_function_security_summary`

Definition 2. For a secure hash function the following security properties need to hold:

Pre-image resistance (one way) Given a d-bit y it is infeasible to find any x such that $y = H(x)$

Weak collision resistance Given an input x it is infeasible to find a different x' such that $H(x) = H(x')$.

Collision resistance It is infeasible to find inputs x and x' such that $x \neq x'$ and $H(x) = H(x')$.

Infeasibility means that an exhaustive search is the best strategy to find such values and the probability to find an x as above is smaller than $\frac{1}{2^{112}}$

Example 2. *Examples for the use of cryptographic hash functions are:*

- *Including hashes of third party content in HTML-tags, e.g. scripts.*
- *Not storing user passwords in plain text.*

Note 3. a) Given a secure hash function H that takes strings as input, we can create a hash function H' that takes lists, structs or byte arrays as input.

b) For pre-image resistance to be useful, the domain of structs should be large and structs should be unpredictable.

c) Given a secure hash function H we can create a family of hash functions H_i by concatenating a number or letter i to the input of H $H_i(x) = H(x||i)$. We can choose i at random.

Note 4. Given a hash function $H : \mathcal{X} \to \mathcal{Y}$ that hashes strings (or byte slices) we can create a hash function $H : \mathcal{X} \times \mathcal{X} \to \mathcal{Y}$ by concatenating the inputs.

We write $x_1||x_2$ for the concatenation of inputs x_1 and x_2. Note that the resulting hash function is not commutative.

$$H(x_1||x_2) \neq H(x_2||x_1)$$

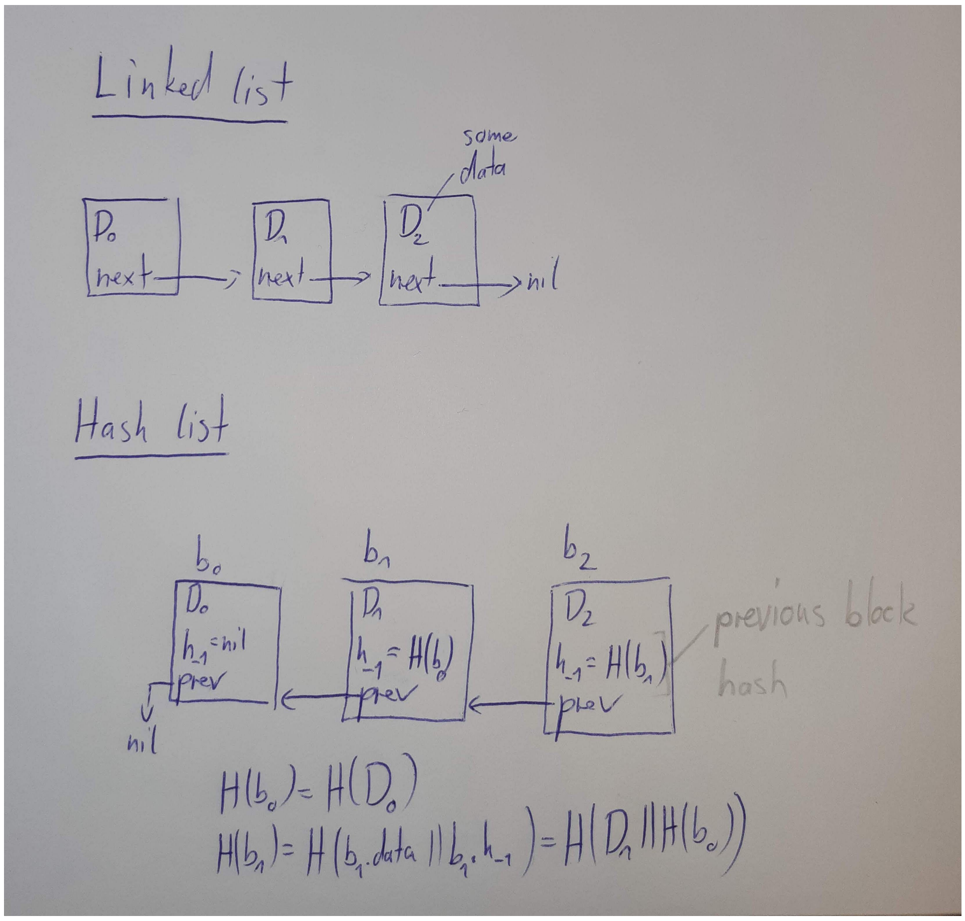
1.2 Hash chain

Assume that H is a secure hash function that maps a struct to a 256-bit.

Definition 3. A **hash-chain** is a data-structure composed of nodes called blocks. Every block b contains a pointer $b.prev$ to a preceding bock and a field $b.h_{-1}$ that contains the hash of the block referenced by $b.prev$. A block also contains a data item $b.data$.

- The hash of a block b is computed as $H(b) = H(b.h_{-1}||b.data)$.
- $b.h_{-1} = H(b.prev)$
- There exist one block b_0, for which $b.prev$ is empty.
- For to block b and b', if $b.prev = b'$ we say that b is a *successor* of b'.

In a **hash-chain** every block has at most one successor.



Lemma 1. *The blocks in a hash-chain can be linearly ordered according to the successor relation with the block b_0 as the first block.*

Note 5. If blocks b_0, b_1 and b_2 contain data D_0, D_1 and D_2 then

$$\begin{aligned}H(b_0) &= H(D_0)\\ H(b_1) &= H(H(b_0)||D_1) = H(H(D_0)||D_1)\\ H(b_2) &= H(H(b_1)||D_2) = H(H(D_0)||D_1)\end{aligned}$$

Example 3. *Assume a notary maintains, a hash-chain of all documents it has signed.*

- *This allows document holders to proof that their document was signed.*
- *This allows to document holders to proof which document was signed first.*
- *Prevent the notary itself to change in which order documents where signed or remove single documents.*

1.3 Merkle trees

By publishing a hash of a document/data we can commit to this document, without revealing it. In the following we look at how to publish many such hashes concurrently.

Example 4. *Given a document describing an invention, I can publish the hash of the document in a newspaper. If someone else then tries to claim my invention, I can proof that I had the document before the date in the newspaper.*

This does not reveal my invention.

Example 5. *In a election you can hash your vote and publish the hash of your vote. After some deadline, you can then reveal your vote to be counted.*

Problem: *An adversary can simply hash all possible values to vote for and thus discover your vote.*

Solution: *Hash vote, concatenated with random value.*

1.3.1 Committing to multiple values

Given data D_1, D_2, D_3 and D_4. We want to commit to all four values. We can either publish individual hashes or a single hash for all:

a) Publish $[H(D_1), H(D_2), H(D_3), H(D_4)]$.

b) Publish $H(D_1||D_2||D_3||D_4)$.

Variant b) requires to publish only a single hash. However to proof that D_1 was committed to, it is necessary to reveal also D_2, D_3, and D_4.

Example 6. *We can build on scheme a) and additionally create* $h_{1,2} = H(H(D_1)||H(D_2))$, $h_{3,4}H(H(D_3)||H(D_4))$ *and* $h_{1,2,3,4} = H(h_{1,2}||h_{3,4})$.

We publish $h_{1,2,3,4}$.

To prove that D_3 *was included we present* D_3, $h_4 = H(D_4)$ *and* $h_{1,2}$. *To check the proof, recompute* $h_{3,4}$ *and* $h_{1,2,3,4}$.

Definition 4. A **Merkle Tree** is a (binary) tree where every note is labeled with a hash h.

- For internal nodes h is the hash of the concatenated labels of its children ($h = H(\mathit{leftchild.h}||\mathit{rightchild.h})$).
- For a leaf node, h is the hash of the data stored in the node ($h = H(D_i)$).

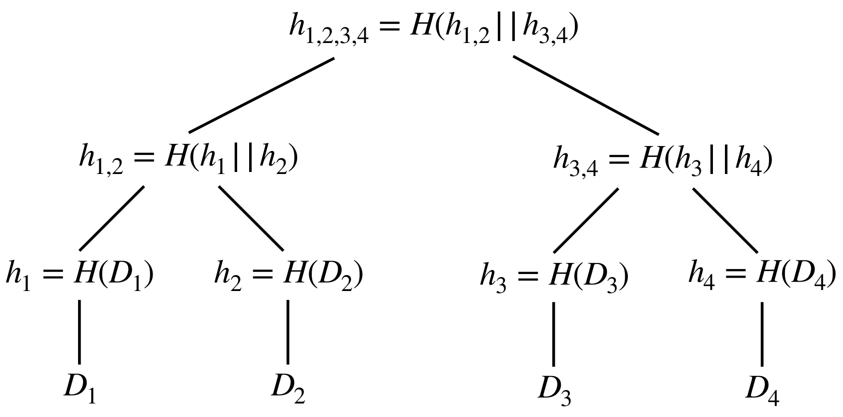


Figure 1.1: Merkle tree with root $h_{1,2,3,4}$

Lemma 2. *If* h_r *is the root of a Merkle Tree with* n *data elements, it is possible to proof that* D_i *is one of the data elements by revealing* D_i *and* $log(n)$ *node labels, i.e., hashes.*

Example 7. *In a Merkle Tree with 16 data elements, an inclusion proof contains 4 hashes.*

In a Merkle Tree with 1000 data elements, an inclusion proof contains 10 hashes.

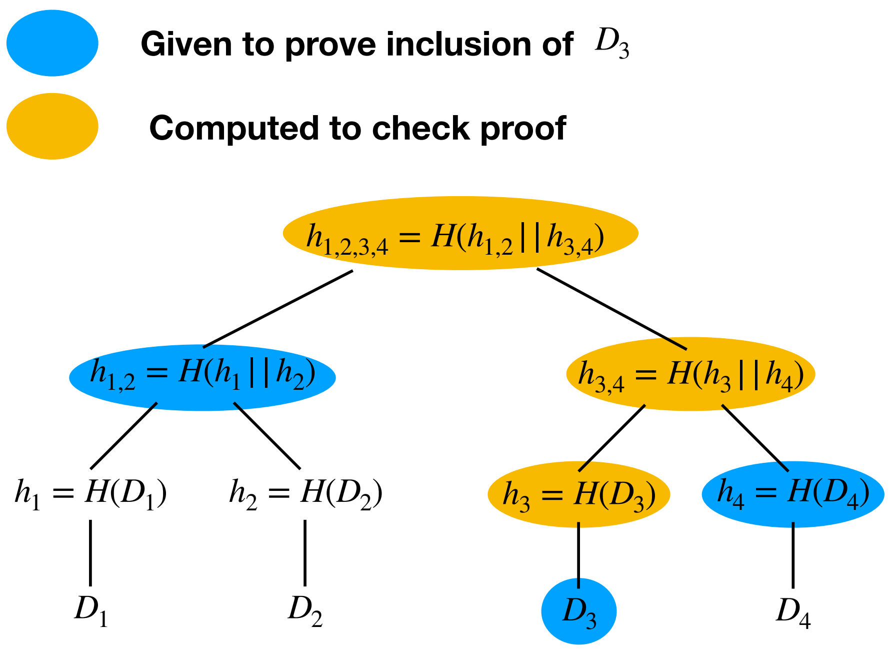


Figure 1.2: Hashes necessary and computed to proof inclusion for D_3.

In a Merkle Tree with 1 million data elements, an inclusion proof contains 20 hashes.

Note 6. It is possible to enhance other tree data-structures, e.g. binary search tree or radix tree with hashes from a Merkle tree.

1.4 Blockchain

The following blockchain definition combines hash-chain with Merkle trees.

Definition 5. A **Blockchain** is a hash-chain where the data entry in every node is the root of a Merkle tree.

Note 7. The blockchain construction was not invented for bitcoin. It is used previously in *linked timestamping*, where block headers are published in newspapers.

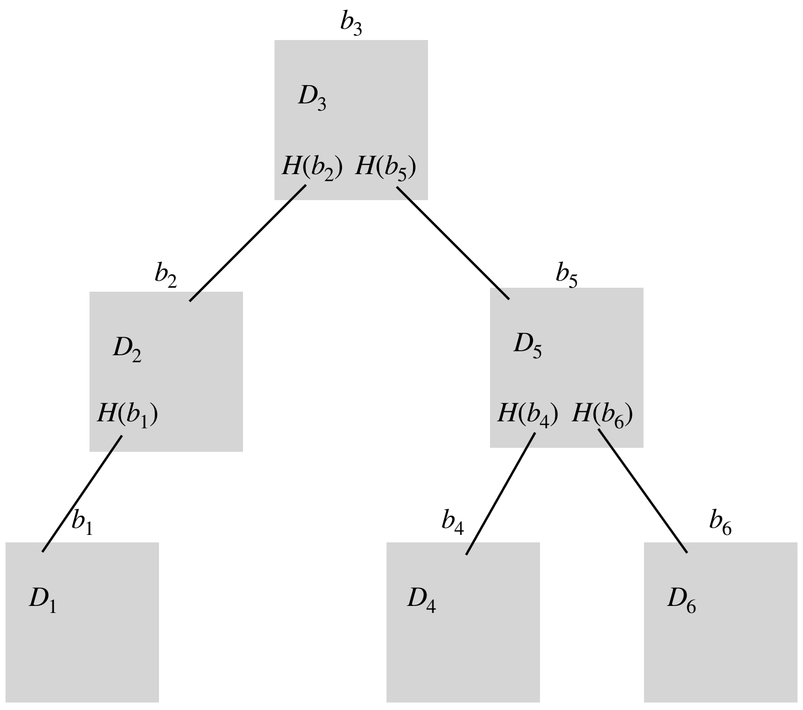


Figure 1.3: A binary search tree enhanced with hash references.

Chapter 2

Transactions

2.1 Digital Signatures

A short recap of digital signatures.

A signature scheme comprises key generation, signing and verification functions. Key generation creates a public and secret (private) key pair (pk, sk).

Signing function takes an arbitrary message m and the secret key sk and outputs a signature σ.

$$\sigma \leftarrow Sign(m, sk)$$

A verify function takes a message m, signature σ and the public key pk. The verification function returns **true**, if σ was produced for m with the secret key sk matching pk. Otherwise verification returns **false**.

$$\{\mathbf{true}, \mathbf{false}\} \leftarrow Verify(m, \sigma, pk)$$

Bitcoin currently uses ECDSA signatures, i.e. signatures where the public key is a point on an elliptic curve.

2.2 Account balances and UTXO

The most prominent use case for blockchains is currently digital currency. We now spend some time to look at how to implement money transfer.

Idea 1. *We can use public keys as identity for users or owners of money.*

Note 8. Idea 1 allows to avoid the use of *public key infrastructure* (PKI), that is otherwise used to bind public keys to identities.

Algorithm 1 show a simple bank that maintains balances. The algorithm has two transactions, Create creates new balances. Transfer allows to transfer money from one account to another. The algorithm assumes that balances that have not been initialized have the value 0.

Authentication of transactions poses a problem:

- For CREATE transactions we assume that they are are only valid during setup of the system.
- For TRANSFER transactions it is desired that all transactions can be submitted by users. E.g. if balances are identified with a public key, we could require transactions to be signed by the owner of *pk-from*.

A bank based on Algorithm 1 may be susceptible to **replay attacks**. A TRANSFER transaction signed by *pk-from* may be submitted multiple times, given that the account of *pk-from* has sufficient fonds. This will result in additional fonds transferred to *pk-to*.

Algorithm 1 Simple Bank using account balances

```
$balances := [pk]\text{uint}$
transaction CREATE($value, pk$)
    $balances[pk] += value$
end transaction
transaction TRANSFER($value, pk\text{-}from, pk\text{-}to$)
    if $balances[pk\text{-}from] > value$ then
        $balances[pk\text{-}from] -= value$
        $balances[pk\text{-}to] += value$
    end if
end transaction
```

Note 9. In a centralized system, a trusted party could process transactions, compute balances and distribute balances to all parties.

Without a trusted party, it is necessary that transactions are distributed to everybody. Every party can then deterministically compute the balances.

One problem in this model is how to role back transactions.

2.2.1 UTXO

We now explain the *unspent transaction output* (UTXO) model.

Idea 2 (UTXO)**.** *The main idea behind the* UTXO *model is that, given a record of transactions, validating that a certain balance is greater than value is the same as verifying that this account has received value funds that have not been spend yet.*

Note that it is usually not necessary to look at all transactions received by the sender, but only a subset. The UTXO *model allows the transaction to specify the subset we should look at.*

Definition 6. A **transaction output** is a tuple (v, pk) that shows, that v funds have been transferred. pk is a *spending condition* that must be met to spend claim v. Typically pk requires a signature with a given public key.

A **transaction input** is a tuple consisting of a reference to a transaction output and an argument that meets the outputs condition. I.e. $(outp_i, \sigma)$ where σ is *redeeming argument* a matching pk, e.g. a signature.

A **transaction** is a tuple containing a list of transaction inputs and a list of new outputs.

Note 10. In bitcoin transactions are implemented in the following way:

- An output from transaction t is identified by a tuple (h_t, i), where h_t is the hash of t and i is the index in the list of outputs in t.
- Algorithm 2 shows how a transaction is validated. For a transaction to be valid, *all inputs must be unspent*, input *signatures must validate* and the *sum of input values must be larger than the output values.*
- Algorithm 2 ensures that a transaction can only be validated once and no two valid transactions can spend the same output.
- The different between transaction inputs and outputs is called transaction *fee.*
- Example 8 gives an example for more complex conditions that may be required to spend an output.
- When the value of inputs is larger than the desired value to be spend, it is common to create an additional output that contains change.

Definition 7. A *double-spend* is a situation where multiple transactions attempt to spend the same output.

Note 11. Note that according to Algorithm 2 only one of double-spend transactions can be validated.

Algorithm 2 Transaction validation and maintenance of UTXO

```
$UTXO := map[(h,i)] \to (value, pk)$
transaction TRANSFER($inputs, outputs$)                ▷ Transaction $t$ with hash $h_t$
    for $((h,i),\sigma) \in inputs$ do
        if $UTXO[(h,i)]$ does not exist then
            abort                                      ▷ invalid transaction
        end if
        if verify($h_t, \sigma, UTXO[(h,i)].pk$) == false then
            abort                                      ▷ invalid transaction
        end if
    end for
    if  sum of values of inputs < sum of values of new outputs then
        abort                                          ▷ invalid transaction
    end if
    for $((h,i),\sigma) \in inputs$ do
        $UTXO[(h,i)] = nil$                            ▷ output spent
    end for
    $h_t := hash(transaction)$
    $UTXO[h_t] = outputs$                              ▷ add new output
end transaction
```

Example 8. *The most common examples for arguments necessary to claim a transaction output are listed here. In Bitcoin they are expressed in a stack based scripting language.*

a) A signature that matches a certain public key.

b) A public key that hashes to a certain value and a signature that matches this key. (Pay to public key hash or P2PKH).

c) Multiple signatures that match a sequence of public keys. (Multisig)

d) Multiple signatures that match m out of n provided public keys.

e) A script that hashes to a certain value and an argument that causes this script to evaluate to true. (pay to script hash)

See here and here for explanation of P2PKH script.

Note 12. To maintain a copy of our transaction based bank, a node has to maintain the set of all unspent transaction outputs $UTXO$.

If variant b) is used instead of variant a) from Example 8 this may significantly reduce the size of the $UTXO$ data structure. The same holds, if d) is used instead of c).

Definition 8. An **address** is either a public key or the hash of a public key. Given the address of a user, it is possible to transfer funds to this user, i.e. create an output that this user can claim by providing a correct signature.

Note 13. Bitcoin and many other cryptocurrencies use Base-58 encoding. This encoding uses small and large letters (a to z) and numbers, omitting 0 (number zero), O (capital o), l (lower L) and I (capital i) because of their ambiguity.

Bitcoin addresses use Base58Check encoding adds a 4 byte checksum before Base-58 encoding, to protect against typos, ...

2.2.2 Privacy in the UTXO model

Different from the account and balance system, the $UTXO$ model encourages the use of different addresses. This makes it harder to identify all transactions done by a single user.

However research has shown, that based on transactions, it is easy to identify different addresses belonging to the same user.

On the other hand, $UTXO$ allows to trace in which transactions a particular value or coin was involved.

Definition 9. A **tainted coin** is a transaction output that is either the result of a transaction considered unethical or illegal or derived from the output of such a transaction by multiple other transactions.

Note 14. Based on the concept of tainted coins it is debatable whether digital cash based on the UTXO model is *fungible.* In economics fungibility is defined as the property that any two units of a good are interchangeable.

The UTXO model allows to create a mixing service:

Definition 10. A *mixing service* can be used to prevent a third party from tracking a specific users transactions. A mixing service would receive payments from many users, and pay them back using new addresses.

Note 15.
- A mixing service makes it hard to see which of the new addresses belongs to which of the users that sent money to the mixing service.
- A mixing service usually requires a high fee.
- A mixing service is usually implemented as a centralized entity.

Chapter 3

Proof of work

In this chapter we discuss how the Transactions, introduced in Chapter **??** are included in a block in the hash-chain, introduced in Chapter 1.

3.1 Centralized straw-man system

We present a straw man solution that relies on a centralized leader to include transactions into a block and issue those blocks. After discussing difficulties with this approach, we present the proof of work based scheme used in bitcoin.

As shown in Figure 3.2, we assume a single leader. Transactions are submitted to the leader and included in a block. The block is then broadcast to all participating nodes, who validate it. *The validation of blocks prevents the leader from including malformed transactions*, however this system still opens several ways for the leader to misbehave:

A) The leader can omit certain transactions on purpose. (Censorship)

B) The leader is a single point of failure.

C) The leader could send different blocks to different processes.

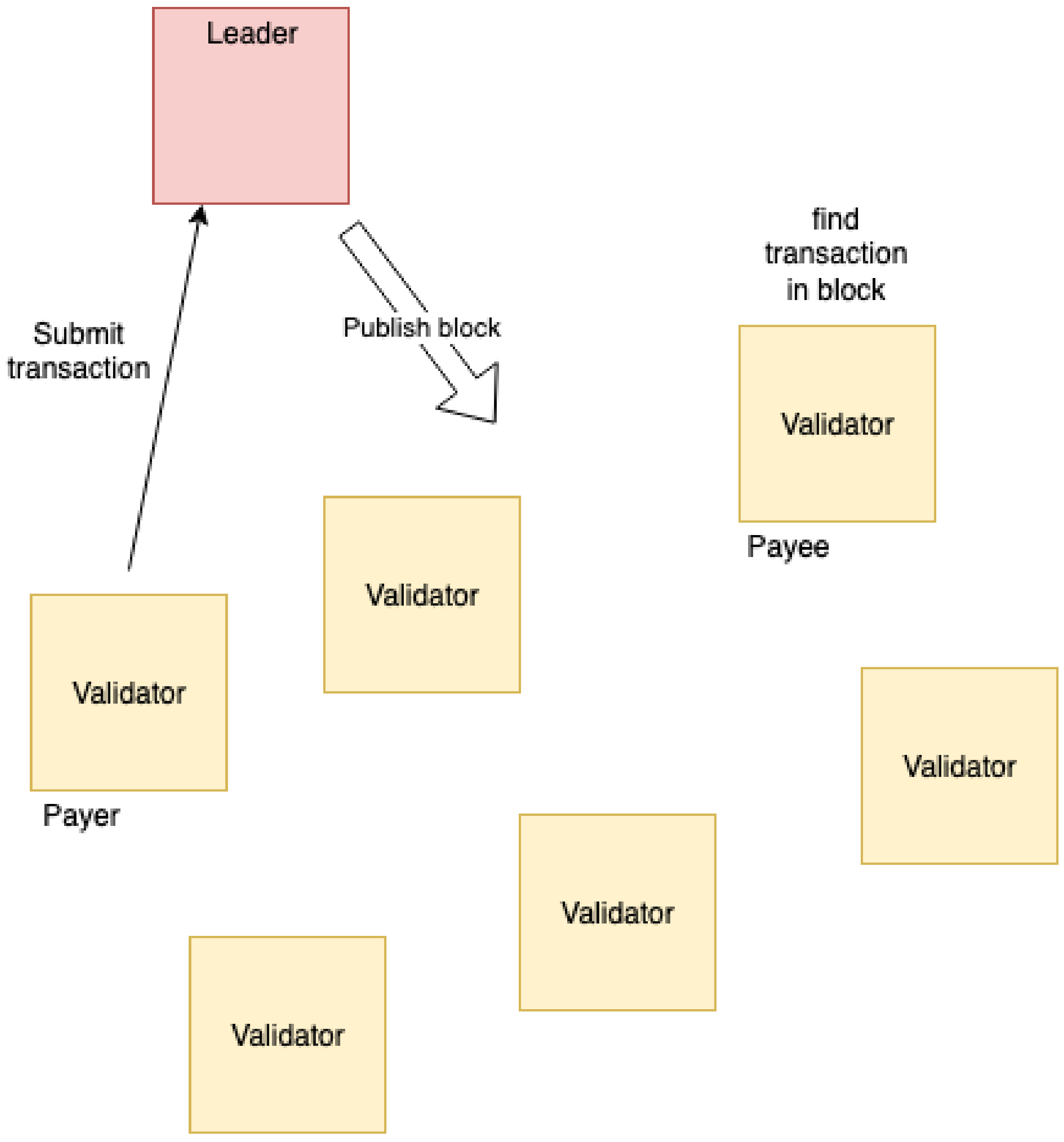


Figure 3.1: Straw-man system with leader

3.2 PoW function

Definition 11. (PoW function version 1) For an integer d, the proof-of-work (PoW) function with difficulty d takes a data item and returns a nonce (random bits) and a hash value:

$$(h_{PoW}, nonce) = f_{PoW}(Data)$$

The proof of work is *valid*, if a) h_{PoW} is the hash of the data, concatenated with the nonce

$$h_{PoW} \stackrel{?}{=} H(Data||nonce))$$

and b) the first d bits of h_{PoW} are 0.

Note 16. The function f_{PoW} is computed by choosing a random *nonce* and checking if the condition b) holds. If it does, we also say that the *nonce solves* the proof of work.

The following is an important conclusion. It follows since the result of hashing one data item is independent from hashing another data item.

Lemma 3. *Given two nonces, chosen at random. The probability that any of them solves the proof of work is independent.*

Theorem 1. *If we conduct multiple, independent Bernoulli trials with success probabilty p, the expected number of trials necessary until success is $\frac{1}{p}$.*

For proof see here.

Note 17. Using version 1 of the proof of work function, adjusting d the difficulty of the proof of work can only be doubled or halved.

Definition 12. (PoW function version 2) For an hexadecimal number d, the proof-of-work function with difficulty d takes a data item and returns a nonce (random bits) and a hash value:

$$(h_{PoW}, nonce) = f_{PoW}(Data)$$

The proof of work is *valid*, if a) h_{PoW} is the hash of the data, concatenated with the nonce

$$h_{PoW} \stackrel{?}{=} H(Data||nonce))$$

and b) h_{PoW} written as hexadecimal number is smaller than d.

$$h_{PoW} < d$$

Note 18. The proof of work function from Definition 12 allows to carefully adjust the difficulty d to achieve a certain expected time.

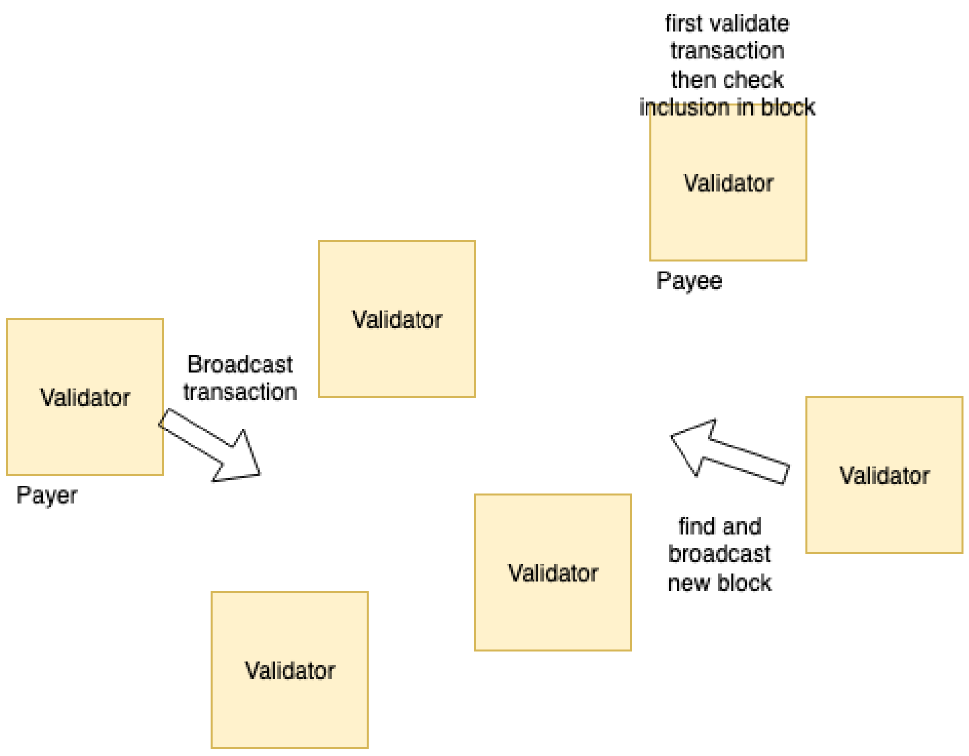


Figure 3.2: Block creation with PoW

3.3 Block creation with PoW

Idea 3. *The idea behind block creation using PoW is to require that any block published includes a nonce, s.t. the block hash and nonce are results of a PoW function.*

We then allow every node to publish a block, if it can find a valid proof of work.

Definition 13. A *proof of work blockchain* is a blockchain as in Definition 5 where additionally, every block contains a *nonce* and the hash of the previous block hash h_{-1}, the root of the merkle tree and the *nonce* solves the difficulty.

Censorship An adversary cannot prevent a node, it does not control from including a certain transaction.

Failures Failure of individual nodes will not prevent the system from running.

Frequency PoW difficulty can be adjusted, to ensure that a block is broadcast to everybody, before the next block arrives.

Conflicting blocks For a single entity to create two different blocks at requires to solve PoW twice. This makes it difficult to publish two conflicting blocks. A high difficulty decreases the probability that two blocks are found concurrently.

3.3.1 Adjustable difficulty

Back in 2010 it was possible to join Bitcoin and create a new block using a simple desktop computer. Today specialized hardware (ASICs) is used to solve the PoW.

Idea 4. *To adjust to nodes joining and leaving the system, and nodes using different infrastructure (i.e. hardware) it is possible to adjust the hardness of the blockchain.*

Definition 14. Additional to the merkle root and nonce Bitcoin includes a timestamp in the block (and in the PoW). This allows to compute the average time between two blocks every 2016 blocks and adjust the difficulty.

Note 19. on timestamps in a PoW blockchain

- The time between blocks can vary significantly but this variance has little effect on the average time, taken over 2016 blocks.
- The timestamp can be set by the nodes when creating the block. When validating the block nodes check that the new block has a timestamp within 2 hours of their local clock.

Lemma 4. *If we assume that the probability that the nodes find a new block within time Δ is constant and independent, and assume that the mean time to block creation is 10 minutes, we can compute the probability, that a block is created within t seconds as*

$$P[\textit{block created within } t \textit{ seconds}] = 1 - \left(\frac{599}{600}\right)^t$$

$$P[\textit{no block created within } t \textit{ seconds}] = \left(\frac{599}{600}\right)^t$$

Proof. Let p be the probability that a block is found within 1 second. Let T be the random variable describing after how many seconds a new block is found. Since finding a block in a specific second is independent we get:

$$P[T = t] = (1 - p)^{t-1}p$$

From Theorem 1 it follows that $E[T] = \frac{1}{p}$. If we assume $E[T] = 10\text{min} = 600$ we get $p = \frac{1}{600}$. $\square$

3.3.2 Fees and mining rewards

Fees and mining rewards give an incentive to solve the PoW function.

Definition 15. When the sum of outputs of a transaction is larger than the sum of inputs, the difference is called *fee*.

Definition 16. Every block in bitcoin contains a *coinbase transaction*. This transaction has no input and a single output. The output value is the sum of a fixed reward (*mining reward*) and the sum of all fees of transactions included in the block.

Note 20. There are different **pro's and con's for fixed rewards and fees:**

- In bitcoin the mining reward is halved every 4 years. Thus, only a finite amount of bitcoin will ever be created.
- The mining reward allows cheap transactions, since fees do not need to cover full mining expenses.
- Some research suggests, that if mining rewards are small compared to fees, mining becomes unstable.[1]
- Mining awards are necessary to get money into circulation.
- Currently in bitcoin mining rewards are 12.5 bitcoin, while fees per block are less than 1 bitcoin.

Note 21. **How big should the fee be?** Fees are determined by market economics, i.e. nodes choose which transactions to include. Usually those with highest fee.

Processing of transactions requires nodes to use network and processing capacities (for relaying and validating transactions). These expenses depend on the size of a transaction, but not on the value transferred. Therefore:

- Fees in bitcoin are usually independent of the value of a transaction. Instead they depend on the size (in bytes) of the transaction.

3.3.3 Forks and longest chain rule

As mentioned above, PoW can reduce the probability for conflicting blocks to be proposed, but cannot prevent this from happening.

Definition 17. A *fork* in a blockchain is when multiple blocks are proposed with the same predecessor. Note that every block in a fork may be extended to a different chain.

[1] `https://freedom-to-tinker.com/2016/10/21/bitcoin-is-unstable-without-the-block-reward/`

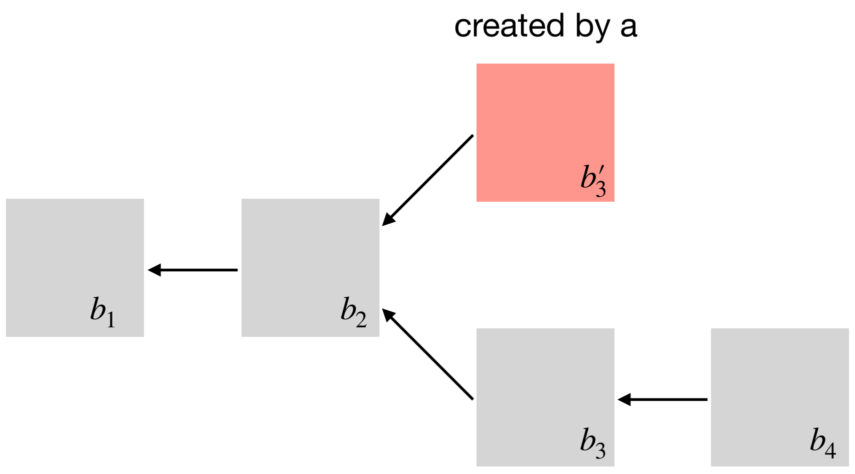


Figure 3.3: A fork in the blockchain. Two blocks extending b_2 where found.

A fork is problematic since the different blocks or chains represent two versions of the world.

Imagine your bank being undecided, wether you did or did not payed your rent.

Definition 18. (Longest chain rule) If a fork exists all nodes should adopt the longest chain.

Note 22. While the Longest chain rule is implemented in the standard bitcoin release, it is not enforced. It is possible to extend a different chain than the given by the longest chain rule. Doing this may change what is the longest chain.

Lemma 5. *Given two chains c_1 and c_2. To find a new block extending c_1 is equally likely as finding a new block extending c_2. This even holds, if nodes already spend significant resources to find a block extending c_1.*

Proof. This follows from Lemma 3. □

Lemma 6. *Assume two chains c_1 and c_2 where c_1 is longer than c_2. Further assume that most nodes follow the longest chain rule, trying to extend the longest chain. A new block is more likely to be eventually part of the longest chain, if it is added to c_1 rather than c_2.*

Proof. Chain c_1 is already longer than c_2. A new block would make c_1 even longer, while it would make c_2 at most equally long to c_1. □

Example 9. *Figure 3.3 shows a fork. Two processes trying to extend block b_2 found a new block, b_3 and b'_3.*

Later another block b_4 was found, that extends b_3. According to the longest chain rule, all nodes will adopt the blocks b_3 and b_4 and discard the block b'_3.

Transactions that where only included in b'_3 but not in b_3 will not be part of the longest chain. They are effectively discarded.

Coinbase transaction *The coinbase transaction in b'_3 is discarded. Thus the node that created b'_3 loses his block reward.*

Double spending *Assume transactions t and t' spend the same output. Assume transaction t' is included in b'_3 and t is included in b_3. t' is discarded together with b'_3. Since b_3 includes t, t' cannot be included in a later block either.*

3.4 PoW and network latency

We now analyze the probability that a fork occurs based on the network delay, i.e. the time it takes to broadcast a block to the different nodes. Results in this section are taken from Decker and Wattenhofer, 2013.

Definition 19. We write δ for the average time it takes for a block until it is validated by a specific node in the network.

Note 23. Based on empirical evaluations, in the current bitcoin network, $\delta = 12.6$ seconds.

In the following theorem we assume that nodes have the same mining power and are following the longest chain rule.

Theorem 2. *Let p be the probability that a block is found within one second. Then the probability for a fork is*

$$1 - (1-p)^{\delta}$$

Proof. The probability for a fork is the probability that the while the block is propagated, another block is found. Thus

$$P[\text{fork}] = 1 - P[\text{no block found during dissemination}]$$

$$P[\text{no block found during dissemination}] = (1-p)^{\delta}$$

□

Corrolary 1. *Let $P[\text{fork}^l]$ be the probability that after a fork, both chains are extended by l blocks. It holds*

$$P[\text{fork}^l] \leq \left(1 - (1-p)^{\delta}\right)^l$$

Proof. Assume one chain c_1 is extended by one block. If a fork of this chain, c_2 is also extended, this has to happen before the new block on c_1 is propagated. Thus the probability that two chains in a fork are extended, is equal to the probability that a block on the second fork is found, while the block on the first fork is disseminated. This is less than the probability of a fork. □

Corollary 1 says that the probability that a block is discarded because of a fork drops exponentially, with the numbers of blocks that are added after this block.

Definition 20. We say that a transaction is *confirmed*, if it is included in a block, and several blocks are added after this block.

In bitcoin, it is recommended to wait for additional 5 blocks, after the block including a transaction.

3.5 Attacks on bitcoin mining

In Section 3.4 we have shown that continued forks are extremely unlikely, if nodes stick to the longest chain rule and do not disturb the network latency.

In the following we first look at two ways to deviate from the longest chain rule, stubborn mining and selfish (hidden) mining. And we look at network attacks that might be performed.

We first look at those attacks from the perspective of fair mining. Then we look at them from the perspective of a double spend.

Definition 21. Mining, or block creation is *fair* if a node that possesses α percent of the hashing power in the network ends up publishing α blocks in the longest chain.

3.5.1 Stubborn mining

A node does perform stubborn mining, if it does not abandon the current chain for the longest chain. Thus it does not follow the longest chain rule.

More precisely, if there exist two blockchains c_1 and c_2 and c_2 contains one more block than c_1. Thus the initial state is as shown in Figure 3.3. A stubborn node that has published a block in c_1 that is not part of c_2 will continue to try and extend c_1 until either c_1 becomes the longest. We assume that if the difference between c_1 and c_2 increases, the stubborn node will abandon c_1.

Theorem 3. *Stubborn mining does not increases the expected outcome of a node, if the node controls less than $\alpha = 0.42$ of the hashing power in the network.*

Proof. We model this system as a markov chain with 4 states.

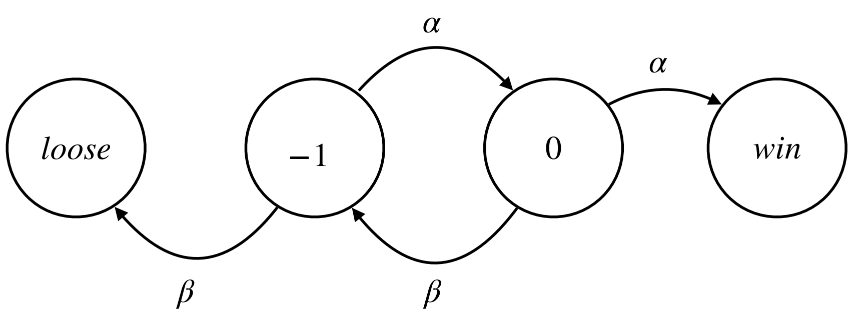


Figure 3.4: Stubborn mining states and transitions.

Loose	where chain c_1 was extended faster than c_2.
-1	where chain c_1 is one block longer than chain c_2.
0	where c_1 and c_2 are equally long.
Win	where chain c_2 became the longest chain.

We ignore the probability that additional forks occur on c_2 but note that they would increase the profitability of stubborn mining. Assume the probability that the attacker finds a block is α, while $\beta = 1 - \alpha$ is the probability that the remaining miners find a block. The states and the transition probabilities are shown in Figure 3.4.

We now calculate the expected number of blocks the Attacker receives with and without doing the attack. For the attack we consider the following cases:

- With probability β the attacker loses in the first step and receives no blocks.
- With probability $\alpha \cdot \alpha$ the attacker mines two blocks. He receive 3 blocks in total.
- With probability $\alpha \cdot \beta \cdot \alpha \cdot \alpha$ the process goes through states $-1 \mapsto 0 \mapsto -1 \mapsto 0 \mapsto win$. In this case the attacker gets 4 blocks.

Extending the above cases, and omitting those that give 0 blocks, E_{Attack}

the expected number of blocks is:

$$\begin{aligned}
E_{Attack} &= \sum_{i=0} (i+3)\alpha^{i+2}\beta^i && (3.1)\\
&= 3\alpha^2 + \alpha\beta \left(\sum_{i} = 0(i+3)\alpha^{i+2}\beta^i + \sum_{i=0} \alpha^{i+2}\beta^i \right) && (3.2)\\
&= 3\alpha^2 + \alpha\beta \left(E_{Attack} + \frac{\alpha^2}{1-\alpha\beta} \right) && (3.3)
\end{aligned}$$

In step (3.2) we used the formula for a geometric sum. Solving the above for E_{Attack} gives

$$E_{Attack} = (3 + \alpha\beta)\frac{\alpha^2}{1-\alpha\beta}$$

To compute the average number of blocks, the attacker would receive if it did not follow the attack, we note that the he gets one block every time an edge with probability α is traversed. We get the following cases.

- With probability $\alpha \cdot \alpha$ the node mines two blocks.
- With probability $\alpha \cdot \beta \cdot \beta$ the process goes through states $-1 \mapsto 0 \mapsto -1 \mapsto$ *loose*. The node mines 1 block.
- With probability $\alpha \cdot \beta \cdot \alpha \cdot \alpha$ the process goes through states $-1 \mapsto 0 \mapsto -1 \mapsto 0 \mapsto$ *win*. In this case the node gets 3 blocks.

Continuing the above analysis, we see that the average number of blocks received when not following the above state machine, but not following the attack, is:

$$E_{NoAttack} = \sum_{i=0} (i+2)\alpha^{i+2}\beta^i + \sum_{i+0} i\beta^{i+1}\alpha^i$$

Using the same techniques as for E_{Attack}, we get:

$$E_{NoAttack} = (2 + \alpha\beta)\frac{\alpha^2}{1-\alpha\beta} + (1 + \alpha\beta)$$

Plotting both graphs we see that $E_{Attack} < E_{NoAttack}$ holds approximately for $\alpha < 0.42$. □

3.5.2 51% attack

If a miner owns $\alpha = 51\%$ of the hashing power in the network, he can attack the network by creating and growing his private chain. Key to this attack is that the attacker will be able to grow his private chain faster than the remaining network can grow the public chain.

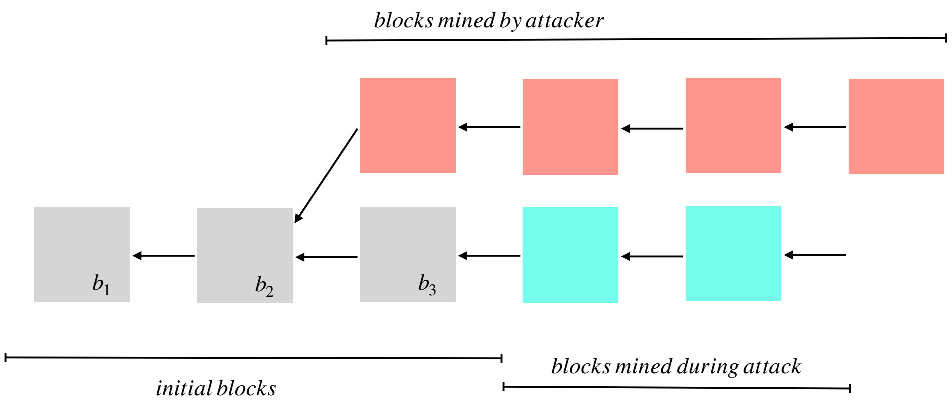


Figure 3.5: A 51% attack.

Example 10. *This example is shown in Figure 3.5. Assume at the begin of the attack the longest chain contains blocks b_1, b_2, and b_3. The attacker picks a recent block, e.g. b_2 and starts secretly extending b_2. Once the attacker has extended his chain longer than the public chain he can publish it and all blocks in the public chain will be discarded.*

3.5.3 Selfish mining

In a selfish mining attack, the attacker does not violate the longest chain rule. Instead he violates the following mandate:

Publication mandate When a node finds a block, it should immediately announce this to the other processes.

The idea behind not publishing the newest block is that it denies the other nodes to try and extend the longest chain. Thus, other nodes waist resources, trying to extend a chain, that is not the longest chain.

For a detailed description and analysis of selfish mining see Chapter 26.1 of these Lecture notes form ETH Zurich.

Note 24. Results above show, that, if the attacker has more than 1/3 of the hashing power, he can increase the ratio of blocks he creates in the blockchain by selfish mining.

- If the attacker has more than $\alpha = 1/3$ of the hashing power, he can increase the ratio of blocks he creates in the blockchain by selfish mining.
- If the attacker has more than $\alpha = 1/4$ of the hashing power, and can reach $\gamma = 0.5$ half of the nodes before another miner can reach them, he can benefit from selfish mining.

Algorithm 3 Selfish mining

```
Idea: Mine secretly, without immediately publishing newly found blocks
Let $l_p$ be length of the public chain
Let $l_s$ be length of the secret chain
if a new block $b_p$ is published, i.e. $l_p$ has increased by 1 then
    if $l_p > l_s$ then
        Start mining on $b_p$
    else if $l_p = l_s$ then
        Publish secretly mined block $b_s$
        Mine on $b_s$ and immediately publish new block
    else if $l_p = l_s - 1$ then
        Push all secretly mined blocks
    end if
end if
```

3.6 P2P networking and network layer attacks

A node in bitcoin does not maintain connections to all 10.000 bitcoin nodes. Instead every node maintains a membership list, with addresses of other nodes. He maintains connections only to a few nodes selected at random from the list. We say that these connections form an overlay network.

In Bitcoin, nodes per default start by establishing 8 connections and extend this to up to 125 connections.

When a block is broadcast, every peer receiving the block first validates it and then forwards it to its neighbors.

3.6.1 Inventory messages and delivery denial attack

The forwarding of a block consumes significant bandwidth. Bitcoin therefore uses INVENTORY messages to announce the a block to neighbors. Receiving an INVENTORY message a node would request to receive the actual message from only one of its neighbors.

Nodes set a timeout when requesting a block. If they do not receive the block within the timeout, it is requested from a different source.

- In bitcoin version 0.10, the timeout for receiving a block is set to 20 minutes.

Definition 22. In a *delivery denial attack* a node does send INVENTORY messages, but when a block is requested, does not forward the block.

For extended details on this attack, see [Gervais et. al. in CSS'15].

Note 25. Due to the static timeout, a delivery denial attack can significantly slow down block propagation.

- Slowing delivery of blocks increases the probability of a fork, as can be seen from Theorem 2. This also increases the probability that a fork extends for several blocks.
- Slowing delivery of competing blocks may increase parameter γ in the selfish mining scheme and thus make selfish mining more appealing and profitable even when $\alpha < 1/4$.

3.6.2 Sybil attack

In the eclipse attack, we assume that an attacker controls many IP addresses and machines, e.g. a botnet.

Definition 23. An attacker performing a *sybil attack* registers multiple peers to receive as many connections as possible. Additionally, he may spread false network addresses.

Note 26. A successful attack has the following **effect:**

- A successful attack allows the attacker to selectively reduce connectivity in the network.
- If the attacker chooses to cooperate in the forwarding of a specific message, block or transaction, this may spread significantly faster.
- Blocks or messages which the attacker chooses not to forward, may spread far slower.
- The increased network delay results in an increased fork probability.
- The selective connectivity gives the attacker an advantage, when performing selfish mining, i.e. increasing the γ parameter.

Idea 5. *An interesting idea would be to select peers among the nodes that have previously published a block.*

Note 27. Using the idea above it is more difficult to introduce sybils. However, this would undo the anonymity of minors. Further, there is a bootstrapping problem.

3.7 Attacks on transactions

It is possible to issue two transactions that both spend the same outputs. Only one of these transactions can be included in a chain, however, in case of a fork, each transaction may be included in a different branch of the fork.

Remember that a transaction counts as confirmed, if it is included in a block and a certain number of blocks is added on top of this block.

Definition 24. In a *double spend attack* on an *unconfirmed transaction* a payee accepts a transaction that is not confirmed, e.g. because he cannot wait for 20 minutes to sell a coffee.

The payer then issues a different transaction an tries to get this different transaction included in the chain.

Definition 25. In a *double spend attack* on a *confirmed transaction* a payee accepts a confirmed transaction.

The payer issues a different transaction an tries to get a fork including this transaction to become the longest chain.

Note 28. We now look at the different attacks from Section 3.5 and if they allow a double spend attack on confirmed transactions.

- A 51% attack allows to perform a successful double spend on a confirmed transaction. The attacker can simply create a secret fork including the second transaction, grow it to be longer than the public chain and publish once the original transaction is confirmed.
- During a selfish-mining attack it may happen that an attacker published a secret chain that is l blocks or longer. Especially, if the attacker already has created a secret chain that is 6 blocks longer than the public chain, he is sure to succeed in a double spend attack.
- A sybil attack may increase the network latency and thus cause forks. However, the probability for a long fork is still quite small.

Note 29. If attacker is doing selfish-mining, what is the probability that he currently is 6 blocks ahead (or more):

- If $\alpha = 25\%$ then about 0.001.
- If $\alpha = 33\%$ then about 0.01

These can be computed using the marginal probabilities from Chapter 26.1

3.7.1 Eclipse attack

The eclipse attack is a special case of the sybil attack. It is performed similarly, but rather than targeting the complete network, the target is a individual node or small group of nodes.

Definition 26. An attacker performing an *eclipse attack* registers multiple peers to receive as many connections as possible. Additionally, he may spread false network addresses to the victims.

The attacker aims to:

I Monopolize all the connections of a single node or small part of the network.

Note 30. A successful attack has the following **effect:**

- A successful attack may allow the attacker to exclude individual nodes from the network. The excluded nodes may no longer receive new blocks and any blocks created by the excluded nodes will probably be discarded when the attack stops.

  This allows the attacker to double spend a confirmed transaction. By creating a chain especially for the attacked node, which confirms the transaction. The main chain will be longer and contain the other double spend transaction.

3.8 Updating a blockchain

Most blockchain protocols are under constant change. There are bugs and security vulnerabilities that are fixed and new features or improvement proposals added.

As with other software, blockchain nodes do not accept and install updates instantly. For some controversial updates, notes might even choose to not update.

Definition 27. An update that changes the rules, which blocks and transactions are valid, is called a *fork*.

Note 31. A software update that changes the code run by a node, but not its output is a *non-fork*. Here nodes implementing new and old versions seamlessly work together.

3.8.1 Soft fork

Definition 28. A *soft fork* is an update that makes some of the initial transactions or blocks, valid under a previous version invalid. However all blocks and transactions under the new version are valid according to the old protocol.

Example 11. *A typical example of a soft fork is a security update, that rules out certain behaviors that were allows on the previous version.*

Note 32. The result of a soft fork depends on whether a majority of the nodes switches to the new fork.

- If the new version in a soft fork is accepted by less than 50% of the nodes, these will create there own chain.
- If the new version in a soft fork is accepted by more than 50% of the nodes, any block published on the old chain will eventually end in a short fork and be discarded.

3.8.2 Hard fork

Definition 29. A *hard fork* is an update that creates blocks that are not valid under the previous protocol. However blocks that are valid under the previous protocol are still valid under the new protocol.

This version of a hard fork is also sometimes called a *strictly extending hard fork.* It is generally considered easier to implement changes as a hard fork, than as a soft fork.

Example 12. *An example for a hard fork is a new feature introduced, e.g. a new Op code for scripts. Blocks and transactions that do not utilize this new feature are still valid under the new protocol.*

Note 33. The result of a hard fork depends on whether a majority of the nodes switches to the new fork.

- If the new version in a hard fork is accepted by less than 50% of the nodes, any block published on the new chain will eventually end in a short fork and be discarded.
- If the new version in a hard fork is accepted by more than 50% of the nodes, two chains will be created.

3.8.3 Hard and soft forks

Definition 30. A *hard and soft fork* is an update that creates blocks that are not valid under the previous protocol and also invalidates blocks from the previous protocol.

This version of a hard fork is also sometimes called a *bilateral hard fork.*

Note 34. In a hard and soft fork, there will always be two chains created.

3.8.4 Analysis

Forks that are caused by software or protocol updates have the potential to create forks of significant length. During such updates it may be easy to perform a double spending attack.

A safe variant is to require transactions to be included in both forks. However this is not possible for all variants, especially when spending outputs that where created in one of the forks.

There exist examples where a fork has resulted in two chains that where maintained separately.

Idea 6. *A common idea is for nodes to first signal their readyness to switch to a new version, e.g. in the blocks they publish. This allows rules like:*

- *Only move to the new version if 95% of the last 2000 blocks have signaled to want to move to this version.*

3.8.5 Pow as voting

The discussion in Section 3.8 shows that solving proof of work can be seen as majority voting, where every node has a voting share equivalent to the hashing power it is using to solve PoW.

It is possible to take this view also for forks that occur during normal operation. A node votes by trying to find a block extending a certain chain.

Note especially that this voting mechanism has a build in sybil resistance, since voting requires limited resources, e.g. CPU or AISCS cycles and electricity. It is not possible to create a higher voting share, by simply creating new identities on the system.

Chapter 4

Alternative PoW

4.1 Improving PoW

4.1.1 Alternative mining puzzles

See Chapter 8 of this book.

Definition 31. A suitable mining function has the following properties:

Adjustable difficulty It must be possible to adjust the difficulty to adapt to a growing network.

Fast verification Every node in the network needs to verify a solution. Thus verification should be easy, compared with computation.

Progress free The probability to solve the PoW function in the next second, should be independent of how long a process has been trying to solve it.

4.1.2 ASIC resistance

Asics resistance is motivated by the fact that bitcoin mining is currently done almost exclusively on specialized hardware, i.e. application specific integrated circuits (ASICs). There are currently produced by a single manufacturer, yielding a single point of failure and trust.

Further, these ASICs are difficult to use for other purposes than mining cryptocurrencies.

Definition 32. A PoW puzzle is *ASICs resistant* if specialized hardware can only provide a small benefit in computing this puzzle, compared to a general purpose CPU.

Note 35. Proposals for ASICs resistant puzzles include

Memory hard functions The idea for this is to design puzzles that require a lot of memory access. The idea is that memory access is harder to optimize than cpu computation, as done for hashing.

CPU benchmarks A recent proposal HashCore ICDCS'19 proposes to use CPU benchmarks, used normally during hardware development, to create PoW puzzles.

Both proposals result in functions that are harder to verify.

4.1.3 Proof of useful work

The idea behind proof of useful work, is that energy and hardware used to compute PoW solutions seems "wasted".

Definition 33. A PoW puzzle is *useful*, if its solution or computation has an application or utility outside of blockchain domain.

Note 36. Proposals for useful PoW are still subject to research. Proposals include:

Finding prime numbers

Evaluate small degree polynomials This can help to determine mathematical problems like vector orthogonality, shortest path problems, ...

Note that while finding problems, e.g. vector orthogonality have applications, it is usually not useful to simply find two random orthogonal vectors. Rather two orthogonal vectors should be found from a given set.

This poses the question, how the set, i.e. the problem should be submitted.

4.1.4 Proof of Storage

Definition 34. A *proof of storage* mining puzzle requires a node to store a certain file to be able to create a block.

Note 37. In a simple variant a proof of storage is a merkle inclusion proof. This can be combined with a classical PoW to adjust hardness, ...

Several problems arise:

- What file to store?
- How to distinguish a stored file from a file, retrieved if necessary.
- How to deal with the increased payload due to merkle proofs.

A probably better alternative is to use the amount of storage a node provides as stake, in a Proof of stake scheme.

4.2 Proof of Stake

Bitcoin uses PoW to avoid centralization. To perform a 51% attack, the attacker has to invest huge amounts of energy and hardware. The idea behind proof of stake is to use the cryptocurrency for this purpose. I.e. to do a 51% attack in PoS, the attacker needs to own 51% of the currency.

Definition 35. Proof of work distributes block rewards to miners, equivalent on the energy and hardware cost they have invested. Proof of stake aims to distributed mining rewards, depending on the amount of money (i.e. cryptocurrency) the miners have frozen for this purpose.

Example 13. *In PPCoin (Peercoin) a miner identified by addr, that has deposited* `coin`$(addr)$ *can supply the current block, if*

$$H(\texttt{prevBlockHash}||addr||\texttt{timeinseconds}) < d_0 \cdot \texttt{coin}(addr)$$

- *Here d_0 is a base difficulty. The probability that a miner with a specific address addr can mine the next block is proportional to* `coin`$(addr)$.
- `timeinseconds` *shows time in seconds. Thus a miner gets a change to submit a solution every second.*

This constructions gives several ***limitations and problems****. Some of these are common for many PoS schemes.*

Predictability *A miner can predict whether he will be able to mine the next block.*

PoW next block *A miner with sufficient resources can try to tweak the current block, s.t. he will be able to also mine the next block.*

Non deciding *In case of a fork, a miner does not have to decide on which block he wants to mine. It is feasible to mine of both chains. Thus forks may prevail for long.*

History rewrite *Theoretically it is feasible to rewrite the complete, or a large part of the history of this chain.*

The above problems are not specific to Peercoin, but are also present in other proof of stake solutions.

Chapter 5

Scaling the Blockchain

5.1 Blocksize and block interval

Bitcoin currently achieves about 7 transactions per second. To increase this, it was heavily discussed, wether the blocksize should be increased, or block frequency can be reduced.

Definition 36. A change in maximum blocksize or target block interval is called a *reparametrization.*

Research, (see resources.md) has suggested that network latency grows linearly with blocksize.

Lemma 7. *A reparametrization that increases blocksize will result in higher network latency and thus in an increased fork probability.*

Proof. Theorem 2 says that increased network latency results in higher fork probability. □

Lemma 8. *A reparametrization that decreases target block interval causes an increased fork probability.*

Proof. A reduced target block interval, i.e., less than the current 10 min value, will result in an increased probability that a block is found within a second (p in Theorem 2). □

5.1.1 Ghost

GHOST is a proposal to avoid the increased vulnerability to selfish mining and double spend attacks. This is especially relevant to maintain security after reparametrization for increased throughput.

Definition 37. The *Greedy heaviest-observed subtree (GHOST)* rule says, instead of selecting the longest chain the root of the subtree with the most leafs should be selected.

Note 38.

- As long as only a single fork exists, the GHOST rule is identical with the longest chain rule.
- In a selfish-mining attack the attackers chain will not contain any forks, since only a single node/agent is trying to extend it. The chain build by honest nodes may show additional forks. Using the GHOST rule, these forks do not give an advantage to the attacker.
- Using network attacks the attacker may cause forks on the honest chain. Using the GHOST rule, these attacks have no impact on the probability of a successful attack.

5.1.2 Inclusive chains and DAGs

An increased fork probability may result in many blocks being discarded. This results in minors not getting their block rewards. It provides an incentive for miners/nodes to form larger groups, rather than mine individually. This is true also when using the GHOST rule.

Definition 38. The published blocks, both on the longest chain and in other forks, build a *tree*, routed at the genesis block. The *previous block hash* h_{-1} pointers point to the parent of a block.

Definition 39. In an *inclusive* blockchain, instead of just including the hash of the parent, a block b_{new} can include hashes of another block b, if:

- b is not an ancestor of b_new
- the depth of b (d_b) (i.e. distance from the genesis block) is less than the depth of b_{new} (d_{new})
- b is a child to one of the ancestors of b_{new}.

Blocks included as additional ancestors are called *uncles.*
Fees and rewards in an inclusive blockchain

- A block b included as uncle in b_{new} receives a fraction of the block-reward. The fraction reduces with the distance between b and b_{new}, i.e. $d_{new} - d_b$.
- A block b_{new} receives a small reward for every uncle it includes.

Theorem 4. *It is possible to define a deterministic order on all blocks in a chain and all uncles in a chain, such that the order extends when the chain is extended. This allows to execute transactions that are included in the uncle blocks, but not in the main chain.*

Proof. The order is done based on the following principles, from highest to lowest priority:

- Ancestors of b_{new} are ordered before uncles of b_{new}.
- Uncles of b_{new} are ordered before b_{new}.
- Uncles are ordered according to their depth.
- At the same depth, uncles are ordered according to their hash.

□

Example 14. *Ethereum uses uncle blocks. An uncle b is rewarded* $1 - \frac{(d_{new} - d_b)}{7}$ *of the block reward. Thus uncles that are more than 6 blocks behind are not rewarded.*

For including an uncle a block creator is rewarded $\frac{1}{32}$ *of a block reward.*

Note 39. Inclusive blockchains have the following effect on scalability:

- If uncle blocks include transactions that are not included or conflicting with the main chain this can further increase scalability. Execution of transaction in uncles is not implemented in Ethereum.
- Given execution of transactions in uncles, it is difficult how to incentivize different blocks in a fork, to include different transactions.
- It is possible to extend inclusive blockchains to allow uncles that are not children of ancestors in the current main chain. Theorem 4 still holds, but becomes more complex.

Security and incentives:

- The possibility to receive a block reward, being not included in the main chain, reduces the urge to form larger mining groups (pools).
- The reward an uncle receives plus the reward for including an uncle is less then the block reward. This should incentivize nodes to try and extend the main chain.
- The reward for uncles is reduces over distance. This disincentivises keeping blocks secret.

Theorem 5. *In an inclusive blockchain, the selfish mining attack becomes profitable at a lower* α *threshold than without inclusive mining.*

This can be leveraged to some extend, if the difficulty is adjusted not based on the frequency of blocks on the main chain, but based on the frequency of blocks on the main chain, and uncles.

Proof. See Selfish Mining in Ethereum

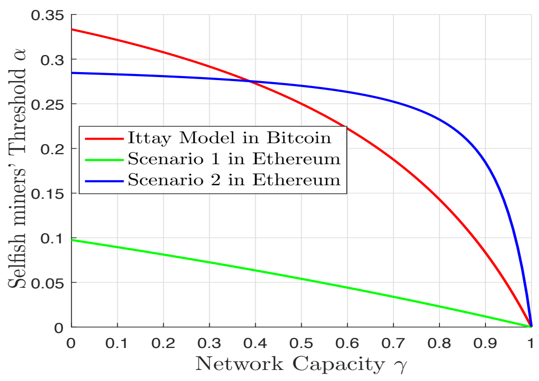


Figure 10. The profitable threshold of hash power in Bitcoin and Ethereum.

In the above figure from [Selfish Mining in Ethereum, ICDCS'19] Scenario 1 is where block difficulty is only adjusted based on the new blocks on the main chain. Scenario 2 adjusts difficulty based on creation of blocks and uncle blocks.

□

5.2 Bitcoin NG

See resources.md on course info for Video, slides, paper, ...

Definition 40. Bitcoin-NG differentiates between *Keyblocks* and *Microblocks*.

Keyblocks Include a proof of work. No transactions, a public key, and the hash of the last Key- or Microblock.

Microblocks Include no PoW but transactions, a previous block hash, and a signature matching public key from last KeyBlock.

Fork resolutions Longest chain rule (or GHOST) is used but only looks at the Keyblocks. **Fee distribution** Fees are distribution between the creator of the microblock (40%) and the creator of the next Keyblock (60%).

Note 40. (Bitcoin-NG)

- Security and fork probability (keyblocks) is decoupled from throughput and transaction rate (microblocks).

- The distribution of fees (40/60) ensures that it is better to reference the latest microblock, rather than keeping the transactions for your own microblocks. Also significant incentive for actually publishing microblocks.

Problem

- Since Bitcoin-NG relies on a leader, progress can be reduced through leader failure. A single leader failure will not be a problem, but if an attacker causes multiple leaders to fail, e.g. by a DDOS (Distributed denial of service) attack, transaction throughput may be significantly reduced.

Example 15. *To understand the choice of 40%, 60% fee distribution, consider the example in Figure 5.1. Squares represent key blocks, while circles represent microblocks.*

In Figure 5.1 the minor of the red block gets 60% of the fees from the yellow transactions from microblock b'_0.

If he instead mines on top of b_0*, he can himself publish a microblock including the yellow transactions. This is shown in Figure 5.2. In this case the red minor receives 40% of the fees from yellow transactions. Additionally, if the red minor has a fraction of the mining power equal to* α*, he has a probability* α *to mine key-block* b_2 *and get additional 60% of the yellow fees. If* $\alpha < 0.33$ *then* $0.4 + \alpha \cdot 0.6 < 0.6$*. Thus the red minor will try to extend* b'_0.

Similarly, if the blue minor does not publish block b'_0 *but instead hopes to publish* b_2 *in Figure 5.2, his expected fraction of the yellow fees is* $\beta \cdot 0.6 < 0.4$*. Thus for a hashing fraction* $\beta < 0.33$ *it is better to publish* b'_0.

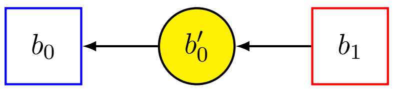


Figure 5.1: Red miner gets 60% of yellow transactions.

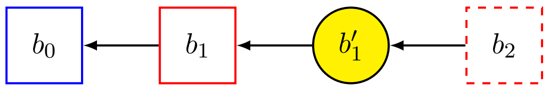


Figure 5.2: Red miner expects to get $40 + \alpha \cdot 60\%$ of the yellow transactions.

5.3 Sharding for blockchains

When sharding, the idea is to divide the blockchain into many smaller chains. Each small chain (i.e. shard) maintains part of the data.

Definition 41. In a *sharded blockchain* every node only maintains a part of the data and processes transactions that access the data he stores.

Note 41. **Problems for sharding:**

A) How to distribute the state?

B) How to process updates that access state in multiple shards?

C) How to avoid that an attacker takes over one shard?

Solutions

A) Using consistent hashing. E.g., shard 1 stores all outputs that belong to addresses that start with 01.

B) There are different cases:

- Payments from one shard to another, can be split into a withdrawal and deposit. Execute withdrawal first. To execute deposit, must reference withdrawal.
- Transactions that are conditioned on multiple shards use a variant of 2 Phase Commit, e.g. first lock required outputs. When all necessary outputs are locked, execute transaction.

Note that variant 2 can diminish the benefit of sharding, since it requires coordination and locks state.

C) There are two ideas:

- Distribute miners to shards according to their hashes, and redistributed frequently.
- Allow miners to participate in multiple shards, and to publish blocks on multiple shards with a single proof of work.

Both of these are problematic. In the first, redistribution is reducing the benefit of sharding. The second encourages minors to form pools.

Chapter 6

System models

In the Bitcoin protocol, any node can join. Additional any entity can create multiple accounts on bitcoin. Accounts are per default anonymous. This model is called **unpermissioned**. No permission is needed to participate.

6.1 Permissioned systems

Permissioned systems assume there exists a list of members. Table 6.1 gives an example. Typically assume that all members have a copy of the membership list.

The existence of a membership list, or possibility to create and distribute one is often referred to as public key infrastructure (PKI). It allows any two members to contact each other (using listed IP addresses), and create an authenticated channel (using public keys).

ID	*pubkey*	IP	...
1	0x1f3		
2	21xf3		

Table 6.1: Membership list example.

We distinguish different permissioned system models, based on the failure assumptions, e.g. what failures may happen.

No failure Assuming that members do not fail, it is possible to use authenticated channels for trusted interactions.

Central node does not fail Given a centralized node, that processes trust to not fail, they can build a centralized system, where the trusted node is responsible to manage state and interaction for the other members.

This model is used in cloud services, where all users connect to the service provider and trust this provider to store and maintain their application state and coordinate interactions with other users.

Crash failure In this model, nodes can stop by crash failures. It is not possible to communicate with a crashed node and the data stored on a crashed node may be lost.

This model is often assumed for machines that cooperatively run a service within a trusted domain, e.g. servers within a data center.

In this model it is possible to implement consistent services, e.g. a blockchain, that work as long as a majority of the nodes is running. Examples are the Paxos algorithm thaught in DAT520.

Byzantine fault tolerance BFT It is assumed that *any node may fail or misbehave*, i.e. a faulty node may not only stop, but may act maliciously, trying to sabotage the application. However, it is assumed that *only a small fraction of the nodes actually fail or misbehave.*

Misbehavior may for example be caused by

- virus or malware
- misconfiguration
- sabotage

The above assumption says, that at most a small fraction of the nodes will be victim to any of these.

In this model it is possible to implement consistent services, e.g. a blockchain, that work as long as a large majority, e.g. 2/3 of the members are correct.

Selfish or rational misbehavior All nodes could misbehave if it benefits them. Such nodes are called *selfish* or *rational.*

In this model it is possible to design algorithms using game theory. For this, every node is assigned a utility function:

For example the utility of participating in bitcoin mining is the block reward and transaction fees, minus costs for computation and networking, e.g. energy and hardware.

The goal for game theory is to show that nodes cannot increase their utility, by deviating from a protocol, e.g. by deviating from the longest chain rule.

Chapter 7

A BFT blockchain protocol

7.1 Proof of certification

Unpermissioned blockchains require that published blocks carry a nonce that causes the block hash to be meet a specific difficulty (e.g. start with a specific number of zeros). The this proof of work has two main functionalities:

1. **Publishing rate:** Requiring a proof of work ensures that blocks are published at a limited rate. This ensures that, whp., a block is propagated throughout the network, before the next block is published.
2. **Fork probability:** If, while a block is propagated throughout the network, another block is published, these blocks create a fork and put the system in an undecided state. Proof of work ensures that the probability that two blocks are found concurrently is small.

In a permissioned system, similar guarantees can be achieved using a certificate.

Definition 42 (System Model)**.** A permissioned system is comprised of N nodes $n_1, n_2, ..., n_N$. We assume that nodes have access to digital signatures, and that each node knows public keys of all other nodes.

Definition 43. A certificate for a block b is a collection of digital signatures for b. A certificate contains signatures from more than $N \cdot \frac{2}{3}$ different nodes. We write c_{min} for the minimum number of signatures contained in a certificate

$$c_{min} = \left\lceil \frac{2N+1}{3} \right\rceil$$

Idea 7. *We now give an intuition how certificates can limit publishing rate and fork probability:*

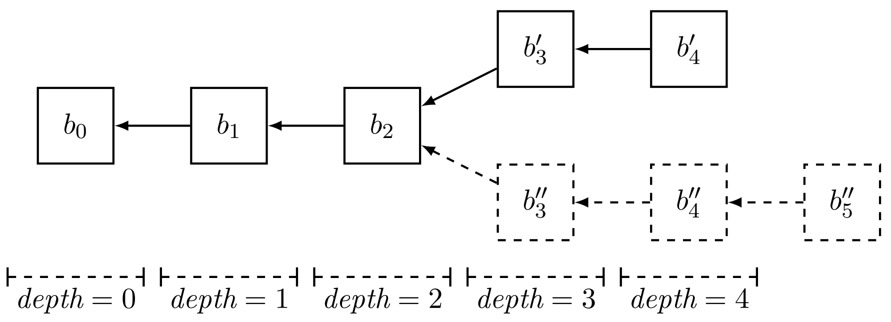


Figure 7.1: Blocks in a tree and depth d of blocks.

1. ***Publishing rate:*** *To publish a block with certificate, this block has to be transmitted, validated and signed by at least c_{min} nodes. Thus, blocks cannot be published faster, than (most) of the nodes can validate them.*

2. ***Fork probability:*** *If less than $\frac{N}{3}$ nodes signed multiple conflicting blocks, no two conflicting blocks can both receive a certificate.*

7.2 Safety

In the following we define a set of rules that correct nodes should follow. We assume that at least c_{min} nodes are following these rules.

The *previousBlock* pointers create a tree structure on blocks. We can thus refer to the *depth* of a block. We write $b.depth$ for the depth of block b. Figure 7.1 shows a tree with different depth levels. Here $b''_4.depth = 4$. Similarly, we refer to the parent, ancestors or descendants of a block. In Figure 7.1, e.g., b''_5 is a descendant of b''_3 and all blocks are descendants of b_0. Further, b'_3 is the parent of b'_4 and b_2 is an ancestor of b'_4.

We can not define the first rule.

Rule 1. *After signing a block at depth d, only sign at depth $d' > d$.*

The first rule says that nodes should only sign at increasing depth. Especially, they should not sign at the same depth twice.

As noted in Section 7.1, to prevent forks we should also add a rule that prohibits changing from one branch of the tree to another, e.g., signing b''_5 in Figure 7.1 after signing b'_3 and b'_4. We further note, that it is possible, that no block at a certain depth receives a certificate. E.g., b'_3 and b''_3 may both be signed by 5 out of 10 nodes. Thus, no block has the required ($c_{min}(10) = 7$) signatures. Therefore it should still be allowed for a node to sign b'_4 after signing b''_3.

Definition 44. The *locked block* at node n_i, $n_i.lock$ is the block b at highest depth, such that n_i has (or has seen) a certificate for b.

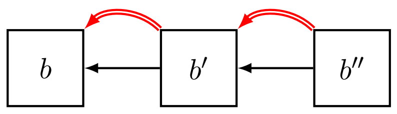


Figure 7.2: Blockes with red justification link, confirm block b.

Rule 2. *A node n_i only signs a block that is a descendant of $n_i.lock$.*

We note that if c_{min} nodes sign a block b, that does not imply that these nodes know the certificate of this block. Certificates have to be collected and disseminated by some node. We therefore define the following:

Definition 45. We assume that every block b_i contains a certificate for a block b_j such that b_j is an ancestor of b_i. We say that $b_i.justify = b_j$.

Lemma 9. *For some node n_i that follows Rule 2, it holds that, after signing block b, $n_i.lock.depth \geq b.justify.depth$ holds.*

Example 16. *In Figure 7.1, if $b'_4.justify = b'_3$, then after signing b'_4 a node following Rule 2 will not sign b''_5.*

Definition 46. We say that a block b is *confirmed*, if there exist blocks b' and b'', such that, $b = b'.justify$, $b' = b''.justify$, and $b.depth = b''.depth - 2$.

We note that the notion of a confirmed block is similar to proof of work blockchains like bitcoin, where a block counts as confirmed if it has been extended by a certain number of blocks, e.g. 6 blocks in bitcoin.

Theorem 6. *If nodes follow Rule 2 and a block b is confirmed, then any certified block at depth $d > b.depth$ is a descendant of b.*

Proof. If b is confirmed, there exist b' and b'' as in Definition 46. b'' contains a certificate for b'. Thus at least c_{min} nodes have signed b'. b' contains a certificate for b. Thus upon signing b', a node n_i sets $n_i.lock = b$.

We have to show that every certified block β with $\beta.depth > b.depth$ is a descendant of b. We proof this by induction over $\Delta_d = \beta.depth - b.depth$.

If $\Delta_d = 1$ then $\beta = b'$ since at most one block at depth $b.depth + 1$ can be certified.

Let $\Delta_d = n + 1$. Both β and b' are signed by c_{min} nodes. Let n_i be a correct nodes (following Rules 1 and 2) that signed both β and b'. Due to Rule 1, n_i signed b' before β. Thus $n_i.lock = b$ did hold. The induction hypothesis implies that when signing β, $n_i.lock$ was either b or a descendant of b. Rule 2 implies that β is a descendant of $n_i.lock$. Thus β is a descendant of b. □

7.3 Liveness

In Section 7.2 we have defined rules, how correct nodes should sign blocks and how we can identify confirmed blocks, based on published blocks.

However, we did not define when and how blocks should be created. Further, we note that if at every depth, many blocks are created, it is possible that no block will ever receive a certificate.

To resolve this, we assign leaders to every depth and only allow the leader to publish blocks at his depth:

$$\text{leader}(depth) = n_{depth \bmod N}$$

Algorithm 4 shows how nodes only sign blocks at the next depth and only if the block is published by leader($depth$). However, based on a timeout, nodes can hop over one depth.

Algorithm 4 Rotating leader

```
1:  $depth = 1$
2:  $timer$ = start()
3:  **on** receive $b$ from leader($depth$)
4:      **if** $b.depth = depth$ and $b$ descendant of $lock$ **then**
5:          sign($b$)
6:          $depth++$
7:          $timer$ = restart()
8:      **end if**
9:  **on** $timer$ finish
10:     $depth++$
11:     $timer$ = restart()
12: **on** leader($depth$) = $self$
13:     ask all nodes for locked certificate
14:     **if** block is missing ad depth $depth - 1$ **then**
15:         create empty block at $depth - 1$
16:     **end if**
17:     create block at $depth$ including deepest certificate
```

Lines 12 to 17 show how a correct node should propose a block. First a correct node needs to query other nodes, especially his predecessor, for certificates they have collected. There are two cases:

a) If a nodes collects a certificate for the last block ($depth - 1$), he can immediately publish a new block including this certificate.

b) If a nodes does only collect certificates for older blocks. He includes the certificate at highest depth in his block. If no block at $depth - 1$ is known to the node, he may also create that block.

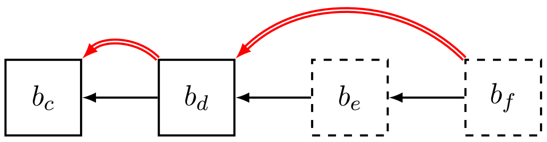


Figure 7.3: Leader for $b_f.depth$ may also create b_e.

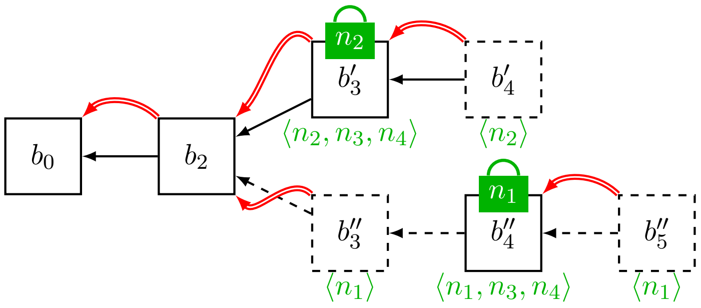


Figure 7.4: Figure illustrating Example 17.

The situation from Case b) is also shown in Figure 7.3.

Example 17. *Assume $N = 4$. Thus $c_{min} = 3$ and we have nodes n_1, n_2, n_3, and n_4.*

In Figure 7.5, green subscript shows the nodes that have signed a specific block. Nodes n_2, n_3, and n_4 have signed b'_3, while n_1 has signed b''_3. Further, n_2 has signed b'_4. Thus $n_2.lock = b'_3$. We assume that only n_2 has seen the certificate for b'_3.

Since n_3 and n_4 have not seen a certificate for b'_3 they have, together with n_1 signed b''_4. n_1 has seen this certificate, when signing b''_5. Thus $n_1.lock = b''_4$. We note that in this example, none of the nodes have behaved faulty.

Example 18. *This example extends Example 17. Thus we again assume $N = 4$. In Figure 7.5 shows a situation as in Example 17.*

Node n_2 has locked b'_3 and node n_1 has locked b''_4. We note that Rule 2 prohibits n_2 from signing b''_6 and n_1 from signing b'_6. This may deadlock the system.

However we note that Rule 2 allows n_2 to sign b''_5 as shown in Figure **??**. *The reason is that b''_5 includes a certificate for b''_4. This allows $n_2.lock$ to be updated.*

In Example 18, it is not helpful, to let n_1, n_3 and n_4 report whether they have voted for b'_4 or b''_4. The reason for this is that one node may violate Rule 1. Thus n_3 or n_4 may also sign b''_4.

Existing systems show two possible solutions for the problem shown in Example 18. The PBFT algorithm allows n_2 to sign b''_6 if he receives messages from n_1, n_3 and n_4 saying that b_2 is their last certificate.

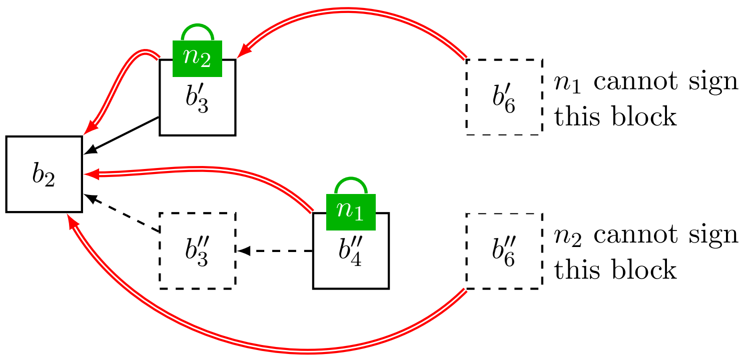


Figure 7.5: Figure illustrating Example 18.

Other systems, like Tendermint, include a long timeout while collecting certificates on Line 13, during which the new leader waits for additional certificates. This ensures in Example 18, if n_1 is correct, the certificate for b_4'' will be forwarded to the new leader.

7.4 Safety revisited

In the following we present a variant of Rule 2 that does not lead to the problem from Example 18 in Section 7.3.

We first define a 3-locked block.

Definition 47. A node n_i sets $n_i.\mathit{lock}_3 = b$, if b is the block at highest depth, s.t. n_i has signed a block b'' with $b''.\mathit{justify} = b'$ and $b'.\mathit{justify} = b$.

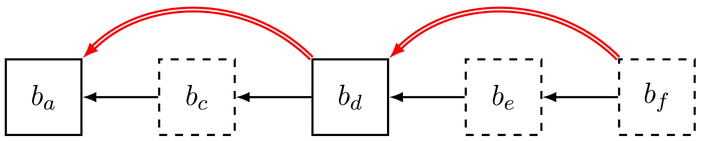


Figure 7.6: Block b_a is locked on signing b_f.

Rule 3 (replaces Rule 2). *A node n_i signs a block b only if*

a) b is a descendant from $n_i.\mathit{lock}_3$

b) $b.\mathit{justify}.\mathit{depth} > n_i.\mathit{lock}_3.\mathit{depth}$.

Example 19. *Figure 7.7 shows position of 3-locks in the same situation as shown in Figure* **??**. *Rule 3 allows to sign any descendant of b_2.*

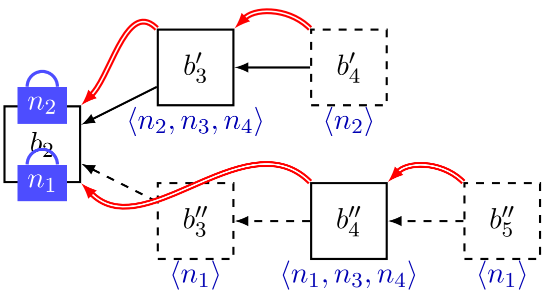


Figure 7.7: Showing 3-locks that allow two sign any block extending b_2.

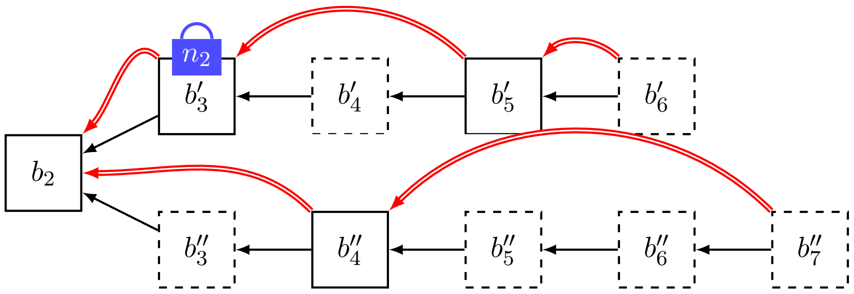


Figure 7.8: Rule 3 b) allows n_2 to sign b''_7.

Figure 7.8 shows a similar situation as in Figure 7.5, only using 3-locks. n_2 has signed b'_6 and set its 3-lock to b'_3. The leader for depth 7 failed to collect the certificate for block b'_5 from n_2. Thus he extended block b''_4. Rule 2 b) still allows n_2 to sign b''_7.

Using Rule 3, it is enough if the leader waits for the last certificate from c_{min} nodes on Line 13 of Algorithm 4.

Definition 48. We say that a block b is *3-confirmed*, if there exist blocks b', b'', and $\hat{b}$ such that, $b = b'.justify$, $b' = b''.justify$, and $b'' = \hat{b}.justify$ and $b.depth = b''.depth - 2$.

Theorem 7. *If nodes follow Rule 3 (instead of Rule 2) and a block b is 3-confirmed, then any certified block at depth $d > b.depth$ is a descendant of b.*

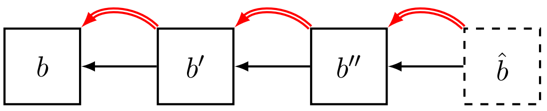


Figure 7.9: Block b is 3-confirmed.

Theorem 8 (Speculation)**.** *Using Rule 3, if a node n_i cannot sign a block b due to Rule 3, then n_i has a proof of misbehavior against the leader that published b.*

Chapter 8

Smart Contracts

8.1 Ethereum and Smart Contracts

See slides

8.2 Smart Contract Security

See slides and `https://github.com/ethereumbook/ethereumbook/blob/develop/09smart-contracts-security.asciidoc`.

8.3 Oracles

8.4 Off chain transactions

Chapter 9

Hybrid blockchains

Blockchains like bitcoin, running in an unpermissioned setting only provide probabilistic guarantees. This applies both to proof of work, proof of stake or proof of utility based systems. This is different from BFT systems used in permissioned settings, that provide strong guarantees.

9.1 Consensus guarantees

In this section we investigate the different guarantees given and techniques to build a hybrid system, providing strong guarantees in an unpermissioned setting.

9.1.1 Bitcoin guarantees

According to Bitcoin-NG, the consensus mechanism deployed by Bitcoin (i.e. Nakamoto Consensus) gives the following guarantees assuming that Byzantine or misbehaving nodes hold at most $f < \frac{1}{4}$ of the mining power:

Termination There exists a time difference function $\Delta(\cdot)$ such that, given a time t and a value $0 < \epsilon < 1$, the probability is smaller than ϵ, that at times $t', t'' > t + \Delta(\epsilon)$ a node returns two different states for the machine at time t.

Agreement There exists a time difference function $\Delta(\cdot)$ such that, given a value $0 < \epsilon < 1$, the probability that at time t, two nodes return different states for $t - \Delta(\epsilon)$ is smaller than ϵ.

Validity If the fraction of mining power of Byzantine nodes is bounded by f, then the average fraction of state machine transitions that are not inputs of honest nodes is smaller than f.

Note especially that Termination and Agreement only hold probabilistically. This may not be a problem in practice, i.e., ϵ can be chosen small enough that the difference between probabilistic and absolute guarantees can be ignored. However, to achieve a small ϵ may require long waiting times, as in bitcoin where the confirmation of a block may take more than 1h. Similarly, if one of these properties, e.g. Termination, is violated it is difficult to say wether this is due to the probabilistic nature of this property, or to the fact that assumptions on the failure threshold f where violated.

9.1.2 BFT guarantees

In comparison, we now analyze agreement and termination properties of the BFT protocol described in Chapter 7. Here we assume that at most $f < \frac{1}{3}$ of the nodes in a permissioned membership misbehave (are byzantine).

We have shown, that once a block is confirmed according to Definition 46 or 48 then all future certified blocks will be descendants of this block. We note that, if we consider a block as confirmed according to these definitions, we can also consider all its ancestors as confirmed. Especially, we get the following agreement and termination property.

Termination safety If a correct node considers a block at *depth* l as confirmed, it will never change this block.

Agreement No two correct nodes will disagree on which block is confirmed at *depth* l.

Termination liveness For some *depth* l, some block at depth $l' \geq l$ will eventually get confirmed.

For Termination liveness to hold, it is necessary that a sequence of blocks is proposed by correct nodes.s

9.1.3 Hybrid BFT

Several blockchain systems propose to use concepts from permissioned BFT to improve Termination and Agreement properties of unpermissioned blockchains.

Resources

ByzCoin is a research system from EPFL that builds BFT on top of a proof of work blockchain.

Casper FFG is a proposal to add a BFT based system, also utilizing Proof of Stake, on top of Proof of Work in Ethereum, to ensure non-probabilistic termination.

The difficulty in running a BFT protocol, as proposed in Chapter 7 in an unpermissioned setting is to identify the set of nodes and avoid sybil attacks. Proof of Work or Proof of Stake can be used for this purpose.

In ByzCoin, nodes must solve a Proof of work puzzle to become a node on the BFT system. ByzCoin applies a fixed window size. A node added with one PoW solution will be removed after the end of this window. We note that in ByzCoin, nodes are not equal. Instead every node has a *voting power* relative to the number of PoW puzzles he submitted in the current window. Thus, instead requiring a certificate with signatures from $\frac{2}{3}$ of the nodes, ByzCoin requires signatures from nodes, that together hold $\frac{2}{3}$ of the voting Power in the current window.

Casper FFG on the other hand, requires nodes participating in the BFT protocol to deposit "Stake", i.e. Ether. A node depositing above a minimum of Ether may participate in the BFT protocol. Again, a node has a *voting power* representing the fraction of the total "Stake". Casper FFG uses Proof of Work to avoid the leader election problem. I.e. proposed blocks need to contain a proof of work. This limits the number of blocks that can be proposed concurrently.