b
UNIVERSITÄT
BERN

# Blockchain and BlockDAG Protocols

**Scaling Bitcoin**

**Bachelor Thesis**

Luca Althaus

from

Lauperswil BE, Switzerland

Faculty of Science, University of Bern

22. December 2020

Prof. Christian Cachin
Ignacio Amores Sesar
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

# Abstract

Bitcoin is an alternative cash system where one does not have to trust any central bank. Bitcoin stores its transactions inside a chain of blocks, the blockchain. Everyone can anonymously join the Bitcoin network to make transactions or participate in the mining process to generate new blocks that secure the transaction order. Bitcoin is the first cryptocurrency that is widely established. However, its scalability is limited due to security reasons and they way it was initially defined. This thesis looks at the three alternative protocols GHOST, Conflux and GHOSTDAG. They aim to improve the scalability of Bitcoin. We compare those protocols to Bitcoin and each other using a simple simulation.

# Acknowledgements

First and foremost, I want to thank Ignacio Amores Sesar for supervising my thesis, discussing the topics with me and his support. I would like to thank Prof. Christian Cachin for providing his expertise. I also want to thank my friends Mathias Fuchs and Sabine Brunner for proofreading this thesis and Dominic Kohler for our code discussions.

# Contents

# Chapter 1

# Introduction

Bitcoin was introduced by Satoshi Nakamoto in 2008 with the aim to create a payment system that avoids needing to trust central banks. Today Bitcoin is the most used cryptocurrency.

Bitcoin stores transactions inside blocks. Each block has a reference to its predecessor block, creating a blockchain that contains the ordered transaction list. When multiple blocks share a common previous block (i.e., they are all linked to the same block) a fork is created. Forks raise the question of what transactions are now to be considered. The reason for the occurrence of a fork may be that someone attacks the blockchain and changes the ordering of the list that people reckon holds.

To prevent such attacks Bitcoin requires that people who want to commit a new block to solve a proof-of-work problem. This is a task they can complete with computational power. When an attacker has not enough power(i.e., under 50% over the overall power) it should not be able to change the transaction list. We always assume that attackers own less than half the overall computational power. To minimize the probability that an attacker can indeed revert a transaction, perhaps by being lucky and solving the proof-of-work problem faster than expected, some waiting time is required until a block and the transactions it contains are considered valid. As the waiting time increases, the chance that the attacker solves a higher number of proof-of-work problems than the rest of the network decreases. In fact, the attacker's chance to revert a block should converge to zero as the waiting time increases.

It is possible that the attacker still has the chance of being successful, even after some waiting time. This is because forks may occur when people create new blocks in parallel, meaning they do not know of each others blocks and hence there is no reference path between them. There is no malicious intend in such forks, but they may happen naturally. The probability for forks to occur cannot be zero and increases when the proof-of-work problem is too easy to solve (i.e., the time between new blocks being created is too short). This leads to a performance bottleneck of Bitcoin. The average time until someone solves the proof-of-work problem is about ten minutes to prevent the forks that have no malicious intent. For a block to be considered valid, six new blocks are to be created in a line after the block. This buildup guarantees that with high probability the initial block will not be reverted. This leads to an overall waiting time of about an hour until a transaction is validated. One notes that the number of transactions inside a given block are limited, otherwise the block would take even more time to propagate due to its larger content. In view of the limited size and the fact that this additional block generation process is relatively slow, Bitcoin cannot be amplified to a higher transaction rate.

In this work we compare the Bitcoin protocol with three alternative protocols that aim to solve the above mentioned disadvantages. The first alternative one is the GHOST protocol that imposes other rules where new blocks should be added and when transactions are validated. The other two protocols are Conflux and GHOSTDAG. They take different approaches with the aim to not follow a single chain in the blockchain. They do this by including forked blocks inside the common transaction list, as well as using a total ordering algorithm over the blockchain. Theory of this is given in Chapter 2. To compare the three protocols we write a simulation using the programming language Python. Chapter 3 provides some insight into the implementation of these simulations while in Chapter 4 a few of the results found in the simulations are discussed and compared.

# Chapter 2

# Background

## 2.1 Distributed blockchain

The goal of the distributed blockchain is to establish a public decentralized transaction history among all participants in a network. We start with some basic definitions based on the Bitcoin protocol.

**The network** is a connected graph where each user represents a node. Each node maintains connections to other nodes. In the Bitcoin network each node tries to maintain eight connections. We initially assume that for every two nodes there is a path between them (i.e., the graph contains no isolated sub graph). Each node keeps a copy of the blockchain. A user that generates new blocks is called a miner.

**The blockchain** aims to store all the transactions in blocks that can then be put together to build a common transaction history. The blockchain starts with the initial block called the genesis block. All other blocks have a reference to a previous or parent block. The transaction history in the blockchain must not have conflicting transactions. This means a transaction is only added to a block if it does not contradict the transactions in the blocks before. Otherwise the block is not going to be accepted by other participants of the network. Fig. 2.1 is an abstract example of such a chain of blocks. We now want to be able to follow this chain to determine how much money each participant has. For example if the user D had no Bitcoins before Block10012 he now has $500 - 27 = 473$ Bitcoins assuming there were no other transactions he was involved in.

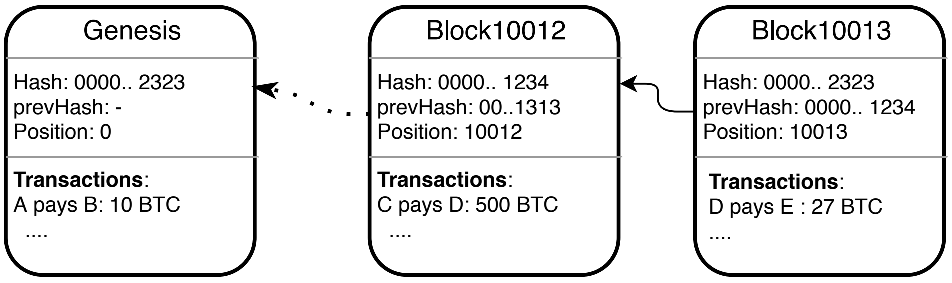

**Figure 2.1.** Example of connected blocks. The dotted arrow means that there are other blocks in between Block10012 and the genesis block. Only a few pieces of information of the blocks body are shown. There can be other attributes in the blocks, like the creation time or the number of transactions.

**Proof-of-work** is used in all the protocols of this work. It is used to secure the content of each block in the chain. A new block can only be added to the chain if a correct input to a hash function (e.g., SHA-256 for Bitcoin) is found. The input is considered correct if the output of the hash function has a given number of leading zero bits. This implies that when a block is created all its pieces of information that have to be immutable for a consistent list of transactions, as for example transactions themselves and the hash of the previous block, are entered into an additional hash function. This resulting hash is then stored in the new block. The time to compute a correct hash grows exponentially with the number of leading zeros needed. In Bitcoin for example a new block is only created every ten minutes on average. Nodes creating a new block usually get a reward for providing their mining power inside the block they find.

**Information inside a block is not alterable.** This is an important assumption we make during this whole work. The assumption states that no other block with the same hash can be found. We make this assumption because the chances to get the same hash out of the hash function for some other input are extremely low. This is important to assume because else one could find a new hash that already represents another block. References to that block would then also reference that new hash leading to ambiguous situations, ambiguities can lead to security problems. Note that we can find another block with the same information but with another hash also beginning with the required number of leading zeros. For example, we generate a random number containing 30 bits and we want the first ten bits all to be zero. We assume that every bit has a 50% chance to be zero. The chance of a random string having ten zero leading bits is $1/2^{10}$. The rest of the twenty bits has no restrictions. Hence there are $2^{20}$ different possibilities to solve the problem. Finding the same hash implies to hit the exact same number again. The verification of a hash is simply done by applying the hash function on the information that is already provided and check if the result is the same as the block's hash. We assume this verification can be done in negligible time compared to finding a new one.

To summarize, we say that the chance of finding a block with the same hash is zero. The time to find a valid hash resulting from the same input does not depend on how many hashes are already found and one can instantly check if a hash is correct.

**The protocol** specifies the rules and how nodes should interact with each other and how the blockchain should be interpreted exactly (e.g., not to accept any block that has conflicting transactions with any block before). Nodes that do follow the rules of the protocol are denoted as honest nodes. Each honest participant in the network broadcasts all valid transactions and blocks it receives to the nodes it is connected to. This also means a node has to check if the proof-of-work of the block it revives is valid. If the honest participant is a miner it collects all new transactions it gets inside a block and tries to find a solution to the proof-of-work function including the previous block's hash. What block to choose as the previous one may not always be clear. The explanation for this is explained in the next section.

## 2.2 Forks

In this section the reason we look at the different protocols is introduced. The problem with a decentralized blockchain is the possibility that blocks are not always built in a chain but that multiple blocks can refer to the same parent Fig. 2.2. This results in problems that raises the following questions:

1. **What transactions are considered valid?** Blocks not being in the same chain e.g., the blocks **G** and **D** in Fig. 2.2 could contain transactions that are conflicting with each other. For example the same money being spent twice by the same person.

2. **What block do miners try to reference as parent while mining for new blocks?** Miners aim to keep a chain and have to choose one of the blocks **G** , **H** , **J** and **F** as parent block.

Another important question is who gets rewarded for mining a new block. If a miner does not get rewarded because a forked block got the reward and it still has to pay for the energy and hardware, it will reconsider mining new blocks. We will put the reward aside and focus on the two main questions given above.

### 2.2.1 Network delay

The natural reason for a fork to occur is when two or more miners mine a new block faster than it takes the network to propagate the blocks to each other. More formally: Let $D_{i,j}$ be the time a block takes from $Node_i$ to $Node_j$ and $t_i$ , $t_j$ the time moment $Node_i$ respectively $Node_j$ mined their block. If

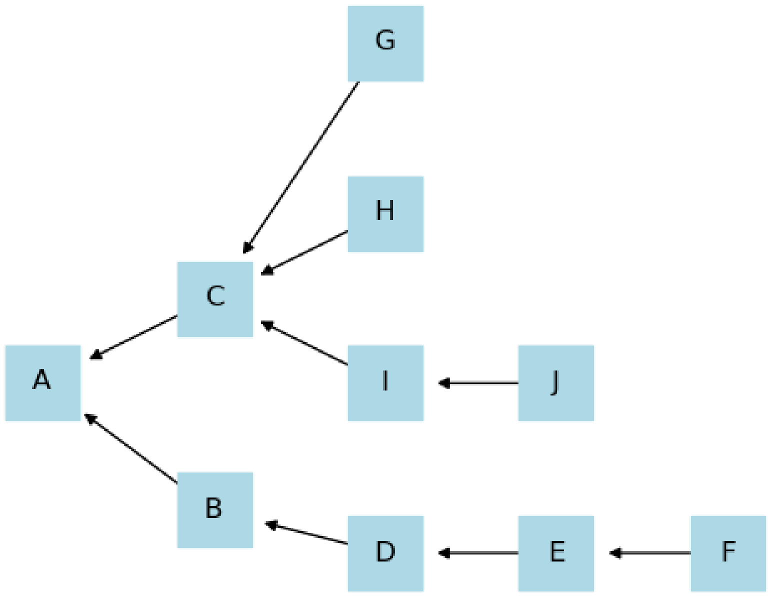

**Figure 2.2.** Blocks that link to a tree instead of a chain.

$Node_i$ finds a block before $Node_j$ and $t_j - t_i < D_{i,j}$ it is not possible that the blocks found by $Node_i$ and $Node_j$ contain a reference to each other and a fork is created.

The number of forks therefore increases when blocks take more time to be transmitted through the network. This results from poor connections between miners, when the miners have fewer total connections or when block sizes increase, meaning there is more data to be sent. A different reason that is specified by the protocol for the occurrence of forks is a shorter time interval between the generations of blocks.

When trying to process more transactions in the same network more forks are created.

To include more transactions either the block size has to increase or the time between creating two blocks has to be lower, given that the blocks have to remain the same size. Hence the probability of forks appearing increases either way.

### 2.2.2 Double-spending attack

Another reason for forks in the blockchain are nodes with malicious intent, meaning that they do not follow the protocol. These nodes do not extend the block that should be extended according to the protocol on purpose. The goal of the double-spending attack is to first make the whole network believe that a block $b$ and its transactions are accepted. Then the attacker commits more blocks that did not follow the protocol with the intent that people change their minds and will extend these blocks such that $b$ is not considered valid anymore and the transaction is reverted. The double-spending attack is an attack against the Bitcoin protocol. A precise example is given in Subsection 2.3.1.

The protocol should be defined such that if over 50% of the nodes are honest the probability of success of such an attack is as small as possible while still having reasonable confirmation times of the blocks.

## 2.3 Protocols

In this section we describe the different protocols. In Subsection 2.3.1 the Bitcoin protocol is introduced and in Subsection 2.3.2 the GHOST protocol. These two protocols rely on using a main chain inside the tree that is created if blocks do fork. The other two protocols GHOSTDAG (Subsection 2.3.4) and Conflux (Subsection 2.3.3) create a directed acyclic graph (DAG) out of the tree and treat the ambiguity created by forks by putting a total order over all the blocks (i.e., the blocks are sorted such that the positions indicate in what order blocks and hence their transactions are considered).

### 2.3.1 Bitcoin

Bitcoin [1] follows the longest path rule. This means if the chain forks into two or more chains the block with the longest path to the genesis block is taken as block to extend the chain on. In case of ties the block that is received first is chosen to be continued.

For example in Fig. 2.2 **A,B,D,E,F** is the longest path so **F** is the block nodes following the protocol will choose to extend when mining a new block. In Bitcoin a block is considered valid if it is contained in the main chain and has six blocks contained in the main chain after it.

#### Double-spending attack on Bitcoin

In certain situations a malicious node is able to revert the validation of a block such that it can spend the money it already spent in that block again. The attacker has to generate a second chain that is longer than the honest chain. It commits this chain after the block it wants to revert is validated (i.e., has six blocks on its top). An example is given in Fig. 2.3.

Note that if the difficulty of the proof-of-work is not set constant, the path with the highest difficulty summation is taken. During this work we assume that the difficulty is constant.

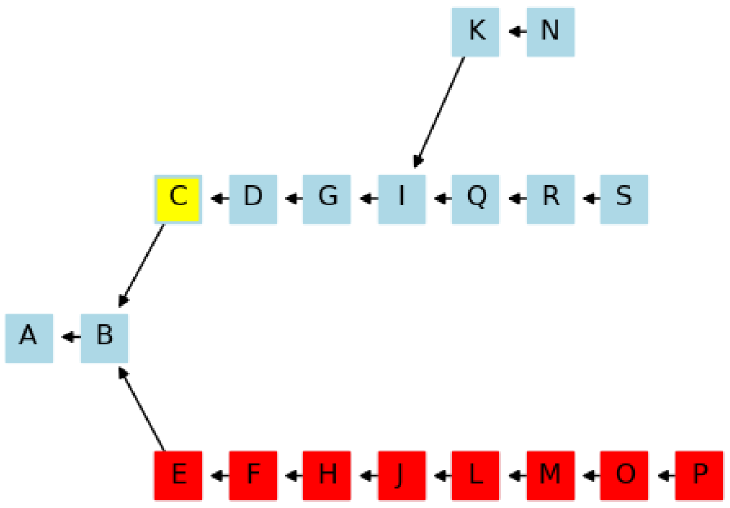

**Figure 2.3.** A chain with two forks. The blue blocks are created by honest miners. Hence the fork in block **I** is created by a network delay. The attacker mined the red blocks and the fork in block **B** comes from a double-spending attack on block **C**. The block **C** was considered valid when the block **S** is found without the red blocks **O** and **P**. After the attacker sends the blocks **O** and **P** to the network, **P** is the new head of the chain and the block **C** is not considered valid anymore.

### 2.3.2 GHOST

GHOST [2] is short for ”Greediest Heaviest-Observable Sub-Tree”. Instead of following the longest path rule as in Bitcoin 2.3.1. To determine what way to take when a fork occurs the path with the block that has the heaviest sub tree appended to it is taken. Fore example in Figure 2.3 the chain starts at **A** followed by **B**. Since there is a fork at **B** we have to determine what path to follow. We need to consider the sizes, i.e., the number of blocks, of the sub trees starting from the blocks **C** and **E**. The blocks in

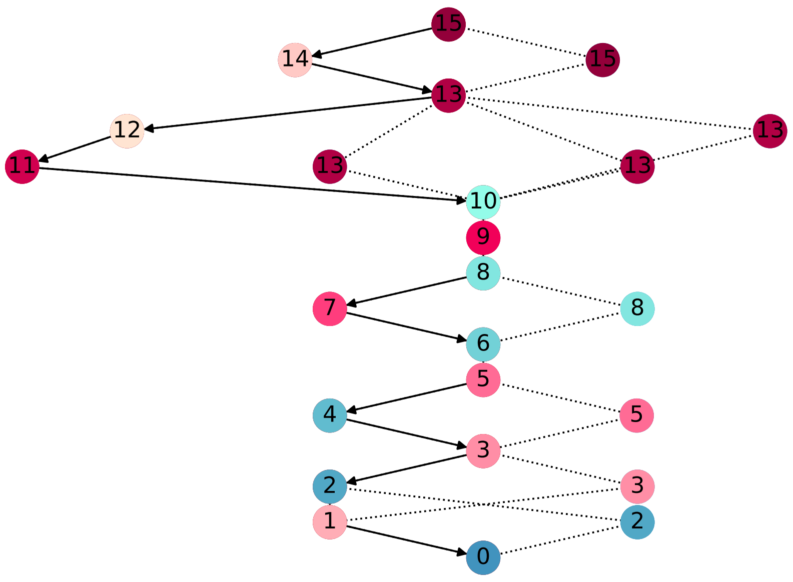

**Figure 2.4.** Graph evaluated with the Conflux protocol built from the bottom. The nodes connected by straight arrows represent the main chain that is calculated with the same procedure as in GHOST. Nodes that are grouped in the same epoch are marked with the same color and same number.

the sub tree of **C** are {**D,G,I,N,K,Q,R,S**} therefore the size of the sub tree is eight. For **E** the sub tree contains the blocks{**F,H,J,L,M,O,P**} and has size seven. Since the sub tree of **C** is bigger, the path over **C** is taken. In the fork at **I**, **Q** is taken to be followed. The final main chain is {**A,B,C,D,G,I,Q,R,S**} and **S** is the block to be continued. We see that the double-spending attack does not succeed on this tree. The blocks in the fork caused by delay inside the network between honest nodes are not ignored and the honest miners have more impact. This example illustrates the goal of the GHOST protocol which is to be safe against double-spending attacks even when many forks occur.

### 2.3.3 Conflux

The Conflux protocol has the aim to handle thousands of transactions per second [3]. In contrast to Bitcoin and GHOST, Conflux establishes a total order over all blocks that are created. Instead of a main chain that does not contain forked blocks, all blocks are put inside one single list. The order of the blocks in that list then indicates how the transactions should be evaluated. If there are no incompatible transactions the list means the same as the main chain in Bitcoin and GHOST. Otherwise transactions that conflict with others earlier in the list are ignored. To achieve this order Conflux directly extends the GHOST (2.3.2) protocol. The block graph is built in the following way: When a new block is mined the parent block is determined as in the GHOST protocol, i.e., following the heaviest sub tree. All other leaves, meaning the blocks that do not have children, are also referred by the block as uncles.

**The total order algorithm** works as follows: First calculate the main chain of the heaviest sub tree. The blocks inside the main chain are now called pivots. For every pivot calculate its epoch. This means for every pivot a list is created by adding all the blocks referred back from the pivot until another epoch is reached. To make sure that previous epochs are already calculated, we simply start with the genesis block and go forward through the list. The genesis block has no uncles and does not include any blocks in its epoch. At the end all these lists are concatenated to make the total order over the graph. An example of the Conflux protocol is shown in Fig. 2.4.

It is important to see that if the pivots do not change, the epochs in between the pivots do not change either. This means if one trusts the main chain as in GHOST, then the order of all other blocks can be trusted likewise.

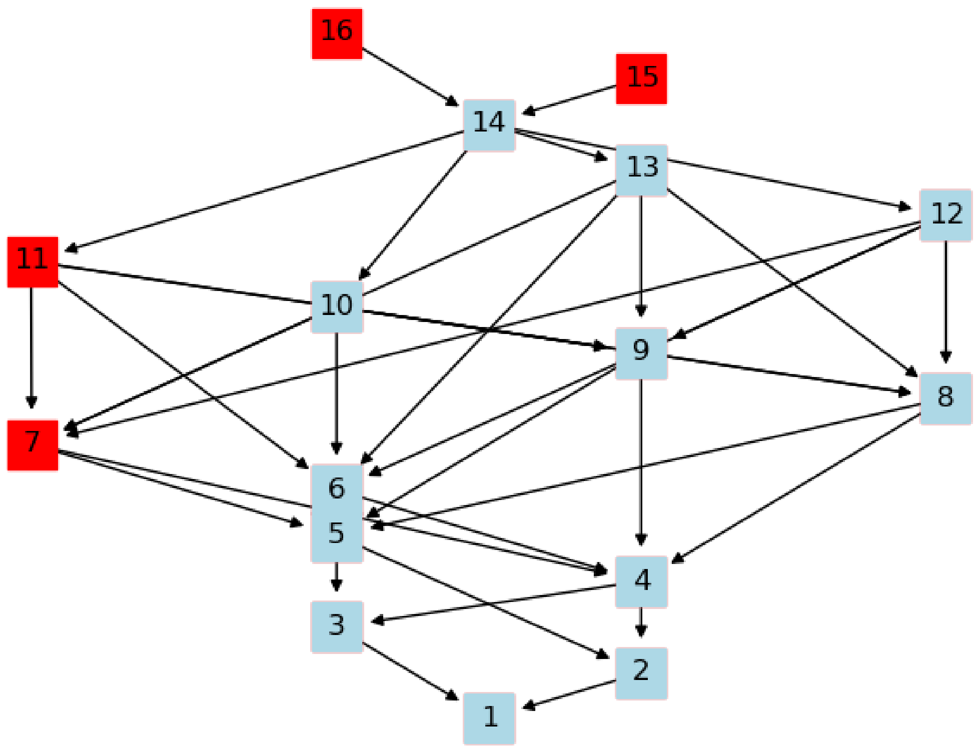

**Figure 2.5.** Sample plot of the GHOSTDAG protocol built from the bottom with a find rate of one and $k = 2$.

### 2.3.4 GHOSTDAG

When a new block in GHOSTDAG is created there is no main parent as in Conflux. All the current leaves simply are referred as previous blocks. The aim of GHOSTDAG is then to split the graph into two sets. A blue set $S_{blue}$ containing trustworthy blocks and a red set $S_{red}$ containing blocks that do not seem trustworthy. The two following definitions are needed to specify the rules of the protocol.

**A anticone of block $b$:** The set of blocks that are neither in the past or the future of $b$ is the anticone of $b$ . Note that the anticone relation is symmetric, if block $a$ is in the anticone of block $b$ then $b$ is also in the anticone of block $a$ but the relation is not transitive. Blocks in each other's anticone are similar as forks that happen in parallel (i.e., the sub tree of other blocks that reference the same parent block). Anticones appear for the same reason as forks in Bitcoin and GHOST. If every miner follows the protocol, blocks are only in each others anticone if none of the miner knows the other block when it mined its own one. Therefore it is a bad sign if a block has a huge anticone and this leads to our next definition.

**$k$-cluster:** A set of blocks $S_k$ is called a $k$-cluster if the size of the anticone is less or equal than $k$ $\forall b \in S_k$.

When a block gets mined the parameter $k$ tells us how many other blocks there can be found from other nodes that do not know this particular block such that this block can still be in the blue set. For example if we set the parameter $k = 0$ the blue set will be exactly the main chain as in Bitcoin since no blocks are allowed to be in the blue set that do not follow each other.

The initial protocol is called Phantom and its goal is to find the best $k$-cluster, i.e., the biggest possible $k$-cluster for a given $k$. This initial problem is NP-hard. The GHOSTDAG Protocol aims to establish the same with a greedy algorithm that does not necessarily find the best possible cluster.

In Fig. 2.5 we see an example of the GHOSTDAG protocol evaluating the DAG. The find rate is set to one and the parameter for the $k$-clusters is set to two. The blue nodes represent the blue set $S_2$. Node 16 and 15 are red because they are not backed up yet. The elements of the anticone set of block 10 for example are the blocks 11, 12 and 13 (the arrow head tells us if a line ends at the block or not). The blocks 14, 15 and 16 belong to the future of block 10 and all other blocks that are neither in the past or anticone belong to its future. This can be checked by following the arrows. To verify that a block belongs to the future of another block we can traverse the arrows backwards. We see 12 and 13 belong to that blue set $S_2$ already. Therefore block 11 can not belong to $S_2$ since in that case it would not be a two-cluster anymore. We would have to remove one of the blocks 10, 12 or 13 but those are actually better connected. The block 7 also can not belong to $S_2$ because the blocks 6, 8 and 9 are all in its anitcone and already belong to the blue set.

**GHOSTDAG algorithm** We try to give a simple description of the algorithm. The blue set and ordered list is defined inductively as follows:

1. The blue set and the ordered list of the genesis block contains only the genesis block itself, i.e., $Genesisblock$ and $[Genesisblock]$.
2. For all other blocks $b$ we select the previous block $b_{max}$ with the biggest blue set. We initialize blue set $set_b$ and ordered list $list_b$ of $b$ equals to the blue set and list of $b_{max}$ and add $b$ to both of them. We then iterate through the anticone of $b$ in an order that no block is visited before any of its past blocks and add them to the $list_b$. If the $set_b$ still maintains the $k$-cluster property if the anticone block would be in it we add it to the $set_b$.
3. For the blue set and ordered list of the whole graph we calculate the set of a virtual block that we add to all leaves as we would if we created a new one and proceed as in step two without adding the block itself.

The set and list of every block are calculated at the time they are created. When looking at the original **Algorithm 1** in [4] we see that they define the algorithm recursively on the paths of the block with the biggest $k$-cluster, showing that the future does not matter. This also implies that due to our assumption that the content of a block is secure by the proof-of-work, we can calculate the blue set and ordered list for every block and need to do this only once.

**Attacking GHOSTDAG.** The double-spending attack is not specified for this protocol. But we still try to simulate a similar attack on GHOSTDAG. The idea is that the attacker will make people believe that the block with the transaction it wants to double-spend on is not a block that can be trusted. Our approach is that the attacker simply does not include the block it wants to double-spend on in the list of its own blocks. The attack still includes all other leaves inside the block it mines.

# Chapter 3

# Implementation

We start introducing the software we use, then explain some principles of the implementations of our protocols and the network. We try to justify our design for the network and how we simulate the mining rate by introducing some real world monitored data. Finally we give our idea of how an attacker could carry out its attacks.

## 3.1 Software

The simulation is written in Python which contains many useful libraries. The `Scipy` library `scipy.stats` [5] that provides many statistical functions is used to simulate the random block generation process. For matrix multiplication and other computations we use `NumPy` [6]. We use the `NetworkX` graph library [7] to plot our blockchain graphs to gain some visual representation. We mainly store graphs inside simple Python dictionaries since it is mostly the only thing we need for our own implemented algorithms and we therefore have more control over the data structure.

## 3.2 Blocks and protocols

**The blocks** are implemented in a simple way. Since Python is not strongly typed we can just define a attribute previous for each block. Depending on the protocol this can be a single block, a list or a set of blocks. Every block has its position given as its minimal number of edges to reach the genesis block. Each block also contains its time step when it's created since one may want that if a fork occurs the earlier created block will be included into the main chain.

Each **protocol** belongs to a node and has an instance of a graph. The protocol class is responsible for managing if a block should be added to the graph or not. Every time a node receives a block $b$ the protocol checks if all the previous blocks of $b$ are contained in the graph. If not, the block is added to a waiting list. Else the block is added to the graph and the waiting list is checked for pending blocks that now may be added. **The graph** contains dictionaries with the blocks pasts and futures and other variables to make use of dynamic programming meaning that we do not want to make some calculations over again. The graph also contains all other methods the protocol needs. For example the method that returns the main chain. Here the graph stores that chain and if a new block is received. If the previous block of the new block is the last block of the chain it simply is added behind it since it will only reinforce the claim of the current main chain to be the best one for each protocol. Else the whole chain has to be recomputed since it is possible that the current main chain is not the right one anymore defined by the protocol. This implies there are slightly different graphs for each protocol.

The protocols mainly aim to answer the question of what blocks are accepted and what block or blocks we want to reference as the parent ones. **The Bitcoin protocol** for example contains a method that allows us to check if a block is accepted. The method checks if the block is inside the main chain without its last six elements.

When a new block $b$ is added to the graph and its distance to the genesis block is bigger than the actual block the chain would be extended on, $b$ is set as the new block that will be continued.

By assumption the past of a block and therefore the positions of a block that is defined by its distance to the genesis block does not change over time. The invariant for this protocol is that at any time the block to continue is the block with the longest path to the genesis block. Since the size comparison is done for every added block and the block to continue gets replaced if the position of the new block is higher the invariant always holds, showing this greedy approach works.

**GHOST** only changes in the way the main chain is calculated. But we need to calculate sub trees and can not simply rely on the highest position. Hence we can not use the same approach as in Bitcoin. From different implementations we choose the one that only computes the main chain when it is needed and not when a new block is received.

Adding a new block to a nodes chain results in the same step as with the Bitcoin implementation, given the block to continue is the previous of this block. In the other case the variable of the block to continue is set to None. If a chain is built the main chain can be returner directly. Otherwise it has to be calculated. We do this with the following recursive algorithm making sure every node of the graph is only visited once such that we do not get an algorithm with exponential run time.

```
1       def calculate_main_chain(self,block):
2           children = self.futures[block]
3           treesize = 1
4
5           if len(children) == 0:
6               return 1 , [block]
7
8           max_size = 0
9           heaviest_chain = None
10
11          for c in children:
12              subtreesize , futurechain  =  self.
                   calculate_main_chain(c)
13              treesize += subtreesize
14              if subtreesize > max_size:
15                   max_size = subtreesize
16                   heaviest_chain = futurechain
17
18          return treesize, [block] + heaviest_chain
```

If a block does not have any children (i.e., the block is a leaf in the graph) the algorithm simply returns the block in a list with a weight of one (until line 6). Else we iterate through all the children of the block and save the results from the one having the biggest sub tree size (lines 11 to 16). We return the summed tree sizes of all children plus one and the longest list concatenated with itself.

**GHOSTDAG.** As we saw in Subsection 2.3.4 the ordered list and blue set is fixed and immutable for each block. Their computation requires a lot of time and is called recursively for a new block. Hence we directly add them to the block when it is created to avoid the computationally challenging recursion and we do not need to store them separately for each node. We assume that the cluster parameter $k$ can not change over time.

We implement the algorithm similar as explained in Subsection 2.3.4 mainly using the set data structure provided by Python to compute the $k$-clusters.

**Conflux.** The Conflux implementation extends the one from GHOST. `block.previous` is an array with the parent at index zero, all the entries that follow are the uncles. The position also gets inherited from GHOST such that a blocks position is the distance to the genesis block by following the parents. Apart from the uncles we also have to add the computation of the epochs. First the main chain needs to be calculated given the GHOST protocol.

The code uses many effective Python features like the list and dictionary comprehension.

```
def createEpochs(self):
    main = self.get_main_chain()

    traverse = lambda block: reduce(lambda a,b: a+b
    ,[traverse (i) for i in self.dottet_past.setdefault(block, [])
       + self.pasts.setdefault(block, []) if i not in main],[block
       ])

    epoch = {i : traverse(i) for i in main}
    \\delete elements that  already are inside previous epochs

return epoch
```

We recursively travel from each main chain block all the paths back until reaching the main chain again, creating a single list at the end. The function `setdefault(block, [])` returns an empty list that can be completed on when a miss in the dictionary occurs instead of throwing an error. The function reduce concatenates lists together inside the lambda function. Different implementation have shown that doing it this way and then simply going through the epochs and remove blocks that are in previous epochs is a fast and effective way.

## 3.3 The Network

The task is to create a proper network. This means that the nodes are connected in a way such that the network behaves close to a real blockchain network. This is not really possible to do since the node number is strongly limited to computational power and storage. Therefore a simpler network with an adjusted mining rate has to be implemented.

### 3.3.1 Real data

To justify our implementation and compare our results we need real data of block transmission times and forks from a blockchain and choose the one from Bitcoin. We take the data provided by the Decentralized Systems and Network Services Research Group from the Karlsruhe Institute of Technology which is monitored in Germany [8, 9]. The data we use is visualized in Fig. 3.1 and Fig. 3.2.

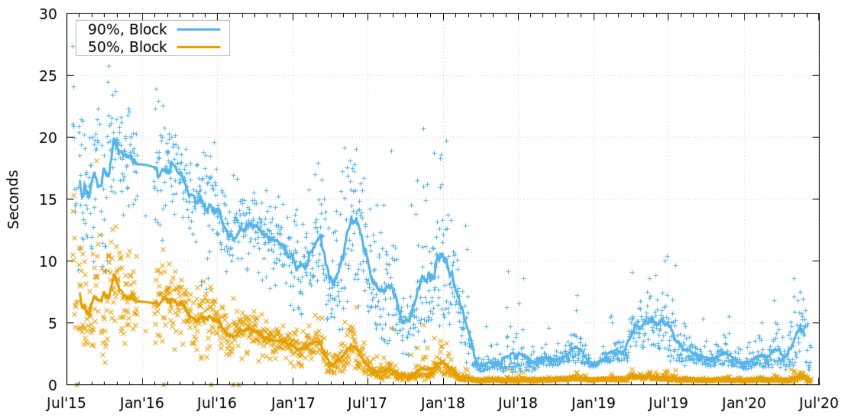

**Figure 3.1.** Block propagation delay history of the time it takes for a block to be announced to 50% and 90% of the network. [9, 8]

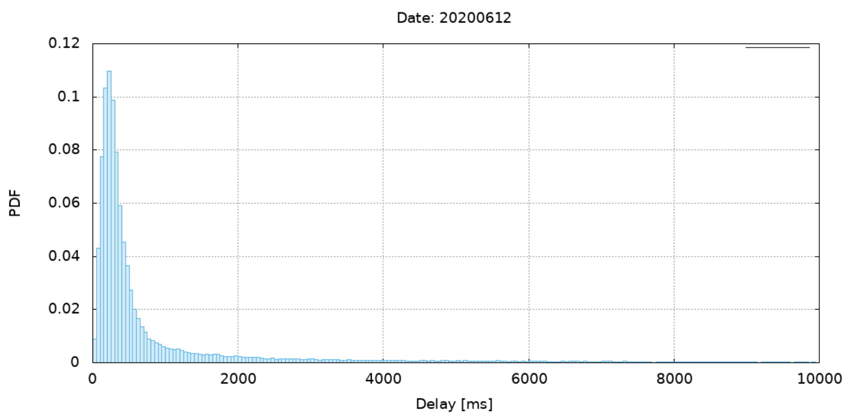

**Figure 3.2.** Distribution of the time interval between creation and sighting of blocks. [9, 8]

### 3.3.2 Our approach

Due to limited resources we simulate a network with 100 nodes only. We want that the nodes are connected randomly. With the aim to run the simulations multiple times on different network topologies. These results should then not differ to much from each other. For the representation of our node connections we choose a transposed connectivity matrix C with Boolean values. This means that if the $i_{th}$ column and the $j_{th}$ row of the matrix is set to `TRUE`. The $j_{th}$ node sends its information to the $i_{th}$ node referred as $Node_j$ and $Node_i$. We divide time into discrete time steps. In one time step each node transmits its new block on its connections. In the next step its connected nodes will transmit the block to their connections and so forth.

**Block broadcasting.** With our connectivity matrix we can then calculate the broadcasting as follows: Create a vector $v^{(1)}$ with its length equal to the number of nodes. Note that this also has to be the number of columns and rows of the matrix. If Node$_j$ finds a new block in the first time step we set the $j_{th}$ position in $v$ denoted by $v_j$ to `TRUE` and all other positions to `FALSE`. In Python `TRUE*TRUE` = True and `TRUE` + `TRUE` = `TRUE` and we can use simple matrix multiplication $v^{(2)} = C * v^{(1)}$ We then see in $v^{(2)}$ what new nodes have to receive the block and will broadcast it the next time step. We can repeat this step: $v^{(3)} = C * v^{(2)}$. We see what nodes know the block at time step three. This is repeated until the vector contains `TRUE` values only. An easy way to find the nodes that have to add a block to their protocol at time n is to calculate the vector $v^{diff} = v^{(n)} - v^{(n-1)}$. To be precise for Boolean values here the minus sign represents the xor operation. $v^{diff}$ has `TRUE` in the entries of the node indices that did not have the block before already. Hence these are the nodes that do know the block now but did not know it before. By setting all the diagonal entries of the $C$ to `TRUE` a `TRUE` position in $v$ will stay `TRUE` after the multiplication with $C$. To calculate the proportion of the network nodes that received a block we simply compute the number of `TRUE` values in the current vector of the block divided by its length $v * I_n/n$ .

**Matrix construction.** The aim of a node in a Bitcoin network is to maintain eight active connections [1]. Putting eight connections for each node to eight randomly chosen nodes with a small number of nodes has the result that the information will be broadcast way too fast to the whole network. There is no tail in the distribution as in Fig. 3.2 at all but the broadcast takes about five time steps for every node. Therefore we leave the need of a node having eight connections with the aim to get closer to the distribution in Fig. 3.2. We also downloaded the data [9] shown in Fig. 3.1 and extracted the data from 2020 and calculated the means of the 50% and 90% propagation delays. That is now about 457ms and 2797ms. We do not want to fix our time step size yet but we also want to get a proportion of around $2797.596/57.204 \approx 6.12$ when we compare the mean of our time steps to reach 50% and 90% of the blocks.

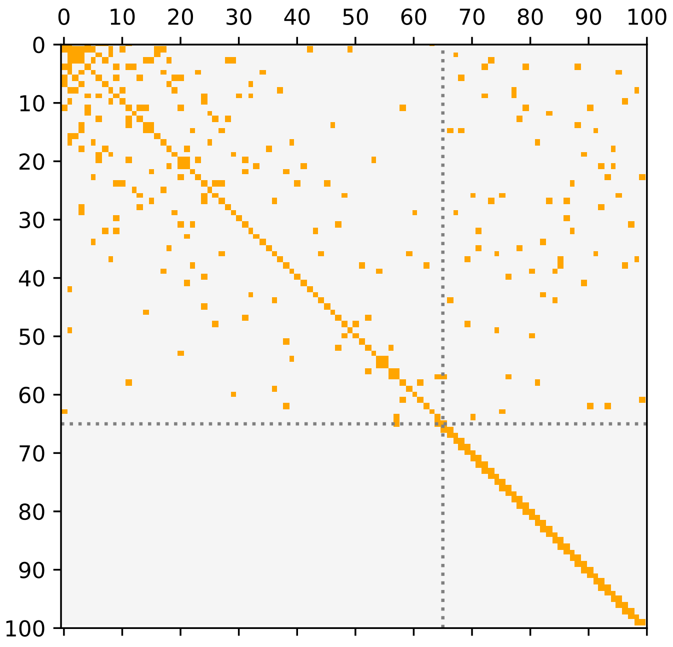

**Figure 3.3.** Connectivity Matrix

The chosen way to approximate a real network behavior is finally done by splitting the matrix into four parts. A cluster that is highly connected that flattens out as shown in the top left of Fig. 3.3. On the bottom right the nodes build a chain and the bottom left of the matrix is empty to simulate the tail of the distribution. A block created by a node with an index below 65 will take many time steps to reach a node with a higher index. On the other hand the top right part contains many connections again to skew the distribution to the right. Without the upper right part the distribution would not have such high values at the beginning as in Fig. 3.2 because many blocks would get lost in the tail. Another approach for this would be to give the nodes with lower connections a lower mining rate. We tried different values for the sizes of the different parts and the density inside the highly connected region. The chosen value for the splits is 66. The average ratio of the 50% and 90% propagation (as in Fig. 3.1) time steps in 10000 different generated matrices is 6.14 and the average number of steps to reach 50% of the network is 4.15 time steps and 25.54time steps for 90% of the nodes. We conclude the real time approximation of our time steps is about $457.204/4.15 \approx 110ms = 0.11s$

A sample distribution of our code to compare with Fig. 3.2 is shown in Fig. 3.4. Since our time step size is 110 ms and the size of a bar in Fig. 3.2 is about 48.78ms we need to flatten our line to compare the two plots. This is shown in the black line below that is scaled by $1/(110/48.78) = 1/2.255$. When we compare our plot with the real data we see that our tail is shorter but the graphs look quite similar otherwise. We try to make simulations with network matrices that have such propagation properties.

### 3.3.3 Nodes

Each node has its protocol and stores its own version of the blockchain. The main purpose of implementing the nodes is that when a node finds a block the block is created with the knowledge of that single node only. The mining behavior and its connections are modeled by the network.

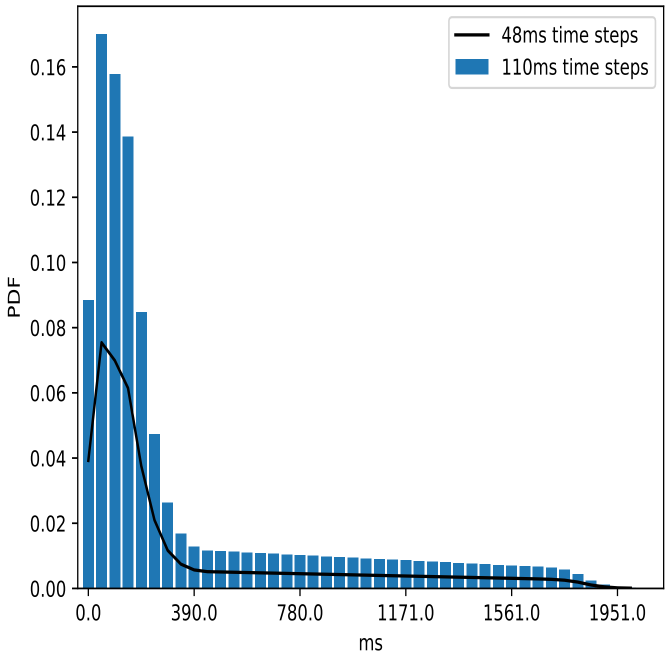

**Figure 3.4.** PDF of our own matrix. The blue bars are the time steps. The black line is adjusted to have the same time intervals as in Fig. 3.2. Since our time step size is 110 ms and the size of a bar in Fig. 3.2 is about 48.78 ms we need to flatten our line by $1/(110/48.78) = 1/2.255$ to compare the two plots.

## 3.4 Mining rate

The mining rate depends on the hash rate of the nodes and the difficulty parameter to find a solution that meets the proof-of-work requirements.

**Approximating the mining process with the Poisson distribution.** Since the number of inputs that can be given to the hash functions is enormous and a solution will is found before even a small part of them is tried out. (e.g., when ten leading zero bits are needed in a string of 30 bits there are $2^{20}$ correct solutions and $2^{30}$ total possibilities). One can assume that each call of the hash function has the same probability $p$ to find a valid hash that satisfies the proof-of-work conditions. Hence this follows a Binomial distribution with probability $p$. Furthermore for many trials and small $p$ the Binomial distribution can be approximated by a Poisson distribution. One can use the Poisson distribution to model the process of mining new blocks in the network. Since this approximation is accurate and the Poisson distribution is much more efficient to compute with. From now on we assume that the block creation process follows the Poisson distribution. For a Poisson distributed random variable the probability to sample the natural number x is given by $P(x) = \frac{e^{-\lambda}\lambda^{x}}{x!}$. $\lambda$ is the parameter for the expected number of outcomes. The Poisson distribution has some nice properties. The sum of multiple Poisson processes is again a Poisson process. For example two processes with an average find rates $\lambda_1$ and $\lambda_2$ is again a Poisson process with the average find rate step of $\lambda_1 + \lambda_2$ and this can be extended to any number of summation. When we simulate the block creation process with a Poisson distribution, then the individual creation time follows a exponential distribution. This distribution is memoryless, meaning for a node to find a block at any time step it does not matter if it found a block in a time step before or not.

**Let $\lambda_N$ be the expected number of blocks mined at any time step by the network.** Assuming that every node has the same mining speed we then for each time step generate a random number for any node according to a Poisson distribution with parameter $\frac{\lambda_N}{\#nodes}$ that is again the expected number of blocks a node finds per time step. From our time step distribution we can now calculate the parameter to plug into the distribution.

**Example $\lambda_N$ for Bitcoin.** We know that for Bitcoin the average time to find a block is $10min = 600$s. If we want to compare our simulation with the current Bitcoin protocol our expected number of blocks per time step is $0.11s/600s = 0.000183$. Denote that we do not care how big the real hash rate of the Bitcoin network is. All we want is a similar block creation process.

**Implementation inside the code.** Our first approach to run the simulation is to generate an array where each entry is the number of blocks found for one node in a single time step. This results in an array where the number of entries is the number of nodes and the number of columns is the number of time steps.

```
def run(timesteps,findrate):
    #100 nodes
    poissonVariables = random.poisson(findrate/100,
    (100, timesteps))

    for row in poissonVariables:
        updateAllNodes()
        for number, index in row:
            nodes[index].findblock(number)
```

This works well for Poisson parameters that are not too small. To find 10000 blocks for a find rate parameter as the one we use to compare Bitcoin we need to compute an array that contains about $10000/0.00183*100$ random generated numbers. The expected number of each entry is $1/100*0.00183$. Iterating and creating this array is not negligible and slows down the simulation.

Improvement for small find rates is given in the code below.

```
def run(timesteps,findrate):
    p_0 = stats.poisson.pmf(0, findrate)
    poissonVariables  = random.random(timesteps) > p_0

    for entry in poissonVariables:
        updateAllNodes()
        if entry:
            for node  in random.multinomial(truncated_poisson(),
                  nodes):
                node.findblock(1)
```

$p_0$ stands for the probability of a Poisson variable being zero given the overall find rate. This array now indicates if at a time step at least one miner mines a new block. If the entry is `TRUE` it indicates the random event that at least one block is found. We now model this event as another random variable modeled by a so called zero truncated distribution. It represents the conditional Poisson distribution $P(X|X > 0)$ [10]. It makes sense to use this random variable now since we know that we found at least one block. In the end we get the same distribution as we would using one single Poisson distribution. Note that then the probability for an entry to be any natural number $k$ is either $p_0$ of the normal Poisson distribution if $k$ is zero. Otherwise we get the probability of the random variable not being zero times the conditional probability, i.e., $P(X > 0)\,P(X = k|X > 0) = P(X = k)$.

The final result is sampled from a multinomial distribution where each node is chosen with the same probability. We can do this since we assume all of them have the same mining rate. One may think that a node needs time to recalculate the block it wants to extend and we may not use this property here. But

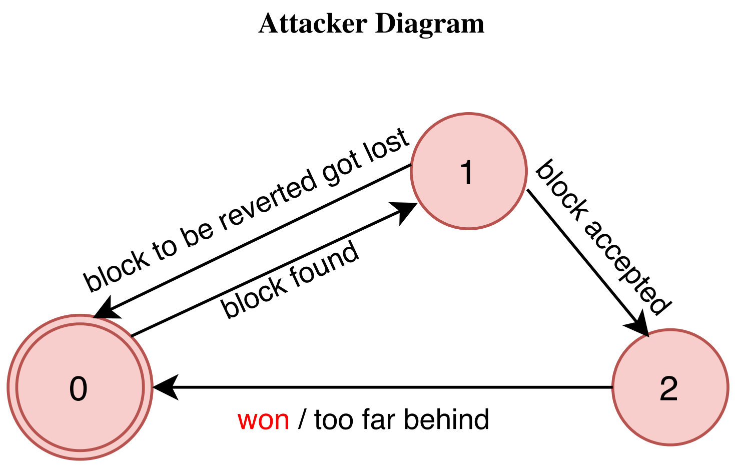

**Figure 3.5.** The implementation of the attacker protocol is done by using three stages in which the attacker can be. In all stages the attacker is mining blocks It starts in **Stage_0** and is mining for a block it then wants to revert. Once it finds the block it sends it to the network and gets to **Stage_1** . While in **Stage_1** it mines on a hidden chain and waits for its block to be accepted by the network (e.g., for Bitcoin having six blocks in a chain after it) once the block is accepted he switches to **Stage_2** . There it tries to revert the block by catching up with its own chain. If the attacker wins by reverting the block or if he is too far behind the main chain and surrenders it gets back to **Stage_0** .

when a node finds a block it instantly would continue mining on the same block it just found. With this method we need to do much fewer computations since if no block is found in a time step only a single function call for a random variable is needed.

## 3.5 Attacker

We give advantage to the attacker by assuming that the attacker has complete knowledge over the network. As opposed to normal nodes, the attacker class has a reference to the network object. The attacker has no own graph but uses the network graph containing every block that exists. When it wants to commit an own block or forward another block it uses the function of the network that sends the block to all nodes instantly. Fig. 3.5 shows how we implemented the attacker as a finite automaton.

Chapter 4

# Simulation

In this chapter, we present some results from our simulations with different parameters on different protocols. We start with the simulations of some blockchains with the Bitcoin and GHOST protocol. We look at the number of forks in Bitcoin and GHOST and compare the expected number of forks in the real Bitcoin network. In Section 4.2 we simulate some attacks of Bitcoin GHOST and Conflux under different parameters. In Section 4.3 we give an example how GHOSTDAG creates its ordered list under a attack and then simulate some additional attacks. Finally because the GHOSTDAG algorithm is not trivial, we want to compare the ordered list of GHOSTDAG with the epochs generated by Conflux.

## 4.1 Forks in Bitcoin and GHOST

Fig. 4.1 shows plotted graphs of some sample runs with 100 Blocks mined following the Bitcoin protocol in each graph. Each point represents a block. The red points are forked blocks. There are four samples for each different find rate shown in the same row. The genesis block is always the block at the bottom. This figure illustrates the impact of the randomness of the model. In the first row we do not have any fork in three of the four samples. These graphs consist of a straight line due to the way the blocks are plotted. The only fork produced in a network with find rate 0.01 is in the second sample. We see a trend of having more forks when the mining rate is increased when comparing between all the graphs. In the third row in sample 10 there are exactly as many forks as in sample 2. Even though the find rate is four times higher. Sample 17 is created with a mining rate four times bigger than the one from sample 11 but contains fewer forks. This illustrates the effect of randomness.

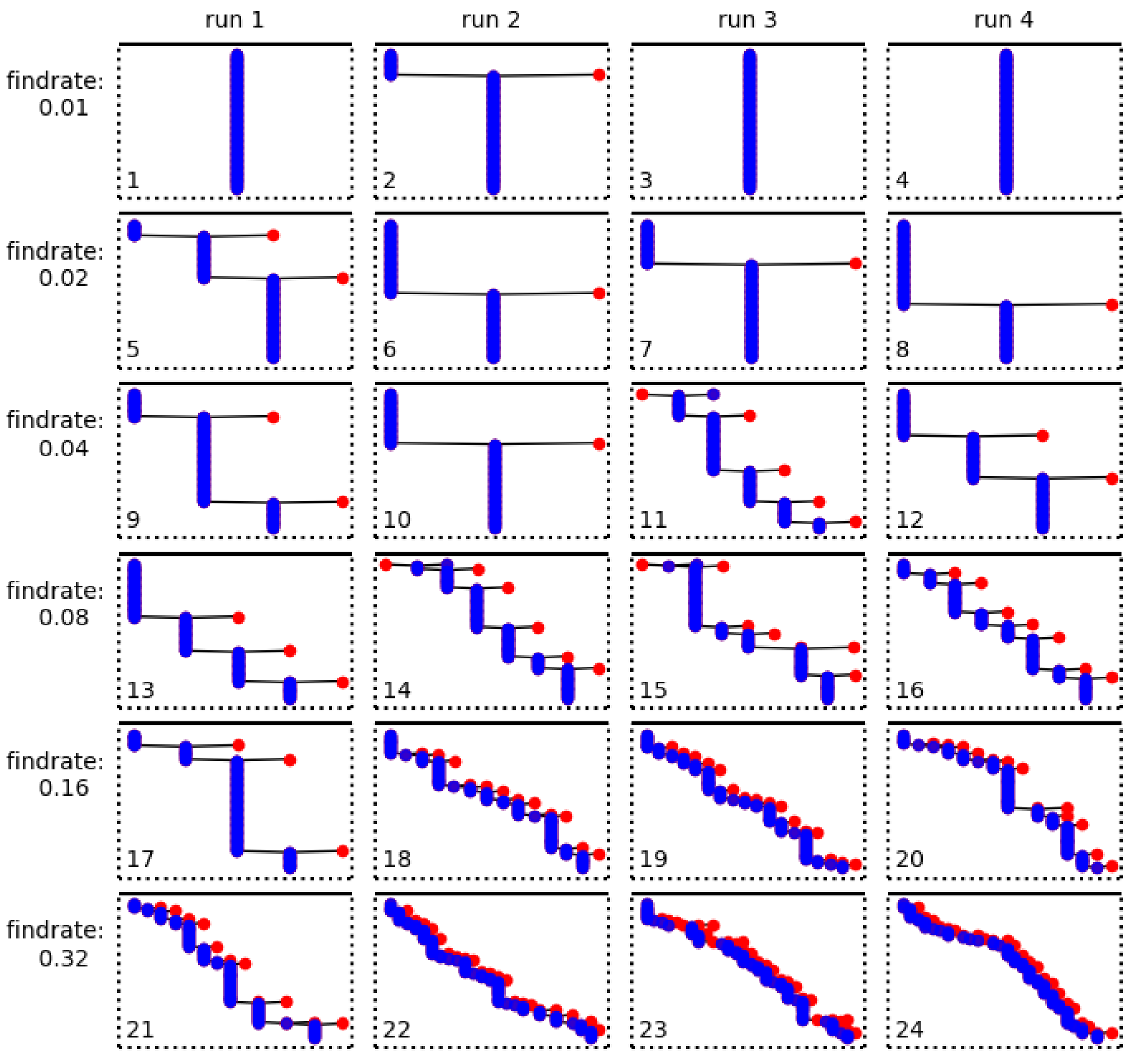

**Figure 4.1.** Sample executions of 100 Blocks mined with different find rates.

In Fig. 4.2 we now see the average number of forks of 20 simulations with 2000 blocks each. For each of the 20 simulations another network is created. We see that doubling the find rate does not directly double the number of forks. We also see that the GHOST and Bitcoin protocol have about the same number of forks as it should be.

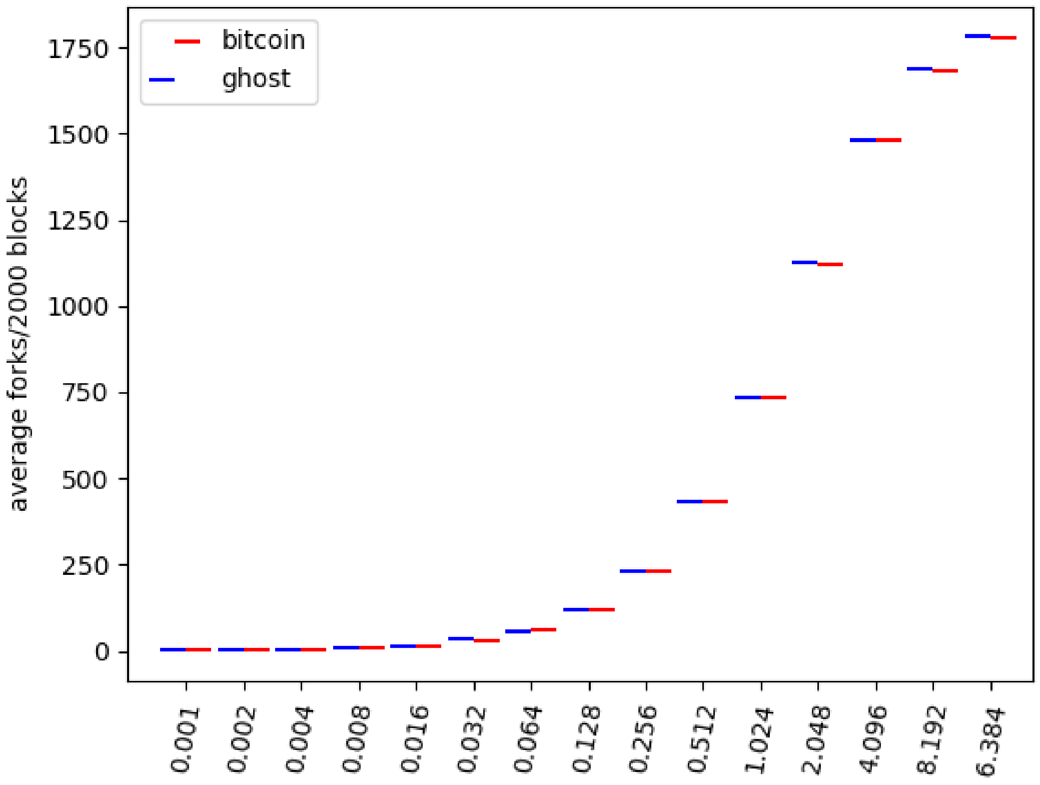

**Figure 4.2.** Bitcoin and GHOST average number of forks in 2000 mined blocks with exponentially increased find rates.

**Analyzing results with the estimated parameter for Bitcoin.** As computed in Section 3.4 the find rate parameter for Bitcoin in our simulation is 0.000183. When running our simulation with the Poisson parameter of 0.000183 as the expected number of blocks mined per time step. We get about **two forks per 1000 blocks** mined. If we compare this with our data from the real Bitcoin network we see that no forks did occur at all in the past half year. We want to check if the number of forks we get is plausible with the theoretical model assumptions our simulation is based on. We let each block save the time step it is created and then compute the time intervals between the different blocks. A simulation with 10000 blocks in Bitcoin and expected find rate per time step of 0.000183 results in an average time interval of 5478 time steps. With our assumption that a time step is 0.11s we indeed get the desired average 10 minutes ($5378 * 0.11s = 602s \approx 10min$). Important for us are the time intervals that are small enough to for forks to happen. If we look at the 10 smallest intervals: $(1, 1, 2, 2, 2, 2, 2, 3, 4, 5)$ we indeed see that it is really likely that a fork may occur in such a time interval. Assuming that the block creations follow a Poisson distribution the probability of finding no block in T time steps is $e^{-\lambda T}$. This implies the probability to find at least one block in a time interval $T$ is $f(T) = 1 - e^{-\lambda T}$.

Table 4.1 summarizes the probabilities of finding a block in a given time interval $T$ with $\lambda = 0.000183$. We see that with a probability of 0.00018 blocks are found in successive time steps. In this case a fork is really likely to happen. With probability 0.00073 there are two blocks within four time steps. As four time steps is the time to reach about 50% of the network on average, this implies that we

**Table 4.1**

| **T** | **1** | 4 | 26 | 43 |
|---|---|---|---|---|
| $f(T)$ | 0.00018 | 0.00073 | 0.0047 | 0.007838 |

have a $50\%$ chance to get a fork. Considering this event alone we realize that when mining 10000 blocks we still expect some forks to occur (i.e., $> (10000 - 1) * 0.00073 * 1/2 = 3.649635$). This is a lower bound only. We see that theoretically forks are still possible to occur even when the mining rates is really low. For $T = 26$ and 43 we get the times to reach 90% and 99% of the network. We still get a really small number of forks and since we prefer our simulation network to perform worse rather than better as the actual network.

We know that in Bitcoin at least six blocks have to follow a block in the main chain such that the transactions inside the block are considered to be safe. If every node would follow the protocol this confirmation time would be too high with the number of forks we observe.

## 4.2 Double-spending attack on Bitcoin compared to GHOST and Conflux

On the two following Figures 4.3 and 4.4 we see the average of 500 runs per find rate and attack power on Bitcoin and GHOST. Since Conflux inherits its security directly from GHOST it also holds for Conflux. $\lambda$ denotes average blocks found per time step over the honest nodes for all future graphics. The reason we do not consider the attacker is that as long it keeps its mining power for itself the difficulty to mine a block will only adjust on the honest nodes. An attacking power of 50 means that it has the same hash rate as the network itself (i.e., the attacker owns 50% of the overall hash rate) and this scales linearly for the other numbers. As we see in the pictures the attack has almost a 100% chance to win if it has a power of 50 meaning we probably highly favour the attacker in terms of it not giving up until being 40 blocks behind the main chain of the network. The aim of this simulation is not the make conclusions about the real world situation but about the comparison between the protocols themselves. We clearly see that with the Bitcoin protocol higher find rates and therefore more forks have a big impact on the attacker's success rate. Conversely in GHOST and Conflux the different success rates seem due to randomness and all have the same trend.

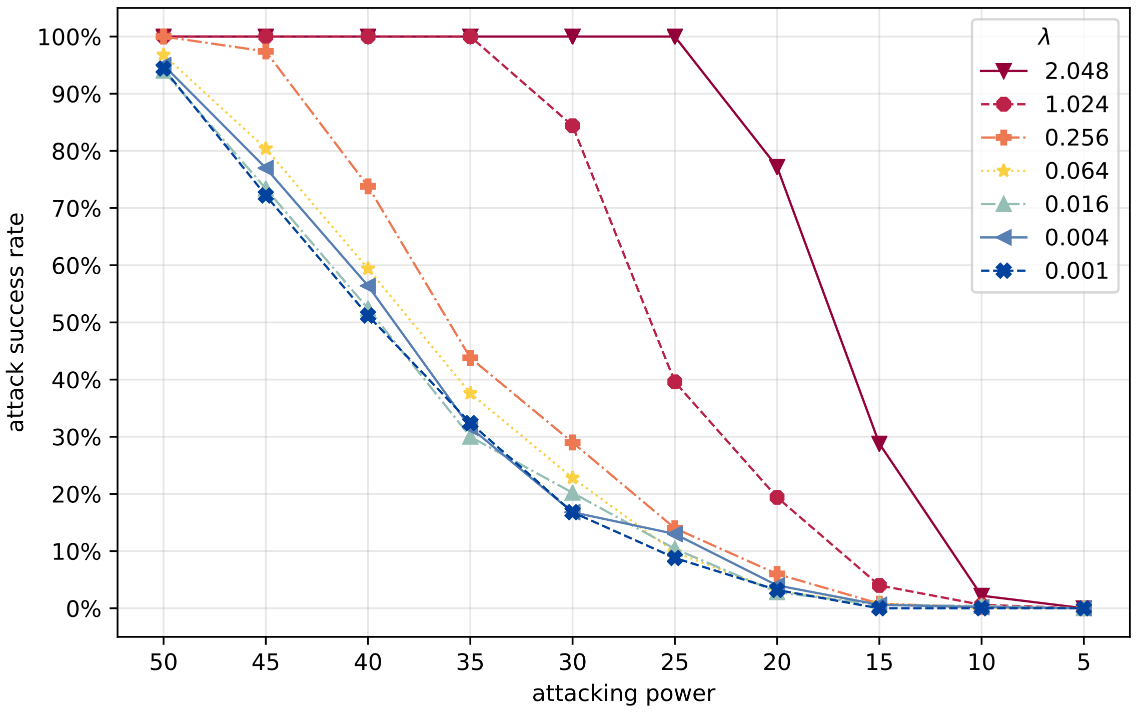

**Figure 4.3.** Average over 500 attacks on the Bitcoin protocol with six blocks confirmation.

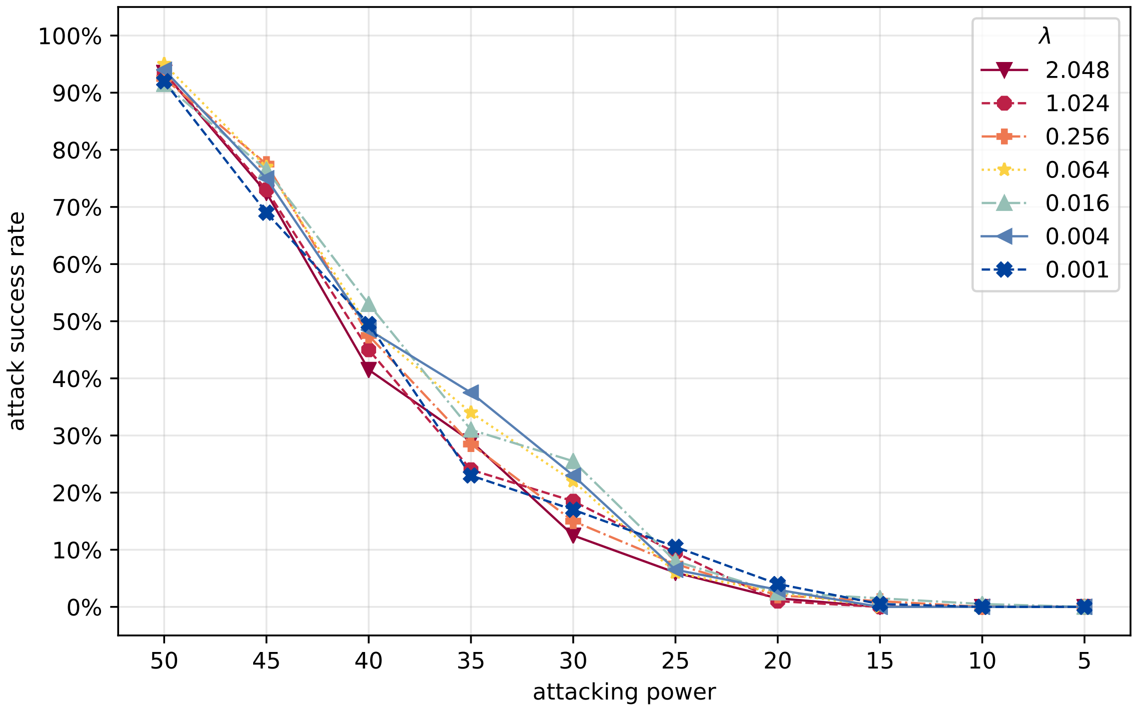

**Figure 4.4.** Average over 500 attacks on the GHOST and Conflux protocol with six blocks confirmation.

We now see different parameters of the expected blocks all of them with the rule that a block is accepted after having six blocks in a chain on top of it and being in the main chain itself. The idea is that with more following blocks needed, the attacking success rate will secure the protocol because the chance of a weak attacker surpassing the honest chain gets smaller. This again increases the confirmation time but we do not need a block to be accepted within some milliseconds. For example with the exaggerated block rate of 2.048 per time step and one time step being equal to 0.11s we would expect to find 18.6 blocks per second. In Fig. 4.5 and Fig. 4.6 we simulate the Bitcoin protocol with more following blocks needed until a block is accepted. In Fig. 4.5 with 20 blocks are needed as confirmation and in Fig. 4.6 100 blocks. In Fig. 4.7 and Fig. 4.8 we simulate the same for GHOST and Conflux. As expected we see that the attacker's chance decreases when the confirmation of blocks is increased.

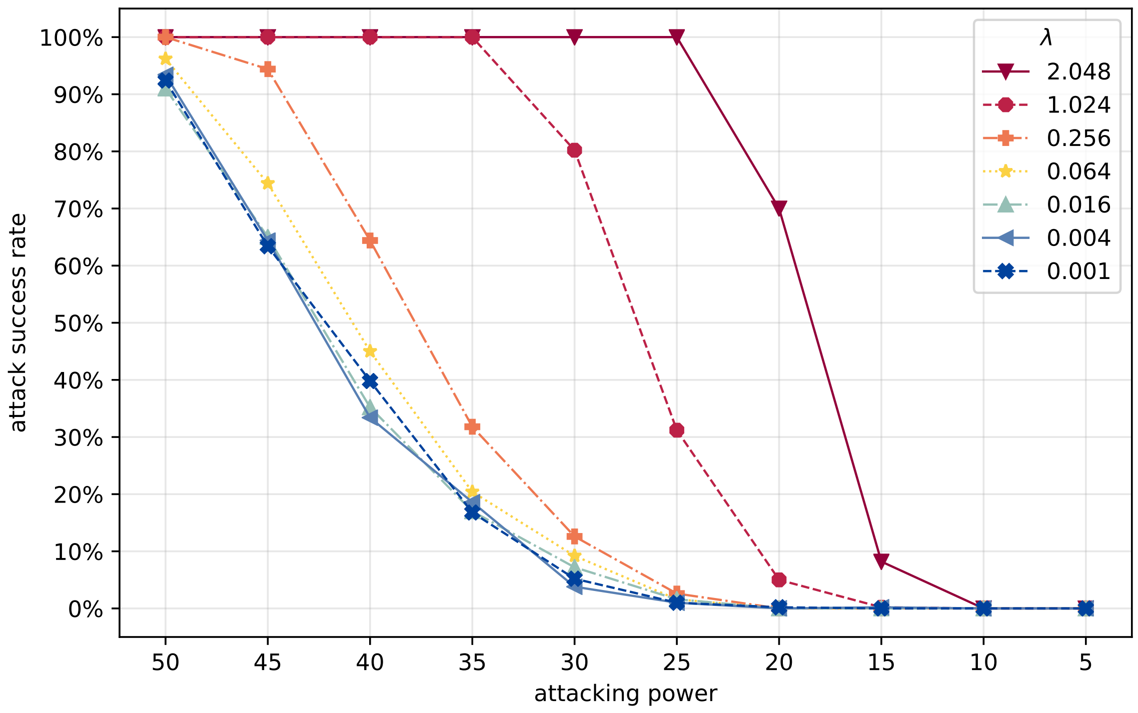

**Figure 4.5.** Bitcoin protocol with 20 blocks confirmation.

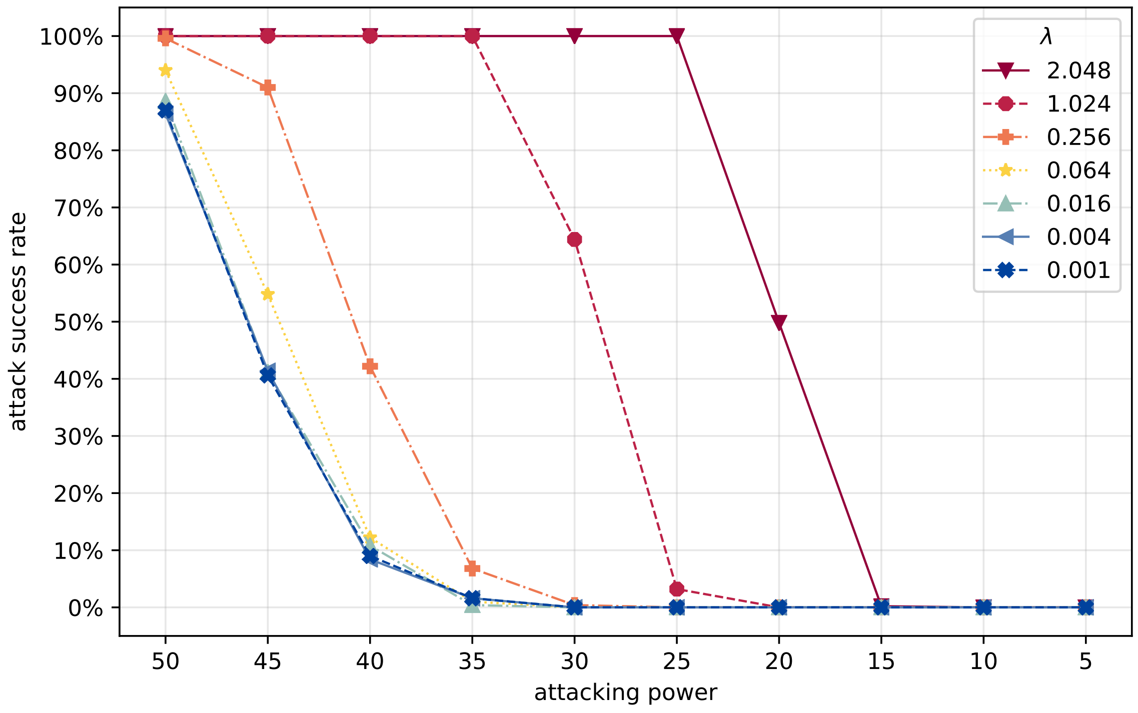

**Figure 4.6.** Bitcoin protocol with 100 blocks confirmation.

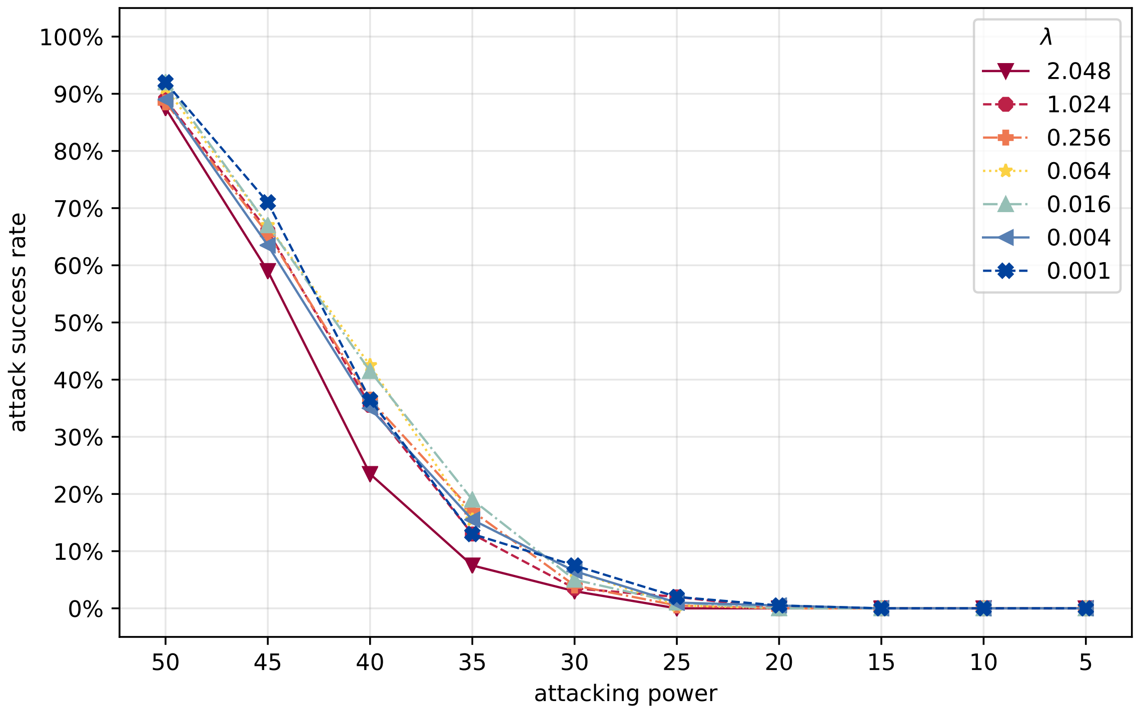

**Figure 4.7.** GHOST protocol with 20 blocks confirmation.

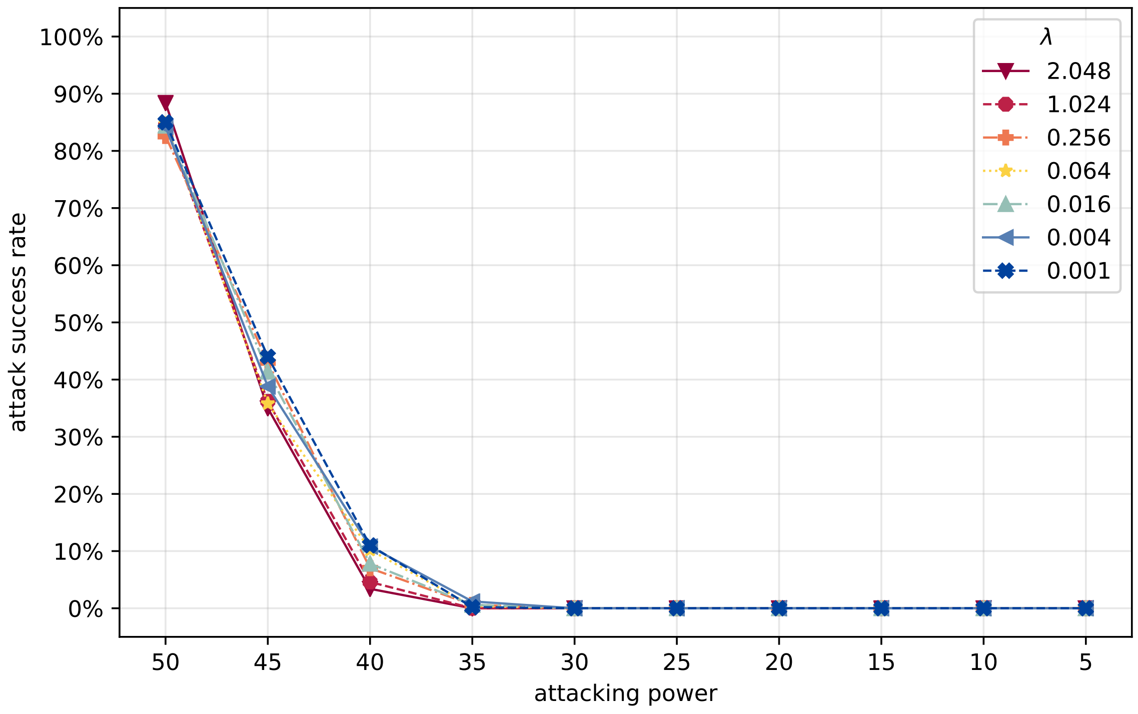

**Figure 4.8.** GHOST protocol with 100 blocks confirmation.

## 4.3 GHOSTDAG Analysis

**Attacking GHOSTDAG.** With our idea given in Subsection 2.3.4 we try to look at how well attacks are performed on the GHOSTDAG protocol. Keep in mind that this is a really naive attack since it just follows the simplest rules as in Bitcoin. We now have even more parameters to vary for our simulation. Beside the find rate and the number of blocks needed on top of a block to be accepted we also have the $k$-cluster parameter $k$. We first want to give an additional example that shows how the DAG is interpreted by the protocol with an attack inside of it. The DAG with the attacker is shown in Fig. 4.9. In Fig. 4.10 we see the same DAG but this time its blue set is shown and we only see the arrows that connect the ordered list together. In the upper graph we see that the attacker wanted to revert block 9. Its first block is block 10 and references block 8, trying to skip block 9. In the second plot we observe that actually all blocks of the attacker are in the last positions in the list and block 9 still belongs to the blue set.

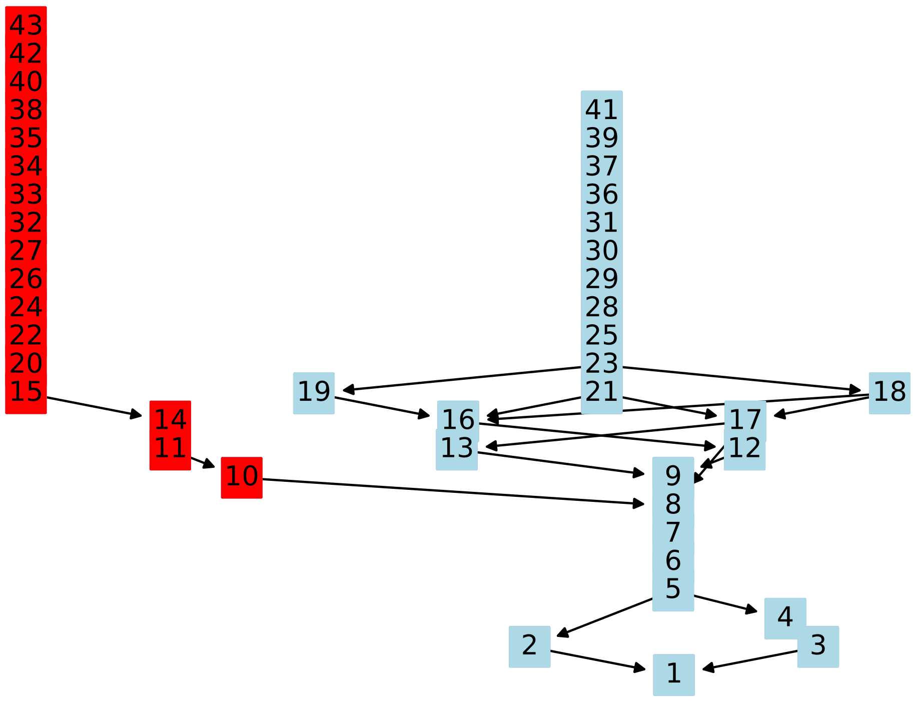

**Figure 4.9.** DAG created with the miners following the GHOSTDAG protocol. The red blocks come from an attacker that wants to revert block 9.

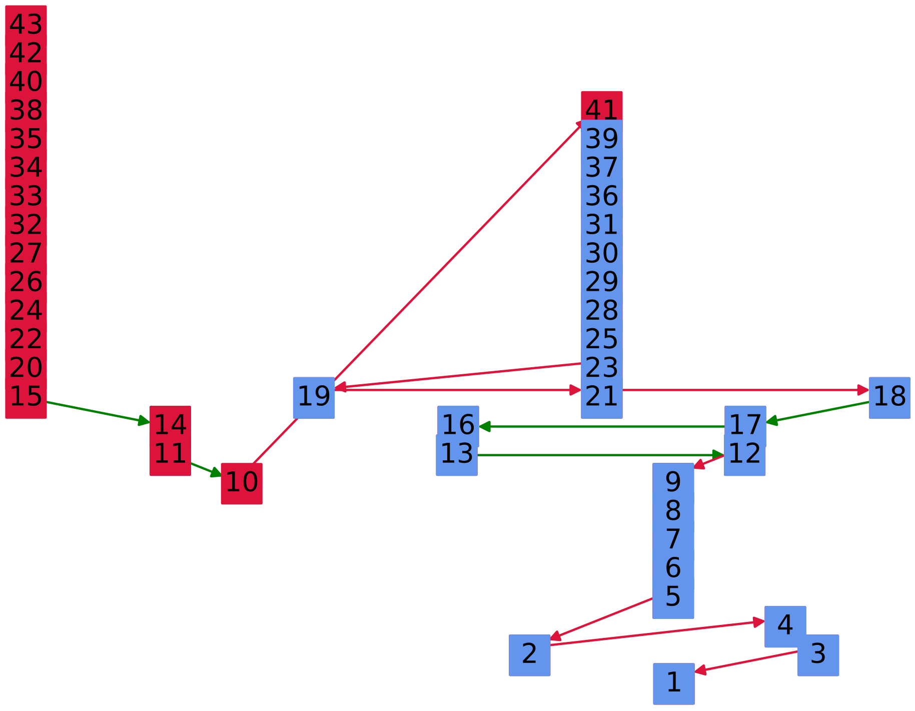

**Figure 4.10.** The same DAG as in Fig. 4.9, showing the blue set and how the ordered list is linked together. The parameter for the $k$-cluster is set to three. The green arrows mean that the two blocks followed one another in the creation as well and the red arrows mean some step is made that is not the same as the order the blocks are created. The actual list is [1, 3, 4, 2, 5, 6, 7, 8, 9, 12, 13, 16, 17, 18, 21, 19, 23, 25, 28, 29, 30, 31, 36, 37, 39, 41, 10, 11, 14, 15, 20, 22, 24, 26, 27, 32, 33, 34, 35, 38, 40, 42, 43].

In our first simulation we check the claim from [4] that when $k$ is set to zero GHOSTDAG should have the same results as Bitcoin. Hence Fig. 4.11 is run with $k$ equal to zero and confirmation blocks needed equals to six. Indeed we see the results is similar to Bitcoin but actually performing even a bit better since the attacker's success rate is dropping a bit earlier than in Fig. 4.3. The next Fig. 4.12 has a $k$-cluster parameter of two and also six blocks confirmation. We see a steady decrease of the attacker's chance of success. Fig. 4.13 shows the results when the $k$-cluster parameter is set to ten. We still see a small decrease of the attacker's chance.

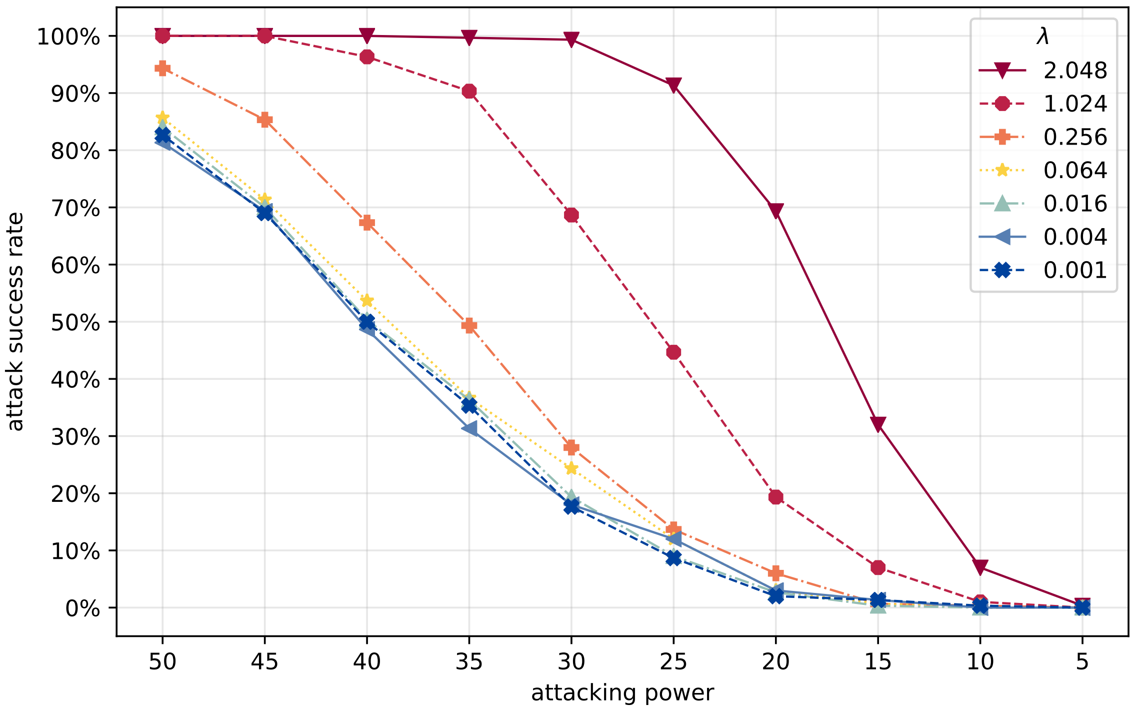

**Figure 4.11.** Attacks on GHOSTDAG with $k$-cluster size of zero and six blocks as confirmation needed.

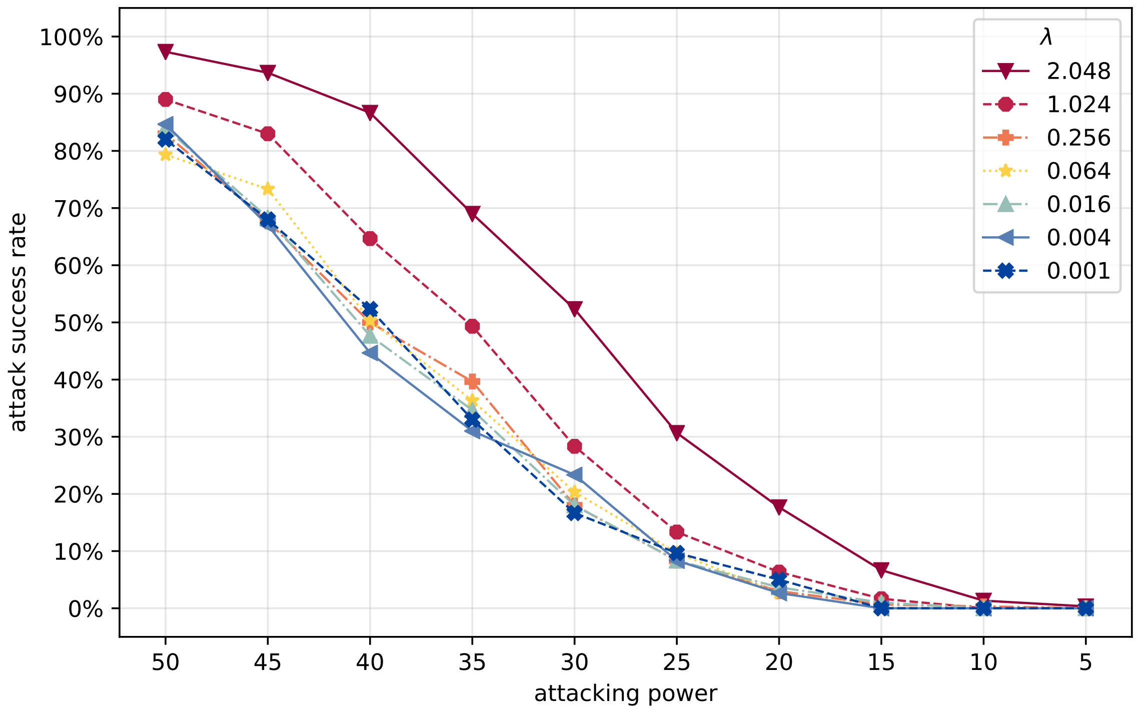

**Figure 4.12.** Attacks on GHOSTDAG with $k$-cluster size of two and six blocks as confirmation needed.

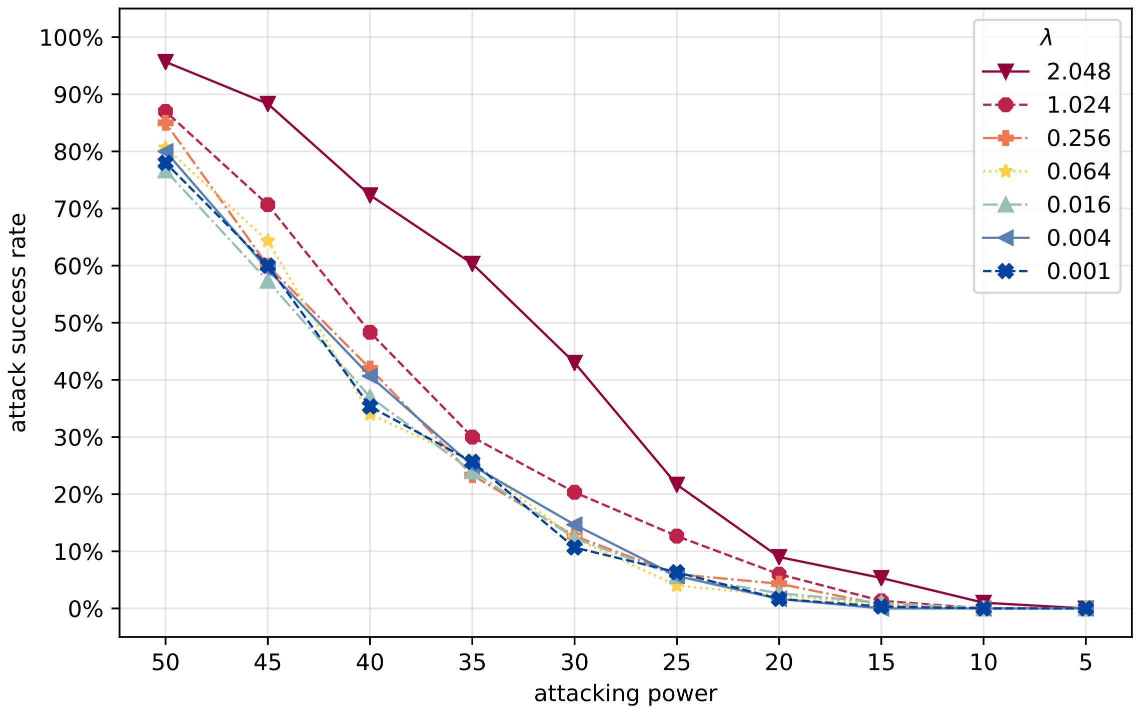

**Figure 4.13.** Attacks on GHOSTDAG with $k$-cluster size of ten and six blocks as confirmation needed.

We want to see the effect when we increase the blocks that we have to wait for to declare that a block is accepted with our attacker oh GHOSTDAG. Fig. 4.14 shows a $k$-cluster size of three and 15 following blocks needed for a block to be valid. In Fig. 4.15 we keep the 15 blocks needed as confirmation and set the $k$-cluster size to ten. We can compare this plot with Fig. 4.14 where the $k$-cluster size is smaller and with Fig. 4.13 where there are fewer blocks needed as confirmation. We do not see a big difference between Fig. 4.14 and Fig. 4.15. The points in both plots are a bit lower than in Fig. 4.13 for the highest find rate. The plot in fig. 4.16 has the same $k$-cluster size as the plot in Fig. 4.14 but with 30 blocks needed as confirmation. We see that our attacker performs terrible when that many blocks are needed for confirmation.

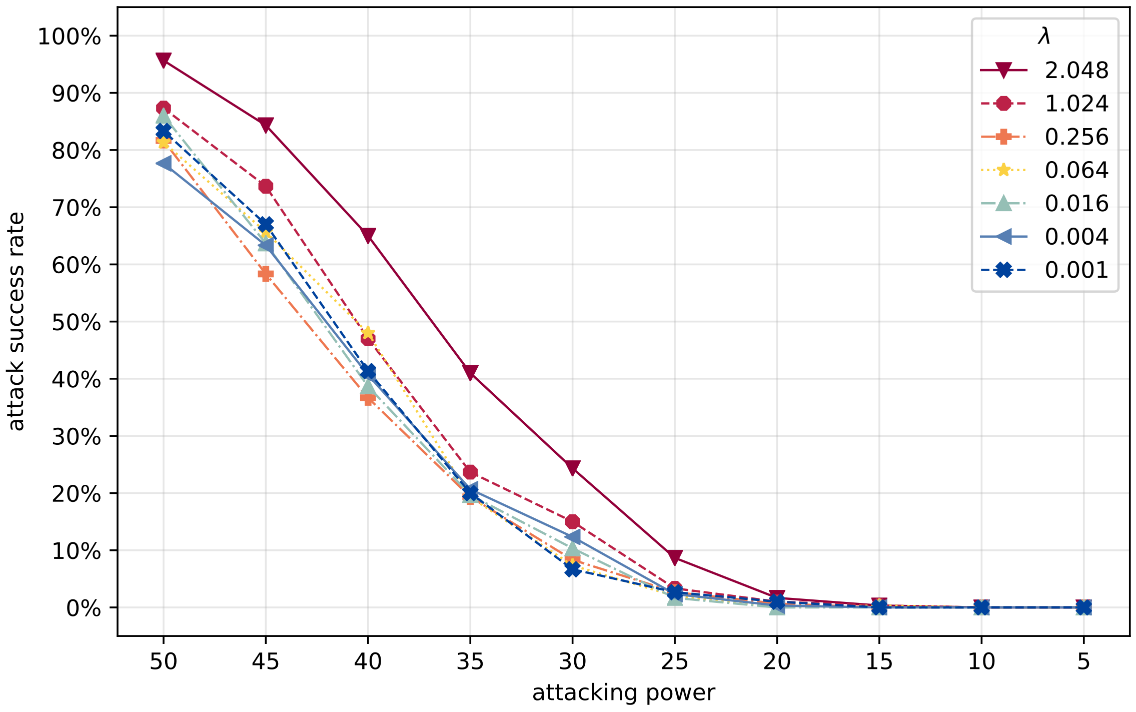

**Figure 4.14.** Attacks on GHOSTDAG with $k$-cluster size of three and 15 blocks as confirmation needed.

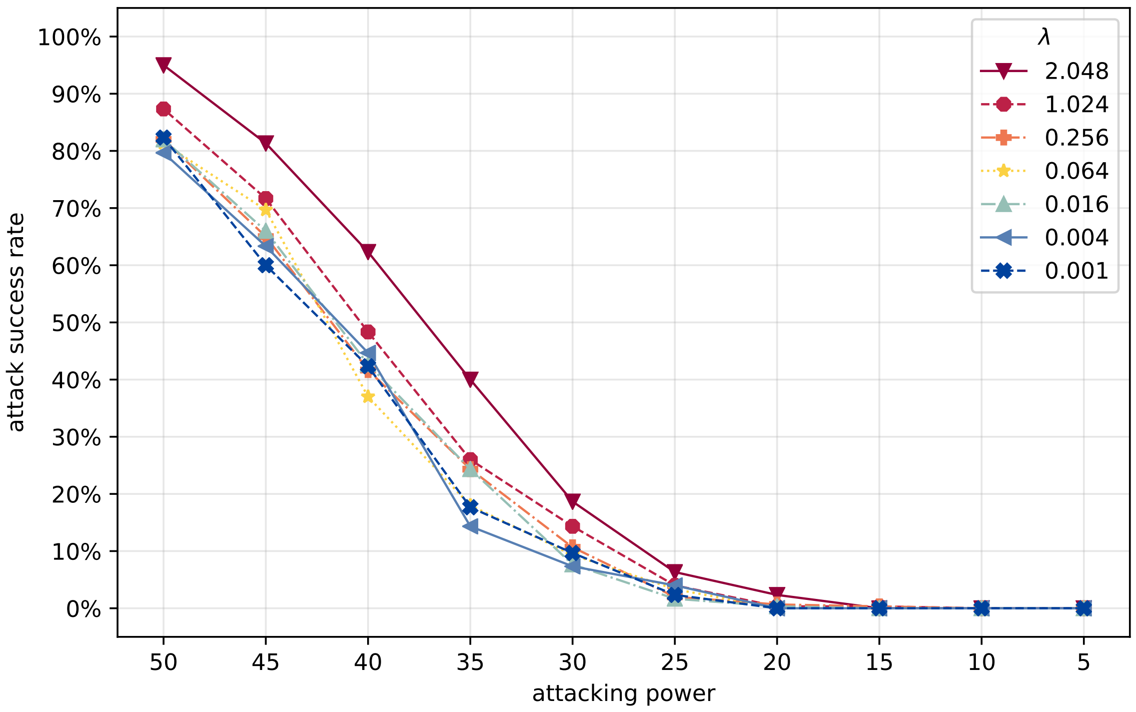

**Figure 4.15.** Attacks on GHOSTDAG with $k$-cluster size of ten and 15 blocks as confirmation needed.

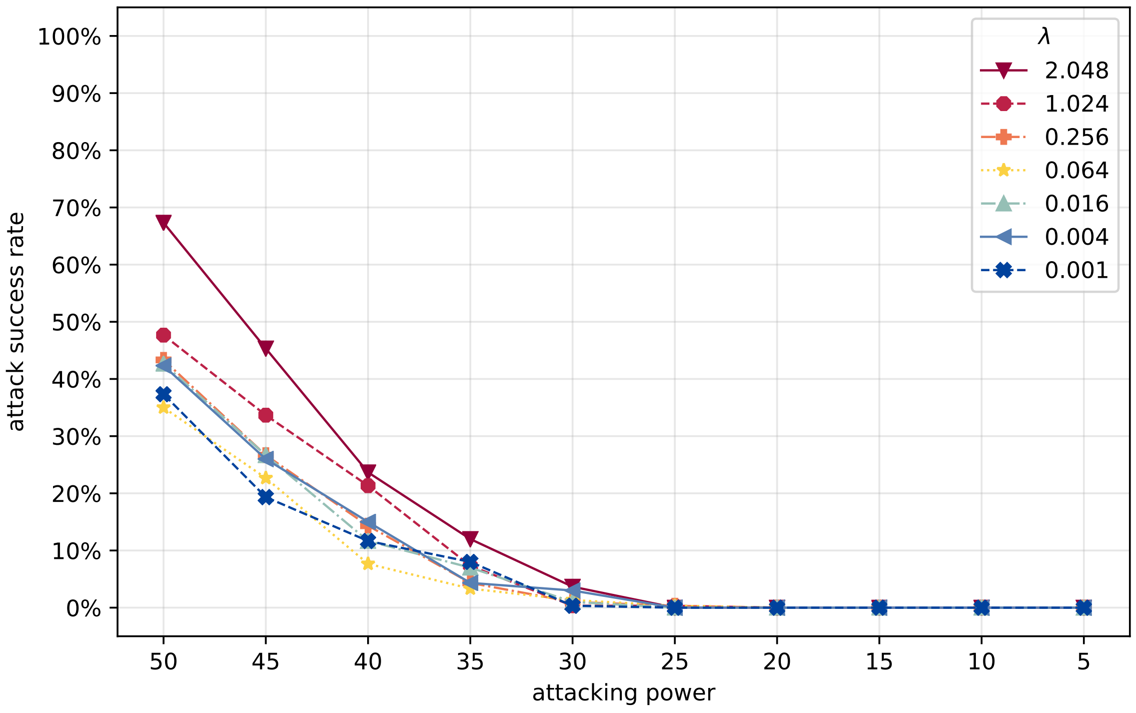

**Figure 4.16.** Attacks on GHOSTDAG with $k$-cluster size of three and 30 blocks as confirmation needed.

**Comparing GHOSTDAG and Conflux.** GHOSTDAG and Conflux have completely different algorithms to achieve their ordering of the blocks. That is why it might be interesting to see how the resulting orderings differ under normal conditions (without attacker). The idea is to check whenever blocks that are inside epochs that are one after the other also all are one after the other in the total order generated by GHOSTDAG. Note that we do not really care how blocks inside an epoch are ordered. We create two networks one that follows the Conflux protocol and one that follows the GHOSTDAG protocol. Both of them have the same number of nodes and the same connections between them. We then always randomly pick an index and let the node with that index find a block in both networks. We then look at both generated blocks as if they were the same and only generated by a different protocol. Fig. 4.17 shows the block in each epoch and their index in the GHOSTDAG main chain. As we see when the epoch index increases the indexes of the blocks positions in the epochs almost always increase as well and the index ranges do not overlap. That means the ordering over the block is almost the same. In epochs 9 and 10 or 29 and 30 we clearly have blocks that are in a different order. But generally the two protocols give really similar results. Many sample runs had no overlaps at all and. In that graph we also can also see the number of forks is about two by looking at the slope of the line.

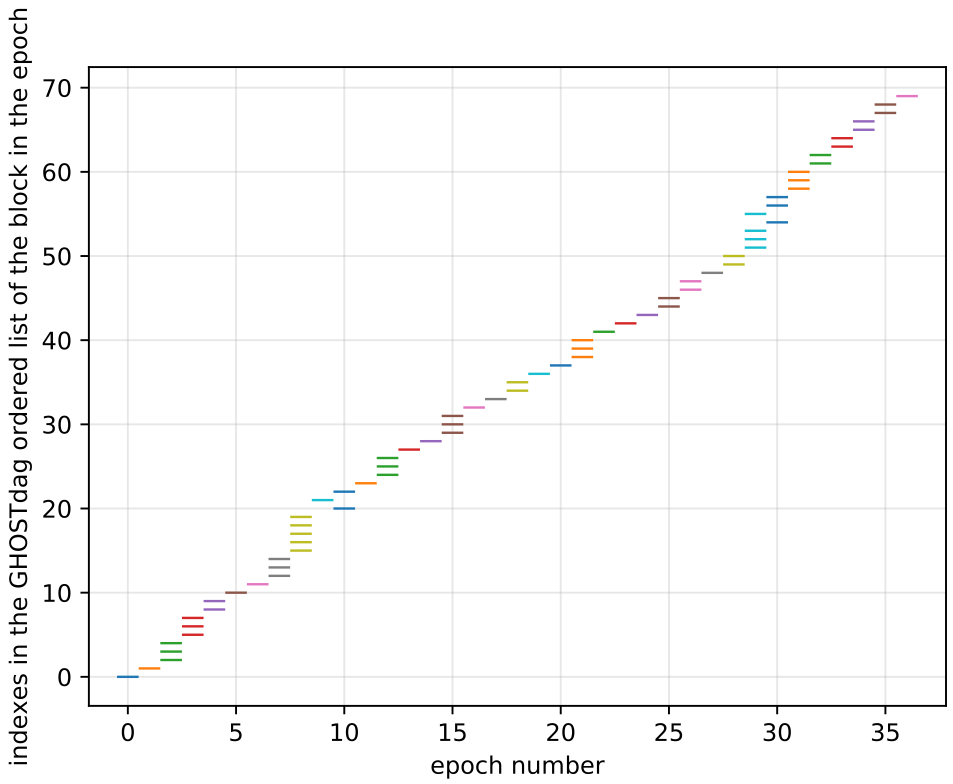

**Figure 4.17.** Comparison of blocks by their epoch in Conflux and their index in the GHOSTDAG ordered list. The cluster parameter $k$ of GHOSTDAG is set to two and the find rate to 1.5.

# Chapter 5

# Conclusion

The GHOST protocol introduces an evaluation of a blockchain that is more secure against double-spending attacks compared to Bitcoin if more forks occur. Conflux and GHOSTDAG extend this scalability by including transactions of all blocks that are mined. First we summarized the concepts of a distributed blockchain that are of interest to our work. We explained how forks are created and what problems they present for a blockchain protocol. We explained how forks interfere with Bitcoin and how the GHOST, Conflux and GHOSTDAG protocols aim to improve the interference with forks. During the process of this thesis a small network with 100 nodes was constructed. The block transmitting time distribution was observed and compared to the real Bitcoin network. By altering the connections between our nodes it was attempted to minimize the difference between the two. For the implementation of the protocols we used dynamic programming, whenever possible, to be able to run longer simulations.

The simulation gave some insight on how the four protocols are similar and where their differences lie. The attacker for Bitcoin, GHOST and Conflux together with the network we simulated on was quite strong but we saw desired trends when varying our parameters. A few claims that were made by the authors of the protocols could be verified; for example that the safety of protocols should converge when the find rate gets lower or the following number of blocks needed for validation increases. The attacker we wrote for GHOSTDAG was not intelligent enough in regard to the complexity of the protocol. Its intelligence is thus one aspect that we can improve on. A further improvement could be the size and complexity of the network to make more general conclusions about the protocols.

.

# Bibliography

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[2] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[3] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:1805.03870*, 2018.

[4] Yonatan Sompolinsky and Aviv Zohar. Phantom, ghostdag. 2018.

[5] Pauli Virtanen, Ralf Gommers, and Oliphant, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[6] *numpy*, 2020 (accessed June 20, 2020). `numpy.org`.

[7] *networkx*, 2020 (accessed June 20, 2020). `https://networkx.github.io/documentation/stable/`.

[8] *bitcoin monitoring*, 2020 (accessed June 12, 2020). `https://dsn.tm.kit.edu/bitcoin/`.

[9] *monitored bitcoin data*, 2020 (accessed June 12, 2020). `https://dsn.tm.kit.edu/bitcoin/data/invstat.gd`.

[10] A. Clifford Cohen. Estimating the parameter in a conditional poisson distribution. *Biometrics*, 16(2):203–211, 1960.

# Erklärung

*Erklärung gemäss Art. 30 RSL Phil.-nat. 18*

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

| | |
|---|---|
| 22.12.2020 | [illegible] |
| Ort/Datum | Unterschrift |