FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Simulation of different selfish mining strategies in Bitcoin

Simulation respecting network topology and reference implementation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Simon Mulser, BSc
Matrikelnummer 01027478

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Edgar Weippl, Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn.
Mitwirkung: Aljosha Judmayer, Univ.Lektor Dipl.-Ing.

Wien, 13. Oktober 2017

Simon Mulser	Edgar Weippl

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Simulation of different selfish mining strategies in Bitcoin

Simulation respecting network topology and reference implementation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Simon Mulser, BSc
Registration Number 01027478

to the Faculty of Informatics

at the TU Wien

Advisor: Edgar Weippl, Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn.
Assistance: Aljosha Judmayer, Univ.Lektor Dipl.-Ing.

Vienna, 13th October, 2017

Simon Mulser

Edgar Weippl

Erklärung zur Verfassung der Arbeit

Simon Mulser, BSc
Dadlergasse 18/1/7, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. Oktober 2017

Simon Mulser

Danksagung

Meine Danksagung. Coming soon...

Acknowledgements

My acks. Coming soon...

Kurzfassung

Meine Kurzfassung. Coming soon...

Abstract

My abstract. Coming soon...

Contents

CHAPTER 1

Introduction

The cryptocurrency Bitcoin started back in the year 2008 with the release of the Bitcoin white paper [Nak08]. As of today, the cryptocurrency has reached a market capitalization of over 20 billion dollars [Mar]. Internally the Bitcoin cryptocurrency records all transactions in a public ledger called *blockchain*. The blockchain is basically an immutable linked list of blocks where a block contains multiple transactions of the cryptocurrency. In Bitcoin, each block needs to contain a so-called proof of work (PoW) which is the solution to a costly and time-consuming cryptographic puzzle. Miners connected in a peer-to-peer network compete with their computation power to find solutions to the puzzle and hence to find the next block for the blockchain. Finding a block allows the miners to add a transaction to the block and gives them right to newly create a certain amount of bitcoins. Additionally, the grouping of the transactions in blocks creates a total order and hence makes it possible to prevent double-spending. After a block is found by a miner, all other miners should adopt to this new tip of the chain and try to find a new block on top. This mining process is considered as incentive compatible as long as no single miner has more than 50% of the total computation power.

[ES14] showed that also miners under 50% have an incentive to not follow the protocol as described depending on their connectivity and share of computation power in the peer-to-peer network. By implementing a so-called selfish mining strategy a miner can obtain relatively more revenue than his actual proportion of computational power in the network. In general, the miner simply does not share found blocks with the others and secretly mines on his own chain. If his chain is longer then the public chain, he is able to overwrite all blocks found by the honest miners. If the two chains have the same length the private miner also publishes his block and causes a block race. Now the network is split into two parts where one part is mining on the public tip and the other part is mining on the now public-private tip. In general, the selfish miner achieves that the other miners are wasting their computational power on blocks which will not end up in the longest chain.

Further research [NKMS16, SSZ16, GRK15, GKW$^+$16, Bah13] explored different modifications of the original selfish mining algorithm by [ES14] and found slightly modifications of the algorithm which perform better under certain circumstances. For example, it could make sense for the selfish miner to even trail behind the public chain.

To prove the existence and attributes of selfish mining different approaches were applied. The researchers used simple probabilistic arguments [ES14, Bah13], numeric simulation of paths with state machines [GRK15, NKMS16], advanced Markov Decision Processes (MDP) [SSZ16, GKW$^+$16] or gave results of closed-source simulations [ES14, SSZ16]. Unfortunately, we cannot discuss the closed source simulations in detail. All other above-mentioned methodologies have the following drawbacks:

- Abstraction of the Bitcoin source code which normally runs on a single node. Since there is no official specification of the Bitcoin protocol it is hard to capture all details. Furthermore, it is hard to keep the simulation software up-to-date because of the ongoing development of the protocol.
- Abstraction of the whole network layer of the peer-to-peer network. The available simulations abstract the network topology by either defining a single connectivity parameter [ES14, Bah13, NKMS16, SSZ16, GRK15] or by using the block stale rate as input for the MDP [GKW$^+$16]. Hence the highly abstract the presence of network delays and natural forks of the chain.

In this thesis, we propose a new simulation approach to more accurately capture the details of the Blockchain protocol under simulation, while allowing for a high degree of determinism. With our simulation, it would be possible to model the selfish mining attack with different network topologies and to use the Bitcoin source code directly in the simulation.

CHAPTER 2

State-of-the-art

Already in the year 2010 the user *ByteCoin* described the idea of selfish mining in the Bitcoin forum *bitcointalk* [Byt]. He provided simulation results of the attack which at that time was called *mining cartel attack*. Nevertheless, the discussions in the thread never caught fire and no further investigations or countermeasures were taken by the community [Bitd, Bah13].

Later in 2014 [ES14] released the paper *"Majority is not enough: Bitcoin mining is vulnerable."* and coined the term selfish mining. The paper gives a formal description of selfish mining and proves how a miner can earn more than his fair share by conducting the attack. Figure 2.1 shows the attack as a state machine where α denotes the mining power share of the selfish miner. The labels of the states are representing the lead of the selfish miner over the public chain. Whenever the public network finds a block and the selfish miner publishes a competing block of the same height a block race occurs denoted with the state *0'*. In the case of such a block race, the variable γ expresses the probability of the selfish miner to win the block race. Hence γ part of the miners are mining on the public-private block and respectively $(1 - \gamma)$ are mining on the public block. The labels on the transitions are representing the transition probabilities between the states. The profitability of the simple strategy of [ES14] was proven by using probability calculations based on the state machine of figure 2.1. Furthermore, results of an undisclosed Bitcoin protocol simulator were given. In the simulation, 1000 miners with the same mining power were simulated and a fraction of these miners formed a pool which applied the selfish mining algorithm. In the case of a block race they artificially split the network where one part is mining on the public block and one part is mining on the block of the selfish pool.

Further research showed that more generalised selfish mining strategies lead to even more relative gain for the selfish miner [NKMS16, SSZ16, GRK15, GKW^{+}16, Bah13]. [NKMS16] provided a comprehensive description of the strategy space and also coined different names for the selfish mining variations:

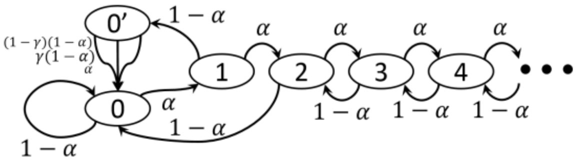


Figure 2.1: Selfish mining state machine with transition probabilities [ES14]

- **Lead stubborn**: This mining strategy compromises the idea to cause as many block races as possible and to never overwrite the public chain with a longer chain. This strategy continuously tries to split the network to mine on different blocks and is therefore especially promising when the probability to win the block race is very high.
- **Equal-fork stubborn**: The mining strategy equal-fork stubborn changes the selfish mining strategy just by one transition. In case the selfish miner finds a block during a block race, he does not publish his block to win the race but he also keeps this block undisclosed to secretly mine on this new tip of the chain.
- **Trail stubborn**: The mining strategies based on trail stubbornness are reflecting the idea to even trail behind the public chain and to eventually catch up. Trail stubbornness is defined with an integer denoting how many blocks the strategy should allow the selfish miner to trail back.

The strategy space for a selfish miner is practically endless and combinations of the aforementioned strategies are possible and are leading to even more relative gain compared to honest miners[NKMS16, SSZ16, GRK15, GKW$^+$16, Bah13].

To find the best strategy for a given mining power share α and connectivity γ researchers used different methodologies. [GRK15, NKMS16] used numeric simulations of paths in the state machine to find optimal selfish mining strategies. [SSZ16, GKW$^+$16] on the other hand used MDPs based on a state machine to find strategies with the most relative gain. The basic structure of the used state machines is for all publications the same. To further validate their results [ES14, SSZ16] used a closed-source simulation.

Besides using variations of the selfish mining strategies, the attack can also be combined with other attacks to achieve better results [GKW$^+$16, SSZ16, NKMS16, GRK15]. If the eclipse attack is used in combination with selfish mining the victim contributes its mining power to the private chain and hence, strengthens the position of the selfish miner [NKMS16, GKW$^+$16]. [NKMS16] additionally shows that the eclipsed victim under certain circumstances can benefit from the attack and therefore has no incentive to stop the attack. Another attack which can be used in combination with selfish mining is double-spending [SSZ16, GKW$^+$16]. Every time the selfish miner starts his selfish mining attack he can publish a transaction and include a conflicting transaction in his first secret block. During the execution of the selfish mining attack, the payment receiver may accept the payment depending on his block confirmation time. Now in the case of a successful selfish mining attempt, the adversary can overwrite the public chain, which additionally results in a successful double spending. The operational costs of unsuccessful double-spending can be seen as low because the adversary still would get goods or a service in exchange for the transaction [SSZ16, GKW$^+$16].

Last but not least also the prevention of selfish mining is part of the current work in selfish mining research [ES14, Hei14, SPB16, ZP17]. A backwards-compatible patch to mitigate selfish mining is uniform tie-breaking [ES14]. This means whenever a node receives two blocks of the same height he randomly select on of the blocks to mine on. [ES14] showed that this would raise the profit threshold to 25% of the computational power and hence mitigating selfish mining. The drawback of this proposed change is that it would increase the connectivity of badly connected attackers to almost 50% with no actual effort for them. Ethereum, the currently second largest cryptocurrency by market capitalization [Mar], has implemented uniform tie-breaking as a countermeasure against selfish mining [GKW$^+$16, uni]. Another countermeasure foresees unforgeable timestamps to secure Bitcoin against selfish mining [Hei14]. This countermeasure would make all pre-mined blocks of the selfish miner invalid after a certain amount of time. The implementation of this patch would require random beacons and hence introduce complexity and a new attack vector [Hei14]. [ZP17] proposes backward-compatible countermeasure by neglecting blocks that are not published in time and allows incorporation of competing blocks in the chain similar to Ethereum's uncle blocks [Woo14]. This enables a new fork-resolving policy where a block always contributes to neither or both branches of the fork [ZP17]. All of this mentioned countermeasures are not planned to be implemented or implemented in Bitcoin [bita, bitc].

CHAPTER 3

Research question

The expected outcome of this thesis is a more accurate simulation of different selfish mining strategies and therefore a better understanding of the potential real world implications of such attacks. The selfish mining strategies used in the thesis include:

- selfish mining [ES14]
- lead stubborn mining [NKMS16]
- trail stubborn mining [NKMS16]
- equal-fork stubborn mining [NKMS16]

For the simulation, these strategies are combined with different distributions of computation power in the underlying peer-to-peer network. The result of the simulations should show which strategy is the best strategy for a certain distribution of mining power and if the selfish mining increases the relative gain of the selfish miner compared to the normal, honest mining. The simulation results should emphasise the recent work in the area of selfish mining and show that the current implementation of Bitcoin protocol is vulnerable against different selfish mining strategies.

The desired outcome of the thesis is supported with the following two research questions:

- **RQ1:** Do the simulations of selfish mining with the proposed software solutions show an increase of the relative gain for the selfish miner compared to the normal, honest mining behaviour?
- **RQ2:** How does the obtained results of the simulation match the outcome of previous research in the area of selfish mining?

An additional outcome of the thesis is the simulation software. The software should allow an accurate and deterministic simulation of the blockchain by using directly the reference implementation and a realistic network topology. Hence, the simulation software could not only be used to simulate selfish mining attacks but could for example also be used to simulate other attacks or new protocol versions of Bitcoin. Since many other cryptocurrencies are derived from Bitcoin, they simulation software could be used also to simulate their behaviour and properties.

CHAPTER 4

Methodology and Approach

First, the different strategies selfish, lead stubborn, trail stubborn and equal-fork stubborn mining from [NKMS16] and [ES14] need to be implemented. This is achieved by implementing a proxy which eclipses a normal Bitcoin client from the other nodes in the network. Now, if a block is found the proxy decides, depending on his selfish mining strategy, if a block should be transmitted from the eclipsed node to the rest of the network or vice versa. The proxy design pattern makes it possible to implement the selfish mining strategies without altering the reference implementation of Bitcoin and is therefore preferred over an implementation directly in the Bitcoin client.

In the next step, a simulation program is implemented. To be able to control when a certain node finds a block, all Bitcoin nodes should be executed in *regtest* mode. In this test mode, the real PoW-algorithm is disabled and every node accepts a command which lets the node create immediately a new block. With this functionality, it is possible to define a block discovery series which basically reflects the computation power of each node. The more blocks are found by a node the more simulated computation power the node has. Additionally to the block generation, the simulation program should also control the network topology and hence the connectivity of each node. For the simulation run, it is important that the connectivity of the nodes stays the same to make the results better comparable. This should be achieved by setting the connections from the nodes by the simulation program itself which is in contrast to normal behaviour. Normally Bitcoin nodes share their connections with other nodes over the Bitcoin protocol and try to improve the connectivity over time.

After the implementation of the selfish mining strategies and the simulation program, the mining strategies are simulated. Different network topologies and distributions of computation power are used to compare the relative gain of the selfish mining strategies over the normal, honest mining.

CHAPTER 5

Simulation software

The simulation framework provides all needed functionalities to orchestrate a peer-to-peer network where on each node is running the *Bitcoin* reference implementation. This peer-to-peer network is implemented as a *Docker* network containing multiple *Docker* containers each running *Bitcoin*. The framework also coordinates the block discovery in the network. Based on a sequence defined in a configuration file the framework sends commands to the nodes which are then generating valid blocks. This is possible because all nodes are executed in the so-called *regtest mode*, where the CPU-heavy proof-of-work is disabled and the nodes are accepting and RPC-call from outside which lets them create immediately a new block. After a simulation run, the framework parses the logs created by the nodes and based on them a report gets created which displays the key metrics of the simulation.

5.1 Tick

A key concept of the simulation framework is a so-called tick. A tick represents a small time span containing information about which nodes should find a new block in this tick. For a simulation run, multiple ticks are generated forming a sequence of ticks. This sequence is basically the simulation scenario for a particular simulation run. The exact duration of a tick is defined on execution of the simulation, hence defining the speed of the simulation. The smaller the defined duration, the higher the execution speed of the simulation.

advantages: fake discrete, we can speed up/slow down

5.2 Configuration files

A simulation executed by the framework needs to be configured with the configuration files *nodes.csv*, *network.csv* and *ticks.csv*. The configuration files are stored in the concise CSV format and on a specific location on the disk to be easily processed by the simulation

	node a	node b	node c
node a	0	1	0
node b	0	0	1
node c	1	1	0

Table 5.1: An example *network.csv* represented as table

framework. This approach gives high flexibility and reproducibility for each simulation run. The configuration files provide the flexibility to be written manually, to be created by a script or a combination of both. Furthermore, the created configuration files can be copied to the output directory of a simulation run providing reproducibility.

The simulation framework already implements for each configuration file a simple script which can be executed with the corresponding commands *nodes* (chapter 5.4.1), *network* (chapter 5.4.2) and *ticks* (chapter 5.4.3).

5.2.1 *nodes.csv*

The *nodes.csv* contains the configuration of every node which should be orchestrated by the simulation framework. Each row in the file reflects one node containing:

- *node_type*: Either *bitcoin* if the node is a normal node or *selfish* if the node should act as a selfish node.
- *share*: The computational share of the node in the network.
- *docker_image*: The Docker image to use when starting the node.
- *group*: Which node group the node belongs to.
- *latency*: The latency of the node in the peer-to-peer network.

5.2.2 *network.csv*

The *network.csv* reflects a connection matrix as shown in table 5.1. The simulation framework starts each node in a way that a node on the y-axis tries to establish an outgoing connection to another node on the x-axis whenever the corresponding value in the matrix is set to 1.

5.2.3 *ticks.csv*

The *ticks.csv* contains all ticks which should be executed by the simulation framework. Each line represents a tick with no, one or multiple block events. Hence the *ticks.csv* has no header describing the columns and the length of the lines varies, depending on the number of block events in the corresponding tick.

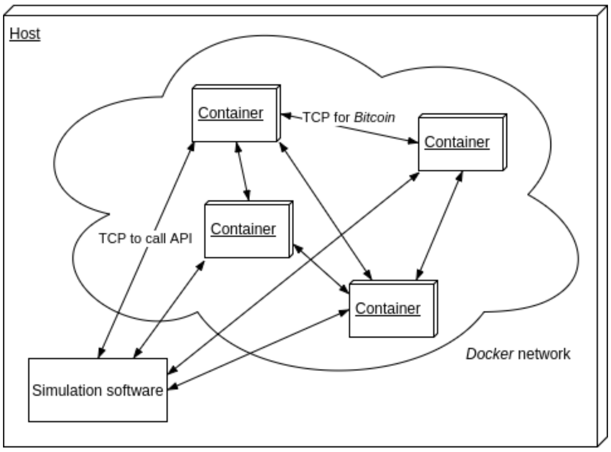


Figure 5.1: Overview of the virtual peer-to-peer network

5.3 Simulation

The main functionality of the simulation framework is to orchestrate a simulation based on the configuration files. This is achieved by using the high-level programming language *Python* and the virtualization software technology *Docker*. *Python* is mainly used to handle the configuration files and to automatically execute *Docker* and other binaries whenever necessary. *Docker* on the other side provides the needed capabilities to run the *Bitcoin* reference implementation and other programs in virtual, lightweight containers on one single host. These containers are using the functionalities of the same kernel in an isolated manner and hence they do not interfere each other as long as the host system provides enough resources to them. The containers can also reuse the networking stack of the kernel, making possible to create a peer-to-peer network needed for the simulation.

To execute a simulation the command *simulate* (chapter 5.4.4) can be used. In that case the configuration files need to be available on disk. The commands *run* (chapter 5.4.5) and *multi-run* (chapter 5.4.6) provide another possibility to execute a simulation, creating all configuration files before starting the simulation.

5.3.1 Preparation

The preparation phase is the first phase of a simulation run. At the beginning, a simulation directory is created and all configuration files are copied into the new directory to assure reproducibility. Subsequently, a virtual peer-to-peer network is build as depicted in figure 5.1. Therefore firstly a *Docker* network with the driver set to *bridge* is created which under the hood configures a new network interface in the networking stack of the host machine. This network interface is used by the *Docker* containers to connect and communicate to other containers. Afterwards based on the two configuration files *nodes.csv* and *networking.csv* the nodes are created as *Docker* containers with *bitcoind* set as command to be executed in the container. Listing 5.1 shows how *Docker* is used to create a container, which executes *bitcoind* on start-up. In line 2 the unique IP of the container is set by using the *--ip* argument. Line 3 shows the usage of a so-called *Docker volume* to mount the folder *data/run-10/node-5/* into the container's folder */data/regtest*. By running *bitcoind* in *regtest mode* (line 6) and setting the data directory of *bitcoind* to */data* (line 7) *Bitcoin* persists all relevant data into */data/regtest*. Hence all the data persisted by *bitcoind* under */data/regtest* will actually be persisted under *data/run-10/node-5/* on the host machine and is therefore still available after the destruction of the container. In line 8 we define to which other nodes the node should connect to by specifying their IP. By using the *-connect* parameters the *Bitcoin* reference implementation automatically stops to listen for incoming connections. Since this is needed to accept incoming connections, it is re-enable in line 9 by setting *-listen* to 1. Lastly, after all nodes are spawned, an RPC-connection to the *Bitcoin* API running in each node is established by using the library *python-bitcoinlib*. These connections are later used to directly send commands to the nodes.

julia

```
1 docker run
2         --ip=240.1.0.5
3         --volume data/run-10/node-5:/data/regtest
4         bitcoind_image
5                 bitcoind
6                         -regtest
7                         -datadir=/data
8                         -connect=240.1.0.2 -connect=240.1.0.9
9                         -listen=1
```

Listing 5.1: Simplified version of how a node is started with *Docker* and *bitcoind*

5.3.2 Execution

In the execution phase the simulation framework iterates over each tick from the *ticks.csv*. If a tick contains a block event, the framework calls *generate* on the *Bitcoin* API of the specific node to generate a new block. Since all nodes are running in *regtest mode* the proof-of-work is deactivated and the block can be created immediately. Some ticks may contain multiple block events. In this case the block events are executed one after another always waiting for the block hash to be returned by the nodes. A simulation run is always executed with a certain tick duration. This tick duration specifies how long a tick should last. Therefore the simulation framework simply keeps track of the passed time during the execution of the block events and sleeps afterwards until the tick is over. In rare case the completion of the block events may last longer then the tick duration. Then the framework immediately starts with the next tick and tries to regain the lost time.

During the execution of the ticks a thread separately collects information about the current CPU and memory usage. For the CPU usage the thread queries periodically the */proc/stat* file which is showing how much time the CPU spent in a certain state. The collected snapshots can later be used to determine the actual utilization of the CPU by calculating the differences between the snapshots. For the memory usage the thread reads periodically the *MemAvailable* value in */proc/meminfo* file. This value provides a heuristic of the current available memory on the machine.

5.3.3 Post-processing

The post-processing phase is the last phase of a simulation run. At the beginning of this phase the consensus chain, denoting the longest chain of blocks all nodes agree about, is calculated. This is done by starting at block height one and asking each node for the hash of the block on this height in their longest chain. If all nodes have a block at this height and the hashes of all blocks are the same, then all nodes reached consensus and the block is added to the consensus chain. In the next step the height is increased by one and the previously described check is repeated. If one node has no block at a certain height or the hashes of the blocks differ then the calculation of the consensus chain stops.

After the calculation of the consensus chain all *Docker* nodes are stopped and removed. Because a separate data directory was mounted on each node by using *Docker volumes* all relevant data, especially the log files, are still available on the host machine after the deletion of the *Docker* nodes. In the next step lines of the logs from nodes and from the log file of the simulation framework are parsed to retrieve information about the simulation run. These log line types are:

- *BlockGenerateLine*: Log line produced by a node when a new block is generated.
- *BlockStatsLine*: A log line displaying various information like block size about a freshly generated block.
- *UpdateTipLine*: Log line produced by a node whenever a block updates a tip of the chain.
- *PeerLogicValidationLine*: Log line produced when the proof-of-work of a received compact block is checked.
- *BlockReconstructLine*: A log line created when a compact block was successfully reconstructed.
- *BlockReceivedLine*: Log line created when node receives a normal block.
- *TickLine*: A log line with information about an executed tick.
- *BlockExceptionLine*: Log line created whenever the simulation framework was not able to successfully execute a block event.

- *RPCExceptionEventLine*: Log line denoting an exception occurred while using the RPC-connection to a node.

The log lines *BlockGenerateLine*, *BlockStatsLine*, *UpdateTipLine*, *PeerLogicValidationLine*, *BlockReconstructLine* and *BlockReceivedLine* are all produced by nodes executing *Bitcoin* where on the other hand *TickLine*, *BlockExceptionLine* and *RPCExceptionEventLine* are created by the simulation framework. Furthermore, the *BlockGenerateLine* log line was added especially for the simulation framework to the *Bitcoin* reference implementation. Normally the reference implementation does not create a log line containing the block hash when it creates a new block. Hence to easily circumvent this fact, a log line was added to the reference implementation to persist the event of the block creation including a hash of the block.

The simulation framework persists all parsed log lines into CSV files where each log line type gets his own file. Subsequently the *preprocess.R* script prepares the CSV files for the final report creation. When a simulation is executed a parameter can be passed which denotes how many ticks at the beginning and at the end should not be evaluated in the post-processing phase (*skip_ticks*). The *R* script then figures out when the first and the last tick to be evaluated occurred and tailors the log line types *BlockGenerateLine* and *BlockStatsLine* respectively. All other types do not need to be tailored because the either are used to calculate some combined statistics like the block propagation time or because the statistics of these types are still calculated over the whole simulation duration. Additionally the *preprocess.R* script sorts all CSV files according to the timestamp of the log line. This is necessary because the parsing of the log files is implemented in a multi-threading manner and thus the ordering from the original log files is lost.

After all CSV files are created the simulation frameworks generates a report by executing a *R Markdown* file. The final report contains:

- general information about the simulation like the start and end time
- specifications and settings of the host machine used in the simulation
- all input arguments passed to the simulation
- summary about planned, executed and parsed block events
- overview of the duration of each phase of the simulation
- chart visualizing CPU and memory utilization
- chart showing the duration of a tick over time
- charts and informations about blocks created during the simulation
- charts and informations about exceptions happened during the simulation

Where most of the informations and charts are just simple representations of the data present in the CSV files, the stale block rate and propagation time of blocks needs to be calculated in the report. The stale rate, describing how many blocks did not end up in the longest chain, is calculated by checking each created block against the consensus chain determined previously by simply merging the *BlockGenerateLine* log lines with the consensus chain. The propagation time of blocks is calculated with *R* as shown in listing 5.2. First the *BlockGenerateLine* log lines are merged with the lines describing the event of receiving a block, namely *UpdateTipLine*, *PeerLogicValidationLine*, *BlockReconstructLine* and *BlockReceivedLine* creating a new data frame. Since for example *UpdateTipLine* is also logged by the node which created the block in line 5 all these elements are filtered out of the data frame. Afterwards the data set is grouped by the block hash and the name of the node (line 7). By filtering out the element with the lowest timestamp, the data frame now represent the points in time when a node heard first about a certain block. Lastly the propagation time is calculated by subtracting the timestamp of the *BlockGenerateLine* log line from the timestamp of the receiving log lines.

```
1 log_lines <- merge(log_lines_receiving_block, block_generate,
2                    by = 'hash')
3
4 log_lines %>%
5   filter(as.character(node.x) != as.character(node.y)) %>%
6   select(-node.y, node = node.x) %>%
7   group_by(hash, node) %>%
8   filter(which.min(timestamp.x)==row_number()) %>%
9   mutate(propagation_time = timestamp.x - timestamp.y)
```

Listing 5.2: Calculation of propagation time with *R*

5.3.4 Multi-runs

When the simulation framework is executed with the *multi-run* command (chapter 5.4.6) multiple simulations are conducted depending on the passed input arguments. After each simulation the created CSV files of the simulation are copied by the simulation software into a own directory. Once the last simulation finishes the framework aggregates all copied CSV files into single CSV files for each log line type. Subsequently the *R Markdown* file, which also is used to create the final report of single simulation, is executed to create a report comparing all simulation runs.

5.4 Commands

The simulation framework exposes six commands to the user. Three of this commands are creating configuration files necessary for the execution of a simulation. One command, the *simulate* command, executes a simulation based on these configuration files. The other two commands, *run* and *multi-run*, are aggregations of the before mentioned commands.

5.4.1 *nodes* command

The *nodes* command can be executed with so-called node groups (eg. *node_group_a*) as input parameters. A node group represents a group with a certain amount of nodes sharing the same node type, Docker image and latency. Alongside these attributes a node group specifies a certain share of the computational power in the network. On execution the simulation framework parses all passed node groups and checks if the shares defined for each group are summing up to a total of 100%. In that case, the framework persists the nodes of each group in a file called *node.csv*, where the share of the computational power of the group is equally distributed over all members of the respective group.

5.4.2 *network* command

When the simulation software gets executed with the *network* command it reads a *node.csv*, which needs to be available, to determine all planned nodes. Based on an additional connectivity parameter (*connectivity*), which defines with how many nodes a node should be connected, the simulation framework creates a matrix reflecting connections between two nodes. The Bitcoin reference implementation itself does not differentiate between established incoming or outgoing connection, hence it suffices to define one connection in the connection matrix if two nodes should be connected. The connection matrix is afterwards persisted in the configuration file *network.csv*.

5.4.3 *ticks* command

The *ticks* command can be used to create the configuration file *ticks.csv*, which contains the ticks to be executed. When executing the *ticks* command the simulation framework accepts one parameter denoting the amount of ticks to create (*amount_of_ticks*) and one parameter about how many blocks per tick should be generated by the nodes (*blocks_per_tick*). Additionally the simulation framework reads the *nodes.csv*, which check with julia needs to be available , to determine all planned nodes and their computational share (*share*). Afterwards the framework parametrizes for each node an exponential distribution as shown in 5.1 with $\lambda = blocks\_per\_tick \cdot share$. From this exponential distribution sufficient samples are drawn, which are denoting points in time when a specific nodes should find a block. With this time series at hand the ticks are created by starting with the 1st tick. For every point in time lower then the number of current tick a block event for the respective node is added to the tick and the point in time is removed from the time series. This is repeated for every tick until reaching *amount_of_ticks*. For example if we calculated the 5 samples 0.4, 0.8, 2.3, 4.1 and 5.8 for an arbitrary node A. Furthermore the desired *amount_of_ticks* would be 5. Then we would get 5 ticks, where in the 1st tick are two block events for node A, and in the 3rd tick and 5th tick one each check with julia . The 2nd and the 4th tick would stay empty. After calculating all ticks, the ticks are persisted in the *ticks.csv*.

$$f(x;\lambda) = \begin{cases} 1 - \exp(-\lambda x) & \text{x} \geq 0, \\ 0 & \text{x} < 0 \end{cases} \tag{5.1}$$

5.4.4 *simulate* command

On execution of the *simulate* command the simulation framework starts a simulation based on the configuration files *nodes.csv*, *network.csv* and *tick.csv*. All these files need to be available and furthermore the duration of ticks (*tick_duration*) and the amount of ticks which events should not be evaluated (*skip_ticks*) are parsed as input arguments. Afterwards the simulation framework executes the simulation as described in section 5.3.

5.4.5 *run* command

When the simulation software is started with the *run* command basically the commands *nodes*, *network*, *ticks* and *simulate* are executed in exactly this order. It is possible to pass all desired input arguments to the specific commands, but since the simulation is started right after the creation of the configuration files, it is not possible to change those files before the simulation.

5.4.6 *multi-run* command

The *multi-run* command accepts an input parameter denoting how often a run should be repeated (*repeat*). The simulation software then creates all configuration files using the passed input arguments and subsequently executes the simulation *repeat* times using the same configuration files as depicted in figure 5.2.

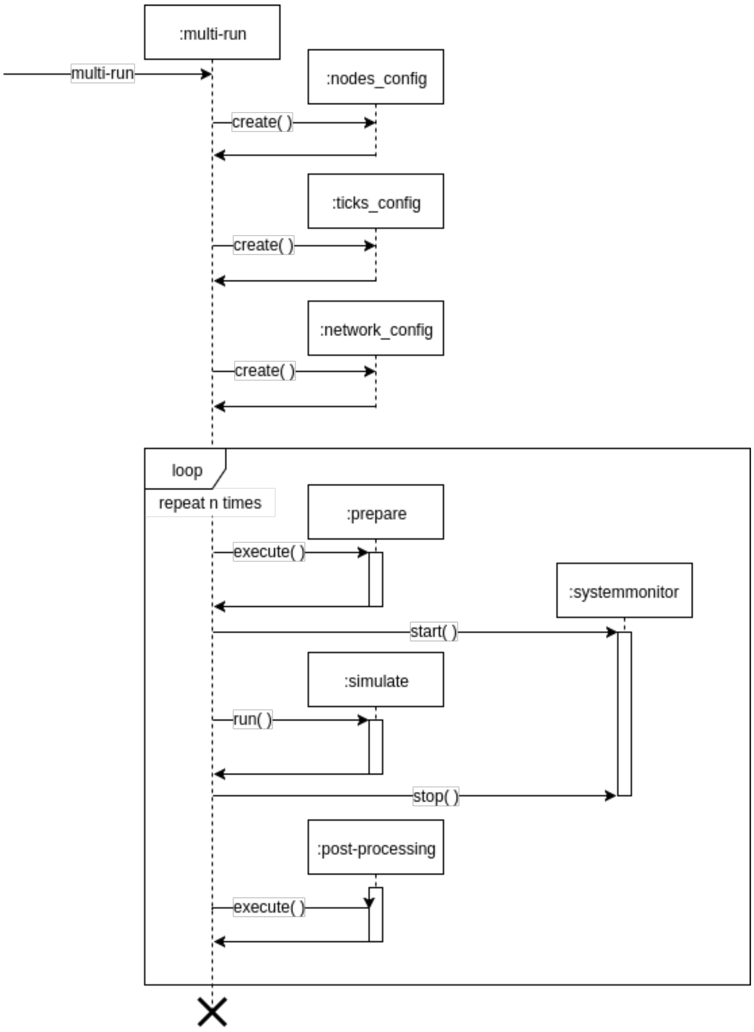


Figure 5.2: Flow chart of the *multi-run* command

CHAPTER 6

Evaluation of simulation software

- we can execute simulation multiple times (100 times would last week) - then we can look at the stale block rate - check which distribution is appropriate - and then we will see :D

how should we do that?

CHAPTER 7

Selfish proxy

The selfish proxy is a node in the peer-to-peer network which performs selfish mining in collaboration with a connected, eclipsed *Bitcoin* node. Together the two nodes are forming a selfish miner as shown in figure 7.1. The proxy implements parts of the *Bitcoin* communication protocol and requests all blocks created either by the honest network or by the eclipsed node. With the retrieved blocks the selfish proxy recreates the chain locally and whenever the public or the private chain changes the node executes the configured selfish mining algorithm. Depending on the outcome of the selfish mining algorithm the proxy afterwards relays blocks to the other part of the network. With this withholding method, the selfish proxy can mimic different selfish mining strategies without creating a single block.

7.1 Network

The selfish proxy is a normal member of the peer-to-peer network and is also executed as a *Docker* container. During the simulation run, the proxy mimics the behaviour of a normal *Bitcoin* node so that all nodes connected to proxy think they are connected to a normal peer. In figure 7.1 a possible network topology with a selfish proxy is depicted. The nodes on the left side are forming the honest, public network working together on the public chain. The two nodes on the right side are forming the selfish miner where the proxy abuses the private chain build by the eclipsed node to execute a particular selfish mining strategy. The topology of the peer-to-peer network is solely established by the simulation software. First, the software starts the selfish proxy which then just listens for new incoming connections. Afterwards, the normal *Bitcoin* nodes are started with the respective *-connect* parameters set. If such a normal node has the IP of the selfish proxy set in a *-connect* parameter, then the *Bitcoin* node simply connects to the listening proxy. The proxy accepts the connection and behaves like a normal *Bitcoin* node during the whole simulation run by obeying the *Bitcoin* communication protocol.

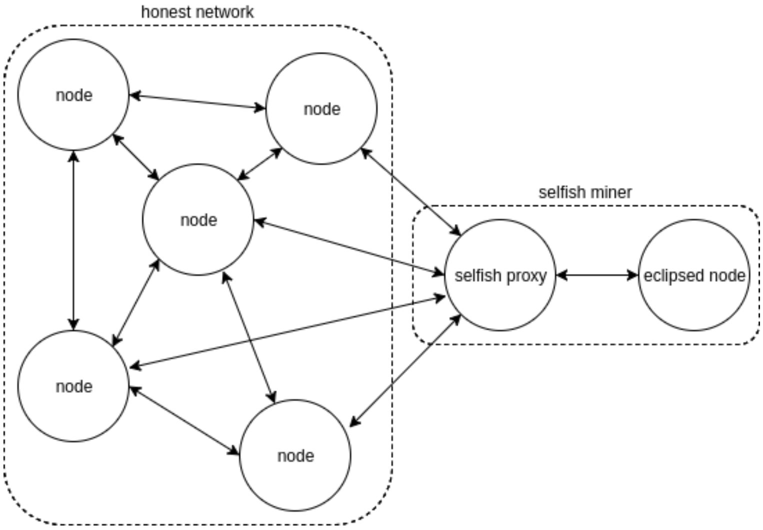


Figure 7.1: Selfish proxy eclipsing a normal node

The implementation of all network related functionalities of the selfish proxy is based on the two open-source libraries *pycoin* [**?**] and *python-bitcoinlib* [**?**]. The library *pycoin* provides simple networking utilities to connect to other *Bitcoin* nodes and to manage those connections. The *python-bitcoinlib* library on the other hand implements functionalities to serialise and de-serialise *Bitcoin* network messages.

7.2 Chain

The selfish proxy continuously collects all block and block headers sent by the connected peers and reassembles the whole chain locally. To execute the selfish mining algorithm efficiently the proxy needs to retrieve updates of the private and public chain as fast as possible. Therefore the proxy uses solely block headers to update the chain despite using whole blocks. The block headers which contain all necessary information to updated the chain can be retrieved faster than the full block because they are just a part of the block and hence smaller. Furthermore, it is secure for the proxy to trust in the validity of the block header since all other nodes in the network are behaving honestly and hence are sending only valid block headers.

When the selfish proxy tries to update the chain with a so far unknown block header, it simply looks at the hash of the previous block stored in the block header. If the previous block hash is in the chain, the newly received block header is appended to the chain. On a programmatic level, the proxy uses for that a one-way linked list with the possibility to navigate to the previous block. In the case the block header has no direct ancestor in the current chain, the header gets preserved as an orphan block. All orphan blocks are checked on every successful insertion of a block if they now can be added to the chain.

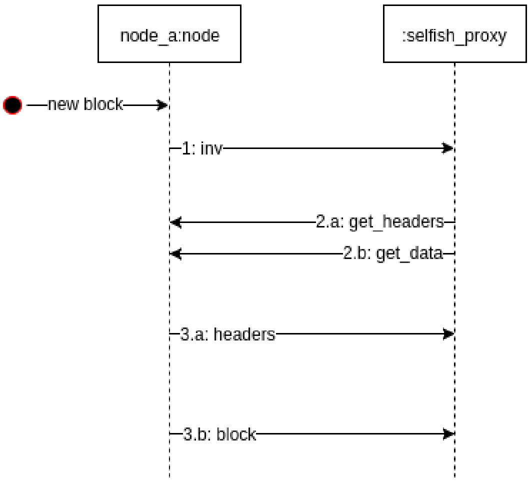


Figure 7.2: Selfish proxy receiving a block from another node

Alongside the information stored in the block, the proxy also keeps track of the block origin and a boolean variable called *transfer_allowed*. The block origin is a simple enumeration if the block was received from the honest network or from the eclipsed node and hence does not change over time. The *transfer_allowed* variable determines if the transfer of a block is allowed and is initially always set to *False*. Depending on the selfish mining strategy the block may be relayed to the other nodes at some later point in time changing the boolean to *True*. These two variables are stored to be able to distinguish between the public chain, the current longest chain known to the honest network and the private chain, the current longest chain known to the eclipsed node. For example to determine the current private chain all blocks originated by the eclipsed node and all blocks with *transfer_allowed* set to *True* are used. The two views of the chain are essential for the selfish mining algorithm to decide which action to take and hence when to relay which block to the other side of the network.

7.3 Receiving blocks

An essential capability of the selfish proxy is to retrieve blocks and block headers from its peers. Figure 7.2 depicts the communication flow between an arbitrary node called *node_a* and the selfish proxy which wants to retrieve the information of a new block. Firstly *node_a* either finds a new block itself or retrieves a new block from some other node in the network. Then, adhering to the *Bitcoin* protocol, the node sends an *inv* message (1) containing the hash of the block to its peers including the proxy. The proxy subsequently checks if it already requested the block from another node or even has the block in the own local chain. In this two cases, the proxy would just ignore the received hash, and the communication flow would end. If the block hash is unknown, the proxy sends a *get_headers* (2.a) message and a *get_data* (2.b) message to the *node_a* as pictured in figure 7.2.

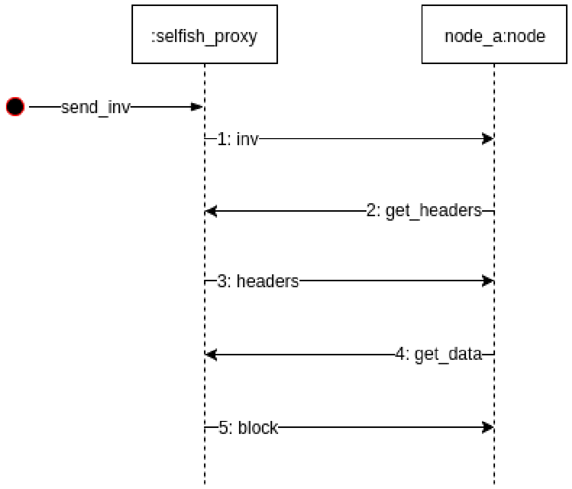


Figure 7.3: Selfish proxy sending a block to another node

The *get_headers* message (2.a) sent by the proxy is composed with an array called block locator hashes and is used to retrieve all block headers after the known block hashes denoted in the array. To create the array the proxy uses either the private or the public chain depending if *node_a* is the eclipsed node or a node of the honest network. The proxy adds then the highest, 2nd, 4th, 8th and 16th highest blocks of the selected chain to the array. If the chain does not provide all needed bocks, then only the available blocks are added to the block locator array. *Node_a*, after it received the *get_headers*, will search the block hashes from the block locator array in its own longest chain starting with the highest block. Once a block hash matches a block in the longest chain of *node_a*, *node_a* collects all block headers after the matched block in a *headers* message and sends the message back to the proxy (3.a). In the normal case the proxy trails just one block behind the highest block known by *node_a*, hence the headers message will contain only one single block header. In the usual cases the proxy actually felt more then one block behind and *node_a* will send multiple block headers back to the proxy. Since the proxy only needs block headers to update the chain, it can immediately update with the received headers the whole chain to the newest tip known tho the *node_a*.

The *get_data* message (2.b) sent by the proxy just contains the block hash of the desired block. As soon as the *node_a* receives the request for the block, it will return the full block in a *block* message (3.b) to the node. The request for the whole block lasts typically longer than the request for the newest block headers with the *get_headers* message as it is pictured in figure 7.2. The proxy request the full block containing all information solely to be able to respond to *get_data* request by other nodes when it advertises the block on later point in time.

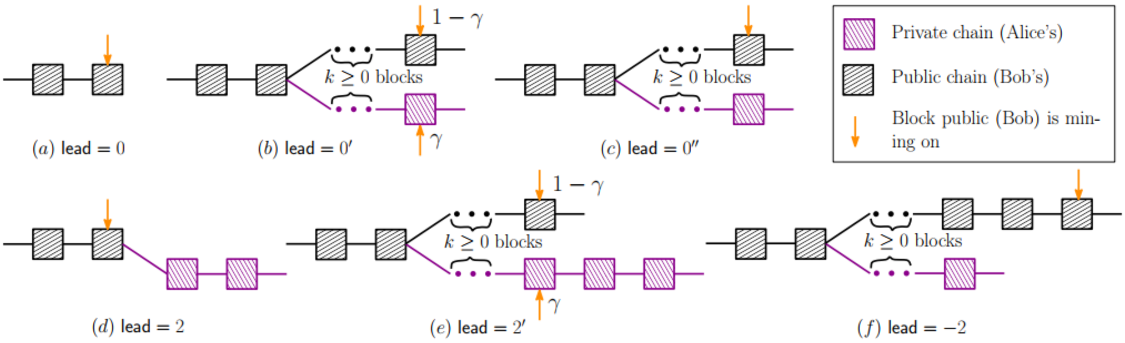


Figure 7.4: Different possible leads of the private chain [NKMS16]

7.4 Sending blocks

After the execution of the selfish mining algorithm, the proxy may want to send a block to the opposite origin of the block. Figure 7.3 shows the communication flow between an arbitrary *node_a* and the selfish proxy which intends to relay a block. The proxy therefore firstly sends the block hash as *inv* message (1) to the *node_a*. The *node_a* after receiving the *inv* message will then reply with a *get_headers* message (2) because it has not seen the withheld block so far. The *get_headers* message contains, similar to the *get_headers* build by the proxy when it receives a block, known block hashes by *node_a*. The proxy then selects either the private or the public chain depending if *node_a* is the eclipsed node or a node of the honest network. In the case that there is no unique, longest chain the selfish proxy prefers the chain where the origin of the highest block is the eclipsed node to promote the blocks of the eclipsed node. Subsequently, the proxy iterates over the selected chain until a block hash from the locator array send by the *node_a* matches. The proxy then returns all headers of the blocks after the matched block until the highest block composed in a *headers* message (3). Afterwards, the *node_a* will iterate over the received headers and request all missing blocks. In the usual case, *node_a* will just lack one block which the node will simply request by sending a *get_data* message (4) to the proxy. The selfish proxy replies to this message then by sending a *block* message (5) containing the full block. Since the selfish mining algorithm already processes the block header before the whole block is available, it could be the case that a block requested by a node is not yet available. In this case, the proxy defers the reply to the node until it receives the entire block from another node.

7.5 Selfish mining

The selfish proxy executes selfish mining in collaboration with an eclipsed node which is only connected to the proxy as shown in figure 7.1. During the simulation, the proxy monitors the honest network which works on the public chain and the eclipsed node which works on the private chain and performs selfish mining by withholding the blocks created by both sides.

7.5.1 Private lead

Every time a block header is inserted in the chain the proxy checks if either the public or private chain altered. In the case that one of these two chains changed the proxy executes the configured selfish mining algorithm. To easier track the changes between the two chains an integer variable called private lead is used which describes the distance between the two tips of the chain as shown in figure 7.4. A positive lead n denotes an advantage of n blocks of the private chain over the chain of the honest network. Conversely, a negative lead n stands for a n block lag of the eclipsed node over the public chain. Furthermore, there exist positive leads annotated with an apostrophe denoting that at the height of the public chain a block race happens. In this block race the possibility that the private chain is extended on the height of the public chain is γ and the probability that the public chain gets extended is $1 - \gamma$. Lastly, a private lead of zero can be annotated with two apostrophes expressing that both chains have the same height but everyone is mining on his own chain.

7.5.2 Actions

The execution of the algorithm outputs one of the four possible actions *adopt*, *override*, *match* and *wait* equivalent defined in the work of [SSZ16]. An action describes which blocks should be advertised and relayed to the other side of the network at a given point in time:

- **Adopt**: The action *adopt* means that the selfish miner adopts the chain of the honest network. This is a typical action if the private lead is zero and the honest network finds a new block. Then it can be sensitive to just adopt to this new block. To execute the *adopt* action the selfish proxy relays the public chain to the eclipsed node by advertising unknown, public blocks.
- **Override**: The *override* action is only possible if the private lead is greater then zero after the insertion of the new block header. In this case, the selfish proxy can override the public chain by sending out the private blocks mined by the eclipse node. Hence, when the proxy executes the *override* action, it sends all private blocks including the first block strictly higher than the public chain. If there are even higher private blocks, the selfish proxy keeps them back for further selfish mining.
- **Match**: The *match* action is only feasible if the private lead previous the insertion of the block header was greater than zero and the origin of the block is the honest network. In this case, the selfish proxy can advertise the private block at the same height to the honest network creating a block race. After the execution of the *match* action, the resulting private lead is annotated with an apostrophe to denote the block race.

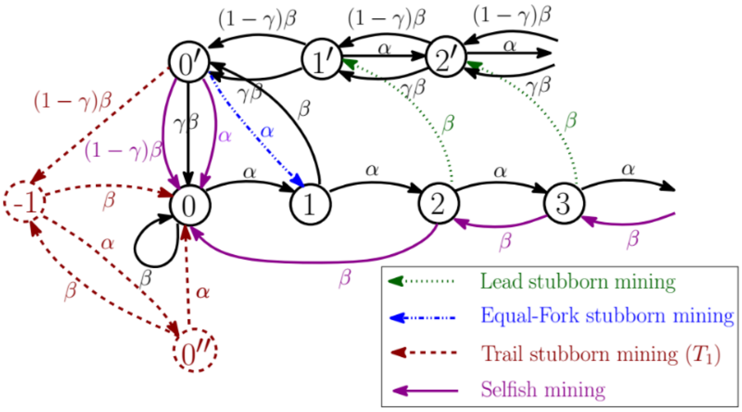


Figure 7.5: Categorization of different mining strategies [NKMS16]

- **Wait**: If the selfish algorithm outputs the *wait* action, then the proxy does simply nothing and waits for the next block which changes either the private or the public chain.

7.5.3 Strategies

The selfish mining strategies are defining which action the algorithm implemented in the proxy should execute after a new block was found in the network. All strategies implemented in the proxy are based on selfish mining strategy described by [ES14] and can be modified with the three modifications lead, equal-fork and trail stubborn by [NKMS16]. Figure 7.5 shows all strategies as a state machine where the label of the states stands for the private lead. The labels α and β used in the transitions are describing the probability that either the eclipsed node or the honest network finds a block. The variable γ represents the likelihood that the part of the honest network which mines on the private chain finds a block.

In the normal selfish mining strategy two possible cases can occur in state *0*. If the honest network finds a block then the selfish miner immediately adopts to the public chain and hence remains in the state *0*. In the other case where the selfish miner finds a block the state 1 is reached because the selfish miner does not share the block with the honest network. The same happens if the selfish miner finds more blocks. Then the selfish miner simple does not share his blocks advancing to state *2* and onwards. In state *1* the selfish miner has a private lead of one block. If then the honest network honest network finds a block the selfish miner immediately releases the private block and starts a block race. Any node in the honest block either chooses the private-public or the public block to mine on top depending which block it sees first. The block race is dissolved after any node in the network finds a block. If the honest network finds a block then the selfish miner adopts to the new tip. In the other case where the selfish miner finds a block it immediately sends out the block to win the race. In both cases, the state *0* is reached again. Lastly, if the private lead is two and the honest network finds a block then the selfish miner immediately publishes the two blocks. This overrides the newly created block by the honest network, and the state *0* is reached.

As introduced by [NKMS16] the selfish mining strategy can be modified as follows:

- **Lead stubborn mining**: In lead stubborn mining the selfish miner tries to cause as many block races as possible. So whenever the private chain is longer than the public chain, and the honest network finds a block the selfish miner releases the competing block with the same high causing a block race. This behaviour is also applied in state *2* where the selfish miner overwrites typically the block appended to the public chain by publishing two blocks. In lead stubborn mining the selfish miner only releases the competing block an starts a block race denoted with the state *2'*. This strategy is promising when γ is high implying that whenever a block race occurs, it is likely that the honest network finds a block on published block of the selfish miner. Hence, the honest network unwillingly helps the selfish miner to succeed the private chain during the block race.

- **Equal-fork stubborn mining**: The equal-fork stubborn mining strategy changes the behaviour of the selfish miner during a block race in state *0'*. Normally the selfish miner would use the created block to overwrite the public chain hence winning the block race. Using the equal-fork stubborn strategy the miner keeps the block back which leads to state *1* and the honest network remains mining on two different tips of the chain. Thus the strategy compromises the idea to keep the honest network split over two chains as long as possible.

- **Trail stubborn mining**: In trail stubborn mining the selfish miner allows the private chain to even trail behind the public chain. If the block race in state *0'* is won by the honest network the selfish miner does not adopt to the public chain and trails back leading into state *-1*. In the case that the selfish miner is able to catch up by creating a new block the state *0"* where both chains have the same length. The trail stubborn strategy finally pays off when the selfish miner finds another block and can override the public chain with the private chain. Trail stubbornness is parametrized with an integer n determining how many blocks the private chain is allowed to trail behind the public chain. If this threshold is reached the selfish miner dismisses his private chain and adopts to the public chain by reaching again state *0*.

The modifications of the selfish mining strategy can lead to even more relative gain for the selfish miner depending on the actual computational share and the parameter γ. Furthermore, the strategies can be combined and since the trail stubbornness can be parametrized the build an infinite strategy space

7.5.4 Algorithm

The selfish mining strategies and its modifications are implemented in selfish proxy by a simple algorithm using normal control flow structures. Since the selfish proxy does not has a holistic overview of when a node finds a block, it only can try to apply selfish mining whenever the public or private chain changes locally after the insertion of a new block header. The algorithm mimics the behavior shown in the state diagram from figure 7.5 by looking at the private lead before the insertion of the block header and the origin of the inserted block header. For example if the private lead before the insertion of the block header was one and the block header was appended to the public chain. This would correspond to the state *1* and the outgoing transition β which leads in the state diagram to the state *0'* denoting a block race. The selfish mining algorithm must now assure that this state is also reached in the simulated network by starting a block race. Thus the algorithm needs to execute the *match* action by publishing the private block to the honest network.

Listing 7.1 shows a part of the algorithm, namely the part where the private lead before the insertion is 0 and an action to be executed is searched. Hence, this part of the algorithm reflects the states 0, 0' and 0" of the state machine pictured in figure 2.1. In the lines 2 to 6 the state 0 is treated by simply looking at the origin of the last block. If the block was mined by the honest network then the proxy just adopts to the public chain. In the other case the block was found by the eclipsed node and the proxy just waits for the next block to be found. The lines 7 to 17 are covering the 0' and 0" states.

```
 1 if private_lead == 0:
 2     if length_private == 0:
 3         if last_block_origin is BlockOrigin.public:
 4             return Action.adopt
 5         else:
 6             return Action.wait
 7     else:
 8         if last_block_origin is BlockOrigin.public:
 9             if self.trail_stubborn < 0:
10                 return Action.wait
11             else:
12                 return Action.adopt
13         else:
14             if self.active and self.equal_fork_stubborn:
15                 return Action.wait
16             else:
17                 return Action.override
```

Listing 7.1: Part of the selfish mining algorithm where private lead is zero

In the case the last block was found by the honest network the proxy adopts to the public chain except the algorithm was configured with trail stubbornness. Then the proxy waits and hopes to catch up with the public chain at a later point (line 10). The other case, implemented from line 7 to 17, reflects the fact when the block is found by the eclipsed node. Then normally the proxy would override the chain by sending the respective private blocks to the honest network. An exception to this is when currently a block race is happens and the algorithm is configured with equal-fork stubbornness. In that case the selfish algorithm currently has set the variable *active* to *True* and applies equal-fork stubbornness by executing the *wait* action.

7.5.5 Configuration

The selfish proxy started with any configuration executes normal selfish mining. On execution time the three modifications lead, equal-fork and trail stubborn mining can be configured by using input arguments:

- *lead_stubborn*: A boolean input argument defining if lead stubbornness should be used.
- *equal_fork_stubborn*: A boolean input argument defining if equal-fork stubbornness should be applied or not.
- *trail_stubborn*: Used with an integer defining how much the selfish proxy should trail back.

CHAPTER 8

Simulation of selfish mining strategies

With the introduced simulation software and selfish proxy, it is now possible to analyse selfish mining and its impact on the relative gain of the selfish miner. As a base, the scenario described in chapter 7 is used where the selfish proxy eclipses one node of the network forming a selfish miner. To obtain a comprehensive overview of the impact of selfish mining various combinations of selfish strategies and different distributions of computation power between the nodes are used.

8.1 Selfish mining scenarios

As strategies, the standard selfish mining strategy and the three modifications lead stubborn, equal-fork stubborn and trail stubborn mining are put into action. The used trail-stubborn strategy is parametrised with 1 meaning that the selfish miner will at the maximum trail one block behind the public chain. Hence, the at least trail stubborn strategy is executed in the different scenarios. Since the modifications of the selfish mining strategies can be combined a total of eight different selfish mining strategies are composed. For the distribution of computation power, five different settings are used where the selfish miner receives either 15%, 22.5%, 30%, 37.5% or 45% of the computation power. The rest of the computation power in each scenario is distributed equally over all remaining, honest nodes. The five used shares are each 7.5% apart covering sensitive shares of the computation share. The scenario with a share of 7.5% and all scenarios above 50% are omitted. The scenario where the selfish miner would get 7.5% is discarded because it is very likely that in that case, selfish mining has no advantages. The other simulations where the selfish miner would get more than 50% are ignored because in that cases the most efficient strategy would be to just mine on the own chain and never to accept blocks from other nodes. Since the miner has more than 50%, it would always create the longest chain copping all mining rewards.

With eight different mining strategies and five different distributions of computation power, a total of 40 different scenarios are obtained. Listing 8.1 shows how a specific scenario is started with the simulation software. In this particular scenario, the selfish miner receives 30% of the computation power (line 4), and the rest of the network consisting of 19 nodes gets with 70% the rest of the power (line 3). As can be seen in line 5 the selfish mining strategy in this simulation run is modified with equal-fork and trail stubbornness. These arguments are passed by the simulation software to the selfish proxy when it gets created. From line 6 to 8 the scenario is configured with the same blocks per tick rate, amount of ticks and tick duration as in the reference scenario described in chapter 7.

```
1 python3 simcoin.py multi-run
2         --repeat 3
3         --group-a bitcoin 19 0.7 25 simcoin/patched:v2
4         --group-b selfish 1 0.3 0 simcoin/proxy:v1
5         --selfish-args '--equal-fork-stubborn --trail-stubborn 1'
6         --blocks-per-tick 0.0333333333333333
7         --amount-of-ticks 60480
8         --tick-duration 0.1
```

Listing 8.1: Command to execute a particular selfish mining scenario

8.2 Simulation

The previously defined selfish mining scenarios are executed on a *x86 Linux* host machine with 16 virtualised cores and 57.718 GB of memory, the same machine used to examine the deterministic behaviour of the simulation software in chapter 7. Each scenario got executed three times by using the *multi-run* command as shown in line 1 and 2 in the listing 8.1. To extract a particular metric from the multiple executions of a scenario the median is calculated. Since the simulation software does not always produce the same results, the median provides a robust method against possible outliers and hence, provides more accurate results are achieved.

Similar as during the evaluation of the deterministic behaviour of the simulation software, also during the execution of the selfish mining scenarios the utilisation of the CPU and the memory of the host machine stayed under 10%. Thus, the specifications of the host machine did not restrict the simulations in any way.

CHAPTER 9

Discussion

relative gain but only if the share is really high

9.1 relative gain

selfish mining is working but only relative gain wait for difficulty adjustment what would happen to the price what would happen if somebody notices that someone is selfish mining combination with other attacks

9.2 performance

current simulation not performing that good because in case of match lambda is small extra hop selfish proxy needs to do some processing other nodes are all connected and have low latency using compact blocks

CHAPTER 10

Further research

10.1 Installation

CHAPTER 11

Introduction to LaTeX

Since LaTeX is widely used in academia and industry, there exists a plethora of freely accessible introductions to the language. Reading through the guide at https://en.wikibooks.org/wiki/LaTeX serves as a comprehensive overview for most of the functionality and is highly recommended before starting with a thesis in LaTeX.

11.1 Installation

A full LaTeX distribution consists of not only of the binaries that convert the source files to the typeset documents, but also of a wide range of packages and their documentation. Depending on the operating system, different implementations are available as shown in Table 11.1. **Due to the large amount of packages that are in everyday use and due to their high interdependence, it is paramount to keep the installed distribution up to date.** Otherwise, obscure errors and tedious debugging ensue.

11.2 Editors

A multitude of TeX editors are available differing in their editing models, their supported operating systems and their feature sets. A comprehensive overview of editors can

Distribution	Unix	Windows	MacOS
TeX Live	**yes**	yes	(yes)
MacTeX	no	no	**yes**
MikTeX	no	**yes**	no

Table 11.1: TeX/LaTeX distributions for different operating systems. Recomended choice in **bold**.

	Description
1	Scan for refs, toc/lof/lot/loa items and cites
2	Build the bibliography
3	Link refs and build the toc/lof/lot/loa
4	Link the bibliography
5	Build the glossary
6	Build the acronyms
7	Build the index
8	Link the glossary, acronyms, and the index
9	Link the bookmarks
	Command
1	`pdflatex.exe  example`
2	`bibtex.exe    example`
3	`pdflatex.exe  example`
4	`pdflatex.exe  example`

5	`makeindex.exe -t example.glg -s example.ist`
	`-o example.gls example.glo`

6	`makeindex.exe -t example.alg -s example.ist`
	`-o example.acr example.acn`

7	`makeindex.exe -t example.ilg -o example.ind example.idx`
8	`pdflatex.exe  example`
9	`pdflatex.exe  example`

Table 11.2: Compilation steps for this document. The following abbreviations were used: table of contents (toc), list of figures (lof), list of tables (lot), list of algorithms (loa).

be found at the Wikipedia page `https://en.wikipedia.org/wiki/Comparison_of_TeX_editors`. TeXstudio (`http://texstudio.sourceforge.net/`) is recommended. Most editors support the scrolling the typeset preview document to a location in the source document by `Ctrl` clicking the location in the source document.

11.3 Compilation

Modern editors usually provide the compilation programs to generate Portable Document Format (PDF) documents and for most LaTeX source files, this is sufficient. More advanced LaTeX functionality, such as glossaries and bibliographies, needs additional compilation steps, however. It is also possible that errors in the compilation process invalidate intermediate files and force subsequent compilation runs to fail. It is advisable to delete intermediate files (`.aux`, `.bbl`, etc.), if errors occur and persist. All files that are not generated by the user are automatically regenerated. To compile the current document, the steps as shown in Table 11.2 have to be taken.

11.4 Basic Functionality

In this section, various examples are given of the fundamental building blocks used in a thesis. Many LaTeX commands have a rich set of options that can be supplied as optional arguments. The documentation of each command should be consulted to get an impression of the full spectrum of its functionality.

11.4.1 Floats

Two main categories of page elements can be differentiated in the usual LaTeX workflow: *(i)* the main stream of text and *(ii)* floating containers that are positioned at convenient positions throughout the document. In most cases, tables, plots, and images are put into such containers since they are usually positioned at the top or bottom of pages. These are realized by the two environments `figure` and `table`, which also provide functionality for cross-referencing (see Table 11.3 and Figure 11.1) and the generation of corresponding entries in the list of figures and the list of tables. Note that these environments solely act as containers and can be assigned arbitrary content.

11.4.2 Tables

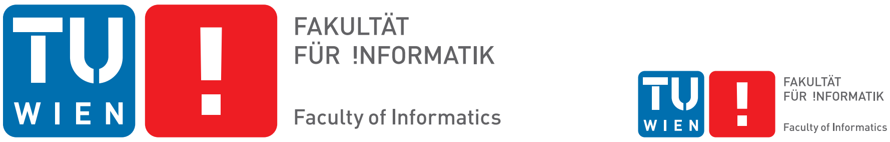
A table in LaTeX is created by using a `tabular` environment or any of its extensions, e.g., `tabularx`. The commands `\multirow` and `\multicolumn` allow table elements to span multiple rows and columns.

Position		
Group	Abbrev	Name
Goalkeeper	GK	Paul Robinson
Defenders	LB	Lucus Radebe
	DC	Michael Duburry
	DC	Dominic Matteo
	RB	Didier Domi
Midfielders	MC	David Batty
	MC	Eirik Bakke
	MC	Jody Morris
Forward	FW	Jamie McMaster
Strikers	ST	Alan Smith
	ST	Mark Viduka

Table 11.3: Adapted example from the LaTeXguide at `https://en.wikibooks.org/wiki/LaTeX/Tables`. This example uses rules specific to the `booktabs` package and employs the multi-row functionality of the `multirow` package.

11.4.3 Images

An image is added to a document via the `\includegraphics` command as shown in Figure 11.1. The `\subcaption` command can be used to reference subfigures, such as Figure 11.1a and 11.1b.



(a) The header logo at text width.

(b) The header logo at half the text width.

Figure 11.1: The header logo at different sizes.

11.4.4 Mathematical Expressions

One of the original motivation to create the TeX system was the need for mathematical typesetting. To this day, LaTeX is the preferred system to write math-heavy documents and a wide variety of functions aids the author in this task. A mathematical expression can be inserted inline as $\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$ outside of the text stream as

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

or as numbered equation with

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}. \tag{11.1}$$

11.4.5 Pseudo Code

The presentation of algorithms can be achieved with various packages; the most popular are `algorithmic`, `algorithm2e`, `algorithmicx`, or `algpseudocode`. An overview is given at `https://tex.stackexchange.com/questions/229355`. An example of the use of the `alogrithm2e` package is given with Algorithm 11.1.

11.5 Bibliography

The referencing of prior work is a fundamental requirement of academic writing and well supported by LaTeX. The BibTeX reference management software is the most commonly used system for this purpose. Using the `\cite` command, it is possible to reference entries in a `.bib` file out of the text stream, e.g., as [**?**]. The generation of the formatted bibliography needs a separate execution of `bibtex.exe` (see Table 11.2).

Algorithm 11.1: Gauss-Seidel

```
Input: A scalar $\epsilon$, a matrix $\mathbf{A} = (a_{ij})$, a vector $\vec{b}$, and an initial vector $\vec{x}^{(0)}$
Output: $\vec{x}^{(n)}$ with $\mathbf{A}\vec{x}^{(n)} \approx \vec{b}$
for $k \leftarrow 1$ to maximum iterations do
    for $i \leftarrow 1$ to $n$ do
        $x_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)} \right)$;
    end
    if $|\vec{x}^{(k)} - \vec{x}^{(k-1)}| < \epsilon$ then
        break for;
    end
end
return $\vec{x}^{(k)}$;
```

11.6 Table of Contents

The table of contents is automatically built by successive runs of the compilation, e.g., of `pdflatex.exe`. The command `\setsecnumdepth` allows the specification of the depth of the table of contents and additional entries can be added to the table of contents using `\addcontentsline`. The starred versions of the sectioning commands, i.e., `\chapter*`, `\section*`, etc., remove the corresponding entry from the table of contents.

11.7 Acronyms / Glossary / Index

The list of acronyms, the glossary, and the index need to be built with a separate execution of `makeindex` (see Table 11.2). Acronyms have to be specified with `\newacronym` while glossary entries use `\newglossaryentry`. Both are then used in the document content with one of the variants of `\gls`, such as `\Gls`, `\glspl`, or `\Glspl`. Index items are simply generated by placing `\index{`⟨*entry*⟩`}` next to all the words that correspond to the index entry ⟨*entry*⟩. Note that many enhancements exist for these functionalities and the documentation of the `makeindex` and the `glossaries` packages should be consulted.

11.8 Tips

Since TEX and its successors do not employ a What You See Is What You Get (WYSIWYG) editing scheme, several guidelines improve the readability of the source content:

- Each sentence in the source text should start with a new line. This helps not only the user navigation through the text, but also enables revision control systems (e.g. Subversion (SVN), Git) to show the exact changes authored by different users. Paragraphs are separated by one (or more) empty lines.

- Environments, which are defined by a matching pair of `\begin{name}` and `\end{name}`, can be indented by whitespace to show their hierarchical structure.
- In most cases, the explicit use of whitespace (e.g. by adding `\hspace{4em}` or `\vspace{1.5cm}`) violates typographic guidelines and rules. Explicit formatting should only be employed as a last resort and, most likely, better ways to achieve the desired layout can be found by a quick web search.
- The use of bold or italic text is generally not supported by typographic considerations and the semantically meaningful `\emph{...}` should be used.

The predominant application of the LaTeX system is the generation of PDF files via the PdfLaTeX binaries. In the current version of PdfLaTeX, it is possible that absolute file paths and user account names are embedded in the final PDF document. While this poses only a minor security issue for all documents, it is highly problematic for double blind reviews. The process shown in Table 11.4 can be employed to strip all private information from the final PDF document.

	Command
1	Rename the PDF document `final.pdf` to `final.ps`.

2	Execute the following command:
	`ps2pdf -dPDFSETTINGS#/prepress ^`
	`-dCompatibilityLevel#1.4 ^`
	`-dAutoFilterColorImages#false ^`
	`-dAutoFilterGrayImages#false ^`
	`-dColorImageFilter#/FlateEncode ^`
	`-dGrayImageFilter#/FlateEncode ^`
	`-dMonoImageFilter#/FlateEncode ^`
	`-dDownsampleColorImages#false ^`
	`-dDownsampleGrayImages#false ^`
	`final.ps final.pdf`

On Unix-based systems, replace # with = and ^ with \.

Table 11.4: Anonymization of PDF documents.

11.9 Resources

11.9.1 Useful Links

In the following, a listing of useful web resources is given.

`https://en.wikibooks.org/wiki/LaTeX` An extensive wiki-based guide to LaTeX.

`http://www.tex.ac.uk/faq` A (huge) set of Frequently Asked Questions (FAQ) about TeX and LaTeX.

`https://tex.stackexchange.com/` The definitive user forum for non-trivial LaTeX-related questions and answers.

11.9.2 Comprehensive TeX Archive Network (CTAN)

The CTAN is the official repository for all TeX related material. It can be accessed via `https://www.ctan.org/` and hosts (among other things) a huge variety of packages that provide extended functionality for TeX and its successors. Note that most packages contain PDF documentation that can be directly accessed via CTAN.

In the following, a short, non-exhaustive list of relevant CTAN-hosted packages is given together with their relative path.

algorithm2e Functionality for writing pseudo code.

amsmath Enhanced functionality for typesetting mathematical expressions.

amssymb Provides a multitude of mathematical symbols.

booktabs Improved typesetting of tables.

enumitem Control over the layout of lists (`itemize`, `enumerate`, `description`).

fontenc Determines font encoding of the output.

glossaries Create glossaries and list of acronyms.

graphicx Insert images into the document.

inputenc Determines encoding of the input.

l2tabu A description of bad practices when using LaTeX.

mathtools Further extension of mathematical typesetting.

memoir The document class on upon which the `vutinfth` document class is based.

multirow Allows table elements to span several rows.

pgfplots Function plot drawings.

pgf/TikZ Creating graphics inside LaTeX documents.

subcaption Allows the use of subfigures and enables their referencing.

symbols/comprehensive A listing of around 5000 symbols that can be used with LaTeX.

voss-mathmode A comprehensive overview of typesetting mathematics in LaTeX.

xcolor Allows the definition and use of colors.

List of Figures

List of Tables

List of Listings

Glossary

editor A text editor is a type of program used for editing plain text files.. 25, 31

. 31

. 31

. 31

Acronyms

CTAN Comprehensive TeX Archive Network. 31

FAQ Frequently Asked Questions. 31

PDF Portable Document Format. 26, 30, 31, 35

SVN Subversion. 30, 31

WYSIWYG What You See Is What You Get. 29, 31

Bibliography

[Bah13] Lear Bahack. Theoretical Bitcoin Attacks with less than Half of the Computational Power (draft). 2013.

[bita] Bitcoin - reference implementation of the bitcoin protocol. `https://github.com/bitcoin/bitcoin`. Accessed: 2017-06-21.

[bitb] Bitcoin - reference implementation release 0.15.0.1. `https://github.com/bitcoin/bitcoin/tree/v0.15.0.1`. Accessed: 2017-06-21.

[bitc] Bitcoin bips - bitcoin improvment proposals. `https://github.com/bitcoin/bips`. Accessed: 2017-06-21.

[Bitd] Thread about mining cartel attack on bitcointalk. `https://bitcointalk.org/index.php?topic=2227.0`. Accessed: 2017-06-21.

[bite] Bitcointicker - charts. `https://charts.bitcointicker.co/#miningpools`. Accessed: 2017-06-21.

[blo] Bitcoin hashrate distribution - blockhchain.info. `https://blockchain.info/en/pools`. Accessed: 2017-06-21.

[BMC$^+$15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *2015 IEEE Symp. Secur. Priv.*, pages 104–121. IEEE, may 2015.

[BS15] Alireza Beikverdi and Jooseok Song. Trend of centralization in Bitcoin's distributed network. In *2015 IEEE/ACIS 16th Int. Conf. Softw. Eng. Artif. Intell. Netw. Parallel/Distributed Comput. SNPD 2015 - Proc.*, pages 1–6. IEEE, jun 2015.

[Byt] User bytecoin on the mining cartel attack. `https://bitcointalk.org/index.php?topic=2227.msg30064#msg30064`. Accessed: 2017-06-21.

[coia] Coin dance | bitcoin nodes summary. `https://coin.dance/blocks#thisweek`. Accessed: 2017-06-21.

[coib] Coin dance | bitcoin nodes summary. `https://coin.dance/nodes`. Accessed: 2017-06-21.

[DW13] Christian Decker and Roger Wattenhofer. Information Propagation in the Bitcoin Network. 2013.

[ES14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 2014.

[GKCC14] Arthur Gervais, Ghassan O Karame, Srdjan Capkun, and Vedran Capkun. Is Bitcoin a Decentralized Currency? 2014.

[GKW$^+$16] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjaň Capkun. On the Security and Performance of Proof of Work Blockchains. 2016.

[GRK15] Arthur Gervais, Hubert Ritzdorf, and Ghassan O Karame. Tampering with the Delivery of Blocks and Transactions in Bitcoin. 2015.

[Hei14] Ethan Heilman. One weird trick to stop selfish miners: Fresh bitcoins, a solution for the honest miner (Poster Abstract). In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 8438, pages 161–162, 2014.

[Mar] Cryptocurrency market capitalizations. `https://coinmarketcap.com/`. Accessed: 2017-06-21.

[Nak08] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

[NKMS16] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi @bullet. Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack. 2016.

[SPB16] Siamak Solat and Maria Potop-Butucaru. ZeroBlock: Timestamp-Free Prevention of Block-Withholding Attack in Bitcoin. 2016.

[SSZ16] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal Selfish Mining Strategies in Bitcoin. 2016.

[TS16] Florian Tschorsch and Bjorn Scheuermann. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Commun. Surv. Tutorials*, 18(3):2084–2123, 2016.

[uni] Release of uniform tie breaking in ethereum. `https://github.com/ethereum/go-ethereum/commit/bcf565730b1816304947021080981245d084a930`. Accessed: 2017-06-21.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[ZP17] Ren Zhang and Bart Preneel. Publish or Perish: A Backward-Compatible Defense against Selfish Mining in Bitcoin. 2017.