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 its 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 its own chain. If its 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 its 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, GRKC15, 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 [GRKC15, 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 framework 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, GRKC15] 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.

The 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 between the participating nodes. For the underlying network a realistic scenario with a certain amount of nodes and network topology is defined.

The result of the executed simulations shows which strategy is the best strategy for a certain distribution of mining power between the selfish miner and the honest network. Furthermore, the relative and total gain of the selfish miner in the different simulation scenarios is observed. The total gain of a miner describes the total amount of received mining rewards, where the relative gain describes the received share of the mining rewards. In the optimal case, where all nodes behave honestly and all miner have the same connection to the network the relative gain of a miner is equal to its computational share. Hence, each miner receives its fair shares of mining rewards. In the simulations is shown that the selfish miner can increase its relative gain by executing a selfish mining strategy, but at the same time its total gain decreases. This is possible because the execution of the selfish mining decreases the amount of blocks which end up in the longest chain. Thus, less mining rewards are distributed between the miners which reduces also the total gain of the selfish miner. To be able to raise the total gain the selfish miner would need to wait for the difficulty adjustment which in Bitcoin happens every two weeks. The attack scenario where the selfish miner waits for the difficulty adjustment is not part of this thesis.

The outcome of the thesis is further analysed by answering 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 framework. 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 framework could not only be used to simulate selfish mining attacks but could for example be used to simulate other attacks or new protocol versions of Bitcoin. Since many other cryptocurrencies are derived from Bitcoin, they simulation framework could also be utilized to simulate their behaviour and properties.

1.1 Structure of this thesis

First, the different strategies selfish, lead stubborn, trail stubborn and equal-fork stubborn mining from [NKMS16] and [ES14] are 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 its 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 are 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 also controls 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 is 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 distributions of computation power between the selfish miner and the honest network are used to find the best selfish mining strategies for different scenarios. Furthermore, the results of the simulation are compared with previous research.

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 [Bitc, 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, GRKC15, GKW+16, Bah13]. [NKMS16] provided a comprehensive description of the strategy space and also coined different names for the selfish mining variations:

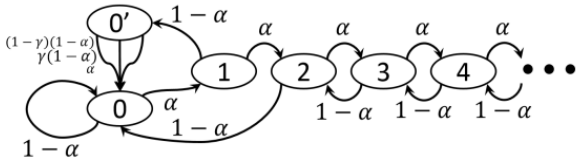

Figure 2.1: Selfish mining state machine with transition probabilities [ES14]

- **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, GRKC15, GKW$^+$16, Bah13].

To find the best strategy for a given mining power share α and connectivity γ researchers used different methodologies. [GRKC15, 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$^+$16, SSZ16, NKMS16, GRKC15]. If the eclipse attack is used in combination with selfish mining the victim contributes its mining power to the private chain and hence, strengthens the position of the selfish miner [NKMS16, GKW$^+$16]. [NKMS16] additionally shows that the eclipsed victim under certain circumstances can benefit from the attack and therefore has no incentive to stop the attack. Another attack which can be used in combination with selfish mining is double-spending [SSZ16, GKW$^+$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, Bil15, 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 [Bil15]. 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 [Bil15]. [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, bitb]. The countermeasures against selfish mining are forming an interesting research field but are not in the focus of this thesis. Nevertheless, all of them would profit from an evaluation method introduced in the next chapter.

CHAPTER 3

Simulation framework

The simulation framework provides all needed functionalities to orchestrate a peer-to-peer network where the nodes are running the *Bitcoin* reference implementation. The whole simulation runs on a single host using the virtualisation software *Docker*. The software furthermore coordinates the block discovery in the network. Based on a sequence defined in a configuration file the software sends commands to the nodes which are then generating valid blocks. To be able to create blocks the nodes are executed in the *regtest mode*, where the CPU-heavy proof-of-work is disabled, and the nodes are accepting a RPC-call from outside which lets them create a new block immediately. After a simulation run, the software parses the logs produced by the nodes and based on them the software generates a report which displays the key metrics of the simulation.

go a little into the details what the rational behind this desgin was. eg. predefined sequence.

The remaining of this chapter starts with the sub-chapter 3.1, where the general concept of a tick is introduced. Afterwards in the chapter 3.2 the different configuration files are explained. In the chapter 3.3 the whole procedure of a simulation with the three main steps preparation, execution and post-processing is explained. Lastly, in the chapter 3.4 the implemented commands to execute the simulation framework are listed.

3.1 Tick

A fundamental 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. Hence a tick forms an abstraction of a certain time span in the real time. All blocks which should be generated in this time span are aggregated into the corresponding tick. Since all blocks created in the same tick are considered to be created at the same time, a tick should never contain the information that one node creates multiple blocks.

For a simulation run, multiple ticks are generated forming a sequence of ticks. This sequence is the simulation scenario for a particular simulation run. The exact duration of a tick is defined on execution time of the simulation and determines the actual speed of the simulation. The concept of a tick helps to have an upper bound for the simulation speed. An upper bound of a certain sequence of ticks is reached when the execution time of at least one tick last longer than the tick duration itself. In that case, the temporal succession of the block events is disturbed resulting in inaccurate or wrong results, and the simulation speed should be lowered.

3.2 Configuration files

A simulation executed by the software 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 processed by the simulation framework. The usage of configuration files as input for a simulation provide the flexibility that they can be written manually or that they can be generated by a small script. Also a combination of both is possible. For example, a small script generates a needed configuration file. Afterwards, the user can adjusted the configuration file by editing it and does not need to implemented a own script for the specific scenario, but does also no need to write the whole configuration file by hand. Furthermore, the created configuration files can be copied to the output directory of a simulation run providing easy method to preserve all input parameters used for run.

The simulation framework already implements for each configuration file a simple script which can be executed by the corresponding commands *nodes* (chapter 3.4.1), *network* (chapter 3.4.2) and *ticks* (chapter 3.4.3).

3.2.1 *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 consisting of:

- *node_type*: Either *bitcoin* if the node is a normal node or *selfish* if the node should act like a selfish node.
- *share*: The computation power proportion of the node in the network.
- *docker_image*: The *Docker* image to use for the node.
- *latency*: The latency of the node in the peer-to-peer network. The node will have this latency to all other nodes with the exact same latency. For other connections the two different latencies are added. Hence, two nodes, one node with 100 milliseconds of latency and one node with 50 milliseconds, will have 150 milliseconds of latency during the simulation.

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

Table 3.1: An example *network.csv* represented as table

- *group*: Which group the node did belong to during the creation of the *nodes.csv* by the script introduce in chapter 3.4. During the actual execution of the simulation this parameter is ignored.

3.2.2 *network.csv*

The *network.csv* reflects a connection matrix as shown in table 3.1. 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 1.

3.2.3 *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 length of lines in a *ticks.csv* varies depending on the number of block events in the corresponding tick.

3.3 Simulation

The main functionality of the simulation framework is to coordinate a simulation based on the configuration files. The software uses therefore the high-level programming language *Python* and the virtualisation software technology *Docker*. *Python* is mainly used to handle the configuration files and to execute *Docker* and other binaries whenever necessary automatically. *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. Furthermore by using *Docker* all containers use the system time of the host machine. This is especially helpful for the aggregation of the log files of all different containers, because it assures that the timestamps in the log lines actually happened in exactly this order.

To execute a simulation the command *simulate* (chapter 3.4.4) can be used. In that case the configuration files need to be available on disk. The commands *run* (chapter

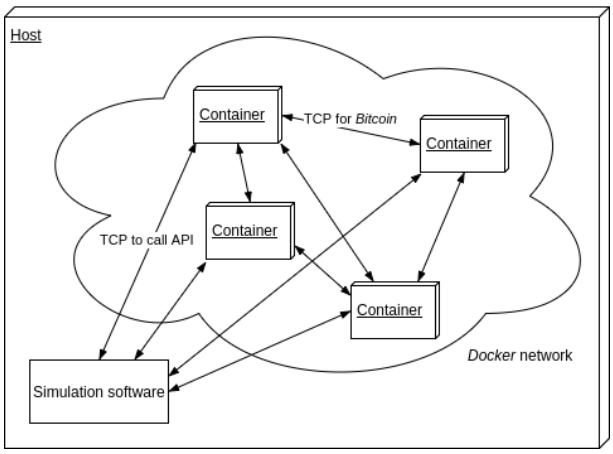

Figure 3.1: Overview of the virtual peer-to-peer network

3.4.5) and *multi-run* (chapter 3.4.6) provide another possibility to execute a simulation, creating all configuration files before starting the simulation.

3.3.1 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 assembled as depicted in figure 3.1. 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 3.1 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-enabled 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.

```
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 3.1: Simplified version of how a node is started with *Docker* and *bitcoind*

3.3.2 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. The sequential execution of the block events is preferred over a parallelisation, which would introduce a new source of uncontrollable indeterminism making the results harder to reproduce even on the same host.

A simulation run is always executed with a certain tick duration. This tick duration specifies how long a tick should last in real time. 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. If the completion of the block events last longer then the tick duration, then the framework immediately starts with the next tick and tries to regain the lost time. The case where the execution of block events last longer then the tick duration, should be a rare exception. If this happens more frequent the simulation scenario is configure too fast, which likely creates inaccurate results as explained in chapter 3.1. Thus, the defined scenario should be created with less block events per tick or a higher tick duration should be configured on simulation start. The simulation framework warns the user about the total amount of exception of this type in the final report.

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* in */proc/meminfo* file. This value provides a heuristic of

the current available memory on the machine.

3.3.3 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 itself. 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. To 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 its 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 they 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 like the memory usage. The *skip_ticks* parameter allows the user to discard ticks which may created distorted data. Right after the start it is likely that the nodes behave differently because the execute some initializing routines or run faster because they just started up. At the end of the simulation run it makes sense to ignore some ticks because blocks created at the end of the simulation would distort the analysis of the simulation. Those blocks probably propagate faster, because right after them no other competing blocks are created. Another problem could be that the simulation frameworks already starts to shut down the peer-to-peer network even though some blocks are still transmitted to other nodes. How many ticks should be omitted depends on the simulation scenario and the right amount is difficult to determine. An additional mitigation is to extend the overall simulation, which would reduce the influence of the distorted data from the first and last ticks.

Additionally to omitting some ticks, 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 information about blocks created during the simulation
- charts and information about exceptions happened during the simulation

Most of the stated information and charts can be directly generated from the CSV files with respective R commands. Only the stale block rate and the propagation time of blocks needs some further processing, which is executed on report creation. The stale block rate and propagation time of blocks needs to be calculated in the report. todo[inline]The term stale block rate should be explained in the State-of-the-Art section when talking about the other simulations - this is a key metric that is important and it should be clear why - maybe also cite Eyals Bitcoin NG

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 3.2. 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 3.2: Calculation of propagation time with *R*

3.3.4 Multi-runs

When the simulation framework is executed with the *multi-run* command (chapter 3.4.6) 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 framework into an own directory. Once the last simulation finishes the software 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 a single simulation, is executed to create a report comparing all simulation runs.

3.4 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.

3.4.1 *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.

3.4.2 *network* command

When the simulation framework 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*.

3.4.3 *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 needs to be available, to determine all planned nodes and their computational share (*share*). Afterwards the software parametrises for each node an exponential distribution as shown in 3.1 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 procedure is repeated for every tick until reaching *amount_of_ticks*. For example if we calculated the five 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 five ticks, where in the 1st tick are two block events for node A, and in the 3rd tick and 5th tick one each. The 2nd and the 4th tick would stay empty. After calculating all ticks, the ticks are persisted in the *ticks.csv*.

$$f(x;\lambda) = \begin{cases} 1 - \exp(-\lambda x) & \text{x} \geq 0, \\ 0 & \text{x} < 0 \end{cases} \tag{3.1}$$

3.4.4 *simulate* command

On execution of the *simulate* command the simulation framework starts a simulation based on the configuration files *nodes.csv*, *network.csv* and *ticks.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 3.3.

3.4.5 *run* command

When the simulation framework 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.

3.4.6 *multi-run* command

The *multi-run* command accepts an input parameter denoting how often a run should be repeated (*repeat*). The simulation framework 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 3.2.

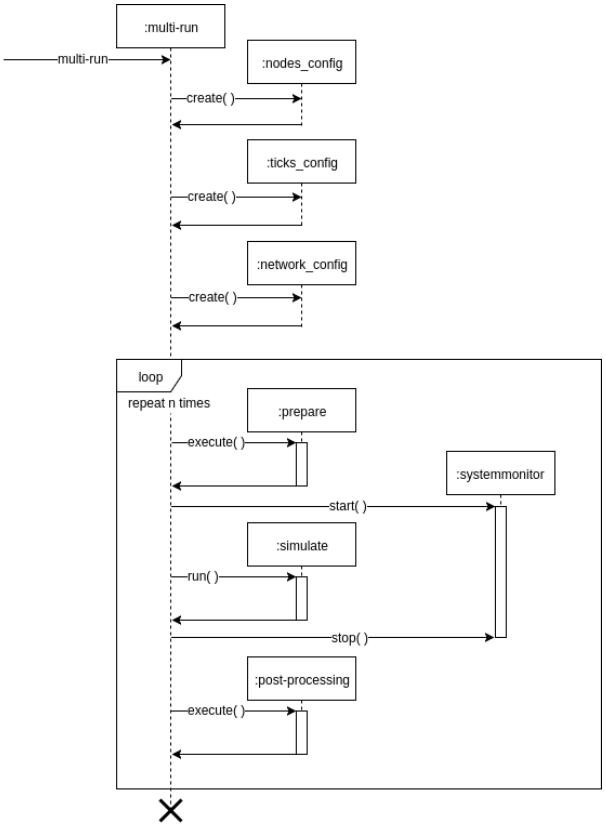

Figure 3.2: Sequence diagram of the *multi-run* command

You could also add some notes on ZFS and alignment - this is relevant for long time storage of simulation data. This is perfectly valid since this chapter is dealing with engineering questions anyway.

CHAPTER 4

Evaluation of simulation software

The simulation framework aims to simulate different simulation scenarios based on supplied configuration files. To be able to trust in the results of such a simulation, the outcome needs to be similar to each repeating execution of particular ticks. During a simulation run, the different nodes are not synchronised in any manner, and the orchestrated containers run entirely independently from each other. For example, the order of low-level operations executed by the different nodes on the host machine depends on many various parameters and circumstances and is likely to change in every simulation run. This natural indeterminism introduced by how the simulation framework works is unchangeable without breaking the whole architecture of the simulation framework. Hence, the results of multiple executions of a particular tick sequence are only evaluated against a specific similarity and not if the outcome is exactly the same.

4.1 Deterministic behaviour

The simulation framework is assessed against a defined deterministic behaviour to have confidence in single executions of particular simulation scenario. The software is considered to behave deterministically if:

The standard deviation of the stale block rate is lower than 0.2% after sufficient executions of the reference scenario.

In this definition solely the stale block rate is used to define a deterministic behaviour because it reflects the outcome of a simulation comprehensively. All configuration parameters of a simulation run influence the stale block rate directly or indirectly[GKW+16]:

- The tick sequence reflects the computational share of the nodes and the block interval time of the overall network. Changes to the tick sequence yield in less or

more block races in the peer-to-peer network. The stale block rate condenses the outcome of these block races.

- The latency in the peer-to-peer network directly influences the propagation time of blocks. With lower latency nodes can faster adopt the currently highest available block, where on the other hand with higher latency nodes more likely create competing blocks and cause block races. The block races are causing then stale blocks reflected by the stale block rate.
- The network topology defines how the peer-to-peer network is connected. Hence, the topology impacts how fast or slow blocks are propagating in the network which impacts the stale block rate.

The definition of the deterministic behaviour is subjective and needs to be considered when trying to conclude something from the results of a simulation framework with such a deterministic behaviour.

4.2 Reference scenario

The reference scenario used to evaluate the simulation framework for its deterministically behaviour, tries to abstract the real *Bitcoin* network. The mining process in the real system is known to be centralised by a few mining pools [**?**, **?**, **?**, **?**]. Current statistics show that about twenty miners create almost every block [**?**, **?**, **?**]. Hence, in the reference scenario a total of twenty nodes are used. These nodes are all directly connected to each other with a latency of 25 milliseconds. The configuration of network topology with direct, fast connections is based on the assumption that every miner wants to hear about new blocks as fast as possible. The rapid propagation time of the blocks reduces the number of stale blocks for each participating peer and thus, increases the revenue gained from the block rewards. For the latency of the connections additionally an upper bound from previous research was taken into account [**?**].

For the sake of simplicity all nodes part of the network run the same version 0.15.0.1 [**?**] of the *Bitcoin* reference implementation. This adjustment is in contrast to the real network where different implementation and version are used [**?**]. Another simplification is taken for the virtual distribution of the computational power in the system. In the reference scenario, all participating nodes receive the same amount of computational power. Consequently, the probability that a node finds the next block is the same for every particular node at every point in time. In the real network, the distribution of the computational power is unevenly, and about five mining pools mine over fifty percent of all blocks [**?**, **?**, **?**, **?**].

The duration of the simulation itself is set to contain about 2016 blocks which correspond to two weeks in the real *Bitcoin* network. This time span is also identical to the time used for the difficulty adjustment of the proof-of-work in *Bitcoin* which is relevant for the simulation of selfish mining. The needed tick sequence is generated by using the

ticks command described in section 3.4.3. The command uses an exponential distribution to determine when a specific node should find a block. The exponential distribution realistically mimics the block intervals caused by the cryptographic proof-of-work puzzle normally executed by every node to search a new block [Nak08, **?**, ES14]. In the reference scenario, the simulation is divided into 0.1 second long ticks. Hence, by setting the blocks per tick to $0.0\dot{3}$, the simulated network finds every three seconds a block. Compared to *Bitcoin*, wherein average every ten minutes a block is created, this results in a 200 times shorter block interval, and instead of two weeks, it takes 100.8 minutes to simulate about 2016 blocks.

4.3 Evaluation

For the evaluation of the deterministic behaviour the reference scenario was executed 100 times to obtain sufficient figures of the stale block rate. The individual simulations were conducted on a *x86 Linux* host machine with *Ubuntu 4.4.0-97-generic* installed. The CPU of the host machine was virtualised with *QEMO 2.5+* and provided 16 cores. Furthermore, the machine provided 57.718 GB of memory. During a particular simulation run, about 4% of CPU and 6% of the storage were utilised as shown in the corresponding figures 4.1 and 4.2. The metrics about the usage of the CPU and storage were continuously collected by the simulation framework as described in chapter 3.4.

The outcome of the evaluation is shown as box plot in figure 4.3 and as a density plot in 4.4. Despite one outlier at 4.521% all data points are inside the whiskers area defined by $[Q1 - 1.5IQR, Q3 + 1.5IQR]$ with the median value equal 4.821%. Even tough the stale block rate is a continuous variable, the calculated 100 values are accumulated at certain values because the fixed amount of blocks only allows certain combinations of accepted and stale blocks.

To evaluate if the simulation framework has a deterministic behaviour as defined in chapter 4.1 the standard deviation is calculated. The calculation of the standard deviation results in 0.159%. Hence, the values satisfies the defined deterministic behaviour, and the simulation framework can be assumed to behave deterministically as defined.

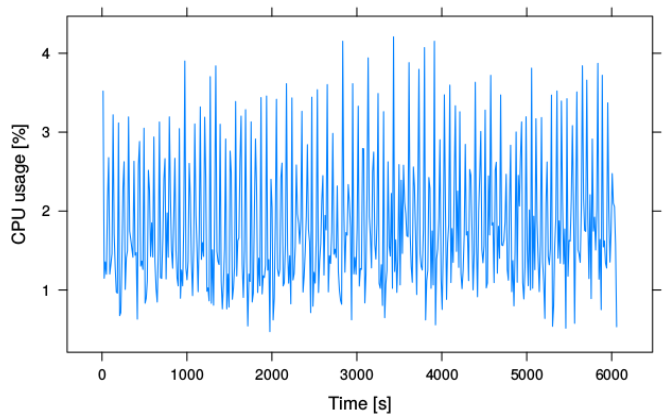

Figure 4.1: CPU usage during a particular simulation run

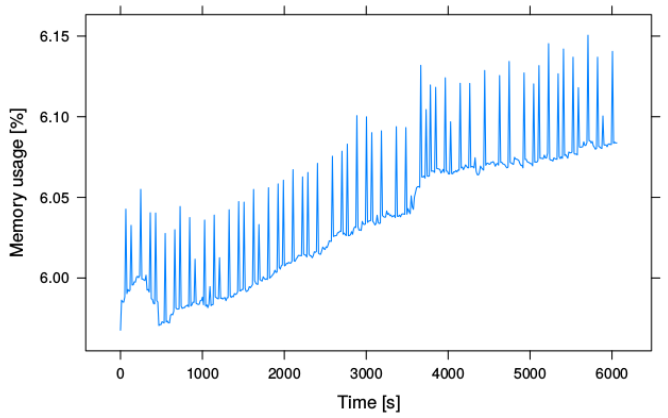

Figure 4.2: Memory usage during a particular simulation run

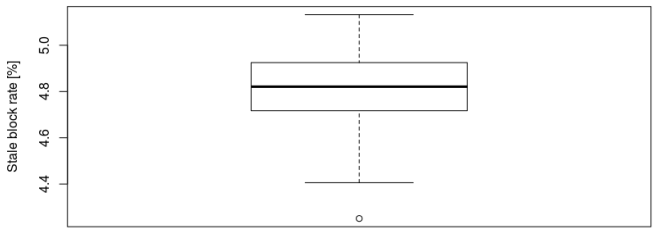

Figure 4.3: Box plot of the stale block rate of 100 executions

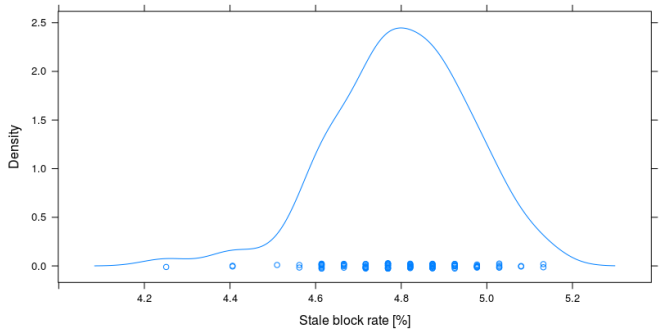

Figure 4.4: Density plot of the stale block rate of 100 executions

CHAPTER 5

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 5.1. 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.

5.1 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 5.1 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 framework. 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.

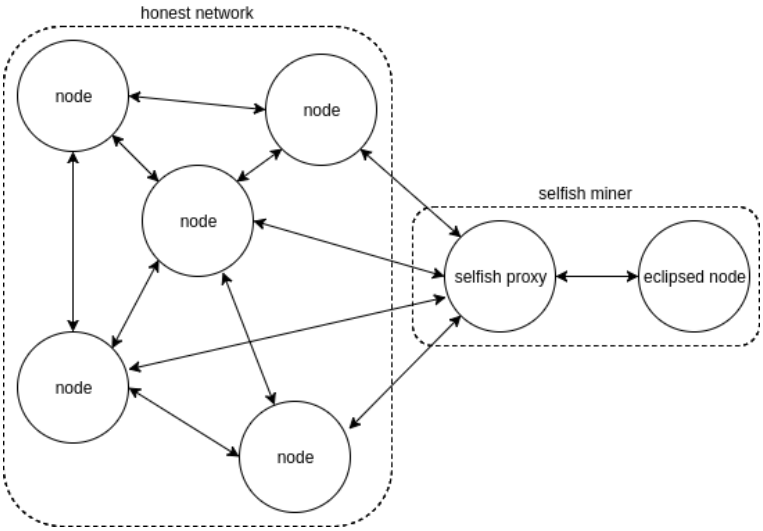

Figure 5.1: Selfish proxy eclipsing a normal node

Is it possible to have multiple eclipsed nodes? I guess not - but then the selfish mining proxy can directly be used as an attack tool by a MiTM attacker. Maybe this should be mentioned

The implementation of all network related functionalities of the selfish proxy is based on the two open-source libraries *pycoin* [pyc] and *python-bitcoinlib* [pyt]. 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.

5.2 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 assumed to behave honestly in our simulation 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

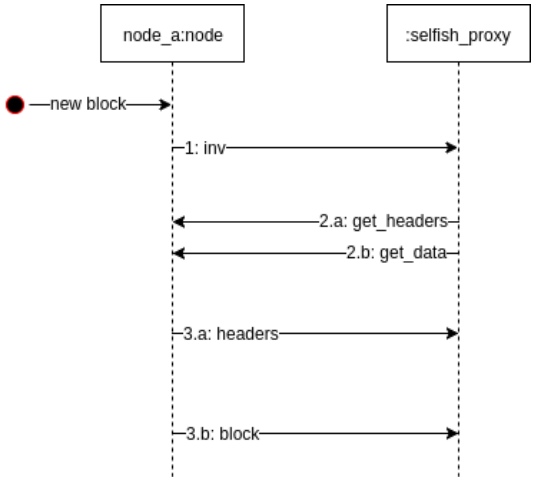

Figure 5.2: Selfish proxy receiving a block from another node

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.

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.

5.3 Receiving blocks

An essential capability of the selfish proxy is to retrieve blocks and block headers from its peers. Figure 5.2 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 5.2.

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 5.2. 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.

5.4 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 5.3 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

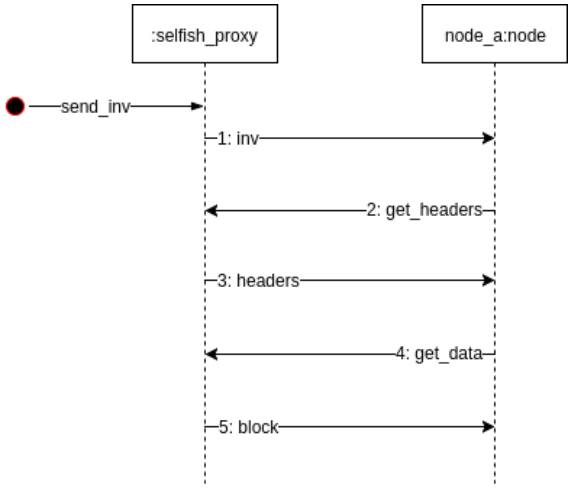

Figure 5.3: Selfish proxy sending a block to another node

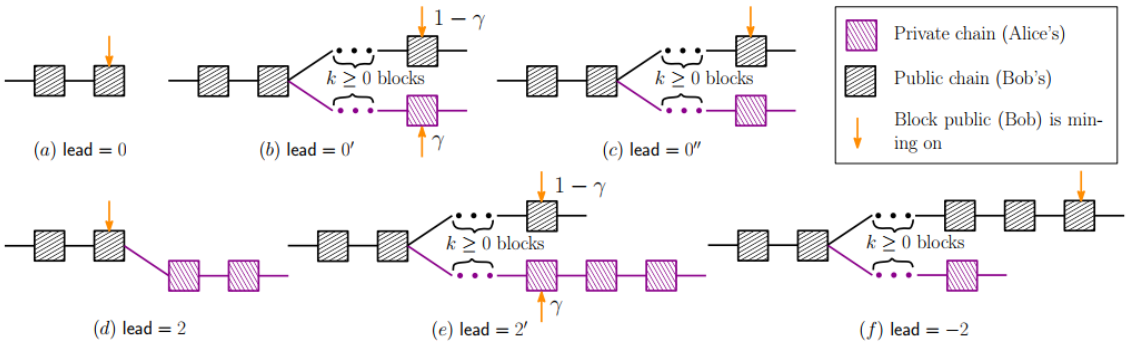

Figure 5.4: Different possible leads of the private chain [NKMS16]

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.

5.5 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 5.1. 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.

5.5.1 Private lead

Every time a block header is inserted in the chain the proxy checks if either the public or private chain was 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 5.4. 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.

5.5.2 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

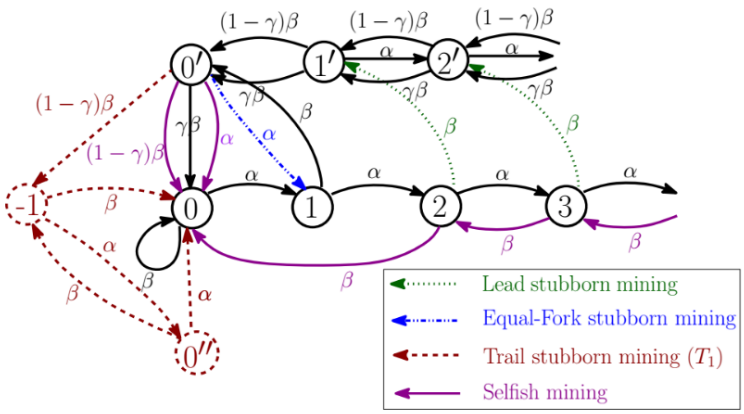

Figure 5.5: Categorization of different mining strategies [NKMS16]

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.

- **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.

5.5.3 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 the selfish mining strategy described by [ES14] (abbreviated with H) and can be modified with the three modifications lead (L), equal-fork (F) and trail stubborn (T) by [NKMS16]. Figure 5.5 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 (*L*)**: 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 (*F*)**: The equal-fork stubborn mining strategy changes the behaviour of the selfish miner during a block race in state *0'*. Usually, 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 (*T*)**: 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 the public chain and trails back leading into state *-1*. In the case that the selfish miner can 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 parametrised 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 parametrised the build an infinite strategy space.

Some comment as in Introduction, explain relative gain

Furthermore, the strategies can be combined and since the trail stubbornness can be parametrized the build an infinite strategy space

5.5.4 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 have 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 behaviour shown in the state diagram from figure 5.5 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 5.1 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 discovered.

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 5.1: 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 happening, 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.

5.5.5 Configuration

The selfish proxy started with any configuration executes the standard selfish mining strategy with no modifications. 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 determining 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 specifying how much the selfish proxy should trail back.

CHAPTER 6

Simulation of selfish mining strategies

With the introduced software solutions, the simulation framework and the selfish proxy, it is now possible to analyse selfish mining and its impact on the relative gain of the selfish miner. For the simulation, the scenario described in chapter 4.2 is adapted. One of the twenty nodes is eclipsed with the selfish proxy forming a selfish miner. To obtain a comprehensive overview of the impact of selfish mining the selfish miners conducts various combinations of selfish mining strategies. Additionally, different distributions of the computation power between the nodes and the selfish miner are applied.

For the twenty nodes you can acutally cite the latest work from sirer `https://arxiv.org/pdf/1801.03998.pdf`

6.1 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 executed during the simulation.

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.

Why not the values 25,33 which are the borders for security in certain studies

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. All possible scenarios where the selfish miner would receive more than 50% are omitted because in that cases for the selfish miner it would be more efficient to launch the so-called 51%-attack copping all mining rewards [Nak08, **?**, **?**]. Additionally, the scenario with a share of 7.5% is discarded because it is very likely that in that case, selfish mining has no advantages as already shown in previous studies [ES14, SSZ16, NKMS16].

With eight different mining strategies and five different distributions of computation power, a total of 40 different scenarios are obtained. Listing 6.1 shows how a specific scenario is started with the simulation framework. 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 mining power (line 3). As shown 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 framework to the selfish proxy when it gets created. Furthermore, it can be seen in line 5 that the strategy modification trail stubborn is set to 1. 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 4.2.

```
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 6.1: Command to execute a particular selfish mining scenario

6.2 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 framework in chapter 4.3. Each scenario gets executed three times by using the *multi-run* command as shown in line 1 and 2 in the listing 6.1. To extract a particular metric from the multiple executions of a scenario, the simulation with the median stale block rate is used. Since the simulation framework can not behave perfectly deterministic due its architecture, the median provides a robust method against possible outliers and hence, more accurate results are achieved.

strategy	share	blocks honest	blocks selfish	stale blocks honest	stale blocks selfish	share selfish	share stale selfish	stale block rate
H	0.150	1560.6	275.4	79.050	13.950	0.15000000	0.1500000	0.04821151
H	0.225	1422.9	413.1	72.075	20.925	0.22500000	0.2250000	0.04821151
H	0.300	1285.2	550.8	65.100	27.900	0.30000000	0.3000000	0.04821151
H	0.375	1147.5	688.5	58.125	34.875	0.37500000	0.3750000	0.04821151
H	0.450	1009.8	826.2	51.150	41.850	0.45000000	0.4500000	0.04821151
S	0.150	1536	78	108	193	0.04832714	0.6411960	0.15718016
S	0.225	1350	166	149	237	0.10949868	0.6139896	0.20294427
S	0.300	1116	328	232	231	0.22714681	0.4989201	0.24278972
S	0.375	919	442	299	271	0.32476120	0.4754386	0.29518384
S	0.450	624	601	446	249	0.49061224	0.3582734	0.36197917
L	0.150	1538	79	106	192	0.04885591	0.6442953	0.15561358
L	0.225	1350	160	149	243	0.10596026	0.6198980	0.20609884
L	0.300	1126	301	222	258	0.21093203	0.5375000	0.25170425
L	0.375	931	415	287	298	0.30832095	0.5094017	0.30295184
L	0.450	648	538	422	312	0.45362563	0.4250681	0.38229167
F	0.150	1542	70	102	201	0.04342432	0.6633663	0.15822454
F	0.225	1356	152	143	251	0.10079576	0.6370558	0.20715037
F	0.300	1120	314	228	245	0.21896792	0.5179704	0.24803356
F	0.375	921	417	297	296	0.31165919	0.4991568	0.30709477
F	0.450	597	597	473	253	0.50000000	0.3484848	0.37812500
T_1	0.150	1534	81	110	190	0.05015480	0.6333333	0.15665796
T_1	0.225	1350	163	149	240	0.10773298	0.6169666	0.20452156
T_1	0.300	1119	325	229	234	0.22506925	0.5053996	0.24278972
T_1	0.375	921	441	297	272	0.32378855	0.4780316	0.29466598
T_1	0.450	648	575	422	275	0.47015536	0.3945481	0.36302083
L, F	0.150	1543	65	101	206	0.04042289	0.6710098	0.16031332
L, F	0.225	1362	136	137	267	0.09078772	0.6608911	0.21240799
L, F	0.300	1160	254	188	305	0.17963225	0.6186613	0.25852124
L, F	0.375	981	306	237	407	0.23776224	0.6319876	0.33350596
L, F	0.450	733	375	337	475	0.33844765	0.5849754	0.42291667
L, T_1	0.150	1537	77	107	194	0.04770756	0.6445183	0.15718016
L, T_1	0.225	1353	153	146	250	0.10159363	0.6313131	0.20820189
L, T_1	0.300	1131	302	217	257	0.21074669	0.5421941	0.24855794
L, T_1	0.375	922	420	296	293	0.31296572	0.4974533	0.30502330
L, T_1	0.450	648	537	422	313	0.45316456	0.4258503	0.38281250
F, T_1	0.150	1540	68	104	203	0.04228856	0.6612378	0.16031332
F, T_1	0.225	1353	151	146	252	0.10039894	0.6331658	0.20925342
F, T_1	0.300	1119	317	229	242	0.22075209	0.5138004	0.24698479
F, T_1	0.375	911	429	307	284	0.32014925	0.4805415	0.30605904
F, T_1	0.450	613	581	457	269	0.48659966	0.3705234	0.37812500
L, F, T_1	0.150	1545	62	99	209	0.03858121	0.6785714	0.16083551
L, F, T_1	0.225	1358	138	141	265	0.09224599	0.6527094	0.21345952
L, F, T_1	0.300	1160	255	188	304	0.18021201	0.6178862	0.25799685
L, F, T_1	0.375	1000	294	218	419	0.22720247	0.6577708	0.32988089
L, F, T_1	0.450	720	389	350	461	0.35076646	0.5684340	0.42239583

Table 6.1: Results of the 40 simulations with the additional honest behaviour H

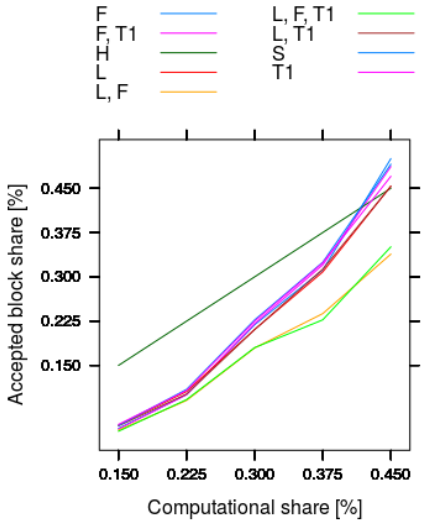

Figure 6.1: Relative revenue of the selfish miner

6.3 Results

The results of the 40 simulations are shown in table 6.1. In the first column, the different selfish mining strategies and its combinations are listed in abbreviated form. The second column reflects the computational share of the selfish miner during each specific simulation. The two columns *blocks honest* and *blocks selfish* contain the number of blocks from each party which ended up in the longest chain. The following two columns reflect the number of stale blocks for the honest network and the selfish miner. The columns *share selfish* and *share stale selfish* are derived from the previous columns and describe the relative proportion of accepted and stale blocks for the selfish miner. Lastly, in the ninth column, the overall stale block rate of each simulation run is stated. Additionally, to the 40 results, the simulation with the median stale block rate from the evaluation of the simulation framework in chapter 4.3 is added with the abbreviation *H*. Since in that simulation the computational share was always distributed evenly amongst all nodes the result of the simulation is multiplied by the corresponding share for each defined distribution of computation power.

In figure 6.1 the relative revenue, given a particular computation share and selfish mining strategy, is shown. The graph shows that the two strategies selfish mining with *lead stubborn, equal-fork stubborn* (*L, F*) and selfish mining with all three modifications (*L, F, T_1*) are the worst performing strategies. The relative share of accepted blocks of the two strategies stays clearly under the relative proportion which could be obtained by behaving honestly. Also with an increase of the computational power the efficiency of the two selfish strategies is not amplified and the curve shows a linear progression. The other six, better performing strategies are all exhibiting a similar behaviour. The performance of these strategies is intensified with the augmentation of the computational

strategy	share selfish	rank	difference to best
F	50%	1	-
S	49.1%	2	0.9%
F, T_1	48.7%	3	1.3%
T_1	47%	4	3%
L	45.4%	5	4.6%
L, T_1	45.3%	6	4.7%
H	45%	7	5%
L, F, T_1	35.1%	8	14.9%
L, F	33.8%	9	16.2%

Table 6.2: Relative share of selfish miner with 45% of computational share

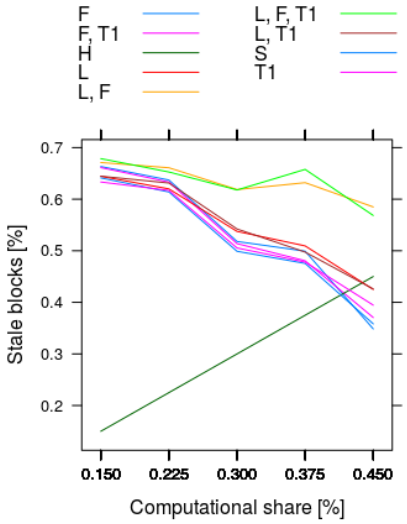

Figure 6.2: Share of stale blocks created by the selfish miner

share underlined by a concave curve. Overall all selfish mining strategies are performing poorly and only with a computational share of about 40% the six better performing strategies can retain a more significant share than the share obtained behaving honestly. In table 6.2 the relative revenue share of the selfish miner with 45% of computation power is shown. The best performing strategies are equal-fork stubbornness (F) and normal selfish mining (S) followed by trail stubborn mining modified with equal-fork stubbornness F, T_1 and the strategy trail stubborn (T_1). These four strategies achieve almost similar results and are only 3% apart.

Figure 6.2 shows the share of stale blocks found by the selfish miner. In the honest case, the share of created stale blocks increases linearly with the computational share. Contrary to the line showing the honest behaviour proceed the lines depicting the

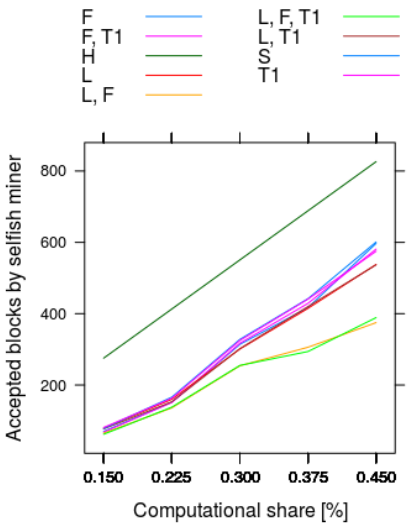

Figure 6.3: Blocks created by the selfish miner and accepted in the longest chain

different selfish mining strategies. Especially if the computational share is low, over 60% of the stale blocks of the network are created by the selfish miner. With an increasing share of computational power, the share of stale blocks declines significantly, except the two strategies lead stubborn combined with equal-fork stubborn (L, F) and selfish mining modified with all stubborn variations (L, F, T_1) which remain on the same level. Additionally, the figure shows that only with a very high amount of computational share the selfish strategies are achieving better results than the normal, honest behaviour.

The total amount of accepted blocks by the selfish miner, given a particular strategy and computational share, is shown in figure 6.3. The graph shows again the gap between the two bad performing strategies and the six other strategies which work slightly better. Additionally, it can be seen that the absolute amount of accepted blocks during the execution of selfish mining is significantly lower than the number of blocks accepted when the nodes behave honestly. Hence, in the short-run, all selfish mining strategies are yielding less mining rewards for the selfish miner than the normal, honest behaviour.

The honest network also creates less accepted blocks during a selfish mining attack as shown in figure 6.4. Since the selfish mining strategies are functioning better with an increasing computational share of the selfish miner, the gap between the case where all nodes behave honestly and the case where one node conducts selfish mining even increases. The performance differences between the selfish mining strategies can also be seen in this figure.

Lastly, in figure 6.5 the relative amount of stale blocks in the network during the simulations is shown. If all nodes behave honestly, the stale block rate is 4.821% as measured during the evaluation of the simulation framework. In the case that a node conducts a selfish mining strategy the stale block rate is significantly higher and increases

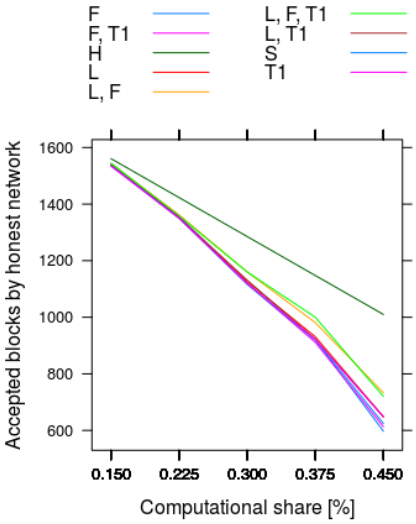

Figure 6.4: Blocks created by the honest network and accepted in the longest chain

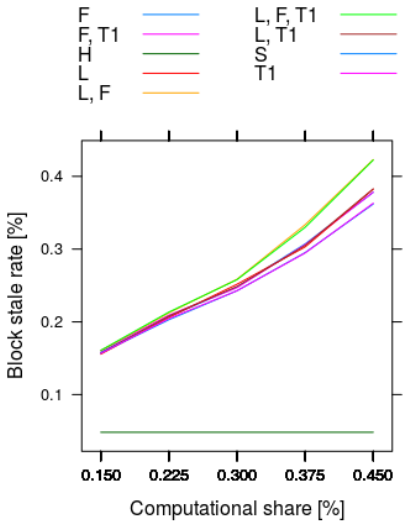

Figure 6.5: Stale block rate

further if the computational power of the selfish miner is augmented. Furthermore, the gap between the two worst performing selfish strategies and the other strategies can be observed. If a node conducts lead stubbornness combined with equal-fork stubbornness $(L,\ F)$ or selfish mining with all three modifications $(L,\ F,\ T_1)$, there are even more stale blocks in the network.

Similar as during the evaluation of the deterministic behaviour of the simulation framework in chapter 4.3, also during the execution of the selfish mining scenarios the utilisation

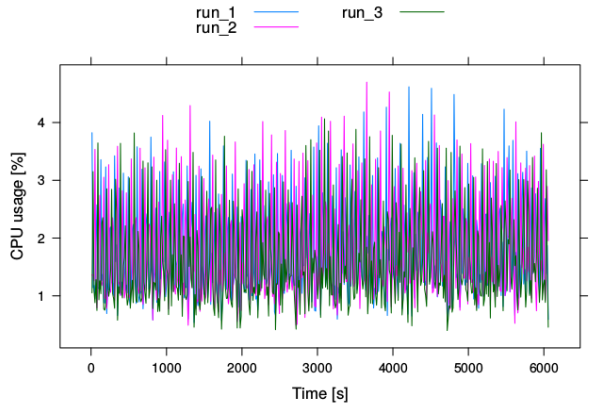

Figure 6.6: CPU usage during the triple execution of a simulation scenario

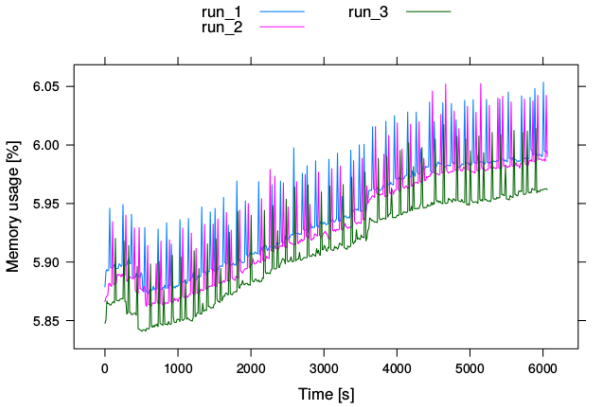

Figure 6.7: Memory usage during the triple execution of a simulation scenario

of the CPU and the memory of the host machine stayed under 10% as shown in figure 6.6 and figure 6.7. Thus, the specifications of the host machine did not restrict the simulations in any way.

CHAPTER 7

Evaluation

The simulation of eight different selfish mining strategies with five distinct distributions of computational power provides a reasonable outcome to argue the two research questions of the thesis.

7.1 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?

The simulation showed that the relative share of mining rewards obtained with selfish mining is higher compared to the honest mining if the miner has over 40% of mining power and conducts a satisfactory selfish mining strategy. To the satisfactory strategies, all strategies expect the combination lead stubborn and equal-fork stubborn (L, F) and selfish mining with all three modifications (L, F T_1) can be counted. The curves of the six good-performing strategies depicted in figure 6.3 show all a similar behaviour where their relative gain increases non-linearly with a higher computational share. Out of these six strategies, the most promising strategies are normal selfish mining (S) and equal-fork stubbornness (F). When these two strategies are applied, the selfish miner can create 49.1%/50% of the blocks of the longest chain even though its share of the mining power is only 45%.

In all cases where the selfish miner has a low share of computational power, the results of the simulation show that the selfish mining does not increase the relative gain. For example, when the selfish miner has 15% of the mining power in the network only about 5% of its blocks end up in the longest chain. Even if the selfish miner has 37.5% share of the computational power the most advantageously strategy (S) only creates 32.4% blocks of the longest chain. The worst performing mining strategy in the scenario where the

selfish miner has 37.5% is the strategy with all three modifications (*L, F, T_1*) generating only 22.7% of the accepted blocks.

Concluding it can be said that with the defined simulation scenario and the proposed software solutions only for a very high computational share an increase of the relative gain could be observed. For all scenarios where the selfish miner has less than 40% percent, the miner would earn relatively more by behaving honestly.

7.2 RQ2

How do the obtained results of the simulation match the outcome of previous research in the area of selfish mining?

When comparing the outcome from the simulations with recent studies, it first needs to be considered that with the introduced simulation framework the outcome of the block races cannot be defined directly, as it is the case in previously used types of simulation. Instead, the block races happen naturally during the execution of a scenario. With the used simulation scenario and implementation of the selfish proxy, it is likely that the parameter γ, denoting if an honest node extends the private chain during a block race, is almost zero. This assumption is based on the configured network topology and the implementation of the selfish proxy. In the scenario, a fully connected network topology is used where each connection has the same latency. Hence, when a node finds a block, it sends the block to all nodes directly. Then it is unlikely that the selfish node can advertise its block faster to other nodes using the same connections before the honest nodes have already adopted to the new public tip. The implementation of the selfish proxy further worsens the position of the selfish miner when trying to match a competing block during a block race. The proxy which eclipses the private node does not implement the fast block propagation mechanism *compact blocks*. Instead, the proxy uses the standard but slower block propagation technique to receive and send blocks. Thus, the proxy receives and sends blocks slower than the rest of the network.

Assuming that the probability of the selfish miner to win a block race is very low, the attained results match the outcome of previous research closely. In figure 7.1 the red line shows the relative revenue of the selfish miner conducting normal selfish mining when it is not able to win any block race [ES14]. The curve shows the same concave course as in figure 6.3 underlining that with an increase of the computational share the efficiency of the selfish mining is amplified. Moreover, the selfish miner achieved more relative revenue compare to the honest miner with about 36% of the mining power, similar to the slightly worse results of 40% showed in chapter 6.3. Comparable outcomes for the selfish mining without any modification were produced in other recent research [NKMS16, SSZ16].

In figure 7.2 optimal selfish mining strategies including stubborn modifications from the research of [NKMS16] are shown. In the case, where the selfish miner loses all block races, the honest behaviour is the most profitable strategy until a computational share of 34% is reached. Afterwards, the most advantageous mining strategy is trail stubbornness (T_1)

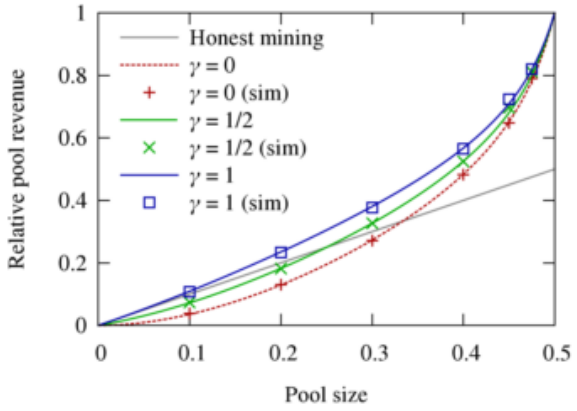

Figure 7.1: Results obtained by [ES14] with normal selfish mining

up to a computational share of 45%. From 45% to 50% the dominant strategy is trail stubborn in combination with equal-fork stubborn (T_1, F), but in many cases, the best strategy is not clear signalised by the black dots in the graphic. The results of the optimal stubborn mining strategy by [NKMS16] slightly differ to the outcome of the simulation from chapter 6.3. In the proposed simulation the best performing strategies were the normal selfish mining (S) and equal-fork stubborn mining (F) with trail stubbornness combined with equal-fork stubbornness (T_1, F) and trail stubbornness (T_1) being only the third and fourth most satisfactory strategy.

[NKMS16] does not propose the actual ranking of different selfish strategies regarding their performance for a specific γ but provides a comparison of the relative gain obtained by selfish mining and selfish mining with stubborn modifications. On the assumption that γ is zero, the relative gain achieved by using stubborn modifications over selfish mining is very low as shown in figure 7.3. Thus, also in the research of [NKMS16] the normal selfish mining forms a viable strategy comparable to the results obtained in the previous chapters. Furthermore, figure 7.2 showed that between 45% and 50% the best strategy is not always known denoted by the black dots. From that concludes, that in the case where all block races are won by the honest network, the optimal strategy is not identifiable, similar to the outcome from the proposed simulations where the four best strategies were 3% apart.

Summarising the obtained simulations results are comparable to recent research even though the outcome of the simulation was slightly worse than results from previous research. Regarding the best performing selfish mining strategies also the research of [NKMS16] showed that differences in the case where γ equals zero are subtle. Nevertheless, both studies showed similar good performing selfish mining strategies.

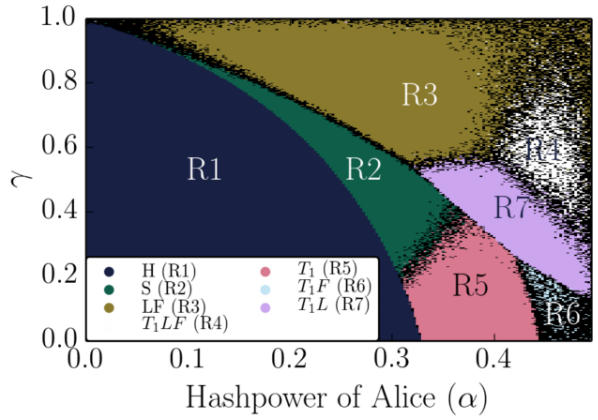

Figure 7.2: Optimal stubborn mining strategies retrieved by [NKMS16]

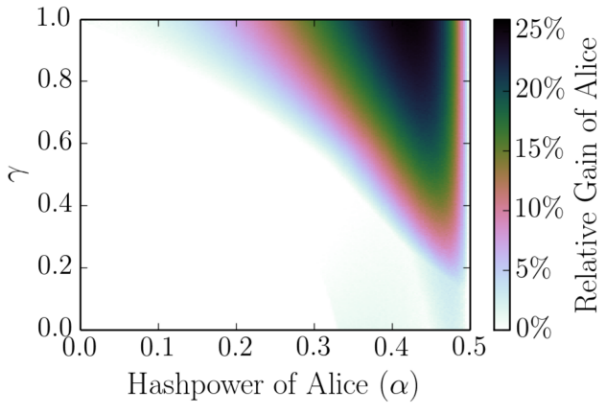

Figure 7.3: Selfish mining compared to optimal stubborn mining strategies by [NKMS16]

7.3 Profitability

As previously evaluated selfish mining can increase the relative gain of the misbehaving miner, but the results of the simulations show further that the absolute amount of accepted blocks is lower as shown in figure 6.3. Hence, the selfish miner is earning less mining rewards even though its relative revenue is increasing. The loss of profit is possible because the stale block rate increases during selfish mining, and thus, fewer blocks end up in the longest chain. To still profit from the attack the miner would need to wait for the difficulty adjustment. After the difficulty adjustment, the nodes in the network can find more blocks and hence, also the selfish miner can create more blocks. Since selfish mining was never observed for such a long time and no research was conducted in this area, it is not clear if such a long attack would be successful [NKMS16].

CHAPTER 8

Further research

The thesis examines the relative gain of a selfish miner in a peer-to-peer network similar to the original *Bitcoin* network. The selfish miner is implemented by eclipsing a normal *Bitcoin* peer with a proxy. The proxy then conducts selfish mining by withholding blocks created by the honest network and the eclipsed node. The network and the containing nodes were realised by using *Docker* and hence, virtualised on one single host. Both approaches, the selfish proxy and the virtualised peer-to-peer network, are novel and are leaving many possibilities for improvement and research directions open.

8.1 Selfish proxy

The selfish proxy needs to receive information of new blocks as fast as possible to conduct selfish mining efficiently. Currently, the proxy requests the whole block and all headers of blocks higher than its local best tip after hearing about a new block as described in chapter 5.3 and 5.4. To successfully relay the block to the other side of the network, the selfish proxy needs to receive both messages entirely where especially the full block request is very time-consuming. By using the compact block relay mechanism, the selfish proxy could speed up this whole communication. The compact block relay mechanism shown in figure 8.1 is implemented in the *Bitcoin* reference implementation since late 2016 [**?**] and allows the node to broadcast blocks to its peers directly [**?**]. The difference to the normal block relay mechanism is that the node relays the block without any transactions. To verify the block, the receiving node first checks which transaction are in the local mempool and asks afterwards with a specific request only for the missing transactions. Compared to the traditional block relay mechanism where the node sends the full block containing all transactions, the compact block relay mechanism reduces the amount of data transferred and accelerates the block relay [**?**, **?**]. In the case of the selfish proxy, the update of the local chain could be executed right after the compact block is received. Then, depending on the outcome of the selfish mining algorithm, the

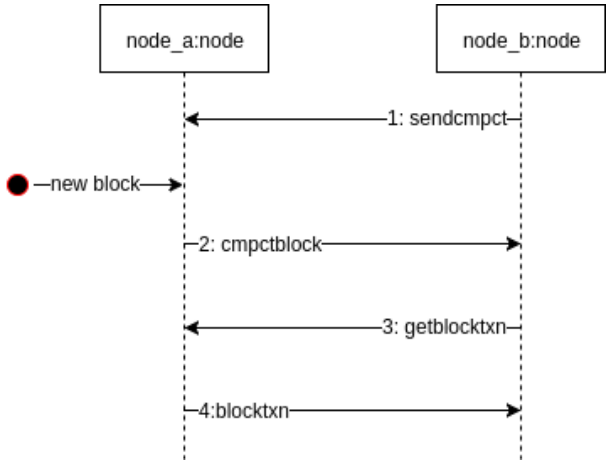

Figure 8.1: Compact block relay mechanism [**?**]

relevant blocks could be relayed using the compact block relay mechanism. Compared to the currently used relay mechanism the whole communication flow would be reduced from six sent messages to two messages because during a simulation only empty blocks are created. It is very likely that the compact block relay would improve the efficiency of the selfish mining strategy performed by the proxy, especially if at some later point also transactions are created during a simulation. Then the transmission of the blocks is even more time consuming and increases the time needed by the proxy to relay blocks.

Another direction of research consists of implementing the selfish mining strategies directly in the code executed by the nodes of the network. For example, the public available reference implementation could be extended to implement selfish mining. In the current setting the selfish proxy adds an extra hop between the communication of the honest network and the eclipse node. Hence, the proxy increases the latency of the eclipsed node significantly. With the raised latency the position of the selfish miner during a block race is worsen resulting in a lower efficiency of the conducted selfish mining. By implementing the selfish mining directly in the reference implementation, the extra hop would be removed, and the selfish miner now consisting only of one node would not have a lower latency compared to the honest network. The downside of this approach would be that the reference implementation is altered which could cause unexpected side-effects.

Last but not least combining attacks with selfish mining provides a possible field of research. The selfish miner could improve the profitability of the selfish mining by combining it with other attacks [GKW+16, SSZ16, NKMS16, GRKC15]. New insights and combinations could be tested by implementing and simulating the attacks directly in the simulation framework or selfish proxy.

8.2 Simulation scenario

Currently, no transactions are included in the simulation runs. Thus, all blocks generated by the nodes are empty and consequently propagate faster in the network than a block with transactions. A further research direction consists of including transactions in a simulation run to simulate the block propagation more realistically. This could be either achieved by filling up the mempool with prepared transactions or by creating the transactions during the simulation run. In the latter approach also the relaying of transaction would be simulated resulting in an even more realistic simulation. Using this method the performance and reliability of the RPC-connections over TCP/IP needs to be considered. The current implementation of the simulation framework has troubles to maintain the connection to the API of the nodes often resulting in *broken pipe* errors. At the moment these errors are recovered by simply reconnecting to the node which costs precious time. Due to the lost time the execution speed of the simulation needs to be lowered resulting in unsustainable long simulation durations. Hence, to support the creation of transaction during a simulation run, the unreliability of the RPC-connections should be dissolved, or it should be considered to use the more performant Unix domain sockets. The Unix domain sockets are providing their performance by bypassing the TCP/IP stack, and are likely to sustain the higher workload when adding transactions to a simulation scenario. The capability to use Unix domain socket to communicate with the provided API by the nodes is planned for the next release of the reference implementation of *Bitcoin* [**?**].

ask aljosha for paper

Another compelling research area is the topology of the network and the nodes participating in the peer-to-peer network. The network topology chosen in this thesis is just an abstract simplification of the real *Bitcoin* network. The actual network changes continuously and contains multiple diverse nodes running different implementations of the *Bitcoin* protocol. Hence, a better capturing of the actual topology of the *Bitcoin* peer-to-peer network containing dissimilar nodes could be an area of research.

8.3 Mitigation

The mitigation of selfish mining is also a possible future research direction. The simulation framework and the proxy can be used to assess different proposed mitigation approaches [ES14, Bil15, SPB16, ZP17] against their impact on selfish mining. Since the simulation framework uses the reference implementation a mitigation proposal like the uniform tie-breaking [ES14] can be implemented directly in the actual code. Hence, those mitigations can be tested quickly and under realistic circumstances providing accurate results.

List of Figures

List of Tables

List of Listings

Bibliography

[Bah13] Lear Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:1312.7013*, 2013.

[Bil15] Saki Billah. One weird trick to stop selfish miners: Fresh bitcoins, a solution for the honest miner. 2015.

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

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

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

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

[ES14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International Conference on Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

[GKW+16] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16. ACM, 2016.

[GRKC15] Arthur Gervais, Hubert Ritzdorf, Ghassan O Karame, and Srdjan Capkun. Tampering with the delivery of blocks and transactions in bitcoin. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 692–705. ACM, 2015.

[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. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 305–320. IEEE, 2016.

[pyc] Minimalistic python implementation of the bitcoin networking stack. `https://github.com/cdecker/pycoin`. Accessed: 2017-10-30.

[pyt] Python2/3 library providing an easy interface to the bitcoin data structures and protocol. `https://github.com/petertodd/python-bitcoinlib`. Accessed: 2017-10-30.

[SPB16] Siamak Solat and Maria Potop-Butucaru. *ZeroBlock: Preventing Selfish Mining in Bitcoin*. PhD thesis, Sorbonne Universit es, UPMC University of Paris 6, 2016.

[SSZ16] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 515–532. Springer, 2016.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58:345–363, 1936.

[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. *Ethereum Project Yellow Paper*, 151, 2014.

[ZP17] Ren Zhang and Bart Preneel. Publish or perish: A backward-compatible defense against selfish mining in bitcoin. In *Cryptographers' Track at the RSA Conference*, pages 277–292. Springer, 2017.