FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Simulation of different selfish mining strategies in Bitcoin

Simulation respecting network topology and reference implementation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Simon Mulser, BSc
Matrikelnummer 01027478

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 13. Oktober 2017

Simon Mulser	Edgar Weippl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Simulation of different selfish mining strategies in Bitcoin

Simulation respecting network topology and reference implementation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Simon Mulser, BSc
Registration Number 01027478

to the Faculty of Informatics

at the TU Wien

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

Vienna, 13th October, 2017

Simon Mulser	Edgar Weippl

Erklärung zur Verfassung der Arbeit

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

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

Wien, 13. Oktober 2017

Simon Mulser

Danksagung

Meine Danksagung. Coming soon...

Acknowledgements

My acks. Coming soon...

Kurzfassung

Meine Kurzfassung. Coming soon...

Abstract

My abstract. Coming soon...

Contents

CHAPTER 1

Introduction

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

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

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

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

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

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

CHAPTER 2

State-of-the-art

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

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

Further research showed that more generalised selfish mining strategies lead to even more relative gain for the selfish miner [NKMS16, SSZ16, GRK15, GKW+16, Bah13]. Figure 2.2 shows a possible categorization of the different selfish mining strategies where α and γ is used equivalent as in figure 2.1 and β expresses (1 - α). Furthermore, the prime

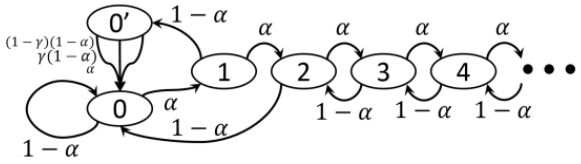


Figure 2.1: Selfish mining state machine with transition probabilities [ES14]

states are standing for states where a block race happens on certain height and the *0"* represent the state where both chains have the same height but the selfish miner and the rest of the network are mining on different branches. The idea behind the different variations of selfish mining strategies in figure 2.2 are:

- Lead stubborn 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 mining strategy 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 mining strategies reflects the idea to even trail behind the public chain and to eventually catch up. The figure 2.2 depicts the trail stubborn mining strategy *T1*. The number one denotes that the selfish miner is allowed to trail one block behind the public chain.

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

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

Besides using variations of the selfish mining strategies, the attack can also be combined with other attacks to achieve better results [GKW$^+$16, SSZ16, NKMS16, GRK15]. If the eclipse attack is used in combination with selfish mining the victim contributes its mining power to the private chain and hence, strengthens the position of the selfish miner [NKMS16, GKW$^+$16]. [NKMS16] additionally shows that the eclipsed victim

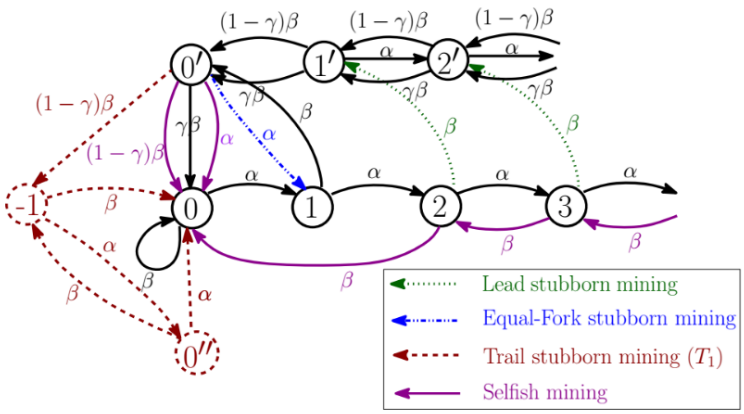


Figure 2.2: Categorization of different mining strategies [NKMS16]

under certain circumstances can benefit from the attack and therefore has no incentive to stop the attack. Another attack which can be used in combination with selfish mining is double-spending [SSZ16, GKW+16]. Every time the selfish miner starts his selfish mining attack he can publish a transaction and include a conflicting transaction in his first secret block. During the execution of the selfish mining attack, the payment receiver may accept the payment depending on his block confirmation time. Now in the case of a successful selfish mining attempt, the adversary can overwrite the public chain, which additionally results in a successful double spending. The operational costs of unsuccessful double-spending can be seen as low because the adversary still would get goods or a service in exchange for the transaction [SSZ16, GKW+16].

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

or implemented in Bitcoin [bita, bitc].

CHAPTER 3

Expected results

The expected outcome of this thesis is a more accurate simulation of different selfish mining strategies and therefore a better understanding of the potential real world implications of such attacks. The selfish mining strategies 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 and different network topologies. The result of the simulations should show which strategy is the best strategy for a certain combination of a network topology and distribution of mining power. Thereby, also the influence of the network topology is studied in more detail compared to previous research. The simulation results should emphasise the recent work in the area of selfish mining and show that the current implementation of Bitcoin protocol is vulnerable against different selfish mining strategies.

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

CHAPTER 4

Methodology and Approach

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

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

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

CHAPTER 5

Simulation software

The simulation software 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.

5.1 Tick

A fundamental concept of the simulation software is a so-called tick. A tick represents a small time span containing information about which nodes should find a new block in this tick. For a simulation run, multiple ticks are generated forming a sequence of ticks. This sequence is 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.

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

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

5.2 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 easily processed by the simulation software. 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 program. Furthermore, the created configuration files can be copied to the output directory of a simulation run providing reproducibility of a run.

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

5.2.1 *nodes.csv*

The *nodes.csv* contains the configuration of every node which should be orchestrated by the simulation software. 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 when starting the node.
- *latency*: The latency of the node in the peer-to-peer network.

5.2.2 *network.csv*

The *network.csv* reflects a connection matrix as shown in table 5.1. The simulation software 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.

5.2.3 *ticks.csv*

The *ticks.csv* contains all ticks which should be executed by the simulation software. Each line represents a tick with no, one or multiple block events. Hence the length of

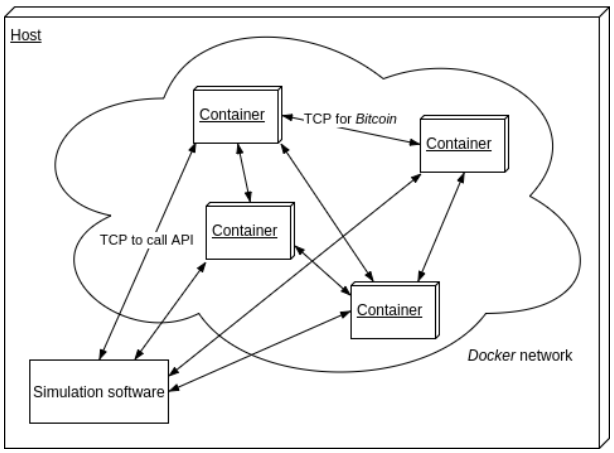


Figure 5.1: Overview of the virtual peer-to-peer network

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

5.3 Simulation

The main functionality of the simulation software 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.

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

5.3.1 Preparation

The preparation phase is the first phase of a simulation run. In the first place, the software creates an output directory and copies all configuration files into the new directory to assure reproducibility. Subsequently, a virtual peer-to-peer network is assembled as depicted in figure 5.1. Therefore 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 with the *Docker* command as shown Listing 5.1. 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*, the *Bitcoin* binary, in *regtest mode* (line 6) and setting the data directory of to */data* (line 7) *bitcoind* persists all relevant data into */data/regtest*. Hence all the data persisted under */data/regtest* will 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*. The simulations software uses this connections later to send commands directly 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 5.1: Simplified version of how a node is started with *Docker* and *bitcoind*

5.3.2 Execution

In the execution phase, the simulation software iterates over each tick from the *ticks.csv*. If a tick contains a block event, the software uses the established RPC-connection to call *generate* on the *Bitcoin* API of the specific node to create a new block. Since all nodes are running in *regtest mode*, the proof-of-work is deactivated, and the nodes can create blocks immediately. Some ticks may contain multiple block events. In this case, the blocks are produced by the nodes one after another consistently waiting for the block hash to be returned. A simulation run is executed with a particular tick duration which specifies how long a tick should last. To ensure the tick duration the simulation software simply keeps track of the elapsed time during the execution of the block events and sleeps afterwards until the tick is over. In the case that the completion of the block events lasts

longer than the tick duration, the execution speed of the simulation was too fast. To still provide the results of the simulation the software continues and just executes the next tick without sleeping.

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 specific state. The collected snapshots can later be used to determine the actual utilisation of the CPU by calculating the differences between the snapshots. For the memory usage, the thread reads the *MemAvailable* value periodically from */proc/meminfo* file. This value provides a heuristic of the currently available memory on the machine.

5.3.3 Post-processing

The post-processing phase is the last phase of a simulation run. At the beginning of this phase the consensus chain, denoting the longest chain of blocks all nodes agree about, is calculated. The chain is determined 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. This procedure is repeated for every block height by simply increasing the height by one after each check. If a 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 computation 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, the logs from the nodes and from the simulation software are parsed to retrieve information about the simulation run. The log lines are parsed by using a particular regular expression for each possible type of a log line:

- *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 created block.
- *UpdateTipLine*: Log line produced by a node whenever a block updates a tip of the chain.
- *PeerLogicValidationLine*: Log line produced when a node checks the proof-of-work of a received compact block.
- *BlockReconstructLine*: A log line created when a node reconstructed a compact block successfully.

- *BlockReceivedLine*: Log line made when a node receives a block.
- *TickLine*: A log line with information about an executed tick.
- *BlockExceptionLine*: Log line produced whenever the simulation software was not able to complete a block event successfully.
- *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 software itself. Furthermore, the *BlockGenerateLine* log line was added especially for the simulation software to the *Bitcoin* reference implementation. Usually the reference implementation does not create a log line containing the block hash when it creates a new block. Hence to easily circumvent this fact, a log line was added to the reference implementation to persist the event of the block creation including a hash of the block.

The simulation software 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. On execution time of a simulation, a so-called *skip_ticks* parameter can be passed. This parameter denotes how many ticks at the beginning and at the end of the simulation should be skipped and not be evaluated. 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 altered because the either are used to calculate some combined statistics like the block propagation time or because the statistics of these types are still calculated over the whole simulation duration. Additionally the *preprocess.R* script sorts all CSV files according to the timestamps of the log line. The sorting 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 the *preprocess.R* script created all CSV files, the simulation software 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 visualising CPU and memory utilisation
- graph showing the duration of a tick over time
- charts and information about blocks created during the simulation
- graphs and information about exceptions happened during the simulation

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

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

Listing 5.2: Calculation of propagation time with *R*

5.3.4 Multi-runs

When the simulation software is executed with the *multi-run* command (chapter 5.4.6) multiple simulations are conducted depending on the passed input arguments. After each simulation, the created CSV files of the simulation are copied by the simulation software into an own directory. Once the last simulation finishes the software aggregates all copied CSV files into separate 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 compose a report comparing all simulation runs.

5.4 Commands

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

5.4.1 *nodes* command

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

5.4.2 *network* command

When the simulation software gets executed with the *network* command, it reads a *nodes.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 software creates a matrix reflecting connections between two nodes. The *Bitcoin* reference implementation itself does not differentiate between incoming or outgoing connection. Hence it suffices to define one connection in the connection matrix if two nodes should be connected. The connection matrix is afterwards persisted in the configuration file *network.csv*.

5.4.3 *ticks* command

The *ticks* command can be used to create the configuration file *ticks.csv*, which contains the ticks to be executed. When executing the *ticks* command the simulation software 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 software 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 5.1 with $\lambda = blocks\_per\_tick \cdot share$. From this exponential distribution sufficient samples are drawn, which are denoting points in time when a specific nodes should find

a block. With this time series at hand, the ticks are created by starting with the 1st tick. For every point in time, lower than 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*.

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

5.4.4 *simulate* command

On execution of the *simulate* command, the simulation software 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 software executes the simulation as described in section 5.3.

5.4.5 *run* command

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

5.4.6 *multi-run* command

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

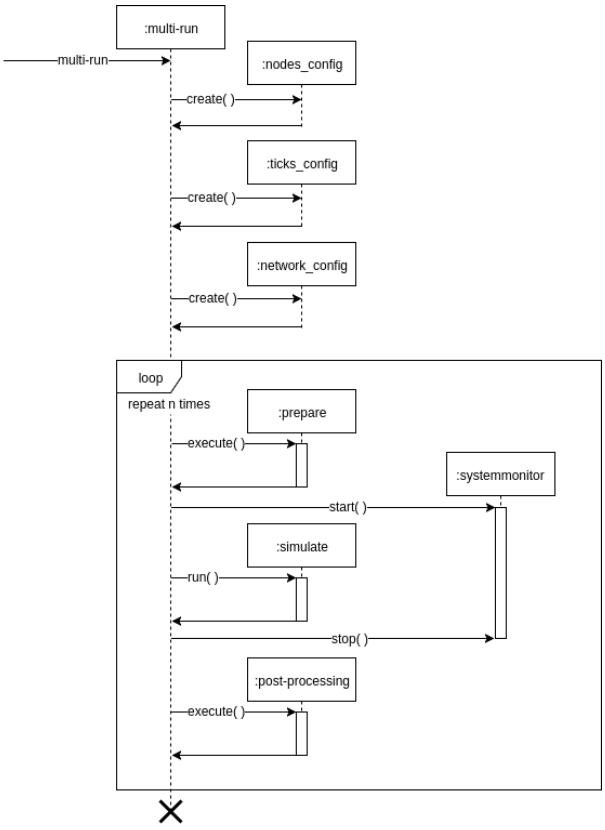


Figure 5.2: Sequence diagram of the *multi-run* command

CHAPTER 6

Evaluation of simulation software

The simulation software 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 software works is unchangeable without breaking the whole architecture of the simulation software. 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.

6.1 Deterministic behaviour

The simulation software 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 software with such a deterministic behaviour.

6.2 Reference scenario

The reference scenario used to evaluate the simulation software 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 [GKCC14, BS15, TS16, BMC+15]. Current statistics show that about twenty miners create almost every block [blo, coia, bite]. 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 [DW13].

For the sake of simplicity all nodes part of the network run the same version 0.15.0.1 [bitb] of the *Bitcoin* reference implementation. This adjustment is in contrast to the real network where different implementation and version are used [coib]. 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 [GKCC14, blo, coia, bite].

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 5.4.3. 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, DW13, 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.

6.3 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 6.1 and 6.2. The metrics about the usage of the CPU and storage were continuously collected by the simulation software as described in chapter 45.

The outcome of the evaluation is shown as box plot in figure 6.3 and as a density plot in 6.4. 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 6.1 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.

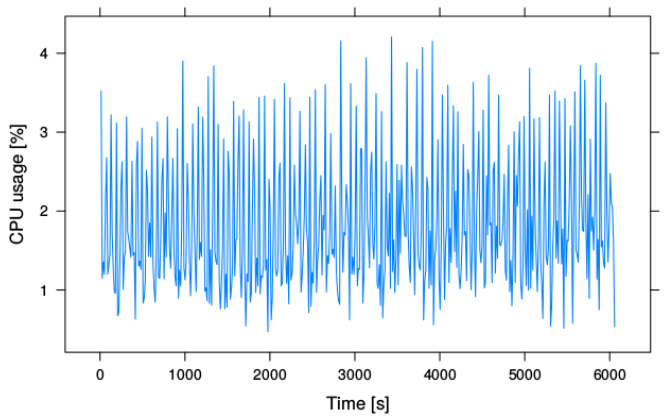


Figure 6.1: CPU usage during a particular simulation run

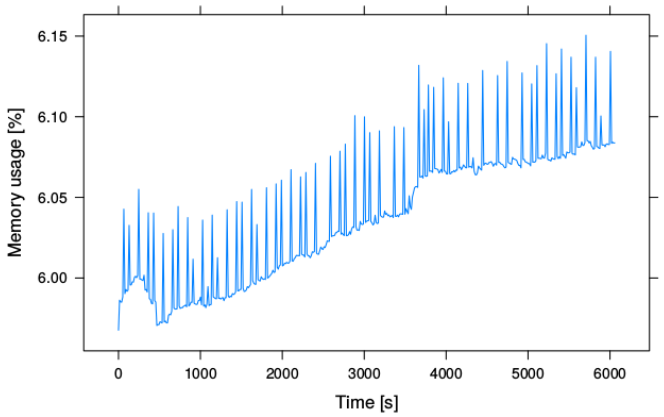


Figure 6.2: Memory usage during a particular simulation run

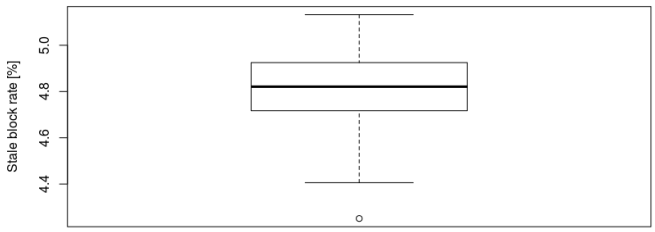


Figure 6.3: Box plot of the stale block rate of 100 executions

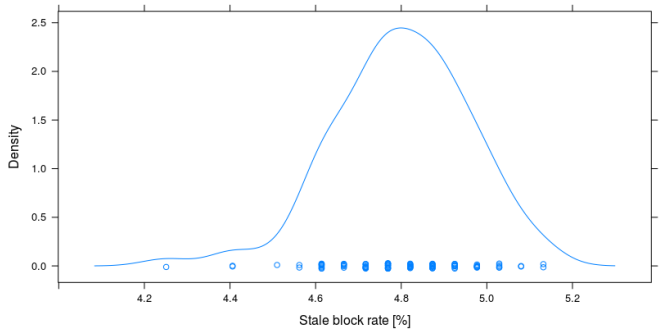


Figure 6.4: Density plot of the stale block rate of 100 executions

CHAPTER 7

Selfish proxy

The selfish proxy is the node in our peer-to-peer network implementing the different selfish mining strategies. He eclipses a normal bitcoin node from the rest of the network and accepts blocks created by the eclipsed node or by the normal nodes. With the collected blocks he recreates the chain locally and on every received block he runs the configured selfish mining algorithm. The algorithm then decides if the received block should be withhold or relayed to the other part of the network.

not relaying tx

create a component diagram - network - chain - strategy create a picture showing how blocks are relayed in bitcoin

any software recommendation for charts/diagrams

7.1 Architecture

7.1.1 Network component

The implementation of network component is based on the two libraries *pycoin* and *python-bitcoinlib*. The library *pycoin* provides simple networking utilities to connect to other bitcoin nodes and to manage those connections. Where on the other hand the *python-bitcoinlib* library implements functionalities to serialize and deserialize bitcoin network messages. In general the network component is responsible for:

reference

- listening and accepting incoming connections
- be able to distinguish between connections from the eclipsed node or connections from the rest of the network
- managing and keep his connections a live by replying to ping messages
- accepting, processing and replying to messages relevant for the selfish relay
- detecting and re-triggering of failed requests

7.1.2 Chain component

The main responsibility of the chain component is to reassemble the chain with the blocks received by the network component. This can be done easily by just looking at the hash of the previous block stored in the received block. If the previous block is already in the chain, the newly received block gets added to the chain. In the case the block identified by its hash is not available in the chain, the block gets preserved as an orphan block. On every successful inserted block all orphaned blocks are checked if they can be added now to the chain. Along side the information stored in the block, the chain component also keeps track of the block origin and a boolean reflecting if the transfer of this block to other nodes is allowed. This information is necessary for the chain component to be able to calculate the current fork between the eclipsed node and the rest of the network.

7.1.3 Strategy component

The strategy component implements the four different selfish mining strategies:

- selfish mining
- trail stubborn mining
- equal-fork stubborn mining
- lead stubborn mining

The strategies are implemented by accepting the current fork as an input and based on that fork the action to be taken is determined with simple control structures like if and else.Additionally the strategy component contains a module for determining which blocks are supposed to be relayed depending on a determined action and a fork.

describe actions here? or in state-of-the-art

7.2 Selfish relaying

The overall concept of the selfish proxy is to do selfish mining without actually mining blocks. By eclipsing a node the selfish proxy is able to withhold and relay blocks corresponding to a selfish mining strategy. Hence the selfish proxy is, as we call it, selfish relaying.

To be able to do selfish relaying the proxy continuously collects all blocks advertised by the connected nodes. In bitcoin newly found blocks are broadcasted over the peer-to-peer network by sending the hash in an inv message. Whenever the proxy receives such a inv msg with a block hash unknown to him, he immediately requests the new block by sending a getdata message. The node, which found the block, replies then with a block message containing the actual block. The proxy, after receiving the block, tries to insert the block in his own local chain. If the block can be inserted on top of a tip in the chain, the proxy runs the selfish mining algorithm. The algorithm determines based on the

current fork between eclipsed node and the rest of the network an action. In the case that the action is one out of match or override, the algorithm further determines which blocks need to be relayed and sends out the corresponding block hashes over a inv message.

When a normal node receives a inv message with a unknown block from the proxy, the node first request all new headers with a getheaders call. The proxy replies to the node then the new headers, which are then

- block relay - how does block relay work - compact blocks - how did we implement it - no compact blocks - assuming all blocks are valid - when we have the whole block pass it to chain - receives height of private and public - searches action to execute - 4 different types of actions (wait, adopt, match, override) - executes by sending inv's - which strategies did we implement? - - how did we implement it? - search for an action - if-else control structure - many unit tests

why we used a proxy? - fast implementation - no need to touch reference implementation

disadvanteges - extra hop - not a real proxy - need to keep track of chain - bitcoin protocol not easy

figure block relay

how works each strategy? describe it here or in section simulation and reference to the section.

move maybe to another section. here we should only DESCRIBE the selfish proxy

CHAPTER 8

Simulation of selfish mining strategies

8.1 Results

CHAPTER 9

Discussion

9.1 Installation

CHAPTER 10

Further research

10.1 Installation

CHAPTER 11

Introduction to LaTeX

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

11.1 Installation

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

11.2 Editors

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

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

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

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

	Command
1	`pdflatex.exe  example`
2	`bibtex.exe  example`
3	`pdflatex.exe  example`
4	`pdflatex.exe  example`
5	`makeindex.exe -t example.glg -s example.ist -o example.gls example.glo`
6	`makeindex.exe -t example.alg -s example.ist -o example.acr example.acn`
7	`makeindex.exe -t example.ilg -o example.ind example.idx`
8	`pdflatex.exe  example`
9	`pdflatex.exe  example`

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

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

11.3 Compilation

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

11.4 Basic Functionality

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

11.4.1 Floats

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

11.4.2 Tables

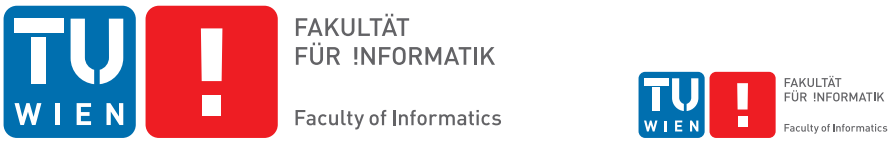
A table in LaTeX is created by using a `tabular` environment or any of its extensions, e.g., `tabularx`. The commands `\multirow` and `\multicolumn` allow table elements to span multiple rows and columns.

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

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

11.4.3 Images

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



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

Figure 11.1: The header logo at different sizes.

11.4.4 Mathematical Expressions

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

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

or as numbered equation with

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

11.4.5 Pseudo Code

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

11.5 Bibliography

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

Algorithm 11.1: Gauss-Seidel

Input: A scalar ϵ, a matrix $\mathbf{A} = (a_{ij})$, a vector $\vec{b}$, and an initial vector $\vec{x}^{(0)}$
Output: $\vec{x}^{(n)}$ with $\mathbf{A}\vec{x}^{(n)} \approx \vec{b}$

```
for k ← 1 to maximum iterations do
    for i ← 1 to n do
        x_i^(k) = (1/a_ii) (b_i − Σ_{j<i} a_ij x_j^(k) − Σ_{j>i} a_ij x_j^(k−1));
    end
    if |x^(k) − x^(k−1)| < ε then
        break for;
    end
end
return x^(k);
```

11.6 Table of Contents

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

11.7 Acronyms / Glossary / Index

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

11.8 Tips

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

- Each sentence in the source text should start with a new line. This helps not only the user navigation through the text, but also enables revision control systems

(e.g. Subversion (SVN), Git) to show the exact changes authored by different users. Paragraphs are separated by one (or more) empty lines.

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

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

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

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

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

Table 11.4: Anonymization of PDF documents.

11.9 Resources

11.9.1 Useful Links

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

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

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

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

11.9.2 Comprehensive TeX Archive Network (CTAN)

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

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

algorithm2e Functionality for writing pseudo code.

amsmath Enhanced functionality for typesetting mathematical expressions.

amssymb Provides a multitude of mathematical symbols.

booktabs Improved typesetting of tables.

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

fontenc Determines font encoding of the output.

glossaries Create glossaries and list of acronyms.

graphicx Insert images into the document.

inputenc Determines encoding of the input.

l2tabu A description of bad practices when using LaTeX.

mathtools Further extension of mathematical typesetting.

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

multirow Allows table elements to span several rows.

pgfplots Function plot drawings.

pgf/TikZ Creating graphics inside LaTeX documents.

subcaption Allows the use of subfigures and enables their referencing.

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

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

xcolor Allows the definition and use of colors.

List of Figures

List of Tables

List of Listings

Glossary

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

. 31

. 31

. 31

Acronyms

CTAN Comprehensive TeX Archive Network. 31

FAQ Frequently Asked Questions. 31

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

SVN Subversion. 30, 31

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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