Technische Universität München

Fakultät für Informatik

Master's Thesis in Information Systems

Bitcoin-like Blockchain Simulation System

Fabian Schüssler

Technische Universität München

Fakultät für Informatik

Master's Thesis in Information Systems

Bitcoin-like Blockchain Simulation System

Bitcoin-ähnliches Blockchain Simulationssystem

Author: Fabian Schüssler

Supervisor: Prof. Dr. Hans-Arno Jacobsen

Advisor: Pezhman Nasirifard

Submission: 07.09.2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 31.08.2018

(Fabian Schüssler)

Abstract

Despite the recent $600 billion crash of the whole cryptocurrency market, interest in development and research on blockchains is growing. Reaching the scalability of payment processors such as PayPal or VISA is a hard problem for decentralized peer-to-peer networks like Bitcoin. Heated discussions about how to solve the scalability problem led to hard forks such as Bitcoin Cash. To better understand the impact of changes to the bitcoin protocol and to optimize bitcoin-like blockchains for scalability, we propose the Bitcoin-like Blockchain Simulation System: a configurable bitcoin-like blockchain simulator for large-scale peer-to-peer networks with the ability to examine the impact of different attacks such as double-spending and transaction spam on the network and its security properties. Important characteristics and metrics of the network and their relationship to various configuration parameters can be explored in an easy to understand user interface.

Inhaltsangabe

Trotz des kürzlichen 600 Milliarden US-Dollar Kursabsturz des Marktes für Kryptowährungen wächst das Interesse in Entwicklung und Forschung von Blockchains. Es ist ein schwieriges Problem für dezentralisierte Rechner-zu-Rechner Netzwerke wie Bitcoin die Skalierbarkeit von Zahlungsverarbeitern wie PayPal oder VISA zu erreichen. Lebhafte Diskussionen über das Problem der Skalierbarkeit haben zu Hard Forks wie Bitcoin Cash geführt. Um die Auswirkungen von Veränderungen am Bitcoin Protokoll und um die Skalierbarkeit von Bitcoin-ähnlichen Blockchains zu optimieren, schlagen wir das Bitcoin-ähnliche Blockchain Simulationssystem vor: ein konfigurierbarer Bitcoin-ähnlicher Blockchain Simulator für große Rechner-zu-Rechner Netzwerke mit der Fähigkeit die Auswirkungen von Angriffen wie Double-Spending oder Transaktionsspam auf das Netzwerk und dessen Sicherheitseigenschaften zu untersuchen. Wichtige Charakteristiken und Metriken des Netzwerkes und deren Beziehung zu verschiedenen Konfigurationsparametern können in einer einfach verstehbaren Benutzeroberfläche erkundet werden.

Acknowledgment

I would like to thank my thesis advisor M.Sc. Pezhman Nasirifard of the Chair of Application and Middleware systems at the Technical University of Munich for supporting me with my thesis, who always took time to help me with invaluable advice. I would also like to thank Prof. Dr. Hans-Arno Jacobsen for giving me the opportunity to write this thesis at his chair and Lyubomir Stoykov for answering questions about his work.

Contents

List of Figures

List of Tables

Abbreviations

ASIC	Application-Specific Integrated Circuit.
AWS	Amazon Web Services, a Cloud-Computing-Provider.
BBSS	Bitcoin-like Blockchain Simulation System.
BIP	Bitcoin Improvement Proposals.
BTC	A Unit of the Cryptocurrency Bitcoin.
d	Attack duration.
n	Confirmations.
non-SegWit	Legacy/before Segregated Witness.
q	Attacker's percentage of the total hash-rate.
SegWit	Segregated Witness.
tpb	Throughput/Transactions per Block.
tps	Transactions per Second.

VIBES	Visualizations of Interactive, Blockchain, Extended Simulations.

Chapter 1

Introduction

There are papers describing blockchain technology as a disruptive innovation [9]. The *Harvard Business Review* argues blockchain technology is not a disruptive, but a foundational technology [10]. Whether blockchain is a disruptive technology or not, it is largely agreed upon that blockchain technology has immense potential and can revolutionize business and redefine companies and economies. But there are still lots of barriers. Currently and in the last years, blockchain technologies such as Bitcoin [11], Ethereum [12], IOTA [13] or Hyperledger fabric [14] are a heavily discussed topic. According to the Gartner Hype Cycle Blockchain technology was undergoing the peak of inflated expectation in 2017 [15]. Price and market capitalization changes of cryptocurrencies are widely covered by the media. This shows blockchain technologies are believed to have a great potential and a big variety of new use cases.

Blockchain technology enables decentralized consensus and can be used for record keeping. Bitcoin is the first digital currency to solve the double-spending problem without the need of a trusted authority. One of the main problems of bitcoin or in general of blockchains is the low number of maximal processed transactions per second. The bitcoin community disagrees about how to solve this scalability problem and supports contradicting bitcoin improvement proposals (BIP). Also, it split already into multiple communities with different approaches to the scalability problem.

VIBES (Visualizations of Interactive, Blockchain, Extended Simulations) is a blockchain simulator, which allows fast, scalable and configurable network simulations on a single computer without any additional resources. It was developed in a master thesis by Lyubomir Stoykov [16] and is the foundation of this master thesis.

1.1 Motivation

The implications of blockchain protocol changes on key figures such as scalability or security are difficult to predict. The reasons are complex relationships between the different parameters of the blockchain network, which can change the way nodes interact with each other or how blocks are propagated. This may lead to getting the impression that the blockchain network is a black box. A configurable simulator can give valuable insights into the inner workings of a blockchain network and helps to answer questions about scalability or security. The goal of this master thesis is to extend and improve VIBES to make more realistic bitcoin-like blockchain simulations with the latest features and attacks possible. In the future, BBSS could be used by developers or heavy blockchain users to simulate changes to different blockchains, so it can help the bitcoin community to agree on bitcoin improvement proposals.

1.2 Problem Statement

Ideally, the users of VIBES could specify their Bitcoin-like Blockchain Simulation. VIBES then presents the results of the simulation in a very detailed and still easily understandable way. In reality, VIBES does not yet support Bitcoin-like Blockchain Simulations. The generic simulation lacks defining traits of bitcoin such as the maximum block size, SegWit, transaction incentives or attack scenarios. Also, other important metrics such as the block time are not yet displayed. To achieve our goals VIBES needs to be changed in a way to allow multiple strategies in the front- and backend, while still having good maintainability of the code. A Bitcoin-like Blockchain Simulation needs to be implemented next to the existing generic simulation. These extensions can improve the quality of the simulations and the use cases of VIBES. For example, the implications of Segwit2x could be analysed. Maybe these extensions could also make it possible to realistically simulate the current bitcoin blockchain with a configuration as similar as possible. The focus of this master thesis is to implement a Bitcoin-like Blockchain Simulation with all important features to make the simulations as realistic as possible, to make it possible to simulate attacks such as double-spending and to visualize all important outputs. The approach is evaluated by seven key figures: correctness, speed, scalability, flexibility, extensibility, powerful visuals, and usability.

1.3 Approach

The presented goals should be achieved while maintaining and extending the design and architecture of the existing VIBES framework, which are described in Chapter 3. The Bitcoin-like Blockchain Simulation fast-forwards the whole network ahead of time and skips heavy computations such as solving a block. The actor model is used to achieve simulation at the event level. The Coordinator or also called MasterActor takes the role of an application-level scheduler to make fast-forwarding possible. The Coordinator, other actors, and the user interface need to be adjusted for the Bitcoin-like Blockchain Simulation and attack scenarios. For the visual side, the pattern of Atomic Design [7] is maintained to add new visualizations to a high-quality and composable user interface.

1.4 Contribution

Before BBSS, there was no multi-purpose bitcoin simulator with good visualisations. BBSS makes bitcoin-like blockchains easy to understand and allows the optimization of bitcoin-like blockchains in scalability and security. Different scalability related changes such as block size limit, SegWit or transaction incentives can be analysed. BBSS also allows the simulation of attacks such as double-spending and transaction spam attacks. The changes were implemented in a way to enable good maintainability to adjust the simulation to future changes of the bitcoin protocol or to add new attacks.

1.5 Organization

The structure of the thesis begins with this chapter as an introduction. The second chapter describes the background and the theoretical foundations which are necessary for the understanding of this thesis. The Chapter 3: Related Work presents related research papers and the architecture and design of VIBES on which this thesis builds on. And it shows not only works related to simulators but also works related to double-spending and transaction spam attacks and puts this work into perspective. Chapter 4 describes the approach and implementation details are explained. Developers or users of the simulator can use Chapter 4 as a documentation to explain questions about the behavior of the simulations and their results. Chapter 5: Evaluation uses empirical and theoretical analysis to test the implementation according to our predefined criteria. In the last chapter, the status, conclusions and, suggestions for future work are summarised.

Chapter 2

Background

This chapter will give an overview of the background concepts necessary to understand the following chapters. The nature of a simulator, the concepts of blockchain, bitcoin and various attacks are explained.

2.1 Simulation and Simulator

A simulation is the imitation of the operation of a real-world process or system [17]. This requires a model representing the key characteristics, behaviours, and functions of the selected system. It is a tool that can be manipulated to observe the outcomes of different assumptions or actions. It models the behaviour of a real system with minimum information loss, less used resources and ideally faster than real-time. Simulators are often used for the optimization of systems, studying and gaining insights into the functioning of simulation models.

2.2 Bitcoin

Bitcoin is a purely peer-to-peer electronic currency published in the paper *Bitcoin: A Peer-to-Peer Electronic Cash System* and as open-source software in 2009 by an unknown person or a group of people under the name Satoshi Nakamoto [11]. It allows online payments to be sent directly from one party to another without going through a financial institution or the need to trust a third party. Bitcoin not only refers to the currency but is also a currency unit and can be shortened to BTC. Bitcoin uses digital signatures and proof-of-work to prevent double-spending. Unless otherwise specified, the original paper [11], the Bitcoin Developer Guide [18], and the Bitcoin Wiki [19] were used as sources for bitcoin-related topics.

2.2.1 Double-Spending Problem

The double-spending problem refers to the problem of electronic cash to prevent some money being spent more than once. Malicious actors can try a double-spending attack to commit fraud. Merchants or users of bitcoin can reduce their double-spending fraud risk by increasing the number of confirmations which they are waiting for. Merchants can also introduce a limitation on how much bitcoin they accept per transaction.

In Chapter 2.2.8 the double-spending attack is explained and in 4.9.2 the maximal safe transaction value is calculated.

2.2.2 Proof-of-work and Blockchain

Digital signatures and proof-of-work provide the main benefit of not requiring a trusted third party to prevent double-spending. The peer-to-peer network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work also called blockchain. This process is called mining. The blockchain can't be changed without redoing the proof-of-work. One single part of this ongoing chain of proof-of-work is called a block. The longest blockchain serves as a proof of the history of transactions and blocks generated by the largest pool of CPU power [11].

VIBES already follows partly bitcoin's protocol [16]. This thesis's goal is to implement more features of the bitcoin protocol.

2.2.3 Block

In the bitcoin protocol, the blockchain is implemented as a directed tree consisting of blocks [16]. As shown in Figure 2.1 each block has certain data such as the hash of the previous block, the block timestamp, the transaction root, and the nonce.

The hash references to the block that came immediately before it. It is necessary to establish a chain of blocks.

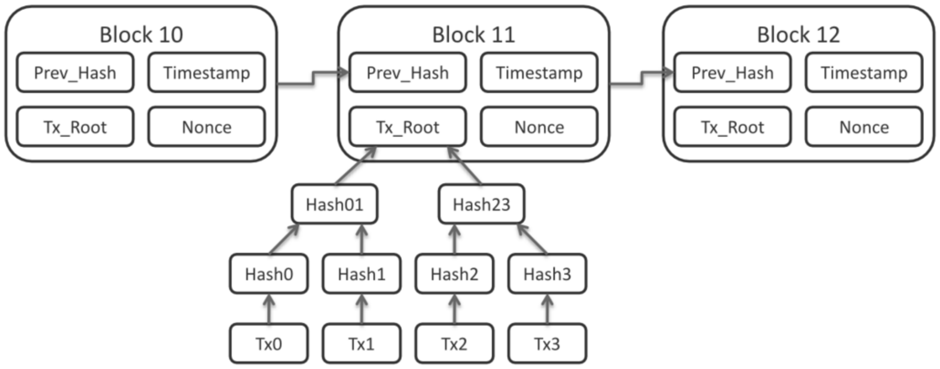


Figure 2.1: Bitcoin Block Data [1]

Each block contains a block timestamp. This timestamp serves as a source of variation for the block hash and makes it also more difficult for an adversary to manipulate the blockchain.

The transaction root is a Merkle root, it is the hash of all the hashes of all the transactions in the block. With this transaction root, it is possible to securely verify that a transaction has been accepted by the network and get the number of confirmations by downloading just the tiny block headers and the Merkle tree, downloading the entire blockchain is unnecessary [2].

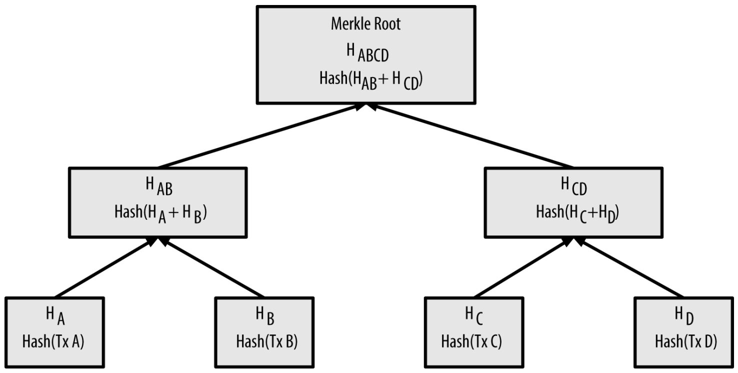


Figure 2.2: Merkle Tree [2]

The nonce is a 32-bit field, just the same as the block hash. It is adjusted by miners to make the hash of the block less than or equal to the current target of the network. This is often called a difficult-to-solve mathematical puzzle, the nonce is unique to each block.

Block Time and Difficulty

The current target of the network is related to the difficulty, the difficulty is a measure of how difficult it is to find a hash below a given target. The difficulty adjustment is necessary for the average block time to be close to the target block time. Deviations of the average block time from the target block time have an effect on the stale block rate.

A block in the bitcoin protocol is supposed to be mined every ten minutes. This block time depends on the difficulty and the hash-rate. If the hash-rate is increasing, then the block time is lower than ten minutes because the difficulty adjustment is delayed. It is supposed to happen every 2016 blocks or approximately every 14 days.

Genesis Block

The first block of a blockchain is called genesis block. Genesis is Ancient Greek and means creation or birth. Of course, the genesis block can't contain a hash reference to a previous block.

Orphan Block and Stale Block

A stale block is a valid block that has a known parent in the blockchain but is not part of the longest chain. Stale blocks occur if two miners mine the next block at the same time or a second miner mines a block while the block of the first miner is still propagated through the network. Despite having mined a valid block, the stale block miner does not receive any block reward since stale blocks are not part of the longest chain. If one part of the network receives a different valid block than another part of the network, then different parts of the network can work on different blocks since both blockchains have the same height. This process repeats until one of the blockchains is longer than the other. Usually, the first propagated block wins this race since it is likely that a larger part of the network is working on it. An orphan block is not to be confused with a stale block. An orphan block is a block which has no known parent in the currently-longest blockchain. This means a node received a block before receiving its parent block, which could be part of the blockchain. Orphan blocks can't be accepted by miners since blocks can only be validated with the knowledge of the parent block. Orphan blocks can occur during forks. Figure 2.3 visualizes these definitions.

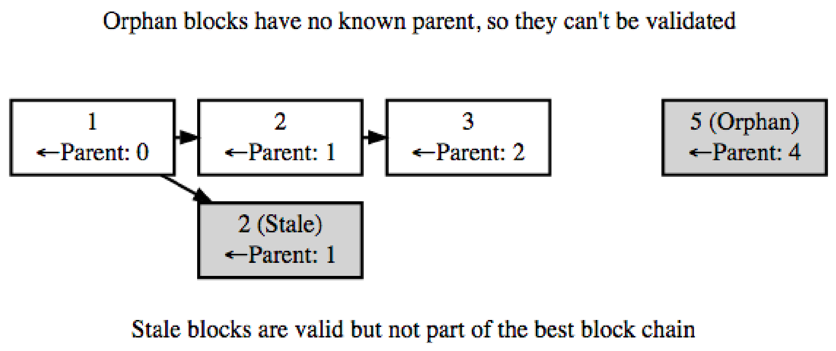

Orphan blocks have no known parent, so they can't be validated
1 ←Parent: 0
2 ←Parent: 1
3 ←Parent: 2
5 (Orphan) ←Parent: 4
2 (Stale) ←Parent: 1
Stale blocks are valid but not part of the best block chain

Figure 2.3: Stale and Orphan Blocks [3]

Block Size Limit

Blockchain protocols can define a block size limit which makes all miners reject all blocks with a higher block size. This limits the number of transactions per block depending on the average transaction size. The number of transactions per block (tpb) is also called throughput. Without a block size limit, a blockchain could theoretically process an unlimited number of transactions, but only a few miners would be able to process these blocks and participate in the network, therefore the network would centralize. This would violate the decentralization goal mentioned in the paper by Nakamoto.

In the original open-source software, the block size was indirectly limited to 32 MiB. In 2010 Satoshi Nakamoto secretly introduced a block size limit of 1 MB. The reason for this secret introduction is assumed to be the protection of the bitcoin network from a DoS attack using blocks of unlimited size. Some nodes would not be willing to accept big blocks and then the chain would split [20]. Until the introduction of SegWit, the maximum size of a Bitcoin block was 1 MB.

2.2.4 Transaction

A transaction is a transfer of BTC that is broadcast to the network and committed into one block. If more transactions are propagated than there is space in the current block, the non-processed transactions are saved in the transaction pools of the miners. A sender of a transaction has to pay a transaction fee to the miner for him to include the transaction into a block. Otherwise there would be no financial incentive for the miner to include transactions, instead, it would only cost him energy and time for the necessary computations. A miner that ignores transactions would be faster and more profitable. Transaction fees are the second financial incentive for miners next to block rewards. A miner can optimize his transaction fee earnings by ordering the transactions in his transaction pool by descending division of transaction fee through transaction size.

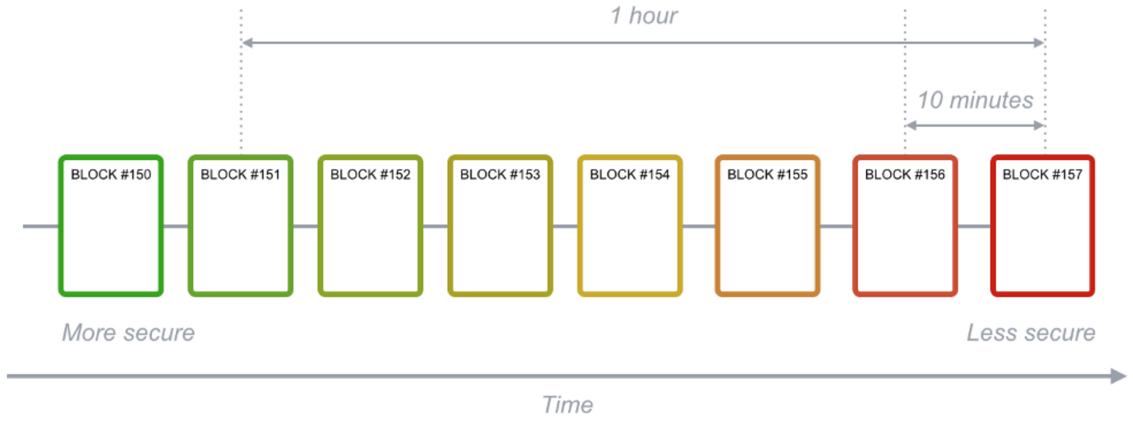


Figure 2.4: Transaction Security [4]

Confirmations

A transaction is confirmed as soon as the transaction is part of a block of the blockchain. A transaction can still be removed from the blockchain by creating a longer blockchain even if the transaction was confirmed in the most recent blocks. This longer blockchain replaces the previous one. For this reason, merchants wait for a certain number of confirmations. As illustrated in Figure 2.5, confirmations are the number of blocks that were created after the block with the transaction. Figure 2.4 shows requiring more confirmations reduces this risk. Six confirmations is a widespread recommendation and require you to wait on average for one hour to be certain about receiving a transaction. Six transactions mean that even if an attacker owns 20% of the hash-rate, he would only have a chance of 2.331% to achieve a successful double-spending attack.

Satoshi

Satoshi is currently the smallest unit of the bitcoin cryptocurrency named after the original creator. 100.000.000 satoshi equals one bitcoin. Transaction fees are most commonly denominated in satoshi.

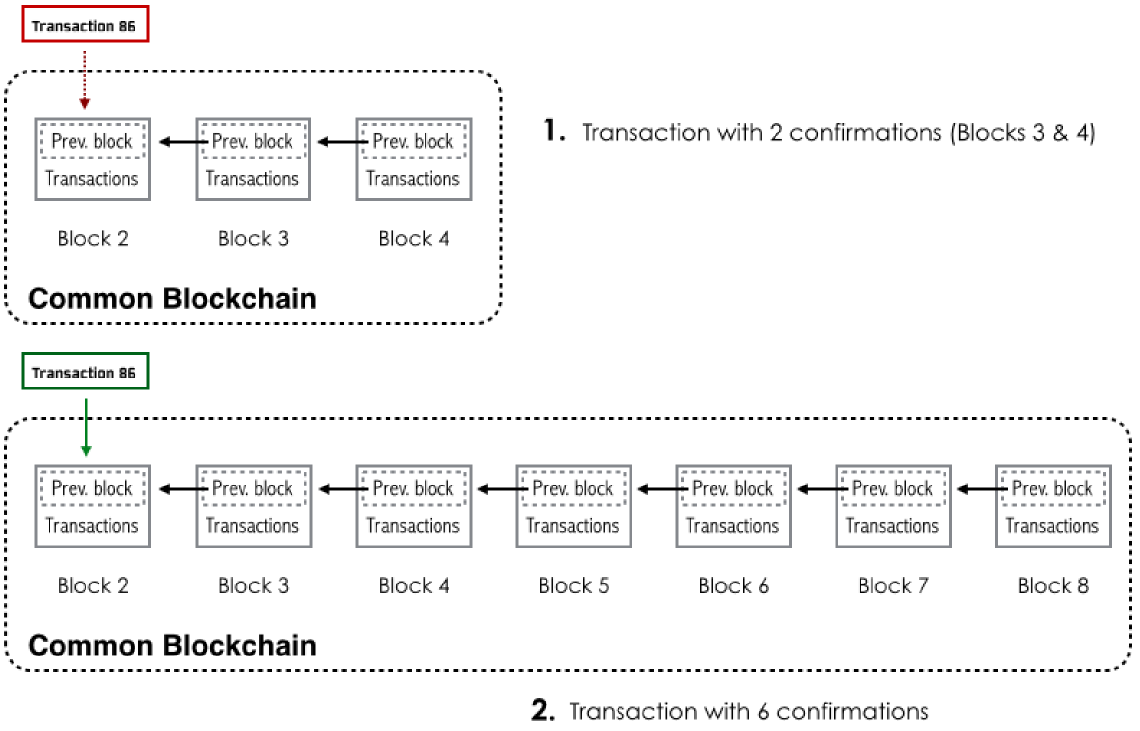


Figure 2.5: Confirmations [5]

2.2.5 Full Node and Miner

Full nodes are clients that have validated the whole blockchain self-sufficiently. They enforce all of bitcoin's rules on any received data and can't be cheated through invalid blocks or transactions.

A full node does not need to keep all blockchain data, it can also run in pruning mode [21]. Pruning mode allows the deletion of all data to make the blockchain size stay under a specified target size.

Also, a miner does not necessarily need the complete blockchain, the miner only needs to have the latest valid block. For example, members of mining pools only need to receive work from the mining pool. A miner is a provider of hash-rate for proof-of-work. Miners want to earn block rewards and transactions fees by adding blocks to the blockchain.

In the approach and the evaluation of this thesis, the terms miner and node are used interchangeably. The reason is VIBES only supports miners and used the term node instead of miner.

Block Rewards and Mining Pools

Miners are incentivized by block rewards to provide their hashing power. Mining is possible with CPUs, GPUs, and application-specific integrated circuits (ASIC). Mining is even possible with a sheet of paper and a pencil but the difficulty of blockchains such as bitcoin or ethereum makes it practically impossible. Since it can take years for most bitcoin miners to generate a block, mining pools were created to pool resources of miners together. The reward is then split equally according to their share of the contributed work.

Bitcoin as Currency

Bitcoins are created or minted from nothing during the creation of a block at a fixed rate of currently 12.5 bitcoins. The bitcoin protocol defines this block reward, every 210,000 blocks or approximately every four years the currency issuance rate decreases by 50%. The winning miner – the miner whose block got accepted by a majority of the network – of a block includes the receiving bitcoin address of the block reward into the block. Bitcoin has a maximum supply of almost 21 million coins which all will be mined approximately in the year 2140.

2.2.6 Fork

There are mainly four distinct meanings for fork. There is the chain fork, it occurs when multiple blocks are mined at the same height. Usually, this results in one of the blocks winning and the other blocks are stale blocks. The softfork is a change to the protocol wherein only previously valid blocks or transactions are made invalid. Softforks are backward compatible, SegWit is an example of a softfork. Thirdly, there is the hardfork. It makes previously invalid blocks or transactions valid. Hardforks are not backward compatible. Bitcoin Cash is an example. The (source) code fork is an altcoin that is a derivative of Bitcoin. For example, Litecoin is a code fork of Bitcoin, but neither is a hardfork nor a softfork. The reason for this is Litecoin and Bitcoin do not share the same genesis block.

2.2.7 Segregated Witness

Segregated Witness (SegWit) is a bitcoin softfork activated on 24th August 2017 as a solution to the scalability problem. After the activation of SegWit, the 1 MB block size limit was replaced with an almost 4 MB big block weight limit. The difference between size and weight is that in size all data types are equal, but weight rates witness data lesser than other types of data. This block weight limit of almost 4 MB is more of theoretical nature, to fill a block with 4 MB it requires all transactions to be very weirdly formatted. The softfork was intended to increase the block capacity, to increase the tps and therefore to increase scalability. This is achieved by defining a new structure called the witness, which is used to check transaction validity and is committed to a block separately from the transaction Merkle tree. The witness structure is not required to determine transaction effects. This approach achieves great backward compatibility, SegWit-enabled and Non-SegWit bitcoin nodes can work on the same blockchain. One of the points of concern is that SegWit is expected to increase the tps only by a factor between 1.8 to 2.3. The average transaction make-up in 2017 would lead to a block size of 2.3 MB if all transactions were SegWit transactions. But SegWit does not only increase the tps, it also allows other scalability solutions such as the Lightning Network to work by adding transaction malleability. Transaction malleability means the signature doesn't encompass all transaction data and a user could potentially change a transaction ID. Another point of concern is the necessary complex software update [22].

SegWit2x or also called SegWit2MB was an abandoned proposal to first activate SegWit and then make a hardfork within six months to increase the block size limit to 2 MB.

2.2.8 Double-Spending Attacks

The entirety of bitcoin's system of blockchain, mining, proof-of-work, difficulty etc. exist to make the history of transactions irreversible and to solve the double-spending problem. When bitcoin is used correctly, the transactions on the blockchain are irreversible and final. There are still scenarios to successfully spend bitcoin twice. These double-spending attacks depend on the number of confirmations a merchant or transaction receiver is waiting for and the hash-rate of the attacker. By redoing the proof-of-work and creating the longest blockchain an attacker can attempt a double-spending attack. An attacker can also abuse low-security confirmation and network settings.

Race Attack

A merchant or a transaction receiver operating his own bitcoin node who accepts a payment on seeing the transaction status "0/unconfirmed" is at risk of a race attack and double-spending fraud. A malicious actor could send a transaction directly to the node of the transaction receiver and a conflicting transaction with a higher transaction fee to the rest of the bitcoin network with a different transaction receiver. The transaction with the higher transaction fee is more likely to be mined into a block, this also depends on the transaction pool.

According to the research paper *Two Bitcoins at the Price of One* [23] an attacker has a high degree of success in performing a race attack

As precautions, a transaction receiver can disable incoming connections and can connect to only well-connected nodes to lessen the risk of a race attack, but the risk can't be eliminated. This is another reason why waiting for six confirmations is recommended. There is a theoretical solution to enable fast and secure bitcoin transactions, alerts in case of double-spending fraud suspicions [23]. But there is no adaptation yet and making fast and secure bitcoin transactions are still very difficult.

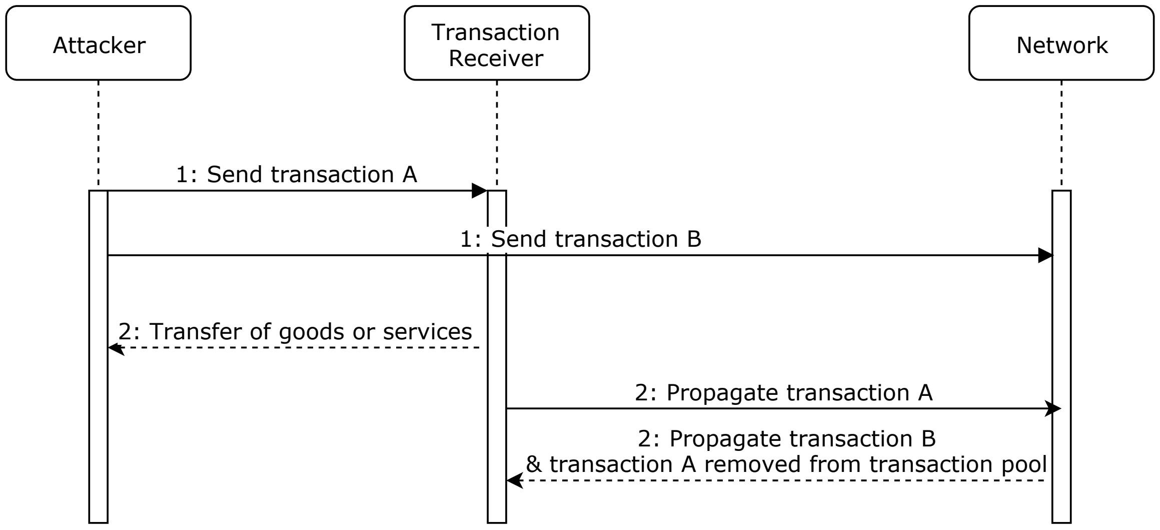

Attacker
Transaction Receiver
Network
1: Send transaction A
1: Send transaction B
2: Transfer of goods or services
2: Propagate transaction A
2: Propagate transaction B & transaction A removed from transaction pool

Figure 2.6: UML Communication Diagram: Race Attack

Finney Attack

The Finney attack is another attack on transaction receivers who accept payments on the transaction status "0/unconfirmed". The Finney attack requires hash-rate. The attacker mines a block and includes a transaction from address A to address B, which he both controls. Now he sends a transaction from address A to transaction receiver's address C, the transaction receiver thinks the transaction is final after receiving the transaction status "0/unconfirmed". But the attacker broadcasts his mined block afterwards. His transaction to address B takes precedence over the transaction to address C.

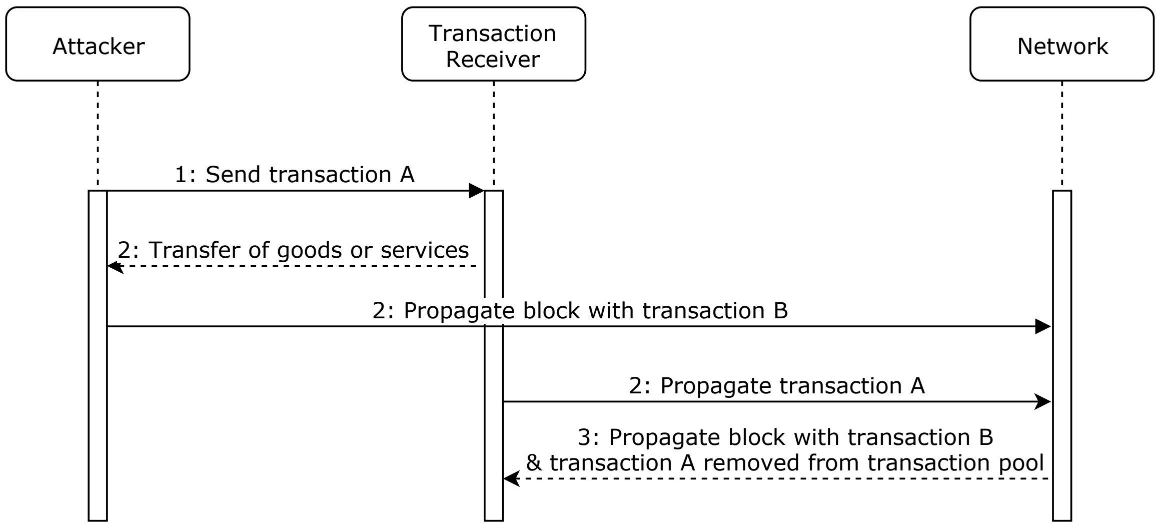

Attacker
Transaction Receiver
Network
1: Send transaction A
2: Transfer of goods or services
2: Propagate block with transaction B
2: Propagate transaction A
3: Propagate block with transaction B & transaction A removed from transaction pool

Figure 2.7: UML Communication Diagram: Finney Attack

Vector76 Attack

The Vector76 attack is a complex attack combining the race and the Finney attack, which can even reverse transactions included in the latest block. The malicious miner has to find an opportunity worth more than the current block reward to make the attack profitable because the attack requires the miner to intentionally let a mined block become a stale block.

The malicious miner solves a block and includes transaction A sending BTC to a victim. Instead of broadcasting the solved block, the miner broadcasts the transaction A via node A (connected directly to the victim) and a second transaction B (via a well-connected) node B. Transaction B does not send money to the victim but to the malicious miner. Eventually, another miner will solve a block and include either transaction A or transaction B. The connectivity of node B and the transaction fees of transaction B make it more likely to take precedence over transaction A. The malicious miner sends his own solved block with transaction A via node A to the victim after seeing the solved block with transaction B via node B. At this moment the victim only sees transaction A in the block mined by the malicious miner, assumes everything is correct and does a beneficial action for the malicious actor. Now there are effectively two branches of the blockchain since it is likely that the majority of the hashing power has the block with transaction B, it will create a new child block and therefore be the longest blockchain and erase the other branch with transaction A. If transaction A gets propagated faster, there should be no/minimal loss for the malicious actor, except for the costs to produce a stale block. In case the block with transaction A gets propagated faster, the malicious actor earns the block reward and transaction fees.

A Vector 76 attack has a very high chance to be profitable, but it is very unlikely to find such an opportunity.

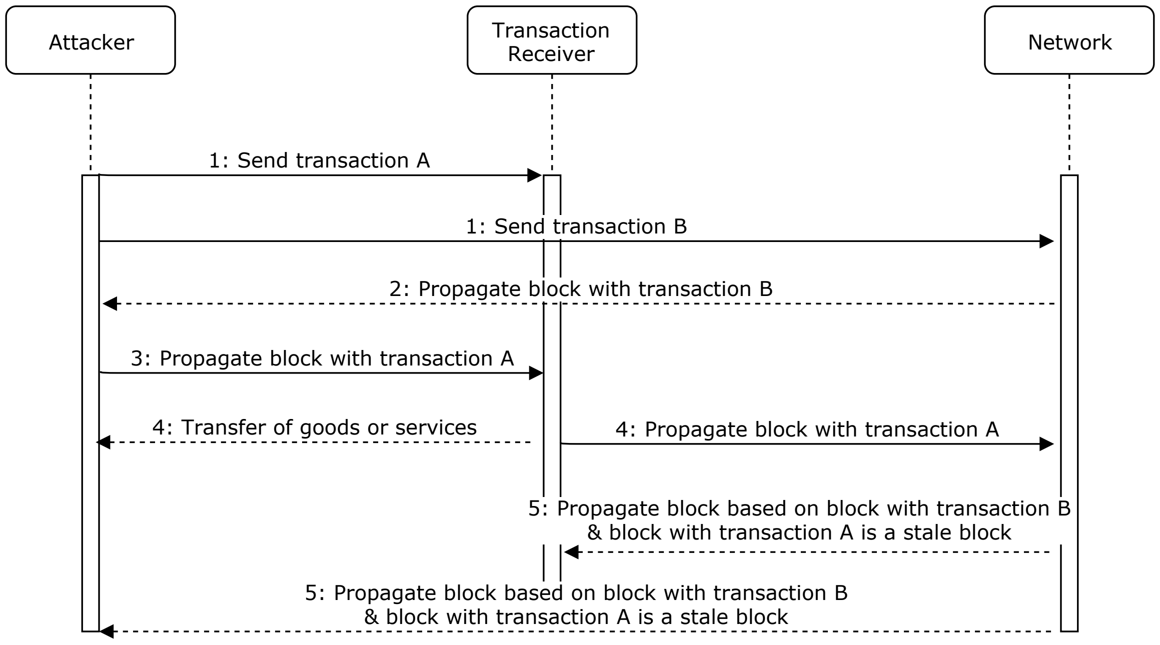

Attacker
Transaction Receiver
Network
1: Send transaction A
1: Send transaction B
2: Propagate block with transaction B
3: Propagate block with transaction A
4: Transfer of goods or services
4: Propagate block with transaction A
5: Propagate block based on block with transaction B & block with transaction A is a stale block
5: Propagate block based on block with transaction B & block with transaction A is a stale block

Figure 2.8: UML Communication Diagram: Vector76 Attack

Alternative History Attack

Figure 2.9 visualizes the concept behind the alternative history and the majority attack. In contrast to the previous attacks, the concept behind these attacks is more well-known.

Comparable to the Finney and Vector76 attacks, the attacker needs a significantly higher hash-rate. The alternative history attack can even work if the transaction receiver waits for some confirmations. The higher the attacker's percentage of the network's total hash-rate, the more confirmations are needed to prevent double-spending.

As the name alternative history attack implies, the miner starts working on an alternative history, his own private blockchain after his transactions were included in a block A. Multiple transactions and double-spending attempts targeting different victims at the same time make the attack more profitable. The private blockchain has the same parent block as block A and the first block includes the fraudulent double-spending transactions. After the victims waited for their number of confirmations, accepted the transactions and did something beneficial for the attacker, the attacker makes his private blockchain public

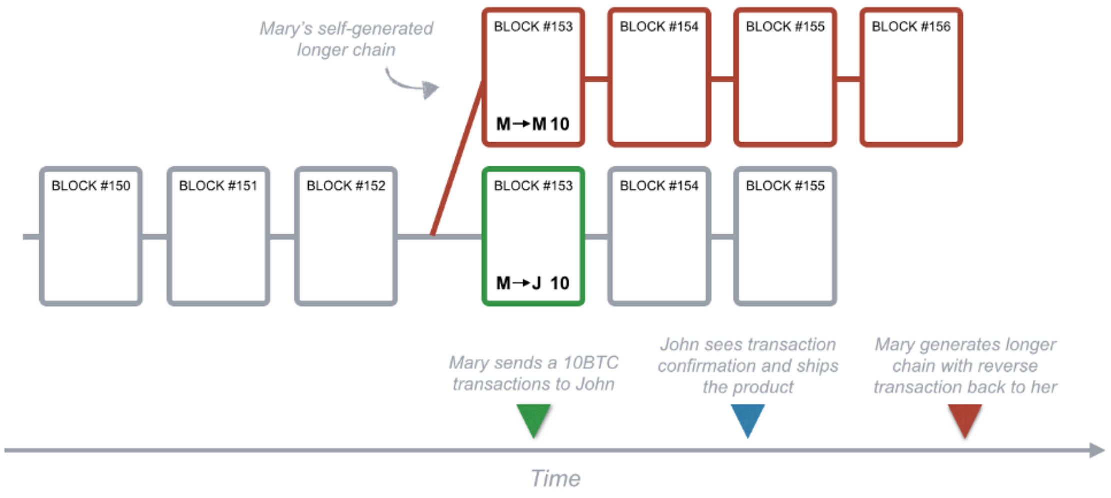
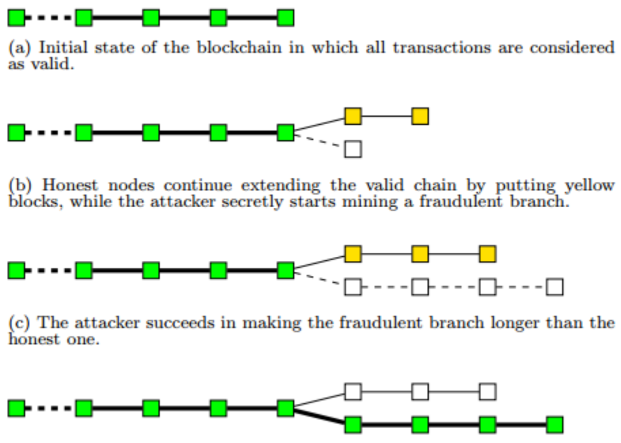

Mary's self-generated longer chain
BLOCK #153
M→M 10
BLOCK #154
BLOCK #155
BLOCK #156
BLOCK #150
BLOCK #151
BLOCK #152
BLOCK #153
M→J 10
BLOCK #154
BLOCK #155
Mary sends a 10BTC transactions to John
John sees transaction confirmation and ships the product
Mary generates longer chain with reverse transaction back to her
Time

Figure 2.9: Double-Spending Attack [4]

as soon as it is longer than the original blockchain and creates hereby an alternative history. In case of success, the attacker regains his spent bitcoins and receives beneficial actions from the victims. In case of failure, the attacker has to bear the hash-rate costs and pay for the bought goods or services.

The success probabilities of an alternative history attack depending on the attacker's hash-rate and the transaction receiver's number of confirmations are displayed in the Table A.1.



(a) Initial state of the blockchain in which all transactions are considered as valid.

(b) Honest nodes continue extending the valid chain by putting yellow blocks, while the attacker secretly starts mining a fraudulent branch.

(c) The attacker succeeds in making the fraudulent branch longer than the honest one.

(d) The attacker's branch is published and is now considered the valid one.

Figure 2.10: Development of a Double-Spending Attack over Time [6]

Maximal Safe Transaction Value

The maximal safe transaction value is the value a participant of the bitcoin network can send safely depending on the success probability of the double-spending attack and other variables such as the block reward as can be seen in the Formula (4.4). The maximal safe transaction values are shown in Table A.2. If a merchant accepts a transaction with a higher value, then it would be profitable under the given parameters to launch a double-spending attack.

Majority Attack

The majority attack is also called 51% attack. The concept behind the alternative history attack and the majority attack is the same, the difference is the attacker's percentage of the total hash-rate of the network. For a majority attack equal to or more than 50% of the total network's hash-rate is necessary. A majority attack has a probability of 100% to succeed, no amount of confirmations can prevent this attack. With at least 50% of the hash-rate, the attacker can work secretly until his private blockchain is longer.

Economic Majority

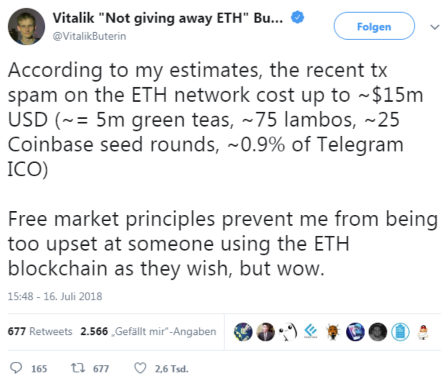
For an attacker, a majority attack on bitcoin can be catastrophic due to the effects on the market and the attacker's very high commitment to ASIC mining hardware. A miner with more than 50% of the hash-rate is therefore incentivized to calm down the market and to reduce his mining power and abstain from attacking to protect the value of his mining hardware.

Altcoins of bitcoin or cryptocurrencies, for which a cryptocurrency with a similar algorithm and higher hash-rate exists, are at risk from majority attacks.

2.2.9 Transaction Spam Attack

In July 2018 the Ethereum Network was affected by transaction spam or also called flood attack. In such a flood attack, the attacker, in principle, trades their own cryptocurrency for increased transaction costs for everyone by only using intended functionality and valid transactions. Vitalik Buterin tweeted about this attack [24], the tweet can be seen in Figure 2.11. This transaction spam is also possible in the bitcoin network. Interesting research questions arise about the costs which a malicious actor has to pay to make the bitcoin network unusable or too uneconomical to use for certain use-cases. Such an attack is only limited to the attacker's number of bitcoin and depends on the target transaction fee, the duration of the attack and the scalability of the network. An attacker with a certain amount of money can make blockchains for other users for a certain time unusable.

There are also other definitions for transaction spam or flood attacks. In this thesis, we assume a transaction spam attack only uses valid transactions.



Vitalik "Not giving away ETH" Bu...
@VitalikButerin

According to my estimates, the recent tx spam on the ETH network cost up to ~$15m USD (~= 5m green teas, ~75 lambos, ~25 Coinbase seed rounds, ~0.9% of Telegram ICO)

Free market principles prevent me from being too upset at someone using the ETH blockchain as they wish, but wow.

15:48 - 16. Juli 2018

677 Retweets 2.566 „Gefällt mir"-Angaben

165 677 2,6 Tsd.

Figure 2.11: Tweet from Vitalik Buterin about Transaction Spam

Chapter 3

Related Work

Different related works were already presented in VIBES. The most similar related work is the Bitcoin-Simulator, its architecture and design are different, it can only simulate up to 6,000 nodes and has no transactions. In contrast, in VIBES simulations with transactions are possible and the number of nodes is not restricted. The two other presented related works are the bitcoin testnet and the back of the envelope approach. Testing and analyzing metrics on the bitcoin testnet is essentially the same as on the public bitcoin network and is almost as resource-intensive. The back of the envelope approach using empirical data to infer certain properties of the network is also impractical due to the very expensive, time-consuming and non-configurable nature [16].

Despite the existence of numerous bitcoin simulators, since the release of VIBES until the writing of this thesis, nothing has been published as similar to VIBES as the Bitcoin-Simulator.

There are other works focusing solely on a single aspect of the bitcoin network, for example, in block mining [25] or network [26] [27]. In contrast to other works, we propose a simulator which addresses multiple aspects of a bitcoin-like blockchain network and new aspects can easily be added. Another difference is the offering of a rich user interface for configuring the simulation environment and investigating the simulation results. Also, the designs and architectures are completely different.

To understand the approach of this thesis, an overview over the existing classes and concepts of VIBES is necessary.

For the double-spending attack the research paper *Analysis of hash rate-based double-spending* is presented and for the flood attack the research paper *Stressing Out: Bitcoin "Stress Testing"*.

3.1 VIBES: Fast Blockchain Simulations for Large-scale Peer-to-Peer Networks

The architecture and design lend itself to being split up into the frontend and the backend. The communication channel between both is HTTP.

3.1.1 Frontend

As JavaScript framework React was chosen for the frontend [28]. The most interesting part of the frontend is the Atomic Design [7]. This approach makes it easy to add new components to the user interface or to remove components from pages. The modularity makes the maintainability easy. Atomic design is a philosophy that encourages the composition of entities. There are four of these entities, the smallest components such as buttons and inputs are called atoms. Atoms, in turn, assemble molecules, several molecules make up an organism. Finally, several organisms create a page. Actually, several organisms create templates which then create pages, but templates are not used in VIBES. The reason for not using templates could be that this abstraction layer was not needed.

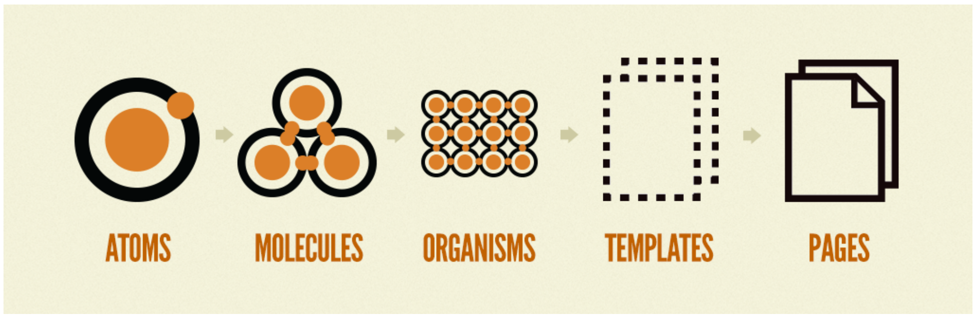


Figure 3.1: Atomic Design [7]

3.1.2 Backend

The backend of VIBES has in principle four main types of classes: actions, actors, models, and utils. The following sub-chapters explain them and the Main object.

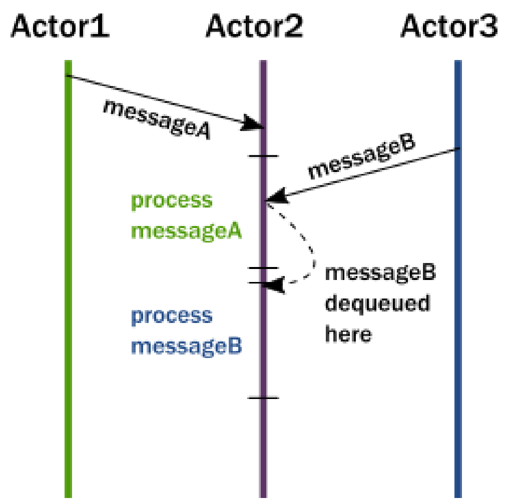


Figure 3.2: The Actor Model - Usage of message passing avoids locking and blocking

Actors and Actions

VIBES uses the Actor Model [29]. An Actor is a computational entity, the primary unit of concurrency and actors communicate via messages. There is no shared state and message-first communication, therefore no locks and no blocking exist as shown in Figures 3.2 and 3.3. This is very important to make sure the simulator is fast.

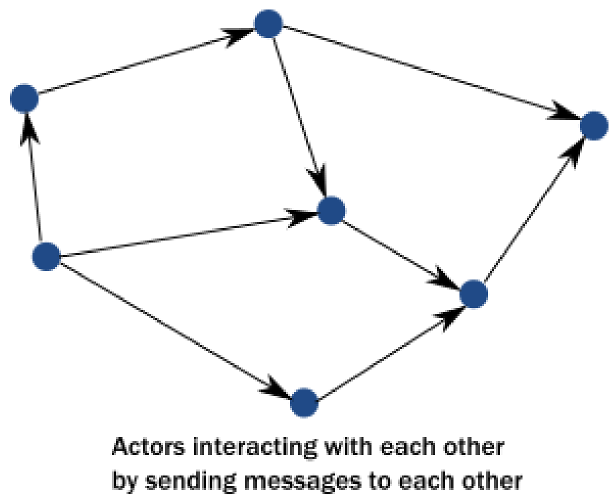

Actors interacting with each other by sending messages to each other

Figure 3.3: The Actor Model - Actors interacting with each other

The implementation in Scala [30] and Akka [31] has five actors with their corresponding actions. Actions describe the methods of Actors. Figure 3.4 shows this architecture.

MasterActor & MasterActions: The MasterActor is also called Coordinator, controls the nodes and the execution order in the network. The MasterActor grants permission to

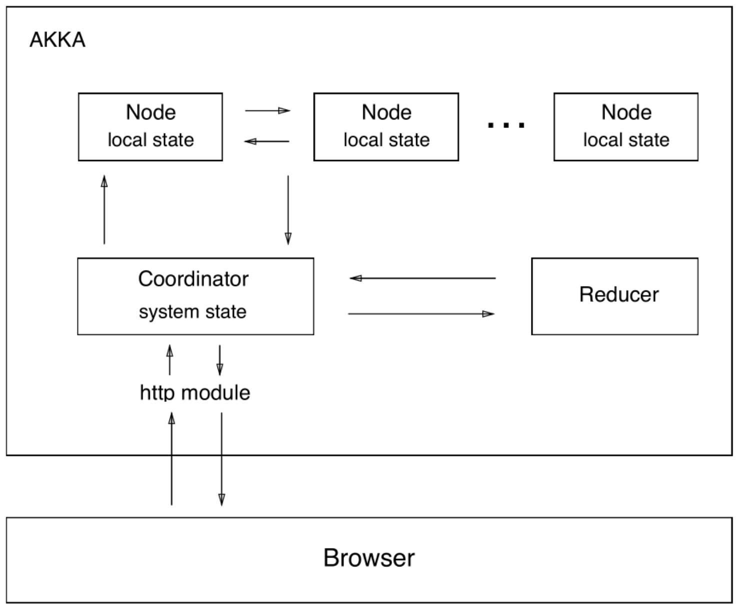


Figure 3.4: VIBES' Architecture

nodes to work on a work request. A **work request** is a piece of computational unit for whose execution the node needs permission, for example, mining a block. If the MasterActor wants to execute a work request with a timestamp in the future, it can **fast-forward** the entire system to this point in the future.

DiscoveryActor & DiscoveryActions: The DiscoveryActor is responsible for assigning and updating the neighbors of nodes.

NodeActor & NodeActions: The NodeActor is the equivalent to a miner in the bitcoin network, interacts with other nodes, blocks, and transactions. The NodeActor is also responsible for the generation of the best guess of a work request. The **best guess** is, for example, the time in the future when a node wants to mine a block.

NodeRepoActor & NodeRepoActions: The NodeRepoActor helps the MasterActor, is the repository for all nodes, starts and ends the simulation.

ReducerActor & ReducerActions: The ReducerActor is called after the simulation to calculate, summarize and prepare the simulation results for the transfer to the user interface.

Models

There are five data structures called models.

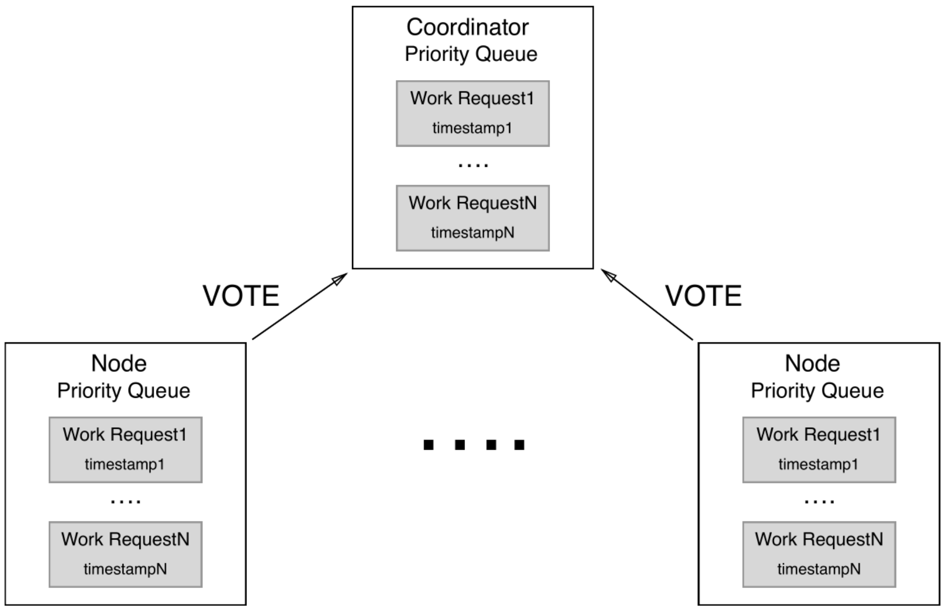


Figure 3.5: Nodes voting to fast-forward

VNode: VNode is the model of a node in the bitcoin network. It has methods to interact with blocks, transactions, nodes, and the blockchain.

VBlock: VBlock is the model of a block in the bitcoin network, it also has a method to calculate the propagation times of a block.

VTransaction: VTransaction is the model of a transaction in the bitcoin network.

VEventTypes: VEventTypes is a special model to provide the frontend with information about the simulation and events. There are three types: *MinedBlock*, *TransferBlock* and *ReducerResult*.

VRecipient: VRecipient is a very simple model of an event receiver consisting only out of two nodes and a timestamp.

Utils

VExecution: Each work request has a **type**, there are four types which are defined in VExecution.

- *MineBlock* is a node's work request to solve the current block.
- *IssueTransaction* is a node's work request to create a transaction.
- *PropagateTransaction* is a node's work request to propagate transactions.

- *PropagateOwnBlock/PropagateExternalBlock* is a node's work request to propagate a block to its neighbours.

VConf: The **configuration parameters** from the user interface are saved in the VConf which can be accessed globally.

Joda: An object for ordering the timestamps.

Main

The Main object starts the backend. It gets the configuration parameters from the frontend, saves them in the VConf and prepares the simulation. Then it starts the simulation. After the end of the simulation, the Main object provides the simulation results to the frontend.

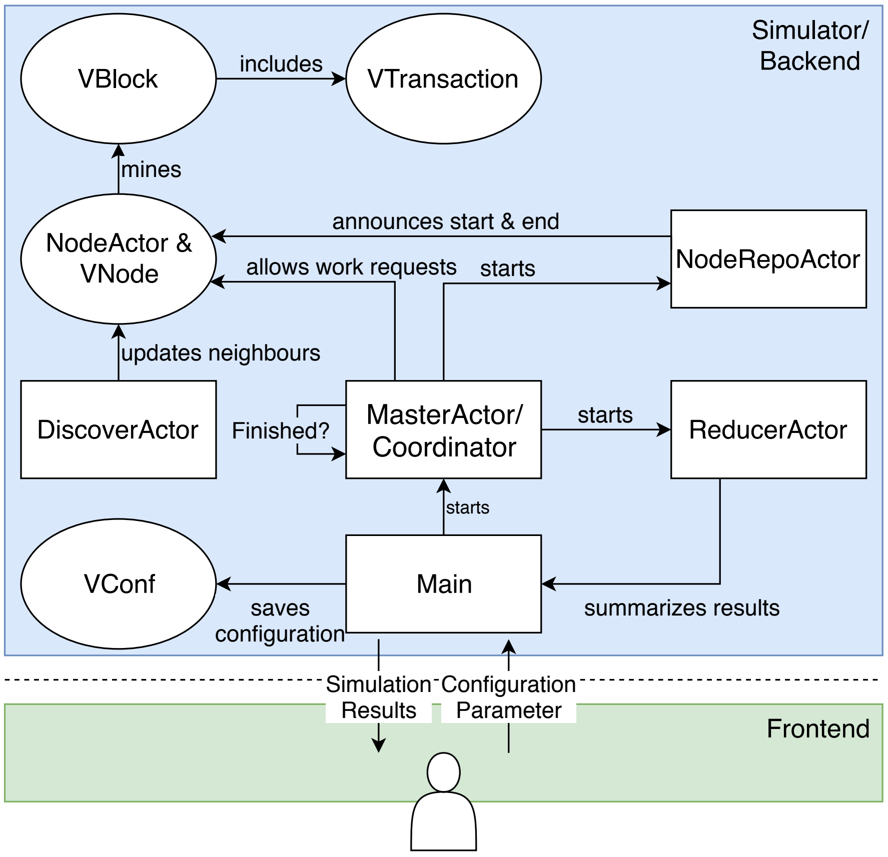


Figure 3.6: Simplified Architecture

3.2 Analysis of hash rate-based double-spending

The research paper *Analysis of hash rate-based double-spending* focuses on the quantitative aspects of bitcoin's double-spending prevention and how they relate to attack vectors and their countermeasures. It takes a look at the stochastic processes underlying typical attacks and their resulting probabilities of success. It provides a formula to calculate the success probability of a double-spending attack depending on the attacker's percentage of the network's total hash-rate and the confirmations the merchants are waiting for. A formula to calculate the maximal safe transaction value is also provided. Both of these formulas are used in this thesis and the models with their configuration parameters are implemented.

3.3 Stressing Out: Bitcoin "Stress Testing"

The research paper *Stressing Out: Bitcoin "Stress Testing"* is an empirical study of a spam campaign in the summer 2015 that resulted in a DoS attack on bitcoin [32]. The attacker tested several different attack vectors with valid and invalid transactions and even managed to crash over 10% of all bitcoin nodes at one point. The impacts were measured, for example, the total costs of the attack and the average price increase for transactions are calculated.

Chapter 4

Approach

In this chapter, the changes to the existing VIBES framework are presented one by one. The reasons for each improvement are described. The prerequisites for every change and the design and architecture are shown. The implementation details are explained.

The implementation is split into frontend and backend. Since the frontend is only displaying the information from the backend, the focus of this thesis is on the backend where the changes to the actual behaviour of the simulation are done. The frontend, the console output, and the log file show the results of the backend and can be used for the evaluation.

4.1 Bitcoin-like Blockchain Simulation

Extending the previous *Generic Simulation* to *Bitcoin-like Blockchain Simulation* made lots of changes necessary, especially in the frontend. More abstract ways to implement different strategies in the backend in Scala were researched. The backend differentiates between the currently only two strategies mainly with If-clauses. This seemed to be the best option, which avoids creating unnecessary complexity. It was also recommended by the author of VIBES. Bloated methods due to having multiple strategies in one method can be avoided by outsourcing strategy-specific parts into their own methods.

4.2 Time between Blocks

The time between the blocks or also called block time is a very important metric. It can be used to check the overall system health. Due to the nature of the bitcoin protocol, a new block can immediately be found after the last one. Very short block times can lead to unusual behaviour, as nodes might not have enough time to send transactions, synchronise their transactions pools or the blockchain. Therefore the block time can explain unusual behaviour. The user interface needs the corresponding data to display the time between blocks chart in Figure 4.1. To calculate the duration of the first block, a new timestamp was introduced to save the start time of the simulation.

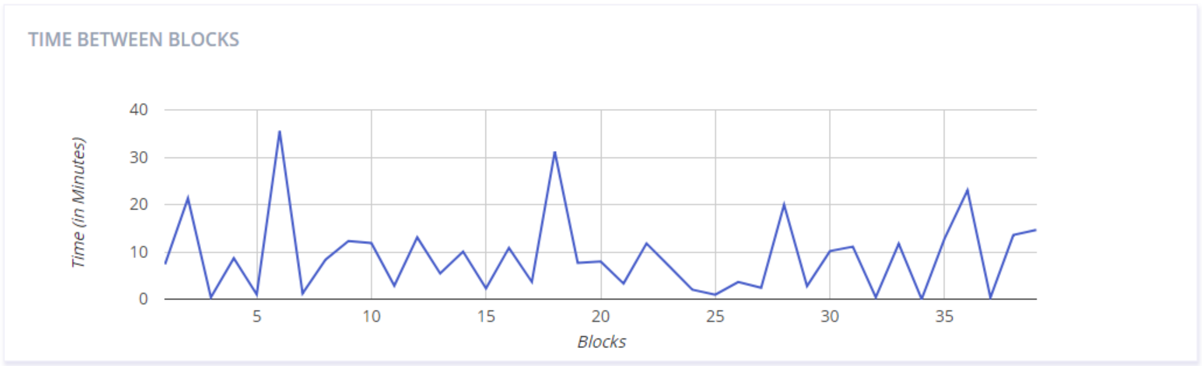


Figure 4.1: Screenshot Time between Blocks

4.3 Block Size Limit

Previously VIBES had no block size limit. This means infinite transactions can be processed and changing input parameters has no effect on the scalability. To be able to investigate the effects of different input parameters on the scalability, the introduction of a block size limit is necessary. This allows a more accurate simulation of bitcoin. The block size limit is, for example, necessary to analyse the implications of Segwit2x according to VIBES [16].

The only prerequisite is to add an additional configuration parameter *maxBlockSize*: the maximal block size in B, the current default value is 50.000 B.

The design and architecture changes of the backend mainly happen in the model VBlock. All generated blocks obey the block size limit depending on if the simulation is a Bitcoin-like Blockchain Simulation. If the block size limit is set to zero, the simulator assumes the block size is unlimited and the maximum number of transactions per block is infinite.

Listing 4.1: VBlock with focus on block size limit

```
object VBlock extends LazyLogging {
  def createWinnerBlock(node: VNode, timestamp: DateTime): VBlock = {
    var maxTransactionsPerBlock : Int = 0
    var processedTransactionsInBlock: Set[VTransaction] = Set.empty

    if (VConf.strategy == "BITCOIN_LIKE_BLOCKCHAIN" && VConf.maxBlockWeight !=
        ↪ 0) {
      maxTransactionsPerBlock = Math.floor(VConf.maxBlockSize /
          ↪ VConf.transactionSize).toInt

      // takes the amount of maxTransactionsPerBlock out of the transaction
          ↪ pool into the winner block
      processedTransactionsInBlock =
          ↪ node.transactionPool.toSeq.take(maxTransactionsPerBlock).toSet
    } else {
      maxTransactionsPerBlock = node.transactionPool.size
      processedTransactionsInBlock = node.transactionPool
    }

    VBlock(
      id = UUID.randomUUID().toString,
      origin = node,
      transactions = processedTransactionsInBlock,
      level = node.blockchain.size,
      timestamp = timestamp,
      recipients = ListBuffer.empty,
      transactionPoolSize = node.transactionPool.size
    )
  }
}
```

The Listing 4.1 shows only the essential lines of code. First, the maximum number of transactions per block is calculated via the maximum block size and the transaction size. For simplicity VIBES chose to have a constant transaction size, therefore this calculation is simple and it could also be done only once. It was a conscious decision to implement it in this way to make the implementation of variable transaction sizes in the future easier. Rounding down the maximal transactions per block makes sure the actual block size is smaller than the limit. Finally, the transactions are taken out of the transaction pool and later this set of transactions is used in the creation of the winner block.

4.4 Segregated Witness

In Chapter 4.3 the block size limit was implemented to make the simulation more accurate and similar to the actual bitcoin network. In this subchapter, one step further is taken. To accurately simulate the bitcoin network the block size limit needs to replaced by a block weight limit such as the actual bitcoin network did with the softfork SegWit.

For a very simple implementation of SegWit, one could maybe just introduce a boolean segWitEnabled and replace the already existing block size limit with a block weight limit and the transaction size with a transaction weight. Since the comparison between non-SegWit and SegWit key figures could be a very interesting use-case, new configuration parameters are introduced instead.

- *blockWeightLimit*: maximal block weight limit in weight unit
- *transactionWeight*: witness data per transaction in weight unit

As can be seen in Listing 4.2 a new condition for SegWit is added to the VBlock object.

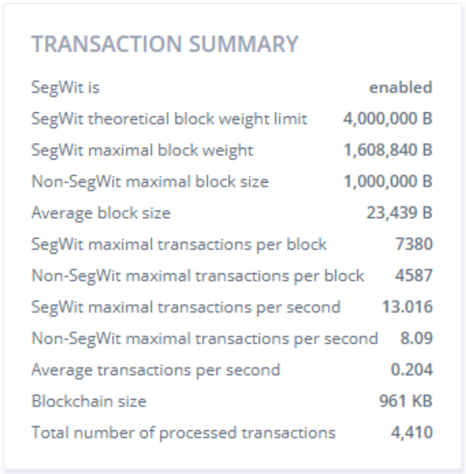
Listing 4.2: VBlock with focus on block weight limit

```
if (VConf.strategy == "BITCOIN_LIKE_BLOCKCHAIN" && VConf.transactionSize
  ↪ != 0) {
  // this part could be moved to Main for constant transaction weight and
    ↪ size to save calculations, but is necessary here for non-constant
    ↪ transaction weight and size
  if (VConf.maxBlockWeight != 0 && VConf.transactionWeight != 0) {
    // SegWit is enabled
    maxTransactionsPerBlock = Math.floor(VConf.maxBlockWeight /
      ↪ VConf.transactionWeight).toInt
  } else if (VConf.maxBlockSize != 0) {
    // SegWit is disabled
    maxTransactionsPerBlock = Math.floor(VConf.maxBlockSize /
      ↪ VConf.transactionSize).toInt
  } else {
    // any number of transactions is accepted
    maxTransactionsPerBlock = node.transactionPool.size
  }

  // sorts the transaction pool by the transaction fee and takes the
    ↪ amount of maxTransactionsPerBlock out of the transaction pool into
    ↪ the winner block
  processedTransactionsInBlock =
    ↪ node.transactionPool.toSeq.sortWith(_.transactionFee >
    ↪ _.transactionFee).take(maxTransactionsPerBlock).toSet

  // sets confirmation status of transaction true
  processedTransactionsInBlock.foreach { _.confirmation = true }

  // sets confirmation level of transaction
  processedTransactionsInBlock.foreach { _.confirmationLevel =
    ↪ node.blockchain.size }
} else {
  maxTransactionsPerBlock = node.transactionPool.size
  processedTransactionsInBlock = node.transactionPool
} ... }
```

Comparisons between the SegWit vs Non-SegWit metrics are provided in the frontend. These are calculated in the ReducerActor as can be seen in Listing 4.3. The SegWit theoretical block weight limit and the Non-SegWit maximal block size are given input values. The SegWit maximal block weight considers the transaction size and is more realistic than the theoretical block weight limit. Additionally, both maximal transactions per block and transactions per second values show the differences between SegWit and Non-SegWit while the actual simulation values are also shown in Figure 4.2.

Listing 4.3: Calculations for the comparisons in the ReducerActor

```
// works only for constant transaction size and weight, otherwise an array is
    ↪ necessary
    var segWitMaxBlockWeight = 0 // nonSegWitMaxBlockSize = VConf.maxBlockSize
    var segWitMaxTransactionsPerBlock = 0
    var nonSegWitMaxTransactionsPerBlock = 2147483647
    var maxTransactionsPerBlock = 0
    var segWitMaxTPS: Double = 0
    var nonSegWitMaxTPS: Double = 0
    if (VConf.transactionSize != 0) {
      // multiplies by 1000 because maxBlockSize is in KB and transaction size
          ↪ is in B
      nonSegWitMaxTransactionsPerBlock = Math.floor(VConf.maxBlockSize * 1000
          ↪ / VConf.transactionSize).toInt
      maxTransactionsPerBlock = nonSegWitMaxTransactionsPerBlock
      nonSegWitMaxTPS = nonSegWitMaxTransactionsPerBlock.toDouble /
          ↪ VConf.blockTime.toDouble
      nonSegWitMaxTPS = (math rint nonSegWitMaxTPS * 1000) / 1000
    }
    if (VConf.maxBlockWeight != 0 && VConf.transactionWeight != 0) {
      segWitMaxTransactionsPerBlock = Math.floor(VConf.maxBlockWeight /
          ↪ VConf.transactionWeight).toInt
      segWitMaxBlockWeight = segWitMaxTransactionsPerBlock *
          ↪ VConf.transactionSize
      maxTransactionsPerBlock = segWitMaxTransactionsPerBlock
      segWitMaxTPS = segWitMaxTransactionsPerBlock.toDouble /
          ↪ VConf.blockTime.toDouble
      segWitMaxTPS = (math rint segWitMaxTPS * 1000) / 1000
    }
```



TRANSACTION SUMMARY

SegWit is	enabled
SegWit theoretical block weight limit	4,000,000 B
SegWit maximal block weight	1,608,840 B
Non-SegWit maximal block size	1,000,000 B
Average block size	23,439 B
SegWit maximal transactions per block	7380
Non-SegWit maximal transactions per block	4587
SegWit maximal transactions per second	13.016
Non-SegWit maximal transactions per second	8.09
Average transactions per second	0.204
Blockchain size	961 KB
Total number of processed transactions	4,410

Figure 4.2: Screenshot Transaction Summary

4.5 Transactions per Second

One of the biggest unresolved issues of bitcoin-like blockchains is scalability. The main metric to measure scalability is transactions per second (*tps*). One drawback of this metric is that it contains no information about the transaction size or the usefulness of the transaction.

According to the research paper Bitcoin-NG the *tps* of bitcoin pre-SegWit was limited to only 1 to 3.5 *tps* for typical transaction sizes due to the block size at 1 MB [33]. For bitcoin heavy transaction loads are an obstacle for a more widespread usage [34]. A payment processor such as VISA handles 4,000 transactions per second on average and has been stress-tested in 2013 to handle 47,000 transactions per second. In comparison, bitcoin can only handle 7 transactions per second, due to the fact that block sizes are restricted to have a maximum size of 1 MB.

Of course, SegWit increased those numbers.

The transactions per second (*tps*) is also called throughput or transaction rate, Figure 4.2 also shows the average *tps*. The average *tps* is - as the Listing 4.4 shows - calculated over the duration of the whole simulation and then given to the frontend.

Listing 4.4: Calculations for the tps in the ReducerActor

```
var actualTPS: Double = longestChainNumberTransactions.toDouble /
    ↪ secondsBetween(VConf.simulationStart,
    ↪ VConf.simulateUntil).getSeconds.toDouble
actualTPS = (math rint actualTPS * 1000) / 1000
```

In our tool the *average transactions per second* can be higher than the *SegWit maximal transactions per second* or the *non-SegWit maximal transactions per second*. This should happen when the simulated block time is shorter than the input simulation time and the blocks are full. The *average transactions per second* is based on the simulated block time, the maximal tps values are based on the input simulation time.

4.6 Processed and Pending Transactions per Block

After introducing the block size limit in Chapter 4.3 and SegWit in Chapter 4.4, it would be very insightful to have visualisations about how many transactions are actually included in blocks and how many transactions are pending. This would also be very helpful for the evaluation of correctness.

Changes in the models VBlock and VEventTypes and the ReducterActor were done for the pending transactions per block chart in Figure 4.3. As a new attribute, every block has the transaction pool size minus the included number of transactions at the time of the block creation.

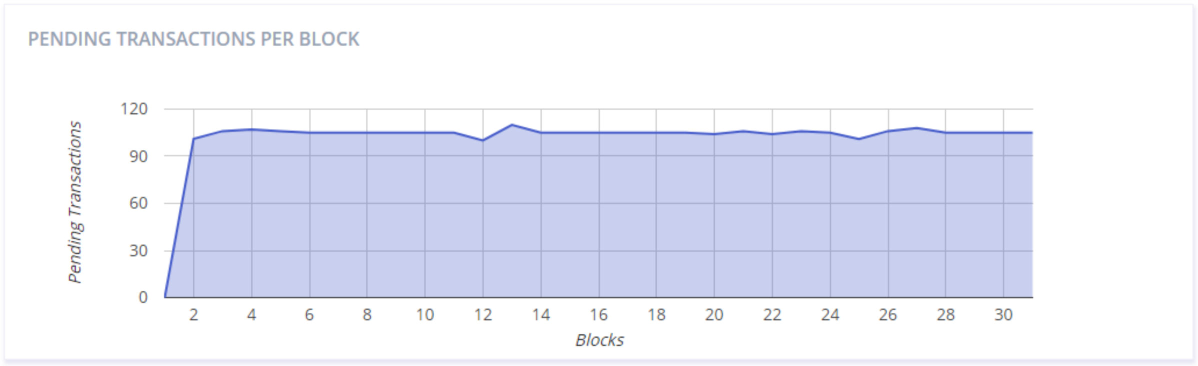


Figure 4.3: Screenshot Pending Transactions per Block

Changes to the same classes and additionally the Main object were done for the processed transactions per block chart in Figure 4.4. The new attribute processed transactions of a block describes the transactions which were included by the winning miner. This is shown as the blue line, the red line shows the maximal possible transactions depending on if there is a block size limit or if SegWit is enabled. If the block size limit and SegWit are disabled (equal to zero), then the transaction limit per block is unlimited and the red line isn't shown.

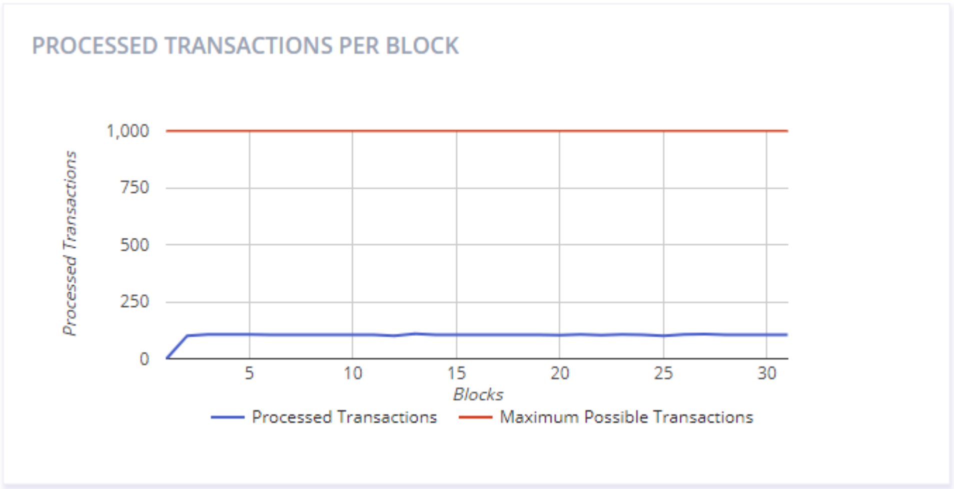


Figure 4.4: Screenshot Processed Transactions Per Block

Both Figures 4.3 and 4.4 show that the first block contains no pending and no processed transactions. The reason for this is that no transactions are received before the genesis block is mined. One reason for having an empty genesis block is the fact that no bitcoin exists before the first block, so no transaction fees can be paid and no bitcoin can be sent. But also transactions with zero transactions fees and zero bitcoin sent can be valid and included in blocks. These transactions would be considered transaction spam. The miners have no financial incentive to include zero fee transactions. Bitcoin's genesis block contains one transaction [35]. In the end, this question about having zero, one or lots of transactions in the genesis block seems to be a question of personal preference. One disadvantage of having only zero or one transaction in the genesis block is that it distorts transactions metrics such as *tps* for a very small number of blocks or transactions.

4.7 Transaction Incentives and Confirmation Status

In bitcoin there are two types of incentives for miners, block rewards and transaction incentives or also called transaction fees. Previously these mining incentives were not considered in VIBES. To make a more realistic simulation, transaction incentives are added to the simulation. This allows analysis, for example, for determining the necessary price to include a transaction in a block within a certain time. It can also be used in future work about mining pools. Since research questions about transaction incentives are closely linked to the confirmation status, new data structures are needed to easily access information about the creation time and confirmation time of transactions.

New variables for the transaction incentives and confirmation status are added to the VTransaction model.

- *transactionFee*: transaction fee in Satoshi
- *confirmation*: confirmation status as a boolean
- *creationLevel*: block level when the transaction was created
- *confirmationLevel*: block level when the transaction was included in a block

Transactions are assigned a random integer between 0 and 124 as the transaction fee in Satoshi. This is about the same range as in reality, but the distribution is different. The real distribution can change from one moment to the next and is difficult to model. When a transaction gets included in a block, its transaction status changes from *false* to *true*.

Two charts were created to show the results of this implementation. For these charts, a slightly higher transaction throughput than block transaction capacity was chosen because this shows an interesting case. In this case, the miners can prefer transactions with high transaction fees over transactions with low fees.

Figure 4.5 shows the transaction confirmation status per transaction fee. The abscissa shows the transaction fees from 0 to 124 in Satoshi, the ordinate shows the number of transactions. The red area shows the unconfirmed and the green area shows the confirmed transactions. It can be clearly seen, the red area is only really big from 0 to 7 Satoshi. This means most of the pending transactions are the ones with the lowest transaction incentives.

The red area exists even if the block size limit or block weight limit is smaller than the actual transaction throughput because the nodes are sending transactions even after the last block of the simulation was mined.

The next Figure 4.6 shows the average transaction confirmation time per transaction fee. The abscissa shows again the transaction fee, the ordinate shows the confirmation time in blocks. Both Figures are from the same simulation. Therefore we can see that the average confirmation time for transaction fees 0 to 3 is zero blocks. The reason is that there are no confirmed transactions in this range. Maybe this visualization is not perfect, since it

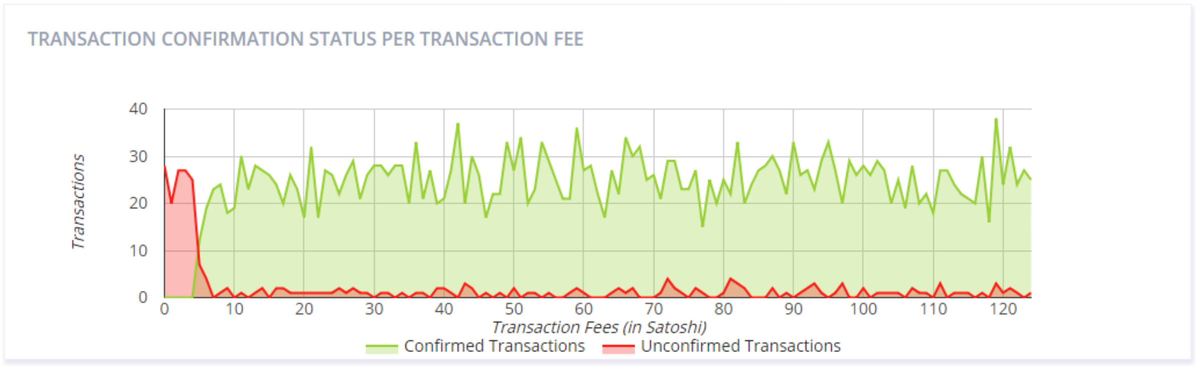


Figure 4.5: Screenshot Transaction Confirmation Status per Transaction Fee

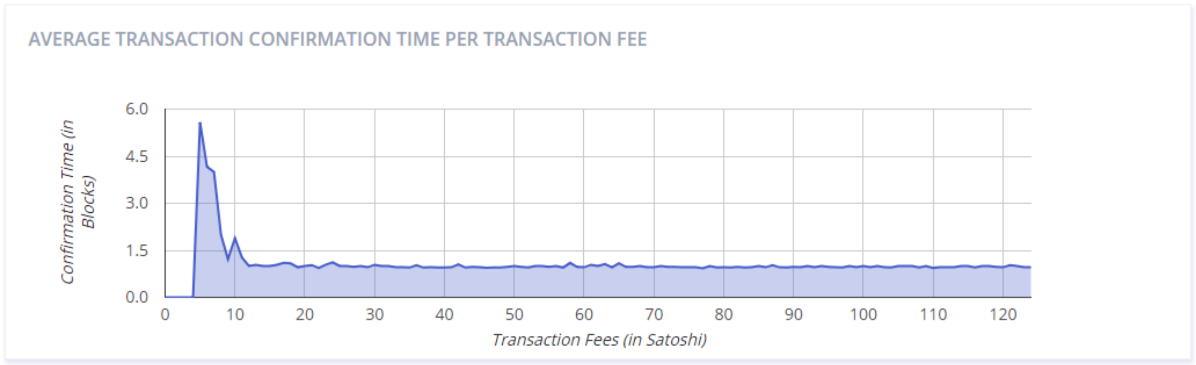


Figure 4.6: Screenshot Average Transaction Confirmation Time per Transaction Fee

may lead to the conclusion that transactions with fees from 0 to 3 are instantly included in a block. But infinite confirmation time in blocks is hard to visualise and taking the first nonzero confirmation time would also be misleading. The highest confirmation time is the point where the transactions barely get included into blocks. After this bottleneck, all transactions get included in about the same time.

For the generation of both charts, all created transactions are sent from the backend to the frontend. The frontend then summarizes the transactions per transaction fee, this could also be done by the backend. Sending all transactions ever created to the frontend is very likely a bottleneck for simulations with a very large number of transactions. The reason for this design decision is the flexibility to create or change charts to analyse different aspects of transactions. So far no issues were found during development. Parts of the previously created frontend are also a bottleneck for simulations with a very large number of blocks or nodes since every created block and every node is sent to the frontend. A solution for this issue could be a database.

VIBES offers the possibility to change the parameters sent to the frontend fast and easily.

For simulations with a very large number of transactions and/or a very large number of blocks the frontend could be changed, or the sought after data can also be output via log file or console.

4.8 Transaction Spam

The transaction spam or also called flood attack has one configuration parameter in the frontend, which is the target transaction fee. The target transaction fee is the minimal fee that the attacker wants everyone in the network to pay for transactions.

Main

In the *Main* class changes were made to set up the flood attack. Currently, the implementation only allows one attack at a time, either double-spending or flood attack. If both attacks have a valid input, the implementation prefers the double-spending attack.

The flood attack transaction buffer is also set up in the *Main*. Very short block times can lead to no regular transactions being sent, therefore a buffer is needed. The size of the buffer is two times the maximal tpb.

NodeActions

NodeActions implements a new case class to issue transactions in a flood attack.

VTransaction

VTransaction implements a new transaction attribute to enable analysis about how many flood attack transactions were pending or processed at a certain time.

VConf

VConf implements *transactionFee* as the target transaction fee and *floodTransactionPool* as an easy way to access the current number of flood attack transactions.

MasterActor

If a flood attack is active, the *MasterActor* is responsible to make the flood attack transaction happen by assigning random nodes to send these transactions. In future work, the nodes could be configurable. The number of sent transactions depends on the number of confirmed flood attack transactions in the last block. In reality, the attacker might not have the time to always check the number of confirmed flood attack transactions, the attacker could mitigate this by a larger buffer.

Listing 4.5: Flood Attack

```
// checks if flood attack is active
if (VConf.floodAttackTransactionFee > 0) {
  logger.debug(s"VConf.floodTransactionPool...
      ↪ ${VConf.floodAttackTransactionPool}")

  (1 to VConf.floodAttackTransactionPool).foreach { _ =>
    val randomActorFrom =
        ↪ actorsVector(Random.nextInt(actorsVector.size))
    val randomActorTo =
        ↪ actorsVector(Random.nextInt(actorsVector.size))
    val now = priorityWorkRequest.timestamp
    randomActorFrom ! NodeActions.IssueTransactionFloodAttack(
      randomActorTo,
      now.plusMillis(50)
    )
  }
}
```

NodeActor

The *NodeActor* needed a new method and a new case for the flood attack.

ReducerActor

The *ReducerActor* calculates how many Satoshi were spent by the attacker on transaction fees, it checks how many transactions were confirmed below the target transaction fee and how many flood attack transactions are in the blockchain.

VEventTypes

The *VEventTypes* had to be adjusted to export the data to the user interface.

Frontend

Figure 4.7 shows the changes to the user interface in case of a flood attack. The number of pending flood attack transactions is added to the pending transactions per block chart, this number should stay constant at about two times maximal tpb.

The green line in the processed transactions per block chart shows the processed flood attack transactions per block. It can be seen on block 11, very short block times can lead to a block full of flood attack transactions.

The flood attack summary shows us the number of confirmed flood attack transactions, the number of bitcoin spent and the confirmed transactions below the target transaction fee.

The next two charts are also interesting, they confirm no transaction below the target transaction fee was confirmed. The target transaction fee in this simulation was 30 Satoshi, as can be seen.

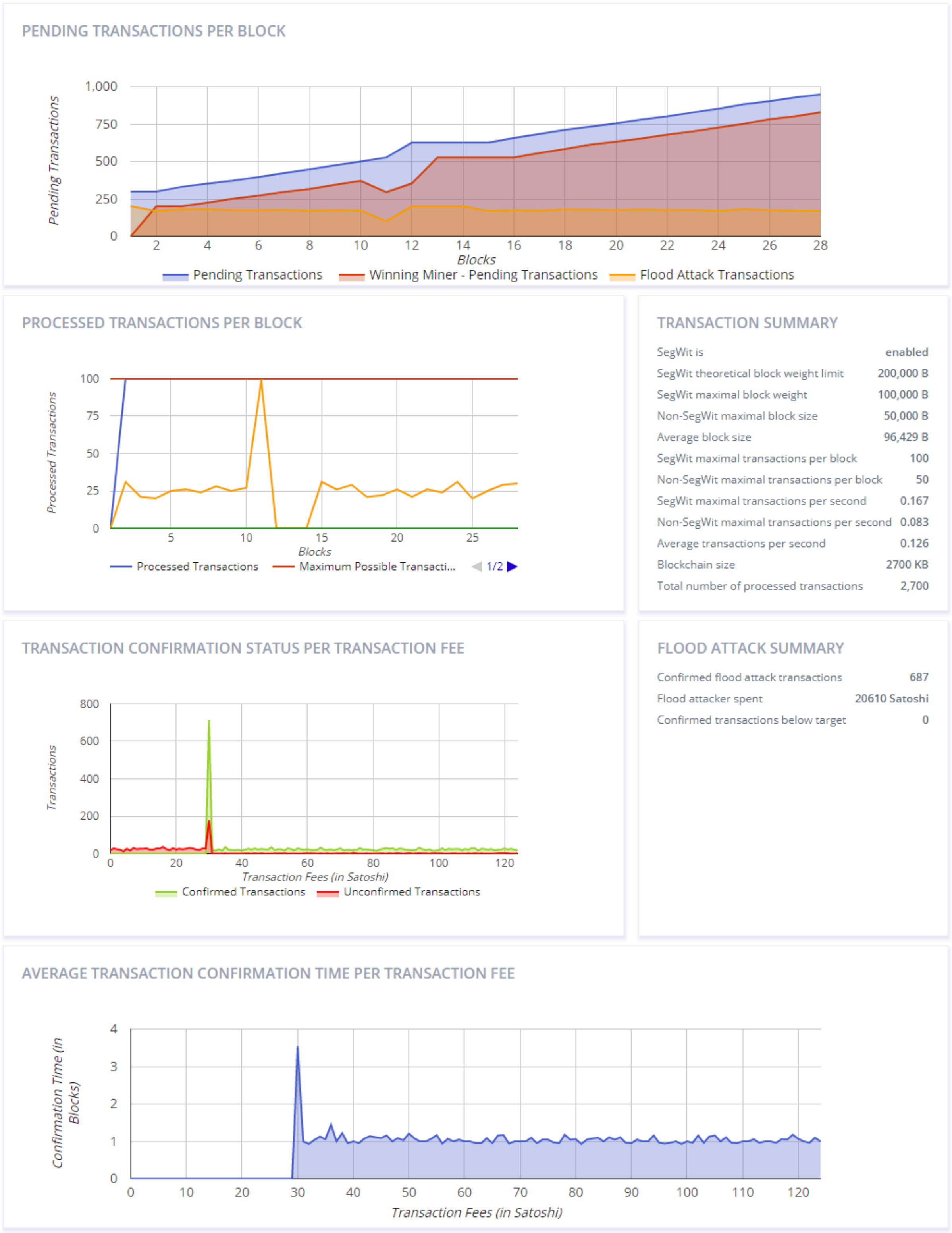
4.9 Alternative History Attack

A 51% attack is one of the most commonly discussed attacks on the bitcoin protocol. It belongs to the group of alternative history attacks. Complex changes are required due to how it functions which is explained in Chapter 2.2.8.

4.9.1 Prerequisites

To simulate an alternative history attack additional configuration parameters are necessary. These parameters are used for the actual simulation of the attack, the calculation of the success probability of the attack and the maximal safe transaction value.

- *isAlternativeHistoryAttack*: if an alternative history attack is simulated as a boolean
- *hash-rate*: attacker's hash-rate as a percentage of the total hash-rate of the bitcoin network


PENDING TRANSACTIONS PER BLOCK
Pending Transactions
1,000
750
500
250
0
2 4 6 8 10 12 14 16 18 20 22 24 26 28
Blocks
Pending Transactions
Winning Miner - Pending Transactions
Flood Attack Transactions
PROCESSED TRANSACTIONS PER BLOCK
Processed Transactions
100
75
50
25
0
5 10 15 20 25
Blocks
Processed Transactions
Maximum Possible Transacti...
1/2
TRANSACTION CONFIRMATION STATUS PER TRANSACTION FEE
Transactions
800
600
400
200
0
0 20 40 60 80 100 120
Transaction Fees (in Satoshi)
Confirmed Transactions
Unconfirmed Transactions
AVERAGE TRANSACTION CONFIRMATION TIME PER TRANSACTION FEE
Confirmation Time (in Blocks)
4
3
2
1
0
0 10 20 30 40 50 60 70 80 90 100 110 120
Transaction Fees (in Satoshi)

TRANSACTION SUMMARY

SegWit is	enabled
SegWit theoretical block weight limit	200,000 B
SegWit maximal block weight	100,000 B
Non-SegWit maximal block size	50,000 B
Average block size	96,429 B
SegWit maximal transactions per block	100
Non-SegWit maximal transactions per block	50
SegWit maximal transactions per second	0.167
Non-SegWit maximal transactions per second	0.083
Average transactions per second	0.126
Blockchain size	2700 KB
Total number of processed transactions	2,700

FLOOD ATTACK SUMMARY

Confirmed flood attack transactions	687
Flood attacker spent	20610 Satoshi
Confirmed transactions below target	0

Figure 4.7: Screenshot Transaction Spam

- *confirmations*: the amount of confirmations the attacked merchants are waiting to accept a transaction
- *attackDuration*: the attacker gives up after mining a certain number of blocks and not succeeding or if it is not possible anymore to surpass the level of the honest blockchain
- *discountOnStolenGoods*: discount of the stolen goods by the attacker, a value from 0 (= full discount) to 1 (= no discount)
- *amountOfAttackedMerchants*: the attack is carried out against a certain amount of merchants at the same time
- *blockReward*: current block reward in BTC

4.9.2 Design and Architecture

In the following the attacker's nodes, blockchain or blocks are interchangeably described as evil, private or malicious and the honest networks' nodes as good or public.

The solution for the simulation of an alternative history attack selects nodes as attacking nodes according to the attacker's hash-rate as a percentage of the total bitcoin network. The good and the evil nodes both can mine the genesis block. The genesis block is then the first block in both the good and the evil blockchain. For simplicity, we assume that the attacker successfully sent the transactions to the attacked merchants in the second block of the honest blockchain. Immediately after the genesis block is mined, the evil nodes start mining together on their own evil blockchain. It is necessary for all nodes to update their neighbor nodes to only have their corresponding nodes as neighbours. The synchronizing of the blockchain is only possible for the same type of node. As a counterexample, the attacker's hash-rate would suffer if the evil nodes could not synchronise their blocks properly if a high percentage of their neighbours are honest nodes.

Finally, the success of the simulated attack is decided if the attacker's blockchain level can surpass the honest blockchain's block level after waiting for the Merchants confirmation and before the attack duration ends. The attack can succeed, fail or be undecided. For example, if a huge percentage of the network is malicious, then the honest network is likely to need a long time to reach the needed block level for the Merchants confirmation.

In the case of success or failure, the two networks need to merge back together by updating their neighbors, allowing the synchronizing of all blocks and taking the winning blockchain.

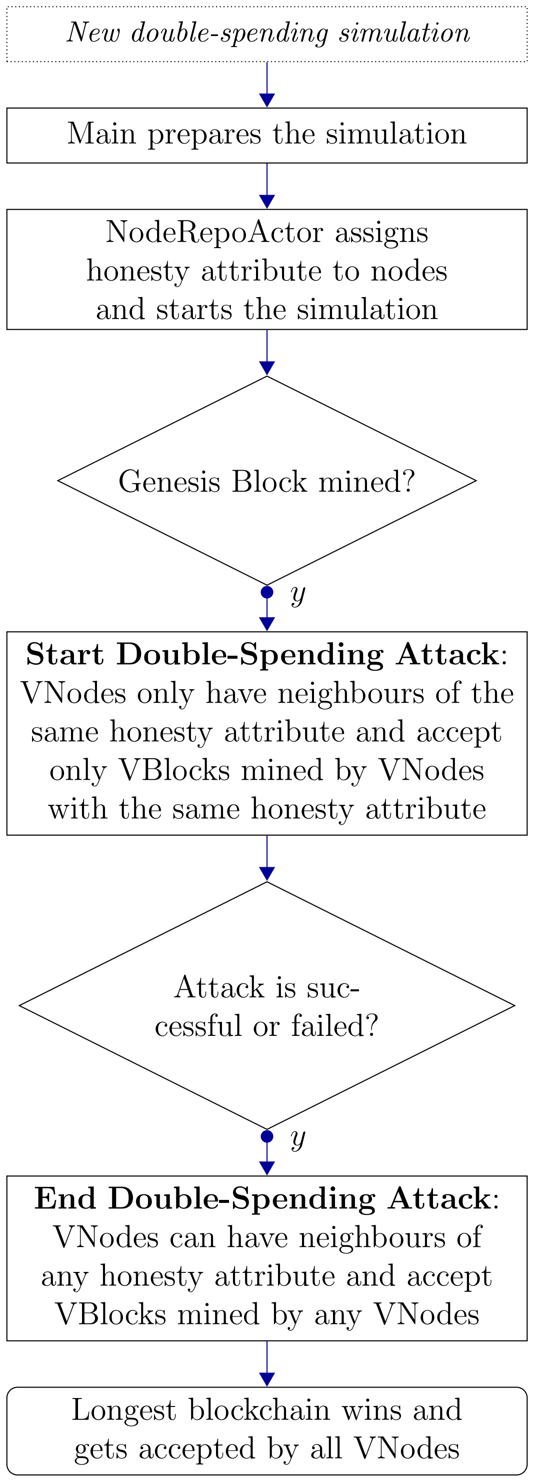

New double-spending simulation
Main prepares the simulation
NodeRepoActor assigns honesty attribute to nodes and starts the simulation
Genesis Block mined?
y
Start Double-Spending Attack: VNodes only have neighbours of the same honesty attribute and accept only VBlocks mined by VNodes with the same honesty attribute
Attack is successful or failed?
y
End Double-Spending Attack: VNodes can have neighbours of any honesty attribute and accept VBlocks mined by any VNodes
Longest blockchain wins and gets accepted by all VNodes

Figure 4.8: Flowchart Double-Spending

Calculating the Success Probability

To be able to validate the results of the Bitcoin-like Blockchain Simulation with an Alternative History Attack a correct reference value for the success probability is required. Therefore the success probability of an Alternative History Attack needs to be calculated.

Before the formula to calculate the success probability of an alternative history attack is shown, the variables need to be explained. q is *hash-rate*, the attacker's percentage of the hash-rate of the total network. p is $1 - q$ and the percentage of the honest network [8].

$$p + q = 1 \tag{4.1}$$

It is the goal to calculate the success probability r. If the attacker's hash-rate q is equal or bigger than p, then the success probability of the attacker is 100%. Due to the implementation, the behavior of the implementation can deviate from the 100%. For example, the variables *attackDuration*, *confirmations* or the simulation duration can have an impact. If $q < p$, then the upper complex formula with binomial coefficients needs to be calculated.

$$r = \begin{cases} 1 - \sum_{m=0}^{n} \binom{m+n-1}{m} (p^n q^m - p^m q^n), & \text{if } q < p \\ \\ 1, & \text{if } q \geq p \end{cases} \tag{4.2}$$

The formula for $q < p$ is transformed for the implementation. This allows the usage of factorial functions instead of binomial coefficients.

One difference between this formula and the implementation is the attack duration. The formula is not limited by an attack duration, while the implementation has one. We assume the difference is negligible. The default value of the attack duration is 20 blocks. The probability of an attacker winning despite being behind 20 blocks is in most cases very small.

$$r = \begin{cases} 1 - \sum_{m=0}^{n} \frac{(m+n-1)!}{m!\,(n-1)!} (p^n q^m - p^m q^n), & \text{if } q < p \\ \\ 1, & \text{if } q \geq p \end{cases} \tag{4.3}$$

Calculating the Maximal Safe Transaction Value

Using the success probability from Equation (4.3) we can calculate the maximal safe transaction value [8]. B is the block reward, o is the attack duration, α is the discount on stolen goods, and k is the number of attacked merchants.

$$\frac{(1-r)oB}{k(\alpha+r-1)} \tag{4.4}$$

4.9.3 Implementation

The implementation of double-spending is complex due to the need of splitting up the nodes and having two separate blockchains running at the same time and after the double-spending attack, the two networks are merging back together to work on one blockchain.

VConf

The VConf implements the new parameters for the alternative history attack which are mentioned in the Prerequisites (Chapter 4.9.1).

Main

The simulation and also the attack starts in Main. Main checks if an alternative history attack happens and sets up the configuration. After the end of the simulation, Main also sends the results back to the frontend.

VNode

The VNode Model needs a new parameter *isMalicious* as Option[Boolean].

NodeRepoActor

In case of an attack, NodeRepoActor creates/registers nodes with their corresponding type Option[Boolean] in the predefined ratio according to the attacker's hash-rate. Malicious nodes are set to Some(true).

DiscoveryActor

The DiscoveryActor updates the neighbours according to the defined Neighbours Discovery Interval. In the case of an active attack, the nodes are only allowed to receive neighbours of the same type (malicious/non-malicious).

NodeActor

The node with the smallest timestamp is allowed to add his block in the NodeActor. Here several conditions are checked to make sure that only in valid cases the blocks are added. In the case of an attack also the status of the attack is determined in the *addBlockIfAlternativeHistoryAttack* function. In the case of a preRestart, the configuration is reset. NodeActor has a new property *isEvil*.

VBlock

Robustness increased by considering that in an attack the recipients of a block can be null.

ReducerActor

The ReducerActor calculates the maximal safe transaction value and the success probability of the alternative history attack. It also prepares the other values connected to the attack.

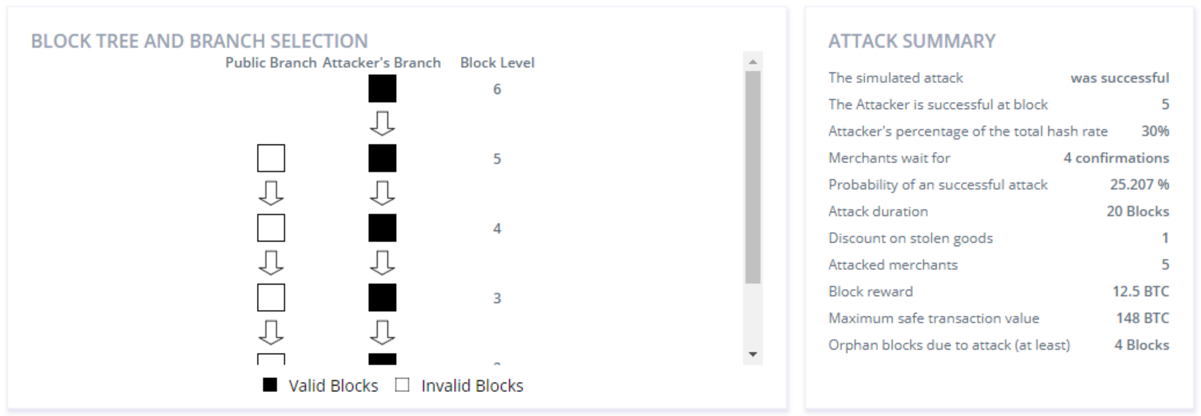
VEventTypes

VEventTypes provides the new figures to the Main.

Frontend

The block tree and branch selection visualization of the frontend was inspired by the double-spending paper [8].

Figure 4.9 shows a successful attack. In this case, the attack was successful immediately after the public branch reached the merchants' confirmation requirement. The blocks of

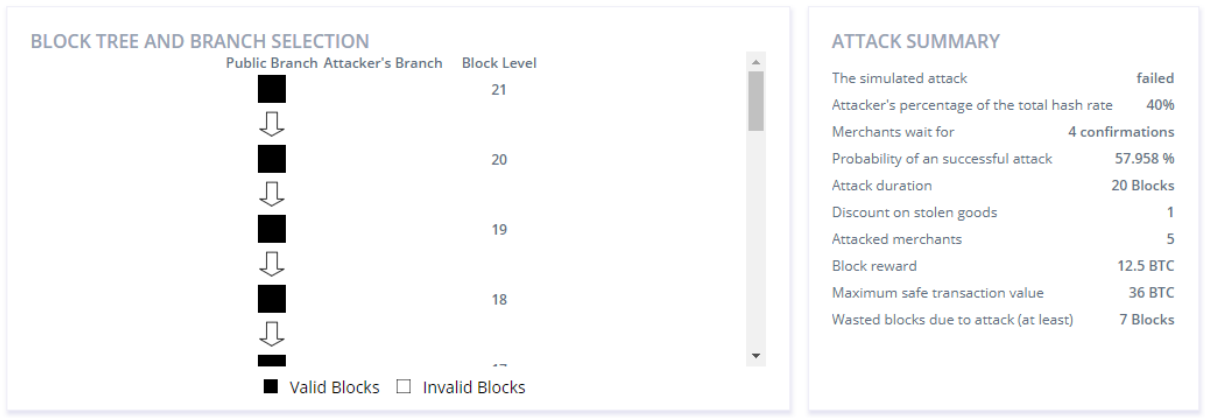


ATTACK SUMMARY

The simulated attack	was successful
The Attacker is successful at block	5
Attacker's percentage of the total hash rate	30%
Merchants wait for	4 confirmations
Probability of an successful attack	25.207 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximum safe transaction value	148 BTC
Orphan blocks due to attack (at least)	4 Blocks

Figure 4.9: Screenshot Block Tree, Branch Selection, and Attack Summary - Attacker wins

the winning branch are shown as black squares and the losing branch's blocks are white. The public branch is always on the left side, the attacker's branch in the middle and the right column shows the block level. In case of a successful attack, only the necessary valid blocks are shown for better visibility.



ATTACK SUMMARY

The simulated attack	failed
Attacker's percentage of the total hash rate	40%
Merchants wait for	4 confirmations
Probability of an successful attack	57.958 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximum safe transaction value	36 BTC
Wasted blocks due to attack (at least)	7 Blocks

Figure 4.10: Screenshot Block Tree, Branch Selection, and Attack Summary - Attacker loses

Figure 4.10 shows a failed attack. As can be seen, the attack failed due to the public branch reaching the end of the attack duration without the attacker's branch overtaking once. The attack duration limit is the reason even attacks with 100% success probability can fail since the attack duration is only considered in the formula for the maximal safe transaction value (4.4) but not in the formula for the success probability (4.3).

Figure A.1 shows an undecided attack, the outcome "not yet decided" can happen if the simulation time was too short.

Zero Confirmations: Race or Finney Attack

In the case of zero confirmations, the simulation assumes the attack was successful because of a race or Finney attack.

4.10 Frontend

The React Google Charts package was used for all the charts [36]. During development, some error messages appeared about *loader.js* which is working correctly now [37] [38].

One of the main focuses of this paper is the frontend. Previously not everything captured by VIBES was visualized. This paper added lots of interesting graphics such as block tree and branch selection, block time, pending transactions, processed transactions, transaction status and transactions fees and also new figures about double-spending, transactions, and staleBlocks to the frontend. But there may still very interesting information that has not yet been analyzed or visualized. These graphics and figures were chosen to validate the correctness of the changes to the backend and can also be used for research questions about double-spending, transaction fees, transaction status, tps, etc.

Chapter 5

Evaluation

For the evaluation of the approach, six criteria from VIBES are taken over: correctness, speed, scalability, flexibility, extensibility, and powerful visuals. These criteria must be at least maintained or improved. Additionally, a new criterion called usability is introduced.

5.1 Correctness

The evaluation of the correctness of the simulator's frontend output is essential. By validating the output of the frontend we also validate the output of the backend.

Some parts of the evaluations are sample testing for consistency between the input parameters, expected and simulated or calculated output. Other parts are empirically tested.

For the calculation of the probability of a successful double spend and the maximal safe transaction value, sample testing is used. For validating the success probability of simulated double spends empirical testing is applied.

5.1.1 Block Size Limit

If there is a block size limit, the total number of processed transactions needs to be smaller or equal to the product of Non-SegWit maximal throughput and the number of mined blocks:

$$\text{Total number of processed transactions} =< \text{Non-SegWit maximal tpb} * \text{Blocks} \tag{5.1}$$

The block size limit was respected in every test. No block with more transactions than the maximal tpb was found.

5.1.2 Segregated Witness

If there is a block weight limit, the total number of processed transactions needs to be smaller or equal to the product of SegWit maximal throughput and the number of mined blocks:

$$\text{Total number of processed transactions} =< \text{SegWit maximal tpb} * \text{Blocks} \tag{5.2}$$

The SegWit block weight limit was respected in every test. No block with more transactions than the maximal tpb was found.

5.1.3 Simulated and Expected Transaction Incentives

If the number of process transactions per block is smaller than the maximum number of transactions per block, then the transaction incentives have no impact. For the evaluation of transaction incentives we take a look at the Figures 4.5, 4.6 and 4.7 which are all based on scenarios with full blocks.

Transactions with higher fees should have a better confirmation status

The first expectation is the transaction with higher transaction fees should have preference over the transaction with lower transaction fees, therefore in the transaction confirmation status chart, the lower transaction fees should have a higher percentage of unconfirmed transactions. The simulated results confirm this expectation.

Transactions with higher fees should have a shorter average transaction confirmation time

The second expectation is transactions with higher transaction fees should have a shorter average transaction confirmation time. The simulated results confirm this expectation.

5.1.4 Transaction Spam

The Screenshot Transaction Spam 4.7 shows the simulated results which match the following expected results.

No transaction confirmed with transaction fee below the target

No transaction below the target transaction fee is expected. The charts about the transaction confirmation status and the average transaction confirmation time and the flood attack summary show that no confirmed transaction has a transaction fee below the target.

All blocks are full

One of the requirements of a transaction spam attack is that the blocks need to be full to block the transactions with fees below the target. All blocks in Figure 4.7 are full, except for the genesis block. There are 28 blocks, the maximum number of transactions per block is 100, so 2,700 transactions are expected and simulated.

Flood attack transaction buffer is not shrinking

To guarantee that no transaction below the target is confirmed, the flood attack transaction buffer should not decrease and stay constant. The number of pending flood attack transactions stays constant in the pending transaction chart.

5.1.5 Expected and Simulated Transactions per Second

For the evaluation of the correctness of the transactions per second (tps) of our simulation, the formula for the calculation of *tps* is compared to the implementation.

$$\text{tps} = \text{transactions per block}/\text{block time} \tag{5.3}$$

Since the *tps* for the whole simulation is an important key metric, the division is not done for one block, but instead for all blocks.

```
  var tps: Double = longestChainNumberTransactions.toDouble /
    ↪ secondsBetween(VConf.simulationStart,
    ↪ VConf.simulateUntil).getSeconds.toDouble
```

As can be seen, the formulas are identical and therefore the result should be correct. Sample testing is done to check for implementation errors.

Sample with a Simulation Duration of Six Hours

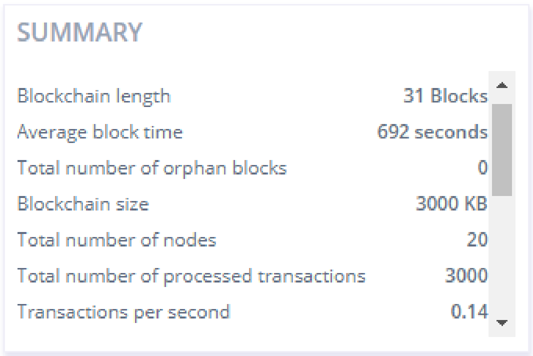
For the first sample, a short simulation of six hours is chosen. The chosen block time is 567 seconds and the chosen throughput is 105.

Calculation of the expected total processed transactions pt:

$$pt = 6\text{h}/567\text{sec} * 105\text{tpb} = 6 * 60 * 60/567 * 105 = 4,000 \tag{5.4}$$

Calculation of the expected tps:

$$\text{tps} = 4,000\text{transactions}/6\text{h} = 0.185 \tag{5.5}$$


SUMMARY

Blockchain length	31 Blocks
Average block time	692 seconds
Total number of orphan blocks	0
Blockchain size	3000 KB
Total number of nodes	20
Total number of processed transactions	3000
Transactions per second	0.14

Figure 5.1: Screenshot Transactions per Second - 6h Simulation Duration

As can be seen in Table 5.1, for such a short simulation the tps is off by about 30%. One reason for this significant difference is the difference in the block time between the simulated and expected values. This can be due to variance. To reduce block time variance a longer simulation time is chosen as a second sample. From these results, we conclude that the tps is correct.

The parameters for reproduction can be found in Chapter A.4.

Table 5.1: Simulated and expected results of transactions per second: Sample with 6h simulation duration

Metric	**Simulation**	**Expectation**
Block time	692	567
Total number of processed transactions	3,000	4,000
Transactions per second	0.14	0.185

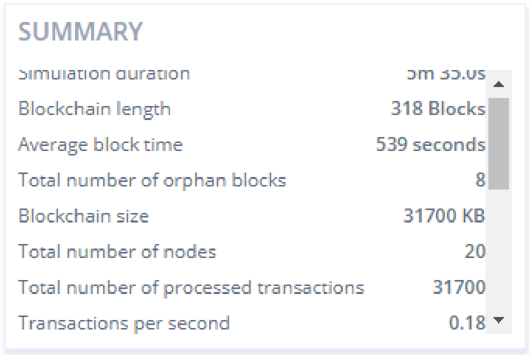
Sample with a Duration of 48 Hours

For the second sample, a longer simulation of 48 hours is chosen. The chosen block time is still 567 seconds and the chosen throughput is also still 105.

Calculation of the expected total processed transactions:

$$\text{pt} = 48h/567\text{sec} * 105\text{tpb} = 48 * 60 * 60/567 * 105 = 32,000 \tag{5.6}$$

Of course, the expected tps stays the same, since only the simulation duration parameter has been changed.



SUMMARY

Simulation duration	5m 55.0s
Blockchain length	318 Blocks
Average block time	539 seconds
Total number of orphan blocks	8
Blockchain size	31700 KB
Total number of nodes	20
Total number of processed transactions	31700
Transactions per second	0.18

Figure 5.2: Screenshot Transactions per Second - 48h Simulation Duration

It can be observed in the second comparative table 5.2, that a longer simulation duration leads the simulated block time to be closer to the expected block time. As a result of the simulated *tps* is close to the expected value.

The parameters for reproduction can be found in Chapter A.4.

Table 5.2: Simulated and expected results of transactions per second: Sample with 48h simulation duration

Metric	**Simulation**	**Expectation**
Block time	539	567
Total number of processed transactions	31,700	32,000
Transactions per second	0.18	0.185

5.1.6 Expected and Calculated Probability of a Successful Double-Spending Attack

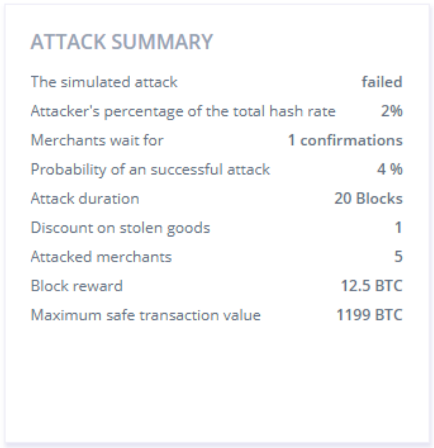
The calculation of the probability of a successful double-spending attack is tested with five samples and the results are compared to the Figure A.1 from the research paper *Analysis of Hashrate-Based Double Spending* [8]. To avoid any confusion about this evaluation, we compare the calculated probability that is shown in the frontend with the expected probability. The calculated probability is later used to test the simulation results against but is not a result of the simulation itself.

The variable q stands for the hash-rate, the variable n is the number of confirmations. The edge cases q = 2% and n = 1, q = 2% and n = 10, q = 50% and n = 1 and q = 50% and n = 10 are tested as well as the common case of q = 30% and n = 6.

Table 5.3: Expected and Calculated Probabilities of a Successful Double-Spending Attack

Sample	**n**	**q**	**Calculated Probability**	**Expected Probability**
a	2	1	4.000%	4.000%
b	2	10	0.000%	0.000%
c	50	1	100.000%	100.000%
d	50	10	100.000%	100.000%
e	30	6	15.645%	15.645%

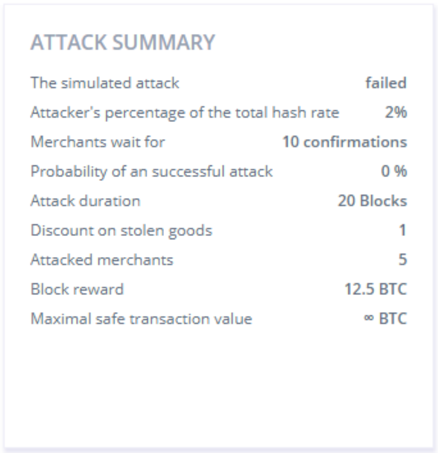
All mentioned test cases in Figure 5.3 match the expected result.



ATTACK SUMMARY

The simulated attack	failed
Attacker's percentage of the total hash rate	2%
Merchants wait for	1 confirmations
Probability of an successful attack	4 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximum safe transaction value	1199 BTC

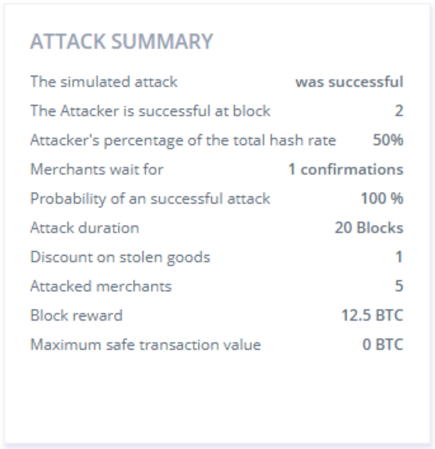
(a) Sample with input: q = 2% and n = 1



ATTACK SUMMARY

The simulated attack	failed
Attacker's percentage of the total hash rate	2%
Merchants wait for	10 confirmations
Probability of an successful attack	0 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximal safe transaction value	∞ BTC

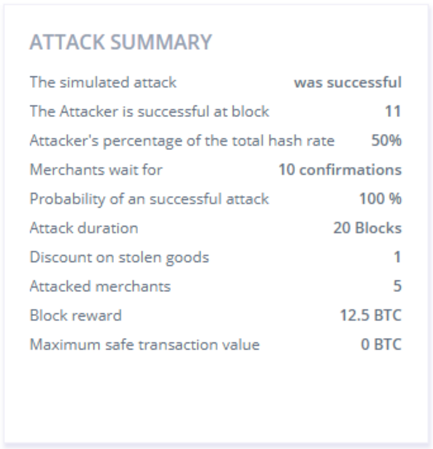
(b) Sample with input: q = 2% and n = 10



ATTACK SUMMARY

The simulated attack	was successful
The Attacker is successful at block	2
Attacker's percentage of the total hash rate	50%
Merchants wait for	1 confirmations
Probability of an successful attack	100 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximum safe transaction value	0 BTC

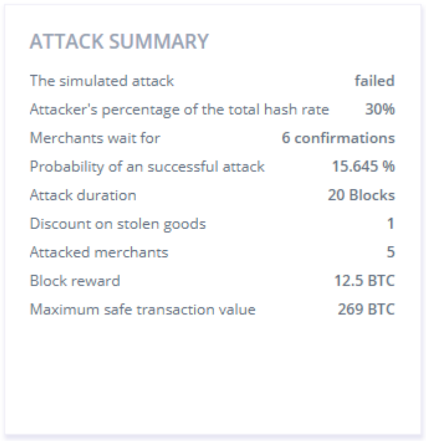
(c) Sample with input: q = 50% and n = 1



ATTACK SUMMARY

The simulated attack	was successful
The Attacker is successful at block	11
Attacker's percentage of the total hash rate	50%
Merchants wait for	10 confirmations
Probability of an successful attack	100 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximum safe transaction value	0 BTC

(d) Sample with input: q = 50% and n = 10



ATTACK SUMMARY

The simulated attack	failed
Attacker's percentage of the total hash rate	30%
Merchants wait for	6 confirmations
Probability of an successful attack	15.645 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximum safe transaction value	269 BTC

(e) Sample with input: q = 30% and n = 6

Figure 5.3: Screenshots Success Probability of Double-Spending

5.1.7 Expected and Calculated Maximal Safe Transaction Value

For the test of the maximal safe transaction value, the samples from the evaluation of the successful double spend probability are reused and compared to Figure A.2.

The additional input parameters were:

- Attack duration: 20 Blocks
- Discount on stolen goods: 1
- Attacked merchants: 5
- Block reward: 12.5 BTC

Compared to the *Analysis of Hashrate-Based Double Spending* paper the block reward was updated from 25 BTC to the current block reward of 12.5 BTC. This means the maximal safe transaction values need to be doubled to compare them with the correct values.

Table 5.4: Expected and Calculated Maximal Safe Transaction Values (MSTV)

Sample	**n**	**q**	**Calculated MSTV**	**Expected MSTV**
a	2	1	1,199 BTC	1,200 BTC
b	2	10	∞ BTC	∞ BTC
c	50	1	0 BTC	0 BTC
d	50	10	0 BTC	0 BTC
e	30	6	269 BTC	269.5 BTC

Ignoring minor rounding differences in samples (a) and (e), all test cases in Figure 5.3 match the expected result. The sample with ∞ BTC is a special case, the frontend displays ∞ BTC in the case of receiving 2,147,483,647 BTC which is the maximal possible 32-bit signed integer value. 2,147,483,647 BTC is orders of magnitudes bigger than the hard supply limit of 21 million BTC.

5.1.8 Expected and Simulated Success Probability of Double Spending

The success probability for the scenario with q = 30% and n = 6 is reused from Chapter 5.1.6.

For the empirical testing of double-spending a script was used, it can be found in Chapter A.4 for reproduction, more information about the scripts and logs can be found in the Appendix. The script starts the simulation, waits for a certain time to let the simulation finish and repeats this for a total of 100 times. The attack duration d is 20 blocks. The simulation needs an attack duration parameter, otherwise, an attack would never fail. The formula for the calculation of the success probability does not consider the attack duration.

Table 5.5: Double-spending outcomes and their simulated and expected probabilities (q = 30, n = 6, d = 20).

Outcome	**Occurrences**	**Simulated probability**	**Expected probability**
Attack not yet decided	45	-	-
Attack successful	8	15.09%	15.645%
Attack failed	47	84.91%	84.355%
TOTAL	**100**	**100%**	**100%**

After counting the occurrences of the outcomes in the log file, they were summarized in Table 5.5. All simulations were finished in the specified time. The not yet decided attacks are not considered in the simulation probabilities. A not yet decided attack could more likely fail than average because it is more likely to have a longer malicious blockchain. The simulated probability of the double-spending attack is with 15.09% very close to the expected probability of 15.645%, which was calculated in Chapter 5.1.6. Hereby is shown that the success probability of the simulation results is very close to the expected probability. But the number of undecided attacks is too high, the simulation duration was too short.

Table 5.6 shows the simulation results with a longer simulation duration, now there are less undecided attacks. But there is a significant difference between the outcome probabilities, this could be due to the fact that the attack duration is too short. With an attack duration of 20, the attacker has only $d - n = 20 - 6 = 14$ blocks to outpace the honest blockchain.

Table 5.6: Double-spending outcomes and their simulated and expected probabilities (q = 30, n = 6, d = 20).

Outcome	**Occurrences**	**Simulated probability**	**Expected probability**
Attack not yet decided	3	-	-
Attack successful	9	9.28%	15.645%
Attack failed	88	90.72%	84.355%
TOTAL	**100**	**100%**	**100%**

Table 5.7 is even more suboptimal as a comparison between the simulation results and the calculated values since now the attacker has only 10 blocks to outpace the honest blockchain.

Table 5.7: Double-spending outcomes and their simulated and expected probabilities (q = 50, n = 10, d = 20).

Outcome	**Occurrences**	**Simulated probability**	**Expected probability**
Attack not yet decided	17	-	-
Attack successful	64	74.05%	100%
Attack failed	18	21.95%	0%
TOTAL	**99**	**100%**	**100%**

Table 5.8 shows the outcome probabilities are as expected in the case of a very low success probability.

Table 5.9 shows the outcome probabilities with a attack duration = 50. The simulated outcome probabilities are very close to the expected values. This shows that the low attack duration previously affected the outcome probabilities significantly. The Table 5.9 proves the correctness of the outcome probabilities of the double-spending attack. The effect of the attack duration on the simulation is with d = 50 negligible.

5.2 Speed and Scalability

The block size limit, the block weight limit and the target transaction fee in the case of a flood attack are in $O(1)$ and do not affect the performance, except for requiring increasing

Table 5.8: Double-spending outcomes and their simulated and expected probabilities (q = 10, n = 10, d = 20).

Outcome	**Occurrences**	**Simulated probability**	**Expected probability**
Attack not yet decided	0	-	-
Attack successful	0	0%	0.001%
Attack failed	101	100%	99.999%
TOTAL	**101**	**100%**	**100%**

Table 5.9: Double-spending outcomes and their simulated and expected probabilities (q = 30, n = 6, d = 50)

Outcome	**Occurrences**	**Simulated probability**	**Expected probability**
Attack not yet decided	0	-	-
Attack successful	14	13.86%	15.645%
Attack failed	87	86.13%	84.355%
TOTAL	**101**	**100%**	**100%**

memory if the throughput is higher than the maximal transactions per block.

For examining the effect of the hash-rate in case of a double-spending attack on the scalability, multiple simulations were executed. Figure 5.4 show there is no significant difference in simulation duration because of a varying hash-rate.

5.3 Comparison with VIBES

To make sure there are no significant performance losses introduced by the changes of BBSS, we compare the simulation results from BBSS with VIBES. The comparison is not easy due to different hardware being used for both theses.

> We were able to simulate on the small MacBook 500 Nodes with 500 Transactions and 8 neighbours 5 times faster than real time. [16]

Table 5.10 used the default number of neighbors of 4 and had a speedup ratio of 5.6 with 500 nodes and transactions. VIBES had a speedup ratio of 5 according to the quote from VIBES. So BBSS is a little bit faster than VIBES but used a lower number of neighbors.

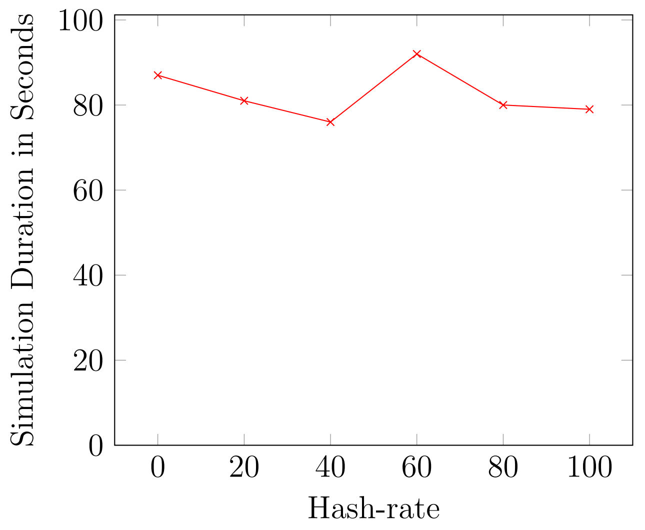


Figure 5.4: Effect of the Hash-rate on the Simulation Duration

To get comparable results, a clean version of VIBES is loaded via git check-out. The clean version of VIBES makes the testing more difficult, since no logs saved into a file, the frontend stops showing the results if the simulation duration is longer than 60 seconds and no scripts can be used. The execution time with 500 nodes and transactions and 4 neighbors was averaged 13 min, the simulation time was 69 min and this results in a speedup ratio of about 5.3. Using the same hardware for testing the speed up ratio shows that the speedup ratio of VIBES and BBSS do not differ significantly. BBSS and VIBES speed up ratios are within the same order of magnitude.

5.3.1 Limitations

There are certain limitations on the simulator. One of the goals of the simulator is that the execution time of the simulation is shorter than the simulated duration. With increasing nodes and transactions, there is a point when the execution time equals the simulation time. In this chapter, we want to examine this relationship and the resulting speedup ratio. The speedup ratio is the simulated duration divided by the execution duration.

All computations were done on a Home PC with Windows Ultimate 64-bit, Intel i7-4770 CPU @ 3.40 GHz and 16.0 GB 1600 MHz DD3, Java Version 1.8.0_144.b01 64bit, Scala Version 2.12.3 and Akka Version 2.5.6 unless otherwise stated.

The simulations were done with an equal number of nodes and throughput. This seems to be a good ratio for the simulator, which allows lots of parallel computation.

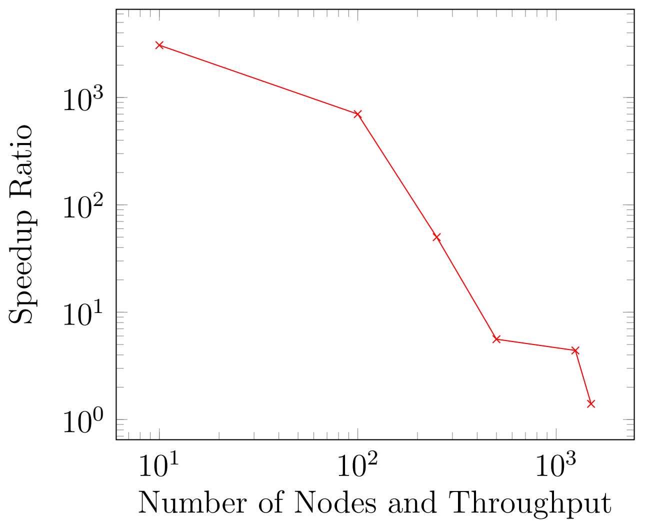


Figure 5.5: Effect of the Number of Nodes and Throughput on the Speedup Ratio

The Table 5.10 and the Figure 5.5 shows the results of the simulations and the declining speedup ratio with increasing nodes and transactions. The values in the table should be considered as approximations since only one simulation per number of nodes and throughput was done and the block time deviations can have an impact on the speedup ratio. One of the consequences deriving from this table is that it is not feasible to do a simulation with the configuration of the real-world bitcoin network with 10,000 nodes and 1,400 throughputs on a Home PC.

Table 5.10: Effect of the Number of Nodes and Throughput on the Speedup Ratio

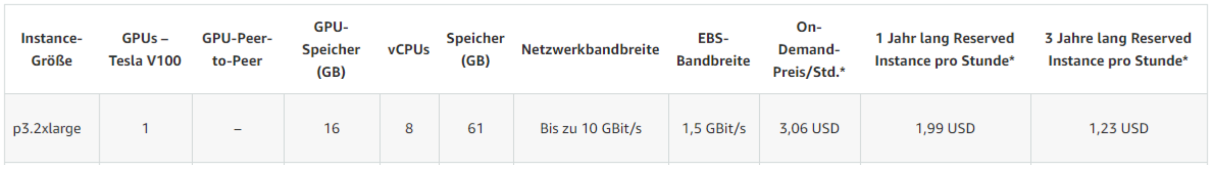
Number of Nodes and Throughput	**Simulation Duration in Seconds**	**Execution Duration in Seconds**	**Speedup Ratio**
10	21,480	7	3,070
100	21,120	30	700
250	19,140	373	50
500	6,900	1,228	5.6
1,250	3,045	697	4.4
1,500	3,060	2,188	1.4

Of course, other configuration parameters than nodes and transactions such as block size limit or block weight limit have a limiting impact due to the increased memory requirement.

It should also be remarked, that the number of nodes scales only in theory linearly, since increasing the number of nodes in a simulation increases the stale block rate. The number of nodes usually does not increase without increasing the number of neighbors.

5.3.2 Realistic Bitcoin Network Simulation with AWS

Amazon Web Service (AWS) is used to find out if it is actually possible to simulate a bitcoin network with 10,000 nodes and 1,400 throughputs. The instance type t2.2xlarge has a similar performance to the used Home PC and therefore doesn't have the required performance. The instance type p3.2xlarge was chosen since it is advertised as "high performance computing" and as an "ideal platform for technical simulations".



Instance-Größe	GPUs – Tesla V100	GPU-Peer-to-Peer	GPU-Speicher (GB)	vCPUs	Speicher (GB)	Netzwerkbandbreite	EBS-Bandbreite	On-Demand-Preis/Std.*	1 Jahr lang Reserved Instance pro Stunde*	3 Jahre lang Reserved Instance pro Stunde*
p3.2xlarge	1	–	16	8	61	Bis zu 10 GBit/s	1,5 GBit/s	3,06 USD	1,99 USD	1,23 USD

Figure 5.6: Screenshot Description of p3.2xlarge AWS Instance Typ

Using various tools we assume the following configuration parameters:

- strategy=BITCOIN_LIKE_BLOCKCHAIN
- simulateUntil=1 hour
- blockTime=600
- numberOfNeighbours=15
- numberOfNodes=10000 [39]
- neighboursDiscoveryInterval=3000
- latency=900
- transactionSize=700 [40]
- maxBlockSize=0
- throughput=1300 [41]
- transactionWeight=1400
- maxBlockWeight=4000000
- networkBandwidth=1
- transactionPropagationDelay=150
- hashRate=0
- confirmations=0

- transactionFee=0

With these configuration parameters, the simulator uses only about 20% CPU. This means realistic parameters do not allow lots of parallel computing. The number of neighbors seems to be too small since the stale block rate is very high which leads slower than real-time simulation.

To avoid having a simulation with a high stale block rate, a good number of neighbors has to be found. According to VIBES, as can be seen in Table 5.11, the optimal number of nodes is four or five for 100 nodes. So there need to be at least four levels of propagation to reach all nodes. To maintain this level of propagation for 10,000 nodes, 15 was chosen as the number of neighbors.

Table 5.11: Optimal Number of Neighbors

Level	4 neighbors	5 neighbors	15 neighbors	21 neighbors
0	$1 = 4^0$	$1 = 5^0$	$1 = 15^0$	$1 = 21^0$
1	$4 = 4^1$	$5 = 5^1$	$15 = 15^1$	$21 = 21^1$
2	$16 = 4^2$	$25 = 5^2$	$225 = 15^2$	$441 = 21^2$
3	$64 = 4^3$	$125 = 5^3$	$3,375 = 15^3$	$9,261 = 21^3$
4	$256 = 4^4$	$625 = 5^4$	$5,0625 = 15^4$	$194,481 = 21^4$

The number of neighbors needs to be between 19 and 22 to reach all nodes after four or five levels of block propagation. In VIBES the optimal number of neighbors for 100 nodes is 4 to 5 neighbors, which is a similar level of block propagation.

But with 21 neighbors the simulation is slower than real-time, the performance of p3.2xlarge is not sufficient. As shown in the Evaluation Chapter of VIBES, increasing the neighbors increases the execution time linearly. All these parameters such as nodes, throughput, and neighbors scale linearly but put together they increase the execution effort too heavily.

The bottleneck of the design and architecture of this simulator is the Coordinator. Only the *propagate transaction* work requests are done in parallel, every other work request is processed serially. This is the reason the CPU utilization is only at 20%. Hundred of thousands of work requests are processed sequentially. The order of execution of every other work request is important, the unordered execution of the other work requests would affect the simulation results.

5.4 Extensibility

The extensibility of the simulator was improved in this thesis. By adding the bitcoin-like blockchain simulation as a second strategy, the frontend and backend are now better prepared for adding a third strategy. Differentiation between the strategies is already built in. The actor model in the backend and the atomic design in the frontend allow high extensibility. New components can be added to the frontend or new functionality can be added to the actors depending on the strategy.

5.5 Flexibility

Previously VIBES supported 8 configuration parameters, BBSS supports now 17 configuration parameters. By increasing the number of parameters the system variability increases, new special cases might need to be addressed and some parts of the code might become too bloated.

- *strategy*: a strategy (choose between *generic simulation* or *bitcoin-like blockchain simulation*)
- *simulateUntil*: a point in the future when the simulation should stop
- *blockTime*: average time to solve a block in seconds
- *numberOfNeighbours*: number of open connections from a node to other peers in the network
- *numberOfNodes*: number of nodes
- *neighboursDiscoveryInterval*: the interval in seconds in which nodes update their neighbours table
- *latency*: the time it takes to propagate a block to other peers
- *transactionSize*: the constant size in byte of all transactions in the network
- *maxBlockSize*: the maximal block size in byte (0 for unlimited)
- *throughput*: average transactions per block
- *transactionWeight*: the constant weight in byte of all transactions in the network
- *maxBlockWeight*: the maximal block weight in byte (0 for no limit)

- *networkBandwidth*: the maximal bandwidth of a single node
- *transactionPropagationDelay*: the time it takes for one transaction to propagate to other peers
- *hashRate*: the attacker's percentage of the total hash-rate in case of a double-spending attack
- *confirmations*: the number of confirmations the transaction receiver is waiting for in case of a double-spending attack
- *transactionFee*: the target transaction fee of an attacker in case of a flood attack

These parameters are the foundation to simulate a bitcoin-like blockchain network with double-spending and flood attacks. In future versions new inputs could be added such as the block reward, the number of attacked merchants, the attack duration or the discount on stolen goods, these inputs are currently hardcoded in VConf to avoid making the configuration panel too bloated. The block reward would be a very important input when adding mining pools or displaying information about mining rewards. New inputs can be added to increase granularity and add functionality. There are lots of great opportunities for future work to improve the simulator.

5.6 Usability

This thesis added usability and made new use cases possible, some of which are presented in the following chapter.

5.6.1 Optimising Transactions per Second

Scalability is one of the biggest issues of bitcoin-like blockchains. The simulator could be used to optimize the tps of bitcoin. The relevant parameters are the block time, the block size limit and the stale block rate. Future works could include the centralization. Our simulation results show that the block time of 10 min could significantly be reduced without affecting the stale block rate. The block size/weight limit could also be increased but our simulator cannot yet display the negative effects of this increase such as the centralization. Our simulation results confirm the findings of the research paper *Block Size Increase* by the BitFury Group [42], who advised for a block size increase.

5.6.2 Securing a Blockchain Merchant

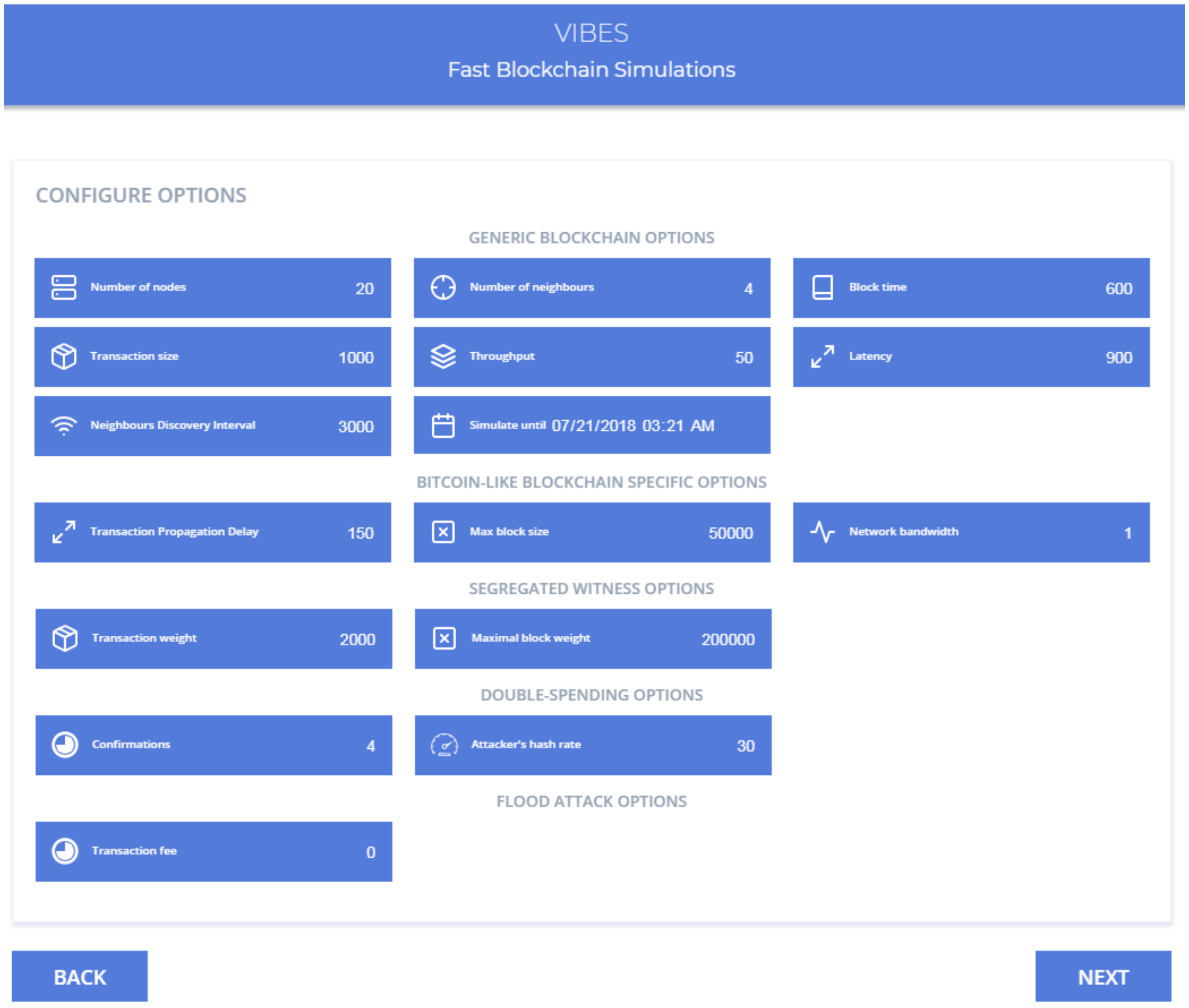
A merchant who interacts with the blockchain to receive transactions or sets up smart contracts can use our simulator to avoid unnecessary risks. One must remember, the output of the simulator is only as good as the input. If the merchant is misinformed about certain parameters, then the simulator won't be able to help. This is the case in the event of a double-spending attack, if the merchant doesn't know the hash-rate of the attacker, then the merchant has problems to set his maximal safe transaction value. If the merchant is knowledgeable about the hash-rate, he can set a maximal transaction value and reduce his risk of a double-spending attack. The outputs of a flood attack scenario can help a blockchain merchant assess if an attacker could make the blockchain unusable and thereby block any revenue for the merchant. Smart contracts set up by the merchant could also be vulnerable to a flood attack if the contracts expect a transaction before a certain point in time.

5.6.3 Choosing Transaction Fees

The simulator outputs charts related to the confirmation time and the transaction fees. Transaction sender could use this to save fees, to optimize their fee/confirmation time ratio. Future work could make some of the nodes use this smart method of setting the transaction fee.

5.7 Powerful Visuals

Our initial goal was to make vital metrics and charts accessible via the web browser. This has been achieved, lots of interesting new facts about the block time, transaction incentives, and attacks can be explored in the user interface. Examples are shown in the following screenshots.



VIBES
Fast Blockchain Simulations

CONFIGURE OPTIONS

GENERIC BLOCKCHAIN OPTIONS

Number of nodes 20	Number of neighbours 4	Block time 600
Transaction size 1000	Throughput 50	Latency 900
Neighbours Discovery Interval 3000	Simulate until 07/21/2018 03:21 AM	

BITCOIN-LIKE BLOCKCHAIN SPECIFIC OPTIONS

Transaction Propagation Delay 150	Max block size 50000	Network bandwidth 1

SEGREGATED WITNESS OPTIONS

Transaction weight 2000	Maximal block weight 200000

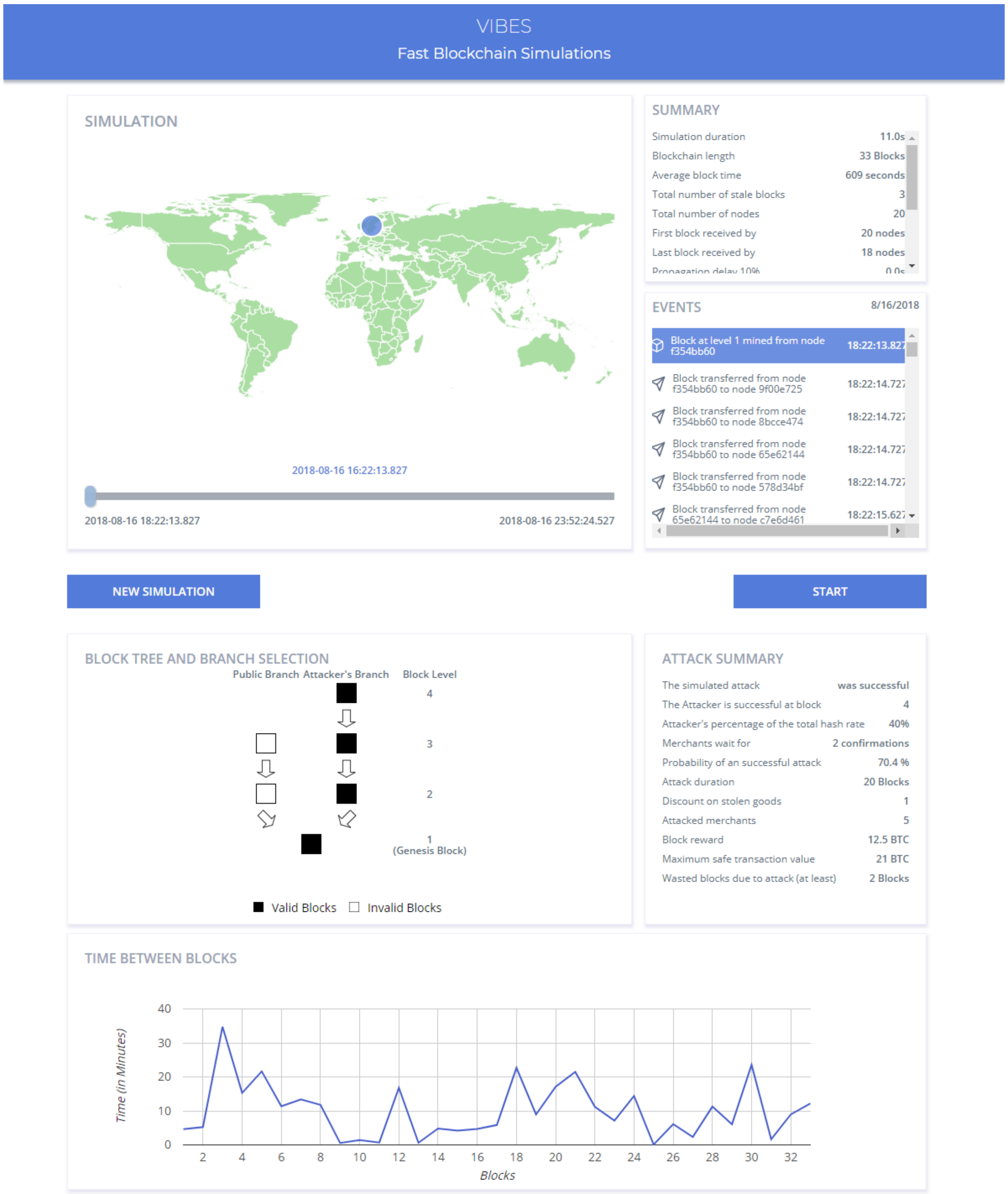
DOUBLE-SPENDING OPTIONS

Confirmations 4	Attacker's hash rate 30

FLOOD ATTACK OPTIONS

Transaction fee 0

Figure 5.7: Screenshot Configuration



VIBES
Fast Blockchain Simulations

SIMULATION

2018-08-16 16:22:13.827

2018-08-16 18:22:13.827 2018-08-16 23:52:24.527

SUMMARY

Simulation duration	11.0s
Blockchain length	33 Blocks
Average block time	609 seconds
Total number of stale blocks	3
Total number of nodes	20
First block received by	20 nodes
Last block received by	18 nodes
Propagation delay 10%	0.0s

EVENTS 8/16/2018

Block at level 1 mined from node f354bb60	18:22:13.827
Block transferred from node f354bb60 to node 9f00e725	18:22:14.727
Block transferred from node f354bb60 to node 8bcce474	18:22:14.727
Block transferred from node f354bb60 to node 65e62144	18:22:14.727
Block transferred from node f354bb60 to node 578d34bf	18:22:14.727
Block transferred from node 65e62144 to node c7e6d461	18:22:15.627

NEW SIMULATION

START

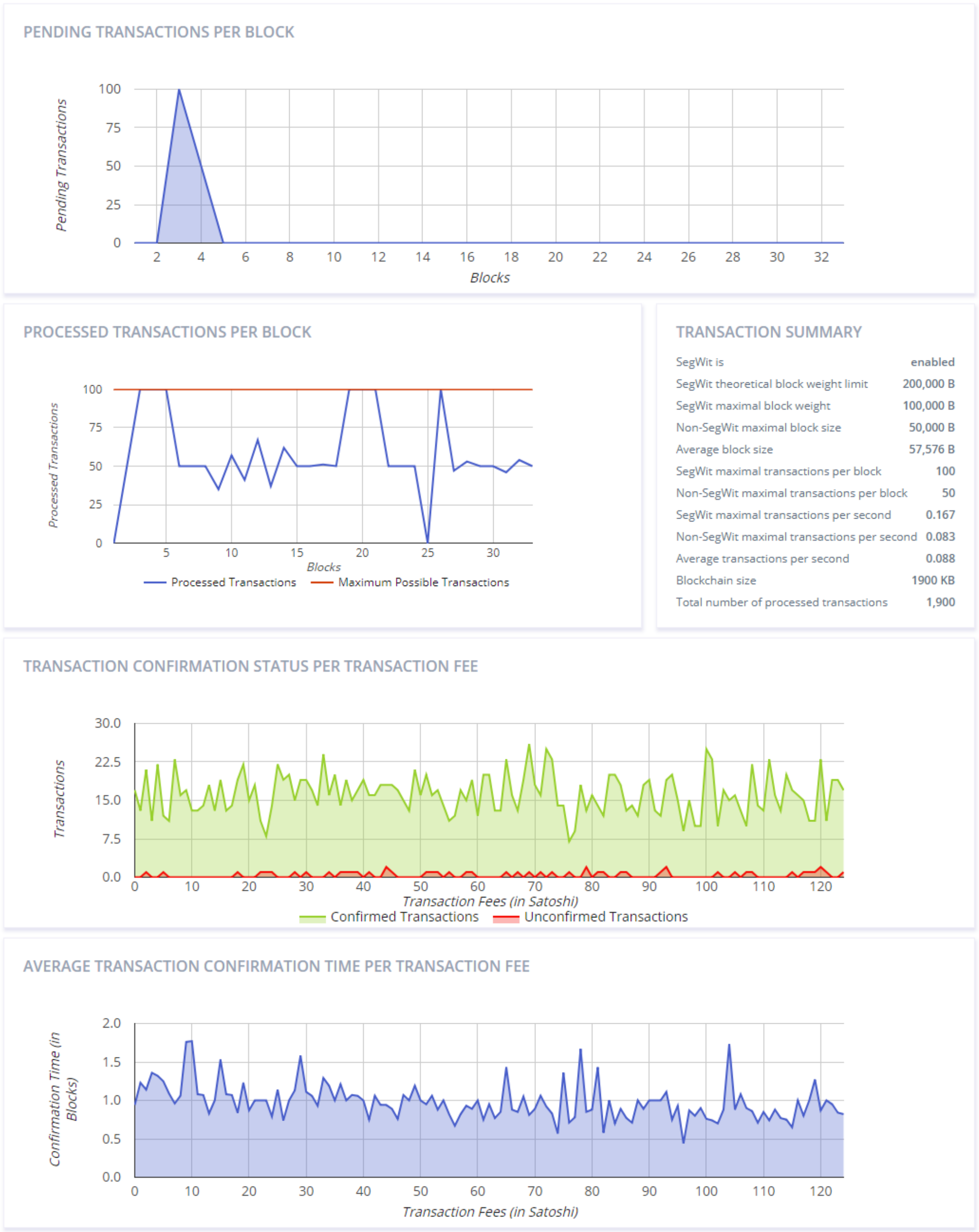
BLOCK TREE AND BRANCH SELECTION

ATTACK SUMMARY

The simulated attack	was successful
The Attacker is successful at block	4
Attacker's percentage of the total hash rate	40%
Merchants wait for	2 confirmations
Probability of an successful attack	70.4 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximum safe transaction value	21 BTC
Wasted blocks due to attack (at least)	2 Blocks

TIME BETWEEN BLOCKS

Figure 5.8: Screenshot Simulation - Part 1


PENDING TRANSACTIONS PER BLOCK
Pending Transactions
Blocks
PROCESSED TRANSACTIONS PER BLOCK
Processed Transactions
Blocks
Processed Transactions
Maximum Possible Transactions
TRANSACTION CONFIRMATION STATUS PER TRANSACTION FEE
Transactions
Transaction Fees (in Satoshi)
Confirmed Transactions
Unconfirmed Transactions
AVERAGE TRANSACTION CONFIRMATION TIME PER TRANSACTION FEE
Confirmation Time (in Blocks)
Transaction Fees (in Satoshi)

TRANSACTION SUMMARY

SegWit is	enabled
SegWit theoretical block weight limit	200,000 B
SegWit maximal block weight	100,000 B
Non-SegWit maximal block size	50,000 B
Average block size	57,576 B
SegWit maximal transactions per block	100
Non-SegWit maximal transactions per block	50
SegWit maximal transactions per second	0.167
Non-SegWit maximal transactions per second	0.083
Average transactions per second	0.088
Blockchain size	1900 KB
Total number of processed transactions	1,900

Figure 5.9: Screenshot Simulation - Part 2

Chapter 6

Summary

In this thesis, a bitcoin-like blockchain simulator with attack scenarios was proposed and implemented. The simulator was evaluated and the limitations were tested. It is very likely to be the first of its kind. The speed and scalability of VIBES are at least maintained, the computations can achieve full CPU utilization, but only in best case scenarios. Only 20% CPU utilization is achieved in a real-world scenario. Double-spending and flood attacks are possible, their impact on the blockchain adoption is shown in Figure 6.1. More attacks, new bitcoin protocol changes or new strategies can easily be added to the front- and backend, because of the usage of atomic design in the frontend and the actor approach in the backend. Important scalability and security metrics are visualized in the frontend.

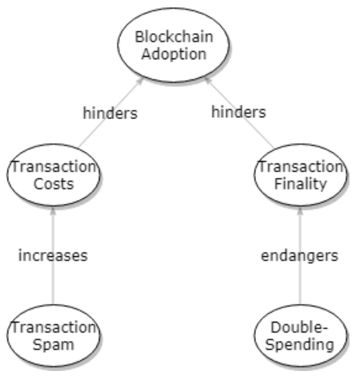


Figure 6.1: Blockchain Adoption

6.1 Status

The main goal of adjusting the generic simulation strategy to a bitcoin-like blockchain simulation was achieved. Compared to the research proposal some ideas and goals were added (attacks) and some were dropped (resource usage by the network, differentiation between miners and full nodes). There are still lots of research topics on bitcoin-like blockchains uncovered, for which the implementation of this thesis could be the foundation. There is very strong evidence that the behavior of the bitcoin simulation is realistic. It is also unlikely that a real-world bitcoin simulation faster than real time is possible with the current architecture and the used computing power, but realistic simulations with a lower number of nodes, transactions, and neighbors are possible.

6.2 Conclusions

Working on blockchains was very interesting since it is not only a relatively new and trendy research area but also because the technology appears to have great potential to improve the real-world with its various use cases. The challenging part of the thesis was validating the behavior of the simulation system and the simulation results. The simulation system can be used to improve the configuration of bitcoin-like blockchains.

6.3 Future Work

There are many ways in which the current implementation can be improved.

Mining Pools

Bitcoin's mining is not done by single miners, but by miners organized in a mining pool. It would be very interesting to simulate mining pools, visualize their share of the hash-rate, their share of the block rewards, and their share of the transaction fees. Mining pools could have different mining strategies.

Requirements for miners

It would be insightful to know the effect on centralization when optimizing the network for tps. Usually, when optimizing for tps, the block size/weight limit is changed. This changes the requirements for the miners. Every miner could be assigned various hardware or bandwidth properties according to a model. The user interface then visualizes the difference in centralization. The research paper *Block Size Increase* by the BitFury Group could be used as a foundation [42].

Resource Usage

The energy consumption of proof-of-work is a very interesting topic, the Bitcoin Energy Consumption Index visualizes insightful facts [43] about the estimated total energy usage of bitcoin, electricity consumed per transaction, number of U.S. households powered for 1 day by the electricity consumed for a single transaction, carbon footprint per transaction, and other interesting facts which could also be implemented in our simulator.

Differentiation between Nodes and Miners

There is no differentiation between nodes and miners in the current implementation. The impact of a varying number of nodes and miners could be interesting.

Varying Transaction Sizes

Currently, transaction sizes are constant. Different models for the transaction size and their effects could be evaluated.

Varying Number of Nodes, Hash-rate and Difficulty

The number of nodes and the hash-rate per miner is constant. Different models about the number of nodes and the hash-rate per miner and their relationship to difficulty adjustments would be interesting.

Smart transaction fees

Transaction senders could adjust their transaction fees to achieve different goals: a certain confirmation time, a certain transaction fee or a good fee/confirmation time ratio.

Selfish Mining

Combined with mining pools, testing different selfish mining strategies and their profitability would be interesting.

Tests

The user interface shows all the important facts about a simulation. It would still be important to be able to run automated tests in order to be sure about the correctness and develop with confidence.

Appendices

Appendix A

Appendix

A.1 Installation and Usage Instructions

A.1.1 Installation Frontend

Install npm

You need to install the npm package manager as documented: `https://www.npmjs.com/`

Install yarn

Then install yarn: `https://yarnpkg.com/lang/en/docs/install/`

Install package.json dependencies

After the installation is done you can navigate to the frontend folder of the project and run *yarn install*.

A.1.2 Installation Backend

SBT

The only thing you need to install is SBT (interactive scala build tool) as documented: `https://www.scala-sbt.org/`

A.1.3 Run

Windows

Navigate to the root folder and run *start frontend and backend.bat* to start the frontend and backend server. Go to `http://localhost:8080/` with your browser.

Linux

Run *yarn dev* to start the frontend, run *sbt server/run* to start the backend.

A.1.4 Inspecting the output of the backend

Start your backend. Then enter the following URL into the browser of your choice after you have adjusted the timestamp of the simulateUntil parameter.

```
http://localhost:8082/vibe?strategy=BITCOIN_LIKE_BLOCKCHAIN&simulateUntil
    ↪ =1534457813308&blockTime=600&numberOfNeighbours=4&numberOfNodes=20&
    ↪ neighboursDiscoveryInterval=3000&latency=900&transactionSize=1000&
    ↪ maxBlockSize=50&throughput=50&transactionWeight=2000&maxBlockWeight
    ↪ =200000&networkBandwidth=1&transactionPropagationDelay=150&hashRate
    ↪ =40&confirmations=2&transactionFee=0
```

The browser shows the simulation results that are fetched by the frontend from the back-end.

A.2 Time-outs and Configuration

Previously the frontend could only display the information from the backend if the simulation results were sent within 60 seconds. After checking the existing time-outs in the project and researching the default time-outs of the used frameworks, the problem was found in the akka.http.server.idle-timeout default setting. This default setting of 60 seconds was changed in *\vibes\server\src\main\resources\application.conf* to infinite.

Listing A.1: application.conf

```
akka.http {

  server {
    idle-timeout = infinite
  }
}
```

Currently, the time-out for providing the information to the frontend is set to 24 hours in *Main.scala*.

A.3 Lazy Logging

Previously logging only occurred in the console. This made debugging of long simulations difficult. Especially for the evaluation of any implementations, a log file is necessary. For this reason, the Scala modules *logback* and *scala-logging* were integrated into the project. Every important event is logged into */logfile.log*.

A.4 Simulations

The simulateUntil parameter of the following URLs to reproduce the evaluation results has to be adjusted. New features might make additional parameters necessary.

Expected and Simulated Success Probability of Double Spending

```
ECHO Start of Loop

FOR /L %%i IN (1,1,100) DO (
  ECHO %%i
  start chrome "http://localhost:8082/vibe?blockTime=567&
      ↪ numberOfNeighbours=4&numberOfNodes=20&simulateUntil
      ↪ =1531411943382&transactionSize=1&throughput=105&latency=900&
      ↪ neighboursDiscoveryInterval=3000&maxBlockSize=100&maxBlockWeight
      ↪ =4000&networkBandwidth=1&strategy=BITCOIN_LIKE_BLOCKCHAIN&
      ↪ transactionPropagationDelay=150&hashRate=30&confirmations=6"
  timeout /t 40
)
```

Expected and Simulated Transactions per Second - 6 hours

```
http://localhost:8082/vibe?blockTime=567&numberOfNeighbours=4&
    ↪ numberOfNodes=20&simulateUntil=1531752759474&transactionSize=1&
    ↪ throughput=105&latency=900&neighboursDiscoveryInterval=3000&
    ↪ maxBlockSize=100&maxBlockWeight=4000&networkBandwidth=1&strategy=
    ↪ BITCOIN_LIKE_BLOCKCHAIN&transactionPropagationDelay=150&hashRate=0&
    ↪ confirmations=4
```

Expected and Simulated Transactions per Second - 48 hours

```
http://localhost:8082/vibe?blockTime=567&numberOfNeighbours=4&
    ↪ numberOfNodes=20&simulateUntil=1531905360000&transactionSize=1&
    ↪ throughput=105&latency=900&neighboursDiscoveryInterval=3000&
    ↪ maxBlockSize=100&maxBlockWeight=4000&networkBandwidth=1&strategy=
    ↪ BITCOIN_LIKE_BLOCKCHAIN&transactionPropagationDelay=150&hashRate=0&
    ↪ confirmations=4
```

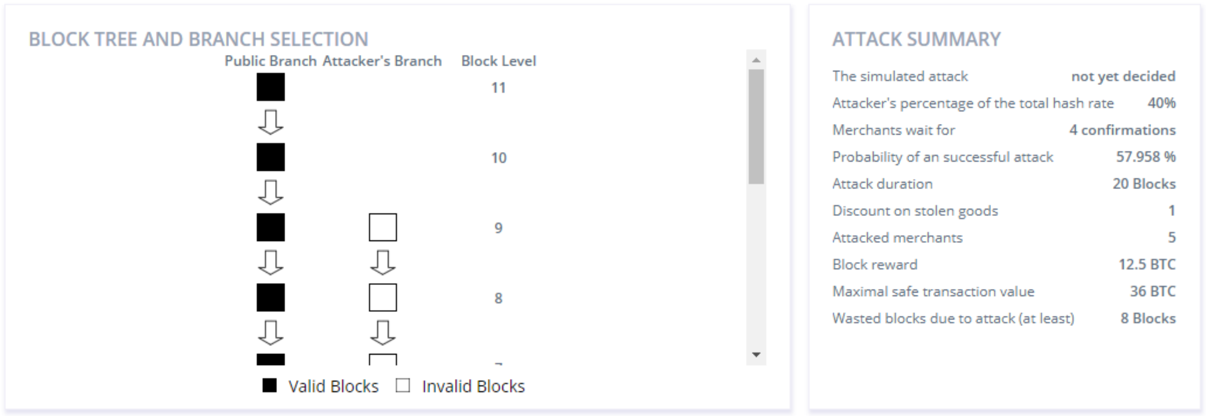
A.5 Tables

q	1	2	3	4	5	6	7	8	9	10
2%	4%	0.237%	0.016%	0.001%	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0
4%	8%	0.934%	0.120%	0.016%	0.002%	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0
6%	12%	2.074%	0.394%	0.078%	0.016%	0.003%	0.001%	≈ 0	≈ 0	≈ 0
8%	16%	3.635%	0.905%	0.235%	0.063%	0.017%	0.005%	0.001%	≈ 0	≈ 0
10%	20%	5.600%	1.712%	0.546%	0.178%	0.059%	0.020%	0.007%	0.002%	0.001%
12%	24%	7.949%	2.864%	1.074%	0.412%	0.161%	0.063%	0.025%	0.010%	0.004%
14%	28%	10.662%	4.400%	1.887%	0.828%	0.369%	0.166%	0.075%	0.034%	0.016%
16%	32%	13.722%	6.352%	3.050%	1.497%	0.745%	0.375%	0.190%	0.097%	0.050%
18%	36%	17.107%	8.741%	4.626%	2.499%	1.369%	0.758%	0.423%	0.237%	0.134%
20%	40%	20.800%	11.584%	6.669%	3.916%	2.331%	1.401%	0.848%	0.516%	0.316%
22%	44%	24.781%	14.887%	9.227%	5.828%	3.729%	2.407%	1.565%	1.023%	0.672%
24%	48%	29.030%	18.650%	12.339%	8.310%	5.664%	3.895%	2.696%	1.876%	1.311%
26%	52%	33.530%	22.868%	16.031%	11.427%	8.238%	5.988%	4.380%	3.220%	2.377%
28%	56%	38.259%	27.530%	20.319%	15.232%	11.539%	8.810%	6.766%	5.221%	4.044%
30%	60%	43.200%	32.616%	25.207%	19.762%	15.645%	12.475%	10.003%	8.055%	6.511%
32%	64%	48.333%	38.105%	30.687%	25.037%	20.611%	17.080%	14.226%	11.897%	9.983%
34%	68%	53.638%	43.970%	36.738%	31.058%	26.470%	22.695%	19.548%	16.900%	14.655%
36%	72%	59.098%	50.179%	43.330%	37.807%	33.226%	29.356%	26.044%	23.182%	20.692%
38%	76%	64.691%	56.698%	50.421%	45.245%	40.854%	37.062%	33.743%	30.811%	28.201%
40%	80%	70.400%	63.488%	57.958%	53.314%	49.300%	45.769%	42.621%	39.787%	37.218%
42%	84%	76.205%	70.508%	65.882%	61.938%	58.480%	55.390%	52.595%	50.042%	47.692%
44%	88%	82.086%	77.715%	74.125%	71.028%	68.282%	65.801%	63.530%	61.431%	59.478%
46%	92%	88.026%	85.064%	82.612%	80.480%	78.573%	76.836%	75.234%	73.742%	72.342%
48%	96%	94.003%	92.508%	91.264%	90.177%	89.201%	88.307%	87.478%	86.703%	85.972%
50%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Table A.1: The probability of a successful double spend, as a function of the attacker's hash-rate q and the number of confirmations n. The abscissa shows the confirmations n and the ordinate shows the attacker's hash-rate q. [8]

q	1	2	3	4	5	6	7	8	9	10
2%	2400	42K	644K	9370K	$\approx \infty$	$\approx \infty$	$\approx \infty$	$\approx \infty$	$\approx \infty$	$\approx \infty$
4%	1150	10K	82K	615K	4437K	$\approx \infty$	$\approx \infty$	$\approx \infty$	$\approx \infty$	$\approx \infty$
6%	733	4722	25K	127K	626K	3018K	14M	$\approx \infty$	$\approx \infty$	$\approx \infty$
8%	525	2650	10K	42K	159K	588K	2144K	7749K	$\approx \infty$	$\approx \infty$
10%	400	1685	5741	18K	56K	168K	503K	1486K	4361K	12M
12%	316	1158	3391	9212	24K	62K	157K	396K	990K	2460K
14%	257	837	2172	5200	11K	27K	60K	132K	290K	632K
16%	212	628	1474	3178	6580	13K	26K	52K	102K	200K
18%	177	484	1043	2061	3901	7202	13K	23K	42K	74K
20%	150	380	763	1399	2453	4190	7039	11K	19K	31K
22%	127	303	571	983	1615	2582	4053	6288	9671	14K
24%	108	244	436	710	1103	1665	2467	3608	5229	7525
26%	92	198	337	523	775	1113	1570	2182	3005	4106
28%	78	161	263	392	556	766	1035	1377	1815	2372
30%	66	131	206	296	406	539	701	899	1141	1435
32%	56	106	162	225	299	385	485	602	740	901
34%	47	86	127	172	221	277	340	411	491	582
36%	38	69	99	130	164	200	240	283	331	383
38%	31	54	76	98	121	144	169	196	224	254
40%	25	42	57	72	87	102	118	134	151	168
42%	19	31	41	51	61	70	80	90	99	109
44%	13	21	28	34	40	46	51	57	62	68
46%	8	13	17	21	24	27	30	32	35	38
48%	4	6	8	9	10	12	13	14	15	16
50%	0	0	0	0	0	0	0	0	0	0

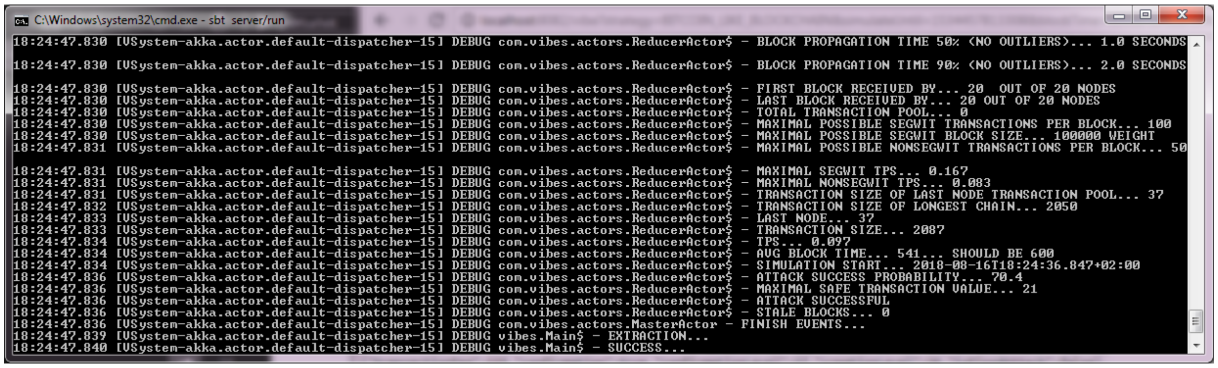
Table A.2: The maximal safe transaction value, in BTC, as a function of the attacker's hash-rate q and the number of confirmations n. The abscissa shows the confirmations n and the ordinate shows the attacker's hash-rate q. [8]



ATTACK SUMMARY

The simulated attack	not yet decided
Attacker's percentage of the total hash rate	40%
Merchants wait for	4 confirmations
Probability of an successful attack	57.958 %
Attack duration	20 Blocks
Discount on stolen goods	1
Attacked merchants	5
Block reward	12.5 BTC
Maximal safe transaction value	36 BTC
Wasted blocks due to attack (at least)	8 Blocks

Figure A.1: Screenshot Block Tree, Branch Selection, and Attack Summary - Attack undecided



```
C:\Windows\system32\cmd.exe - sbt  server/run
18:24:47.830 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - BLOCK PROPAGATION TIME 50% (NO OUTLIERS)... 1.0 SECONDS

18:24:47.830 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - BLOCK PROPAGATION TIME 90% (NO OUTLIERS)... 2.0 SECONDS

18:24:47.830 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - FIRST BLOCK RECEIVED BY... 20  OUT OF 20 NODES
18:24:47.830 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - LAST BLOCK RECEIVED BY... 20 OUT OF 20 NODES
18:24:47.830 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - TOTAL TRANSACTION POOL... 0
18:24:47.830 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - MAXIMAL POSSIBLE SEGWIT TRANSACTIONS PER BLOCK... 100
18:24:47.830 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - MAXIMAL POSSIBLE SEGWIT BLOCK SIZE... 100000 WEIGHT
18:24:47.831 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - MAXIMAL POSSIBLE NONSEGWIT TRANSACTIONS PER BLOCK... 50

18:24:47.831 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - MAXIMAL SEGWIT TPS... 0.167
18:24:47.831 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - MAXIMAL NONSEGWIT TPS... 0.083
18:24:47.831 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - TRANSACTION SIZE OF LAST NODE TRANSACTION POOL... 37
18:24:47.832 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - TRANSACTION SIZE OF LONGEST CHAIN... 2050
18:24:47.833 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - LAST NODE... 37
18:24:47.833 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - TRANSACTION SIZE... 2087
18:24:47.834 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - TPS... 0.097
18:24:47.834 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - AVG BLOCK TIME... 541... SHOULD BE 600
18:24:47.834 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - SIMULATION START... 2018-08-16T18:24:36.847+02:00
18:24:47.836 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - ATTACK SUCCESS PROBABILITY... 70.4
18:24:47.836 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - MAXIMAL SAFE TRANSACTION VALUE... 21
18:24:47.836 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - ATTACK SUCCESSFUL
18:24:47.836 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.ReducerActor$ - STALE BLOCKS... 0
18:24:47.836 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.MasterActor - FINISH EVENTS...
18:24:47.839 [VSystem-akka.actor.default-dispatcher-15] DEBUG vibes.Main$ - EXTRACTION...
18:24:47.840 [VSystem-akka.actor.default-dispatcher-15] DEBUG vibes.Main$ - SUCCESS...
```

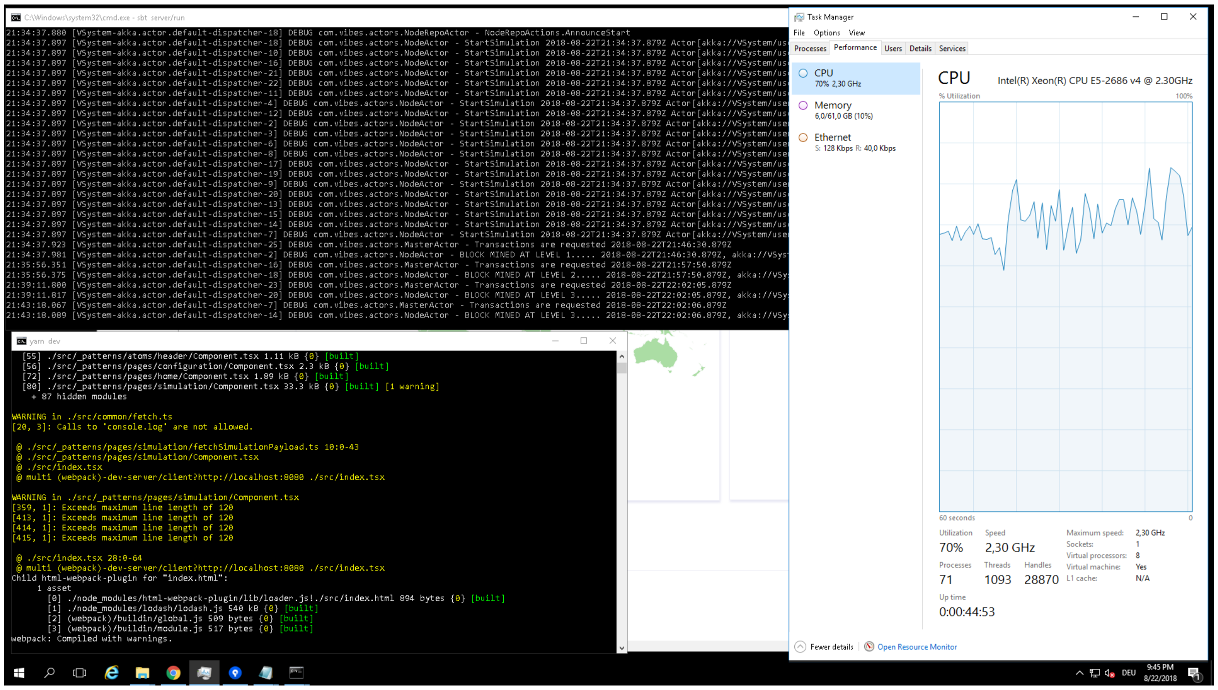
Figure A.2: Screenshot Backend Console

A.6 Source code

Available at `https://github.com/i13-msrg/vibes/tree/FabianSchuessler`.

A.7 Scripts and logs for the evaluation

The logs of the simulation results that are summarized in the evaluation section are available as a proof at `https://github.com/i13-msrg/vibes/tree/FabianSchuessler/testing` and also the scripts for reproduction.



C:\Windows\system32\cmd.exe - sbt server/run

```
21:34:37.880 [VSystem-akka.actor.default-dispatcher-18] DEBUG com.vibes.actors.NodeRepoActor - NodeRepoActions.AnnounceStart
21:34:37.897 [VSystem-akka.actor.default-dispatcher-18] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-10] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-16] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-21] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-22] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-11] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-4] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/user
21:34:37.897 [VSystem-akka.actor.default-dispatcher-12] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-2] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/user
21:34:37.897 [VSystem-akka.actor.default-dispatcher-3] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/user
21:34:37.897 [VSystem-akka.actor.default-dispatcher-6] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/user
21:34:37.897 [VSystem-akka.actor.default-dispatcher-8] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/user
21:34:37.897 [VSystem-akka.actor.default-dispatcher-17] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-19] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-9] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/user
21:34:37.897 [VSystem-akka.actor.default-dispatcher-20] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-13] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-15] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-14] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/us
21:34:37.897 [VSystem-akka.actor.default-dispatcher-7] DEBUG com.vibes.actors.NodeActor - StartSimulation 2018-08-22T21:34:37.879Z Actor[akka://VSystem/user
21:34:37.923 [VSystem-akka.actor.default-dispatcher-25] DEBUG com.vibes.actors.MasterActor - Transactions are requested 2018-08-22T21:46:30.879Z
21:34:37.981 [VSystem-akka.actor.default-dispatcher-2] DEBUG com.vibes.actors.NodeActor - BLOCK MINED AT LEVEL 1..... 2018-08-22T21:46:30.879Z, akka://VSys
21:35:56.351 [VSystem-akka.actor.default-dispatcher-16] DEBUG com.vibes.actors.MasterActor - Transactions are requested 2018-08-22T21:57:50.879Z
21:35:56.375 [VSystem-akka.actor.default-dispatcher-18] DEBUG com.vibes.actors.NodeActor - BLOCK MINED AT LEVEL 2..... 2018-08-22T21:57:50.879Z, akka://VSy
21:39:11.800 [VSystem-akka.actor.default-dispatcher-23] DEBUG com.vibes.actors.MasterActor - Transactions are requested 2018-08-22T22:02:05.879Z
21:39:11.817 [VSystem-akka.actor.default-dispatcher-20] DEBUG com.vibes.actors.NodeActor - BLOCK MINED AT LEVEL 3..... 2018-08-22T22:02:05.879Z, akka://VSy
21:43:18.067 [VSystem-akka.actor.default-dispatcher-7] DEBUG com.vibes.actors.MasterActor - Transactions are requested 2018-08-22T22:02:06.879Z
21:43:18.089 [VSystem-akka.actor.default-dispatcher-14] DEBUG com.vibes.actors.NodeActor - BLOCK MINED AT LEVEL 3..... 2018-08-22T22:02:06.879Z, akka://VSy
```

yarn dev

```
  [55] ./src/_patterns/atoms/header/Component.tsx 1.11 kB {0} [built]
  [56] ./src/_patterns/pages/configuration/Component.tsx 2.3 kB {0} [built]
  [72] ./src/_patterns/pages/home/Component.tsx 1.89 kB {0} [built]
  [80] ./src/_patterns/pages/simulation/Component.tsx 33.3 kB {0} [built] [1 warning]
    + 87 hidden modules

WARNING in ./src/common/fetch.ts
[20, 3]: Calls to 'console.log' are not allowed.

 @ ./src/_patterns/pages/simulation/fetchSimulationPayload.ts 10:0-43
 @ ./src/_patterns/pages/simulation/Component.tsx
 @ ./src/index.tsx
 @ multi (webpack)-dev-server/client?http://localhost:8080 ./src/index.tsx

WARNING in ./src/_patterns/pages/simulation/Component.tsx
[359, 1]: Exceeds maximum line length of 120
[413, 1]: Exceeds maximum line length of 120
[414, 1]: Exceeds maximum line length of 120
[415, 1]: Exceeds maximum line length of 120

 @ ./src/index.tsx 28:0-64
 @ multi (webpack)-dev-server/client?http://localhost:8080 ./src/index.tsx
Child html-webpack-plugin for "index.html":
     1 asset
       [0] ./node_modules/html-webpack-plugin/lib/loader.js!./src/index.html 894 bytes {0} [built]
       [1] ./node_modules/lodash/lodash.js 540 kB {0} [built]
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
webpack: Compiled with warnings.
```

Task Manager

CPU Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz

```
CPU       70% 2,30 GHz
Memory    6,0/61,0 GB (10%)
Ethernet  S: 128 Kbps R: 40,0 Kbps

Utilization   Speed                  Maximum speed:      2,30 GHz
70%           2,30 GHz               Sockets:            1
                                     Virtual processors: 8
Processes     Threads   Handles      Virtual machine:    Yes
71            1093      28870        L1 cache:           N/A

Up time
0:00:44:53
```

Figure A.3: Screenshot BBSS on AWS

A.8 NodeActor's source code

Listing A.2: NodeActor's source code

```
package com.vibes.actors

import java.util.UUID

import akka.actor.{Actor, ActorRef, Props}
import akka.util.Timeout
import com.typesafe.scalalogging.LazyLogging
import com.vibes.actions._
import com.vibes.models.{VBlock, VNode, VTransaction}
import com.vibes.utils.{VConf, VExecution}
import org.joda.time.DateTime

import scala.collection.SortedMap
import scala.concurrent.duration._

/*
* The NodeActor represents both a full Node and a miner in the blockchain
    ↪ network. As
* described in VIBES Chapter 6, it is a good opportunity for future work to
    ↪ differentiate between
* both.
* In VIBES, each NodeActor has its own blockchain, pool of pending transactions
* (candidates for the next block) and neighbours. It also works to solve the
    ↪ next block in
* the chain. A Node has its own priority queue of executables and the next
    ↪ solution of
* a block by this Node is represented by the first executable of type
    ↪ MineBlock in the queue.
* As a reminder, a NodeActor's executables types are either MineBlock,
    ↪ IssueTransaction,
* PropagateTransaction or PropagateBlock.
* A vote is represented by a best guess (timestamped work request) sent to the
* Coordinator. The NodeActor sends votes to the Coordinator and is only
    ↪ allowed to
* execute a piece of work once the Coordinator has granted a permission based
    ↪ on the votes.
* Moreover, this Actor receives and propagates blocks from other Nodes in the
    ↪ network. If
* a received block comes from a longer chain, the Actor takes care to follow
    ↪ synchronization
```

```
* steps described in Section 2.7. The Node synchronizes its blockchain, pool
    ↪ of transactions,
* rolls back any orphan blocks and adds any valid transactions from orphan
    ↪ blocks back
* to the transaction pool if they are not already included in the chain.
    ↪ Besides blocks, the
* NodeActor also takes care to propagate and create transactions in the
    ↪ network.
*
* In BBSS Chapter 4, the approach for transaction spam and alternative history
    ↪ attacks is
* described. In case of an active alternative history attack the nodes only
    ↪ accept blocks by
* miners of the same honesty attribute type. The status of the attack is
    ↪ checked and if the
* attack is finished, the neighbours are updated to allow neighbours of a
    ↪ different honesty type.
*
*/
class NodeActor (
  masterActor: ActorRef,
  nodeRepoActor: ActorRef,
  discoveryActor: ActorRef,
  reducerActor: ActorRef,
  lat: Double,
  lng: Double,
  isEvil: Option[Boolean]
) extends Actor with LazyLogging {
  implicit val timeout: Timeout = Timeout(20.seconds)
  private var node = new VNode(
    id = UUID.randomUUID().toString,
    actor = self,
    blockchain = List.empty,
    transactionPool = Set.empty,
    neighbourActors = Set.empty,
    nextRecipient = None,
    lat = lat,
    long = lng,
    isMalicious = isEvil
  )

  /**
    * Every node has a priority queue of executables, current implementation
        ↪ utilizes TreeMap / SortedMap instead
    * because I'd also need to filter the Queue by criteria later. Node
        ↪ workRequest with those executables to the MasterActor
```

```
   * and the Node with the smallest timestamp for workRequested executable
     ↪ receives the right to fast-forward the network
   * aka run the executable and its Queue. The executable is then removed
     ↪ from the Queue, then the appropriate Nodes
   * recompute their Queues and workRequest again recursively
   */
 private var executables: SortedMap[VExecution.WorkRequest, VExecution.Value]
   ↪ = SortedMap.empty

 var blockList: Set[VBlock] = Set.empty

 private def addExecutablesForMineBlock(now: DateTime): Unit = {
   val timestamp = node.createTimestampForNextBlock(now)
   val exWorkRequest = node.createExecutableWorkRequest(self, timestamp,
     ↪ VExecution.ExecutionType.MineBlock)
   val value = () => {
     val newBlock = VBlock.createWinnerBlock(node, timestamp)

     if (VConf.isAlternativeHistoryAttack) {
       addBlockIfAlternativeHistoryAttack(timestamp, newBlock)
     } else {
       logger.debug(s"BLOCK MINED AT LEVEL ${node.blockchain.size + 1}.....
         ↪ $timestamp, ${self.path}")
       node = node.addBlock(newBlock)
     }

     reducerActor ! ReducerActions.AddBlock(newBlock)
     addExecutablesForPropagateOwnBlock(timestamp)
     nodeRepoActor ! NodeRepoActions.AnnounceNextWorkRequestAndMine(timestamp)
   }
   executables += exWorkRequest -> value
 }

 private def addExecutableForIssueTransaction(toActor: ActorRef, timestamp:
   ↪ DateTime, isFloodAttack: Boolean): Unit = {
   val transaction = VTransaction.createOne(self, toActor, timestamp,
     ↪ node.blockchain.size, isFloodAttack)
   val exWorkRequest =
     transaction.createExecutableWorkRequest(self, self, timestamp,
       ↪ VExecution.ExecutionType.IssueTransaction)
   val value = () => {
     node.addTransaction(transaction)
     addExecutablesForPropagateTransaction(transaction, timestamp)
     self ! NodeActions.CastNextWorkRequestOnly
   }
   executables += exWorkRequest -> value
```

```
}

private def addExecutablesForPropagateOwnBlock(timestamp: DateTime): Unit = {
  node.neighbourActors.foreach { neighbour =>
    val exWorkRequest = node.createExecutableWorkRequest(  neighbour,
       ↪ timestamp.plusMillis(VConf.blockPropagationDelay),
       ↪ VExecution.ExecutionType.PropagateOwnBlock)
    val hash = node.createBlockchainHash()
    val value = () => {
      neighbour ! NodeActions.ReceiveBlock(node, node.blockchain.head,
         ↪ exWorkRequest.timestamp, hash)
      self ! NodeActions.CastNextWorkRequestOnly
    }
    executables += exWorkRequest -> value
  }
}

def addExecutablesForPropagateTransaction(transaction: VTransaction,
    ↪ timestamp: DateTime): Unit = {
  node.neighbourActors.foreach { neighbour =>
    val exWorkRequest = transaction
      .createExecutableWorkRequest(self, neighbour,
         ↪ timestamp.plusMillis(VConf.transactionPropagationDelay),
         ↪
         ↪ VExecution.ExecutionType.PropagateTransaction)
    val value = () => {
      neighbour ! NodeActions.ReceiveTransaction(node, transaction,
         ↪ exWorkRequest.timestamp)
      self ! NodeActions.CastNextWorkRequestOnly
    }
    executables += exWorkRequest -> value
  }
}

private def addExecutablesForPropagateExternalBlock(now: DateTime): Unit = {
  node.neighbourActors.foreach { neighbour =>
    val workRequest = node.createExecutableWorkRequest(neighbour,
       ↪ now.plusMillis(VConf.blockPropagationDelay),
       ↪ VExecution.ExecutionType.PropagateExternalBlock)
    val hash = node.createBlockchainHash()
    val value = () => {
      if (node.blockchain.isEmpty) {
        logger.debug(s"empty ${self.path}")
      }
      neighbour ! NodeActions.ReceiveBlock(node, node.blockchain.head,
         ↪ workRequest.timestamp, hash)
      self ! NodeActions.CastNextWorkRequestOnly
    }
    executables += workRequest -> value
  }
}

// method only for alternative history attack
private def addBlockIfAlternativeHistoryAttack(timestamp: DateTime,
    ↪ newBlock: VBlock): Unit = {
  if (node.isMalicious.contains(true)) {
    logger.debug(s"EVIL BLOCK MINED AT LEVEL..... ${node.blockchain.size +
        ↪ 1}")
  } else {
    logger.debug(s"GOOD BLOCK MINED AT LEVEL..... ${node.blockchain.size +
        ↪ 1}")
  }

  // checks if to add block
  if (VConf.attackSuccessful) {
    logger.debug(s"addBlockIfAlternativeHistoryAttack:
        ↪ VConf.attackSuccessful")
    node = node.addBlock(newBlock)
  } else if (VConf.attackFailed) {
    logger.debug(s"addBlockIfAlternativeHistoryAttack: VConf.attackFailed")
    node = node.addBlock(newBlock)
  } else if (newBlock.level == 1 && node.isMalicious !=
      ↪ newBlock.origin.isMalicious) { // block zero is the common block
    logger.debug(s"addBlockIfAlternativeHistoryAttack: newBlock.level == 1")
    node = node.addBlock(newBlock)
  } else if (newBlock.level == 0) {
    logger.debug(s"addBlockIfAlternativeHistoryAttack: newBlock.level == 0")
    node = node.addBlock(newBlock)
  } else if (node.isMalicious == newBlock.origin.isMalicious &&
      ↪ node.blockchain.head.timestamp.isBefore(newBlock.timestamp)) {
    logger.debug(s"addBlockIfAlternativeHistoryAttack: added newBlock")
    node = node.addBlock(newBlock)
  } else {
    logger.debug(s"addBlockIfAlternativeHistoryAttack: Didn’t add newBlock
        ↪ ${newBlock.level}, ${VConf.attackSuccessful} and
        ↪ ${VConf.attackFailed}")
  }

  // checks if attack is finished
  if (!VConf.attackFailed && !VConf.attackSuccessful) {
    // sets good and evil chain length
    if (node.blockchain.size == 1) {
      VConf.evilChainLength = node.blockchain.size
      VConf.goodChainLength = node.blockchain.size
    } else if (node.isMalicious.contains(true)) {
      if (node.blockchain.size > VConf.evilChainLength) {
        VConf.evilChainLength = node.blockchain.size
      }
    } else if (node.isMalicious.contains(false)) {
      if (node.blockchain.size > VConf.goodChainLength) {
        VConf.goodChainLength = node.blockchain.size
      }
    }

    // prints chain lengths
    logger.debug(s"GOOD CHAIN LENGTH: ${VConf.goodChainLength}; BAD CHAIN
      ↪ LENGTH ${VConf.evilChainLength}")

    // checks if attack succeeded
    if (VConf.evilChainLength > VConf.goodChainLength &&
      ↪ VConf.goodChainLength > VConf.confirmations) {
      logger.debug(s"ATTACK IS SUCCESSFUL.....")
      VConf.attackSuccessful = true
      VConf.attackSuccessfulInBlocks = VConf.evilChainLength
      discoveryActor ! DiscoveryActions.AnnounceNeighbours
    }

    // checks if attack failed
    if (((VConf.evilChainLength > VConf.attackDuration &&
      ↪ VConf.goodChainLength > VConf.confirmations) ||
      ↪ VConf.goodChainLength > VConf.attackDuration) &&
      ↪ !VConf.attackSuccessful) {
      logger.debug(s"ATTACK FAILED..... $timestamp, ${self.path}")
      VConf.attackFailed = true
      discoveryActor ! DiscoveryActions.AnnounceNeighbours
    }

    // updates neighbours to make only evil nodes work with evil nodes and
      ↪ good nodes with good nodes after one common block
    if (newBlock.level == 0) {
      discoveryActor ! DiscoveryActions.AnnounceNeighbours
    }
  }
}

override def preStart(): Unit = {
  logger.debug(s"NodeActor started ${self.path}")
```

```
    discoveryActor ! DiscoveryActions.ReceiveNode(node)
  }

  override def preRestart(reason: Throwable, message: Option[Any]): Unit = {
    VConf.attackSuccessful = false
    VConf.goodChainLength = 0
    VConf.evilChainLength = 0
    VConf.attackFailed = false
    logger.debug(s"REASON.... $reason")
    logger.debug(s"MESSAGE.... $message")
    logger.debug(s"PRERESTART.... ${self.path}")
  }

  override def receive: Receive = {
    case NodeActions.StartSimulation(now) =>
      logger.debug(s"StartSimulation $now $self")
      addExecutablesForMineBlock(now)

      self ! NodeActions.CastNextWorkRequestOnly

    case NodeActions.CastNextWorkRequestOnly =>
      masterActor ! MasterActions.CastWorkRequest(executables.head._1)

    /**
      * Comes from a sender() that just mined a block, so his block
          ↪ executables should be empty and all other
      * nodes should remove their MineBlock executables so that they can do
          ↪ their best guesses via
      * addExecutablesForMineBlock(timestamp) again
      */
    case NodeActions.CastNextWorkRequestAndMine(timestamp, sender) =>
      val count = if (sender != self) 1 else 0
      assert(
        executables.count(executable => executable._1.executionType ==
            ↪ VExecution.ExecutionType.MineBlock) == count
      )
      if (sender != self) {
        // delete previous guess to generate a new one, only the broadcaster /
            ↪ sender doesn't delete his block
        executables = executables
          .filter(executable => executable._1.executionType !=
            ↪ VExecution.ExecutionType.MineBlock)
      }

      addExecutablesForMineBlock(timestamp)
      self ! NodeActions.CastNextWorkRequestOnly
    case NodeActions.ProcessNextExecutable(workRequest) =>
      val head = executables.head

      assert(
        workRequest.id == head._1.id ||
          (
            workRequest.id != head._1.id &&
              workRequest.executionType ==
                  ↪ VExecution.ExecutionType.PropagateTransaction &&
              head._1.executionType == workRequest.executionType
          )
      )

      executables = executables.tail

      head._2()

    case NodeActions.ReceiveBlock(origin, block, now, hash) =>
      if (VConf.isAlternativeHistoryAttack) {
        // in case of alternative history attack, only propagate external
            ↪ blocks for certain conditions
        if (block.level + 1 > node.blockchain.size &&
            ↪ (node.isMalicious.contains(true) ==
            ↪ block.origin.isMalicious.contains(true) || block.level == 0) &&
            ↪ !VConf.attackSuccessful && !VConf.attackFailed) {
          val incomingBlock = block.addRecipient(origin, node, now)

          if (NodeActor.shouldSynch(node, hash)) {
            node = node.synch(origin, origin.blockchain, now)
          } else {
              logger.debug(s"NodeActions.ReceiveBlock: Added newBlock
                  ↪ ${node.isMalicious}, ${origin.isMalicious},
                  ↪ ${incomingBlock.level}, ${VConf.attackSuccessful} and
                  ↪ ${VConf.attackFailed}")
              node = node.addBlock(incomingBlock)
          }
          addExecutablesForPropagateExternalBlock(now)
        } else if (block.level + 1 > node.blockchain.size &&
            ↪ (VConf.attackSuccessful || VConf.attackFailed)) {
          val incomingBlock = block.addRecipient(origin, node, now)

          if (NodeActor.shouldSynch(node, hash)) {
            node = node.synch(origin, origin.blockchain, now)
          } else {
            logger.debug(s"NodeActions.ReceiveBlock: Added newBlock
                ↪ ${node.isMalicious}, ${origin.isMalicious},
                ↪ ${incomingBlock.level}, ${VConf.attackSuccessful} and
                ↪ ${VConf.attackFailed}")
            node = node.addBlock(incomingBlock)
          }
          addExecutablesForPropagateExternalBlock(now)
        }
      } else {
        if (block.level + 1 > node.blockchain.size) {
          val incomingBlock = block.addRecipient(origin, node, now)

          if (NodeActor.shouldSynch(node, hash)) {
            node = node.synch(origin, origin.blockchain, now)
          } else {
            node = node.addBlock(incomingBlock)
          }
          addExecutablesForPropagateExternalBlock(now)
        }
      }
      self ! NodeActions.CastNextWorkRequestOnly

    case NodeActions.ReceiveNeighbours(neighbours) =>
      node = node.exchangeNeighbours(neighbours)

    case NodeActions.IssueTransaction(toActor, time) =>
      addExecutableForIssueTransaction(toActor, time, false)

    case NodeActions.ReceiveTransaction(origin, transaction, timestamp) =>
      if (node.isTransactionNew(transaction)) {
        val incomingTransaction = transaction.addRecipient(origin, node,
            ↪ timestamp)
        node = node.addTransaction(incomingTransaction)
        addExecutablesForPropagateTransaction(transaction, timestamp)
      }

      self ! NodeActions.CastNextWorkRequestOnly

    case NodeActions.End =>
      reducerActor ! ReducerActions.ReceiveNode(node)

    case NodeActions.IssueTransactionFloodAttack(toActor, time) =>
      addExecutableForIssueTransaction(toActor, time, true)
  }
}
```

```
object NodeActor {
  def props(
    masterActor: ActorRef,
    nodeRepoActor: ActorRef,
    discoveryActor: ActorRef,
    reducerActor: ActorRef,
    lat: Double,
    lng: Double,
    isEvil: Option[Boolean] = None
  ): Props = Props(new NodeActor(masterActor, nodeRepoActor, discoveryActor,
      ↪ reducerActor, lat, lng, isEvil))
  def shouldSynch(node: VNode, hash: Int): Boolean = {
    node.createBlockchainHash() != hash
  }
}
```

Bibliography

[1] "Bitcoin block data." https://de.m.wikipedia.org/wiki/Datei:Bitcoin_Block_Data.png, 2018. Accessed: 2018-08-07.

[2] "Merkle root." https://bitcoin.stackexchange.com/questions/10479/what-is-the-merkle-root, 2018. Accessed: 2018-08-08.

[3] B. Peh, "Confirmation times, stale blocks, reverse transaction, double spending and the 51% attack in simple terms." https://medium.com/@blockchain101/confirmation-times-stale-blocks-reverse-transaction-double-spending-and-the-51-attack-in-simple-bd65a32d32b3, 2018. Accessed: 2018-08-07.

[4] M. D'Aliessi, "How does the blockchain work?." https://medium.com/@micheledaliessi/how-does-the-blockchain-work-98c8cd01d2ae, 2016. Accessed: 2018-08-01.

[5] A. Barrera, "Confirmations." http://tech.eu/features/926/bitcoin-ecosystem/, 2018. Accessed: 2018-08-08.

[6] "Two new models for double spending attacks on bitcoin's blockchain." https://www.deepdotweb.com/2016/12/31/two-new-models-double-spending-attacks-bitcoins-blockchain/, 2018. Accessed: 2018-08-25.

[7] B. Frost, "Atomic web design." http://bradfrost.com/blog/post/atomic-web-design/, 2013. Accessed: 2018-08-01.

[8] M. Rosenfeld, "Analysis of hashrate-based double spending," *CoRR*, vol. abs/1402.2009, 2014.

[9] M. Atzori, "Blockchain technology and decentralized governance: Is the state still necessary?," *SSRN Electronic Journal*, Dec. 2015.

[10] M. Iansiti and K. R. Lakhani, "The truth about blockchain." https://hbr.org/2017/01/the-truth-about-blockchain, 2017. Accessed: 2018-08-06.

[11] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." http://www.bitcoin.org/bitcoin.pdf, 2009.

[12] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform." `https://github.com/ethereum/wiki/wiki/White-Paper`, 2014. Accessed: 2018-08-06.

[13] J. L. Newcomb, S. Chandra, J.-B. Jeannin, C. Schlesinger, and M. Sridharan, "Iota: A calculus for internet of things automation," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, (New York, NY, USA), pp. 119–133, ACM, 2017.

[14] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, (New York, NY, USA), pp. 30:1–30:15, ACM, 2018.

[15] K. Panetta, "Top trends in the gartner hype cycle for emerging technologies, 2017." `https://www.gartner.com/smarterwithgartner/top-trends-in-the-gartner-hype-cycle-for-emerging-technologies-2017/`, 2017. Accessed: 2018-04-15.

[16] L. Stoykov, "Vibes: Fast blockchain simulations for large-scale peer-to-peer networks," Master's thesis, Technische Universität München, 2018.

[17] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation (3rd Edition)*. Prentice Hall, 3 ed., 2000.

[18] "Bitcoin developer guide." `https://bitcoin.org/en/developer-guide`, 2018. Accessed: 2018-08-05.

[19] "Bitcoin wiki." `https://en.bitcoin.it/wiki/Main_Page`, 2018. Accessed: 2018-08-07.

[20] Dinkins, "Satoshi's best kept secret: Why is there a 1 mb limit to bitcoin block size." `https://cointelegraph.com/news/satoshis-best-kept-secret-why-is-there-a-1-mb-limit-to-bitcoin-block-size`, 2017. Accessed: 2018-08-07.

[21] "What is the meaning of the term full node." `https://bitcoin.stackexchange.com/questions/48436/what-is-the-meaning-of-the-term-full-node`, 2018. Accessed: 2018-08-08.

[22] F. Schüssler and Y. Kandalaft, "Bitcoin-like blockchain scalability issues and improvements." Seminar Paper.

[23] G. O. Karame, "Two bitcoins at the price of one? double-spending attacks on fast

payments in bitcoin," in *In Proc. of Conference on Computer and Communication Security*, 2012.

[24] V. Buterin, "Transaction Spam." `https://twitter.com/VitalikButerin/status/1018990773042405376`. Accessed: 2018-07-21.

[25] M. Carlsten, H. Kalodner, S. M. Weinberg, and A. Narayanan, "On the instability of bitcoin without the block reward," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 154–167, ACM, 2016.

[26] T. Neudecker, P. Andelfinger, and H. Hartenstein, "A simulation model for analysis of attacks on the bitcoin peer-to-peer network," in *IM*, pp. 1327–1332, IEEE, 2015.

[27] M. B. Sarrias, "Bitcoin network simulator data explotation," Master's thesis, Open University of Catalonia, 2015.

[28] "React - a javascript library for building user interfaces." `https://reactjs.org/`, 2018. Accessed: 2018-08-13.

[29] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[30] "Scala." `https://www.scala-lang.org/`, 2018. Accessed: 2018-04-15.

[31] "Akka." `https://akka.io/`, 2018. Accessed: 2018-04-15.

[32] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver, "Stressing out: Bitcoin "stress testing"," in *Financial Cryptography Workshops*, vol. 9604 of *Lecture Notes in Computer Science*, pp. 3–18, Springer, 2016.

[33] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, (Berkeley, CA, USA), pp. 45–59, USENIX Association, 2016.

[34] R. Beck, J. S. Czepluch, N. Lollike, and S. Malone, "Blockchain - the gateway to trust-free cryptographic transactions," in *24th European Conference on Information Systems, ECIS 2016, Istanbul, Turkey, June 12-15, 2016*, p. Research Paper 153, 2016.

[35] "Details on genesis block." `https://bitcoin.stackexchange.com/questions/18454/details-on-genesis-block`. Accessed: 2018-07-23.

[36] "React google charts." `https://github.com/RakanNimer/react-google-charts`. Accessed: 2018-07-21.

[37] "React google charts - issue 197." `https://github.com/rakannimer/react-google-charts/issues/197`. Accessed: 2018-07-21.

[38] "React google charts - issue 202." `https://github.com/rakannimer/react-google-charts/issues/202`. Accessed: 2018-07-21.

[39] "Network snapshot." `https://bitnodes.earn.com/nodes/`, 2018. Accessed: 2018-08-15.

[40] "Transaction size." `https://charts.bitcoin.com/btc/chart/transaction-size`, 2018. Accessed: 2018-08-15.

[41] "Average number of transactions per block." `https://www.blockchain.com/charts/n-transactions-per-block?`, 2015. Accessed: 2018-08-15.

[42] B. Group, "Block size increase." `https://bitfury.com/content/downloads/block-size-1.1.1.pdf`, 2015. Accessed: 2018-08-15.

[43] "Bitcoin energy consumption index." `https://digiconomist.net/bitcoin-energy-consumption`, 2018. Accessed: 2018-08-25.