EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

BACHELOR

Brabocoin
an educational cryptocurrency based on Bitcoin

van den Berg, Dennis P.; Dekker, D.J.C.; van den Eerenbeemt, Sophie; Wessel, Sten

Award date:
2019

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technology
Department of Mathematics and Computer Science

2WH40 Bachelor Final Project

Brabocoin

An educational cryptocurrency based on Bitcoin

May 13, 2019

Authors

D.P. van den Berg	0949036
D.J.C. Dekker	0936100
S. van den Eerenbeemt	0954445
S. Wessel	0941508

Supervisors
B.M.M. de Weger
B. Skoric

Abstract

Bitcoin is an elaborate and complicated cryptocurrency with a strong mathematical foundation. Bitcoin is designed to be usable and efficient, which means the cryptography is hidden in the internal functionality of the system. This suggests the need for an educational implementation of a cryptocurrency based on Bitcoin, which exposes both the inner workings and the mathematics that supports the Bitcoin system. In this document, we provide such an educational application, called Brabocoin. Brabocoin provides a way for interested parties, such as Bitcoin enthusiasts and students, to explore the structure and functioning of Bitcoin. We provide a main overview of the workings of Bitcoin, followed by a detailed exploration of all mathematical subjects on which Bitcoin is built. To this end, we provide an analysis of elliptic curve cryptography, hash functions, digital signatures, and the mathematics behind cryptocurrency wallets. We explain various parts of the Brabocoin software, cover design decisions made during development and motivate deviations from Bitcoin. A list of future work is provided, which contains features that can be added to Brabocoin at a later stage.

Authors

Section	Author
Section 1	All
Section 2	All
Section 3	S. Wessel
Section 4 excluding 4.2.3 and 4.3.10	S. van den Eerenbeemt
Section 4.2.3 and 4.3.10	D.P. van den Berg
Section 5	D.J.C. Dekker
Section 6	D.P. van den Berg
Section 7	All
Section 8	All

Contents

1 Introduction **7**

2 An overview of Bitcoin **8**
2.1 Proving ownership . . . 8
2.2 Creating transactions . . . 8
2.3 Mining . . . 9
2.4 The blockchain . . . 10
2.5 The transaction pool . . . 10
2.6 The UTXO set . . . 10
2.7 Orphan transactions . . . 11
2.8 Chain reorganization and forks . . . 11
2.9 The 51% attack . . . 11

3 Elliptic curve cryptography **13**
3.1 Introduction . . . 13
3.2 Discrete logarithm problem . . . 13
3.3 Elliptic curves . . . 13
3.3.1 Group structure . . . 14
3.4 Discrete logarithm problem for elliptic curves . . . 16
3.5 The Bitcoin curve . . . 17
3.6 Efficient implementation . . . 18
3.6.1 Projective coordinates . . . 18
3.6.2 Scalar point multiplication . . . 20
3.7 Point compression . . . 26

4 Hashing **28**
4.1 Introduction . . . 28
4.2 Hashing methods . . . 28
4.2.1 Definitions . . . 28
4.2.2 Merkle Damgård construction . . . 29
4.2.3 MD5 . . . 29
4.2.4 SHA-256 . . . 31
4.2.5 RIPEMD-160 . . . 34
4.3 Applications of hashing in Bitcoin . . . 36
4.3.1 Double hashing . . . 36
4.3.2 Block and transaction identifier . . . 37
4.3.3 Proof-of-work . . . 38
4.3.4 Private key generation . . . 38
4.3.5 Converting public keys to addresses . . . 39
4.3.6 Base58Check encoding . . . 39
4.3.7 Bloom filters . . . 40
4.3.8 Merkle trees . . . 42
4.3.9 Digital signatures . . . 44
4.3.10 Message authentication codes . . . 45

5 Digital signatures **47**
5.1 Current digital signature schemes . . . 47
5.1.1 Introduction . . . 47
5.1.2 ElGamal signatures . . . 48
5.1.3 DSA . . . 49
5.1.4 ECDSA . . . 51
5.1.5 Public key recovery . . . 52
5.1.6 Bitcoin ECDSA . . . 53
5.1.7 Issues in the post-quantum era . . . 55
5.2 Hash-based digital signature schemes . . . 56
5.2.1 Introduction hash-based signatures . . . 56

5.2.2 Notation . . . 56
5.2.3 Lamport-Diffie one-time signature . . . 56
5.2.4 An alternative approach for the Lamport-Diffie one-time signature . . . 58
5.2.5 Winternitz one-time signature . . . 60
5.2.6 W-OTS$^+$ . . . 62
5.2.7 Few time signature schemes . . . 64

6 Wallets **66**
6.1 Wallet types . . . 66
6.2 Hierarchical deterministic wallet . . . 66
6.2.1 Extended keys . . . 67
6.2.2 Obtaining the master extended private key . . . 67
6.2.3 Child key derivation . . . 67
6.2.4 Hardened child key derivation . . . 68

7 Brabocoin **71**
7.1 Software overview . . . 71
7.2 Data model . . . 71
7.2.1 Hash . . . 72
7.2.2 Transaction . . . 73
7.2.3 Block . . . 74
7.2.4 Network messages . . . 74
7.2.5 Wallet . . . 76
7.3 Advanced data structures . . . 76
7.3.1 Blockchain . . . 76
7.3.2 Chain UTXO set . . . 77
7.3.3 Transaction pool . . . 78
7.3.4 Orphan blocks and transactions . . . 79
7.3.5 Recently rejected blocks and transactions . . . 80
7.4 Protocol specification . . . 80
7.4.1 Network design . . . 80
7.4.2 Protocol description and gRPC . . . 80
7.5 Node environment and peers . . . 82
7.5.1 Message queue . . . 82
7.5.2 Initialization . . . 82
7.5.3 Maintaining a set of peers . . . 83
7.5.4 Initializing the transaction pool . . . 83
7.5.5 Updating the blockchain . . . 83
7.5.6 Message propagation . . . 83
7.5.7 Handling orphan blocks . . . 84
7.6 Consensus and configuration . . . 84
7.6.1 Consensus . . . 84
7.6.2 Configuration . . . 88
7.7 Validation and verification . . . 91
7.7.1 Transaction validator . . . 91
7.7.2 Block validator . . . 97
7.8 Mining . . . 102
7.8.1 Configuration . . . 102
7.8.2 Mining refresh . . . 102
7.8.3 Continuously mine vs. Mine single block . . . 103
7.8.4 Mining procedure . . . 103
7.9 Processors . . . 104
7.9.1 Starting the node . . . 104
7.9.2 Block processor . . . 104
7.9.3 Transaction processor . . . 105
7.10 Wallet . . . 106
7.10.1 Key pair management . . . 106
7.10.2 Balance calculation . . . 106
7.10.3 Transaction history . . . 107

7.11 Deviations from Bitcoin 107
7.11.1 General 107
7.11.2 Network 107
7.11.3 Transactions 108
7.11.4 Blocks 109
7.11.5 Blockchain 110
7.11.6 Transaction pool 111
7.11.7 UTXO sets 111
7.11.8 Validation 111
7.11.9 Wallet 112
7.12 Future work 112

8 Conclusion **114**

1 Introduction

Bitcoin was developed to be a payment system that allows online payments between parties without a central authority: a decentralized peer-to-peer cash system or *cryptocurrency*. Because the design of Bitcoin is based on cryptographic proofs instead of blind trust [48], Bitcoin is based on various mathematical principles. Bitcoin is designed to be comprehensible by the average user, which means the cryptography is hidden in the internal functionality of the system. Its software is an open source project, meaning its design is public. The Bitcoin core has matured and experienced multiple adjustments in order to add features and improve efficiency, making it challenging to understand the principles and foundation it is based on.

This suggests the need for an educational implementation of a cryptocurrency based on Bitcoin, which exposes both the inner workings and the mathematics that supports the Bitcoin system. Such an application provides a way for interested parties, such as Bitcoin enthousiasts and students, to explore and experiment with the structure and functioning of Bitcoin.

To this end, we developed the Brabocoin application. It is based on the foundation of Bitcoin and provides interested parties with an environment where they can explore and experiment with the workings of cryptocurrencies.

In order to achieve this, we investigated the Bitcoin core implementation and used this to design Brabocoin's software. Brabocoin's main features are similar to those of Bitcoin, just like Brabocoin's basic software structure. However, we made some essential adjustments in order to make the Brabocoin application usable for educational purposes.

First of all, Brabocoin is an attempt at an implementation of a cryptocurrency based on Bitcoin, that is more transparent. Bitcoin does not expose the data and internal state of the application to the user; it hides these details from the user in order to make the software comprehensible. In Brabocoin, we expose the data and inner functionalities to the user. This way, the user can view the internal data and processes that are instrumental to Bitcoin and other cryptocurrencies. Brabocoin users can experiment and manipulate the data and observe the effects when doing so. This also allows, for example, attacks that are hypothetically possible on Bitcoin to be performed in practice.

Furthermore, the cryptography that supports the Bitcoin system is adequately visualized in Brabocoin. Brabocoin makes the user aware of which elements of cryptography are used, when they are used and what it is used for. It teaches the user of Brabocoin about the mathematical foundation of cryptocurrencies.

Next to this, Brabocoin's main functionality is a stripped version of Bitcoin's functionality. It contains the essential features Bitcoin provides, disregarding optimizations and additional features in Bitcoin that are not essential to a cryptocurrency. This gives the user the opportunity to properly experiment with the main features that Bitcoin provides, in a controlled and limited environment.

Finally, the user has the ability to purposely send invalid data over the network. For example, a user can create an invalid transaction. For obvious reasons, in Bitcoin these kind of features are not directly available to the user. In Brabocoin, however, we made these features available, such that users can experiment with creating invalid data and observe its effects.

In this document, we first provide an overview of Bitcoin's overall functionality in Section 2. We then explore various subjects with regards to Bitcoin's mathematical foundation. In Section 3 we introduce elliptic curves and explore the elliptic curve used by Bitcoin. Afterwards, we elaborate on the subject of hash functions, and more specifically, the hash functions used in Bitcoin, in Section 4. We provide a list of where these hash functions are used in Bitcoin, and analyze the properties of these hash functions. In Section 5, digital signatures are discussed. We discuss several signature schemes, including the schemes used by Bitcoin. We analyse their correctness, efficiency, and possibilities for attacks. Next, we discuss cryptocurrency wallets and their mathematical foundation in Section 6. In particular, the subject of hierarchical wallets is examined.

Section 7 elaborates upon the Brabocoin implementation. All software components are introduced and discussed in detail. Design decisions made during the creation of Brabocoin as well as the main deviations from Bitcoin, are described and motivated. We provide a list of future work, containing features that can be added to Brabocoin.

2 An overview of Bitcoin

Bitcoin belongs to a large group of cryptocurrencies. It allows us to safely pay digital money to others, without the presence of a trusted authority, like a bank or government. There is no single database maintained by the bank which contains all the information. Rather, all information is stored and verified by many users of the Bitcoin network. This is why Bitcoin is called a *decentralized* system, where users are connected in a *peer-to-peer* network. Each user is a node in the network and each node has a number of nodes it communicates with, these are called its *peers*. The advantage of this set-up is that it is almost impossible for one or a few users to abuse the system, without the need for a trusted authority. There is a consensus: as long as enough honest users verify all transactions and store them correctly, fraudulent data will simply be ignored.

The unit of currency is called bitcoin. All transactions transfer bitcoin from one person to another. These transactions are grouped and stored in blocks. Each block is linked to a previous one, forming a *blockchain*. Creating a valid block is time-consuming and resource-intensive. Trying to create valid blocks is known as *mining*; the users that try to create valid blocks are *miners*. Every time a block is mined, new bitcoin is created and transferred to the miner who mined the block. This is the *mining reward*. Mining is part of the mechanism in Bitcoin that allows the emergence of consensus in a network of individuals. This is further explained in Section 2.4. The amount of bitcoin that is created in a new block halves every time another 2016 blocks have been mined on the blockchain. This means that only a fixed, maximum amount of bitcoin can exist, namely 21 million [1]. In the next sections, we describe Bitcoin in more detail.

2.1 Proving ownership

Users have secret keys, which are called private keys. These are long, random integers, which users can use to prove ownership of their money, and thus, spend the money they own. Proving ownership is done using a digital signature, which is calculated using the user's private key. Details about digital signatures are discussed in Section 5.

From each private key, one can calculate a public key using elliptic curve cryptography. Details about elliptic curve cryptography are discussed in Section 3. The basic idea is that we want to transfer bitcoin to a person's non-secret public key. Calculating this public key using the private key is done using a one-way function: it is easy to calculate the public key from the private key, but computationally infeasible to do so the other way around. When you want to transfer bitcoin to another address, you must first prove ownership of the public key the money currently belongs to, by means of the corresponding private key. This is done using a digital signature.

The basic idea is now covered, but Bitcoin uses an extra layer of security. We usually do not pay bitcoin to a public key (although it is possible), but rather, we pay to an address. The reason why this provides additional security is discussed in Section 5.1.6. An address is calculated from a public key using another type of one-way function: hash functions. Details about hash functions are discussed in Section 4.

Users may own multiple keys, which is why Bitcoin has *wallets* that manage the keys that users own. They serve as a user interface to simplify the creation of new transactions and new private and public keys. More details on wallets and the generation of keys can be found in Section 6.

In summary, we pay bitcoin to an address. The owner can derive this address from the public key with a hash function and can derive the public key from the private key with elliptic curve cryptography. Deriving the public key from the address, and deriving the private key from the public key, are both hard problems that are computationally infeasible to solve. When we want to spend bitcoin, we must prove ownership of the spent money using a digital signature. Bitcoin uses wallets to manage the user's keys.

2.2 Creating transactions

Transactions in Bitcoin consist of so-called *inputs* and *outputs*. An output consists of an amount that we pay to an address. In an input, we reference an output of another transaction. In order to spend the money of this output, we prove that we own the address the money was paid to. Note that we always use the entire output as a new input, and need to transfer back some change to yourself. In order to do

this, we create an extra output. This output pays the change back to an address of the creator of the transaction.

Transactions are identified by their hash, which is calculated using a one-way hash function. In order to reference an output we wish to spend, we use the hash of the transaction we are referencing, together with the index of the output in that referenced transaction. We do not need to specify the amount that we use, as this is equal to the amount specified in the referenced output.

As an example: we want to pay someone 0.3 BTC (bitcoin), but we want to reference an output with an amount of 1 BTC. We must reference the output in an input, using the transaction hash and output index of that referenced output. We must also provide a digital signature, proving that we indeed own the 1 BTC we are referencing. We create one output paying 0.3 BTC to the address of the receiver, and another output for the change: we send 0.65 BTC back to one of our addresses. Note that we do not spend the entire 1 BTC that we referenced. To give miners more incentive to include the transaction in the block, they receive a *transaction fee* from the creator of the transaction, for each transaction they mine. This fee is the difference between the bitcoin we referenced in our input and the bitcoin we output in the transaction. In our example, that would be $1 - 0.3 - 0.65 = 0.05$ BTC. The user decides the value of the fee in the transaction. The total reward a miner receives for mining a block is the block reward plus the sum of all fees of the transactions within the block.

Currently, these fees are low compared to the mining reward for creating a new block. As discussed earlier, mining a block is difficult, and is rewarded with new bitcoin. However, this reward halves approximately every four years. Transaction fees ensure that miners will still receive a compensation for mining blocks when the mining reward has become very low. It is worth noting that it is technically not required to add a fee in a transaction in Bitcoin. However, if no fee is added, the chance that your transaction is accepted in the blockchain is low. This is because the fee in a transaction is an incentive for a miner to add the transaction to a new block. It is beneficial for a miner to add transactions with a higher fee, as the total reward for new blocks will be higher.

2.3 Mining

Transactions must comply with a number of rules that everyone adheres to in the network. These rules are called the consensus parameters. When we are finished creating our transaction, we send it to our peers. They will validate it first. If the transaction is valid, they send it to their peers, propagating it through the entire network. However, this does not mean that the transaction is stored in the blockchain. The transaction must still be included in a block for it to be considered spent. If a transaction is invalid, it is rejected and ignored.

Miners assemble new blocks containing valid transactions. These new blocks can then be stored in the blockchain. They can choose which transactions they would like to include in their new block. Typically, the transactions with the largest fees are included.

Before creating the actual block, miners create one special transaction, the *coinbase transaction*, that pays the fees and mining reward to the miner. This transaction has one output and zero inputs. This coinbase transaction is also included in the block, together with the earlier chosen transactions. To mine the block, miners spend much time to make sure that the block is valid, which is done using the *proof-of-work* algorithm.

Bitcoin is tuned such that one new block is mined approximately every ten minutes. That means that it must be hard to create a new, valid block. The challenge is to create a block with a smaller hash than a certain *target value*. The hash of a block is calculated over the *block header* of the block, which is defined as the block without the included transactions. The target value is the same for every miner and is adjusted throughout the network, such that approximately every ten minutes a new block is mined. This target value is adjusted every 2016 blocks and is based on the average time it took to mine one block (over these 2016 blocks) [1]. When the hash of a block is not small enough, the block is slightly changed and the hash is calculated again. To support this, each block has a *nonce* field that can be filled with arbitrary data. By trying many nonces, a miner will eventually find a nonce that gives a small enough hash. Due to the properties of hash functions (see Section 4), every nonce has an equal chance of resulting in a valid block. This process is known as the proof-of-work algorithm.

If the miner finds a nonce that gives a valid block, he sends the block to his peers. They will validate it and send it to their peers, and the block will be propagated through the network. All users will stop

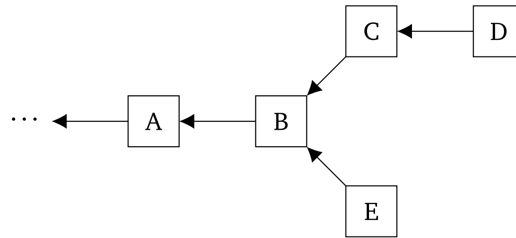

Figure 1: A fork in the blockchain.

mining: each block references the previous one, so all miners have to start over with a new block that points to the just added block. This is why mining is a competition for finding a nonce that yields a block hash less or equal to the target value.

2.4 The blockchain

The blockchain is not a simple chain of blocks. Every block points to a previous one, but nothing prevents two blocks from pointing to the same parent. This situation is called a fork. An example is displayed in Figure 1: both blocks C and E point towards block B.

Every user always considers the longest chain to be the *main chain*. When a fork occurs, for example when two users mine a new block almost simultaneously, it will resolve automatically. One of the two chains will be extended with a new block. Therefore, one of the chains will be longer, and thus, be the main chain. In the example, the chain with blocks C and D is the main chain, because that is the longest chain. Block E is on a fork and will be ignored.

This means that it is not a good idea to create a fork on purpose; your block will be ignored, unless your fork becomes longer than the current main chain. It also means that a transaction might not be included in the main chain, even when it is mined in a block. Hence, it is common to wait for six blocks to be mined on top of the block containing your transaction, which are called confirmations. After six confirmations, it is safe to assume that your block (and hence, your transaction) stays on the main chain.

Confirmations are also required for spending block rewards. After a miner mines a new block, the block rewards are transferred to the miner via the coinbase transaction of a block. However, the miner can only spend his block rewards after 100 blocks have been mined on top of his mined block. This value is called the *coinbase maturity*.

2.5 The transaction pool

When we send a transaction to our peers, they validate the transaction before propagating it through the network. Afterwards, they add the transaction to their *transaction pool*. This pool contains all valid transactions that are not in a block contained in the blockchain. In Bitcoin, this pool is usually called the *mempool*. When we start mining, we can select some transactions from the transaction pool to add to the new block. When a block is mined, transactions in the block are removed from the transaction pool.

When someone receives a transaction with outputs that he can spend, it is recommended to wait some time before spending. However, we could immediately spend the outputs of a transaction which is still in the transaction pool. In this situation, we cannot create a block with the transaction that depends on the other transaction, without including them both. Bitcoin uses specific data structures to store transactions in the transaction pool in order to make the selection of transactions for mining as fast as possible.

2.6 The UTXO set

While verifying a transaction, we must find the referenced transactions of the inputs in the transaction, in order to know how much bitcoin was referenced in each input. In order to find the referenced output, we need to find the block that contains the transaction with this output. Note, however, that only the hashes of these referenced transactions are stored in the inputs, not the block height of the block they

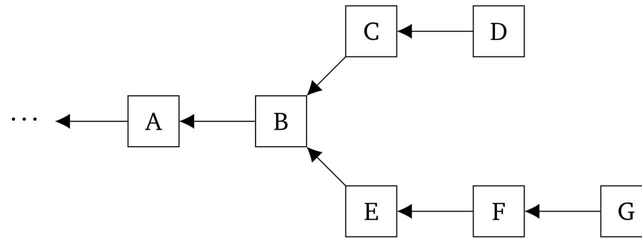

Figure 2: A blockchain with a longest chain and a fork.

were included in. Scanning through the entire blockchain to find the referenced transactions would cost too much time. To keep track of unspent transaction outputs (UTXO for short), *all* unspent outputs are stored in a set, known as the *UTXO set*. Although this is a large set, it makes validation of transactions, and therefore blocks, a lot faster. Note that the size of the UTXO set is considerably smaller than the size of the blockchain. Since we can also spend outputs that are still in the transaction pool, we need an extra UTXO set to keep track of the unspent outputs created in the transaction pool. This set is called the *pool UTXO set*.

2.7 Orphan transactions

It could happen that we receive a transaction, but one of the referenced transactions is neither in the blockchain UTXO set, nor the pool UTXO set. This can have multiple reasons. First, it could be that the referenced output has already been spent, making this a *double spending* transaction, which is considered invalid. It could also be the case that we are missing the referenced transaction, because either our blockchain or our transaction pool is not up-to-date. This is why we call these transactions *orphan*, which are not considered valid at this point. If we would receive the missing referenced transaction later, or receive the block in which the referenced transaction is recorded, the orphan transaction might become valid.

2.8 Chain reorganization and forks

The presence of forks could cause a situation where the main chain changes, which is called a *reorganization*. Recall the example from Figure 1, where the main chain contained blocks A, B, C and D. It is possible that two new blocks are added to the fork containing block E. This situation is shown in Figure 2. The chain with blocks E, F and G is now longer than the main chain. Since we always follow the longest chain, we must switch to the chain with blocks E, F and G, which becomes the main chain now. We refer to this as a main chain reorganization.

In such a reorganization, the blocks on the current main chain need to be disconnected (blocks C and D in our example), and the blocks on the (longer) fork need to be connected (and thus processed) as the new main chain (blocks E, F and G in our example). The chain UTXO set needs to be updated in such a situation, undoing all the transactions in blocks C and D, and processing all transactions in the connected blocks E, F and G. The transactions in the blocks C and D must be added to the transaction pool and the pool UTXO set must be updated with these newly added transactions, and with the transactions that were removed from the pool by disconnecting blocks C and D.

Notice that a transaction that was originally included in a block on the main chain might become an orphan now. If another transaction in a block on the new main chain references the same output, the transaction in the original block is now a double spending transaction and is ignored.

2.9 The 51% attack

A well-known attack on Bitcoin is the so-called ‘51% attack’. Suppose an attacker ‘Eve’ has more than half of all hashing power in the network, hence the name ‘51% attack’. This means Eve is able to mine blocks faster than the rest of the network. Obviously, Eve need not be one individual but can be a group of attackers. In this scenario, Eve is able to double spend her outputs on purpose. Also, she can invalidate some transactions included in blocks on the current chain.

To illustrate this attack, we will again use the situation as depicted in Figure 1. The main chain contains

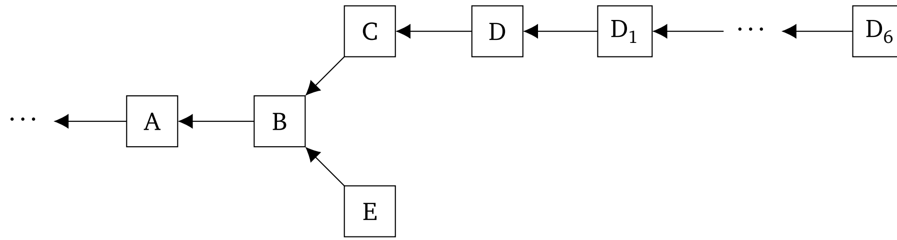

Figure 3: Six confirmations for block D.

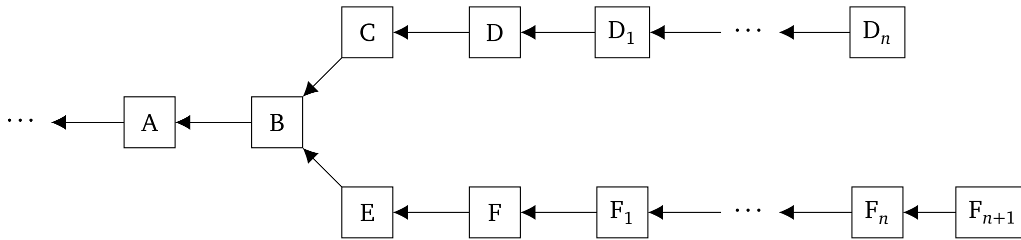

Figure 4: Main chain switch after 51% attack.

blocks A, B, C and D. Under normal circumstances, blocks will be mined on top of the main chain and therefore the next block will reference block D as parent. Suppose, however, Eve starts mining on block E instead. Since she owns the majority of the hashing power in the network, she will mine faster than the rest of the network. She will eventually catch up with the main chain and cause a main chain reorganization in the network.

Now suppose block A contained some transaction that has an output referencing one of Eve's addresses. That is, Eve has some output, say O_1, she can spend. Now, suppose Eve buys coffee from Bob by creating a transaction T_1 that spends O_1, i.e. T_1 contains an input referencing the output O_1 and T_1 contains some output to Bob's address. Suppose D was the block that contains this transaction T_1. Bob wants to make sure the transaction stays on the main chain and waits for the recommended six confirmations before giving Eve her coffee. That is, Bob waits for six blocks to be mined on top of block D. Let these blocks be named $D_1, \ldots, D_6$, resulting in Figure 3.

Eve will now try to spend O_1 again, by creating a new transaction T_2 with an input referencing O_1. For example, she can create T_2 by having one input referencing O_1 and creating an output O_2 to some other address she owns. Now, she creates a block F that contains T_2 and references block E as its parent. She mines and finds a valid nonce for block F. Eve now continues to mine on top of F until she catches up with the main chain in the network and causes a chain reorganization. Since she owns the majority of the hashing power in the network, this will eventually happen. Suppose at this point block D has n blocks mined on top of it, i.e. D has n confirmations. This means that Eve has mined $n + 1$ blocks on top of F, as depicted in Figure 4.

When reorganization happens, blocks C, D, $D_1, \ldots, D_n$ are reverted and blocks E, F, $F_1, \ldots, F_{n+1}$ are connected. This means that transaction T_1 (contained in block D) is invalidated, as T_2 is now contained in the main chain and both transactions now contain an input that references output O_1. Since D was reverted, the output of T_1 is removed from the UTXO set and Bob loses his unspent output, i.e. he loses his coins earned for giving Eve her coffee. Meanwhile, Eve has an output O_2 she can spend again. She has successfully double spent output O_1.

As a side effect, since blocks C, D, $D_1, \ldots, D_n$ are reverted, all the outputs in the coinbase transactions of these blocks are also removed from the UTXO set. This means that every miner in the network that mined one of these blocks loses these mining rewards and fees as well. It should be avoided that one entity has more than half of all the hashing power in the network.

3 Elliptic curve cryptography

3.1 Introduction

In Bitcoin, it is important that the data of a transaction is protected and its integrity can be verified. An asymmetric cryptographic system is used, which accomplishes two important security features: *authentication*, which verifies the identity of the sender of the message, and *encryption*, where a secret message can be sent over an untrusted network. The latter is not applicable in the Bitcoin network, but is an important feature of asymmetric cryptography nonetheless.

Asymmetric cryptography uses a pair of keys: a *private* and a corresponding *public* key. The private key is to remain secret, while the public key can be shared with other (untrusted) parties. When a message is encrypted with the public key, only the holder of the private key can then decrypt the message. Moreover, the same system can be applied for authentication. Messages that are signed with the private key can be verified with the corresponding public key, thus providing a way of authentication.

Since the private key is to remain secret, it is important that it is not possible to construct the private key from the public key. At least it should be sufficiently hard to do so. Hence, at the basis of asymmetric cryptography systems lies a so-called *trapdoor* function that is easy to compute in one direction, but difficult to compute in the other direction.

3.2 Discrete logarithm problem

A well-known and widely used method to compute public keys is based on the *discrete logarithm problem*. The public key used here is interpreted as an element of a group G with multiplication. Given a randomly generated private key, the positive integer k, we determine public key K with

$$K = g^k,$$

where $g \in G$ is a generator element of G.

Now, given the public key K and generator element g for G, in some groups G it is considered computationally infeasible to determine the private key $k = \log_g K$. The fastest known algorithms for computing the discrete logarithm run in sub-exponential time, and it is unknown whether a polynomial-time algorithm exists. This makes this method of creating key pairs useful for cryptographic purposes.

Usually, for group G the finite field $\mathbb{F}_p = \{0, \dots, p-1\}$ is chosen, where p is a large prime number. However, choosing a group where the elements are defined over an elliptic curve ensures that the discrete logarithm problem becomes even harder to solve since no sub-exponential time algorithms are known for elliptic curve fields.

3.3 Elliptic curves

We first define the elliptic curve group formally before we describe how we can use this group for cryptographic purposes.

Definition 3.1 (Elliptic curve)**.** An *elliptic curve* E over a field F, denoted by $E(F)$, is the set of all points $(x, y) \in F^2$ that satisfy the equation

$$E\colon y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1}$$

where $a_1, \dots, a_6 \in F$, together with the *point at infinity* $\mathcal{O}$.

When we place some restrictions on field F, we are able to simplify the elliptic curve equation considerably [44]. We assume that the characteristic of field F is not equal to 2 or 3. This is for instance the case when $F = \mathbb{F}_p$ for $p > 3$ prime, or when $F = \mathbb{R}$. This assumption is acceptable for our purposes of using elliptic curves for cryptography. Then, the change of variables

$$(x, y) \mapsto \left(x, y - \frac{a_1}{2}x - \frac{a_3}{2}\right)$$

is admissible, because 2 has an inverse in F. This leads to the simplified formula

$$E\colon y^2 = x^3 + b_2x^2 + b_4x + b_6$$

where

$$b_2 = a_2 + \frac{a_1^2}{4}, \quad b_4 = a_4 + \frac{a_1 a_3}{2}, \quad b_6 = a_6 + \frac{a_3^2}{4}.$$

Now, apply the transformation

$$(x, y) \mapsto \left(\frac{x - 12b_2}{36}, \frac{y}{216} \right)$$

which is admissible because $36 = 2^2 \cdot 3^2$ and $216 = 2^3 \cdot 3^3$ have an inverse in F (because $\mathrm{char}(F) \neq 2, 3$). This gives

$$E\colon y^2 = x^3 + ax + b, \quad a, b \in F, \tag{2}$$

where

$$a = 216^2 \cdot \left(\frac{b_4}{36} - \frac{b_2^2}{108} \right), \quad b = 216^2 \cdot \left(\frac{2b_2^3}{27} - \frac{b_2 b_4}{3} + b_6 \right)$$

according to the transformation.

Elliptic curves of the form of Equation 2 are said to be in *short Weierstrass form*. From now on, we will assume that $\mathrm{char}(F) \neq 2, 3$ and only consider short Weirstrass equations for elliptic curves. We define the following quantity.

Definition 3.2 (Discriminant)**.** The *discriminant* D of an elliptic curve E in short Weierstrass form is defined as

$$D = -16(4a^3 + 27b^2). \tag{3}$$

When the discriminant of an elliptic curve E is non-zero, the curve is *non-singular*. This guarantees that there are no points on E at which the curve has multiple distinct tangent lines [32].

From now on, we will only consider elliptic curves with non-zero discriminant. This allows us to define an arithmetic over the curve, which in fact forms a group structure.

3.3.1 Group structure

We can define an group structure over elliptic curves, which will give us the ability to do algebraic manipulations with the points defined on the curve.

For this, we first define a special *point at infinity*, denoted by $\mathcal{O}$, that acts as the identity element in the group. We use $E(F)$ to denote the set of all points on the elliptic curve, together with the identity element $\mathcal{O}$.

The *addition* of two points $P, Q \in E(F)$ is based on the following geometric interpretation over the reals. When $P \neq Q$, draw a straight line through these points, which intersects the curve at a third point. Reflect this point in the x-axis to obtain $R = P + Q$.

When $P = Q$, the tangent line to the curve in P is used instead. The point at which the tangent line (which is well-defined, because the discriminant is non-zero) intersects the curve again is reflected in the x-axis to obtain $R = 2P$, the *double* of P.

In this geometric interpretation, $\mathcal{O}$ is a point with an infinite x and y coordinates. Then, adding $\mathcal{O}$ to $P = (x, y)$ would require drawing a vertical line through P, which intersects the curve at $-P = (x, -y)$, the *negative* or *inverse* of P. These operations are displayed in Figure 5.

Algebraically, we find explicit formulas for these operations as follows [56].

Point inverse

Since the inverse $-P$ of a point $P = (x_1, y_1)$ on the curve is the reflection of the point in the horizontal axis, we have that

$$-P = (x_1, -y_1).$$

We have that $P - P = \mathcal{O}$ and $-\mathcal{O} = \mathcal{O}$.

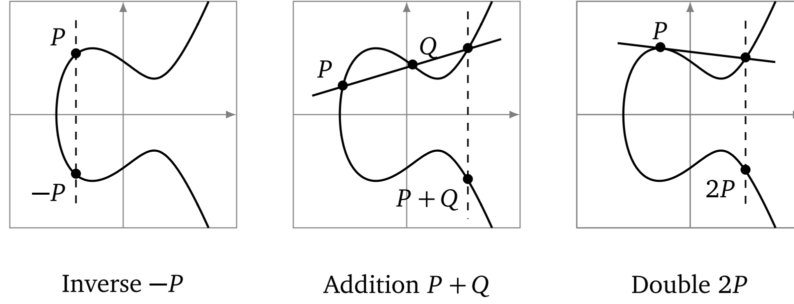

Figure 5: Geometric interpretation of elliptic curve point operations. Adapted from TikZ for Cryptographers [36].

Point addition
To find $P+Q=(x_3,y_3)$ for two points on the curve $P=(x_1,y_1)$, $Q=(x_2,y_2)$ where $P\neq\pm Q$, we first look at the equation of the straight line ℓ through P and Q given by

$$\ell\colon y=\lambda(x-x_1)+y_1$$

where

$$\lambda=\frac{y_2-y_1}{x_2-x_1}.$$

Substituting ℓ in the equation for E (Equation 2 in short Weierstrass form) gives

$$(\lambda(x-x_1)+y_1)^2=x^3+ax+b,$$

or, in expanded form,

$$x^3-\lambda^2x^2+(a+2\lambda x_1-2\lambda y_1)x+b-\lambda^2x_1^2+2\lambda x_1y_1-y_1^2=0. \tag{4}$$

Note that, since Equation 4 is the substitution of ℓ in E, it has exactly the solutions x_1,x_2,x_3. This means that Equation 4 can also be written as

$$(x-x_1)(x-x_2)(x-x_3)=0.$$

We thus have

$$(x-x_1)(x-x_2)(x-x_3)=x^3-\lambda^2x^2+(a+2\lambda x_1-2\lambda y_1)x+b-\lambda^2x_1^2+2\lambda x_1y_1-y_1^2.$$

We now compare the coefficient of x^2, which must be equal on both sides of the equation. Note that from the right hand side we obtain the coefficient $-\lambda^2$ and from the left hand side (after expansion of the terms) the coefficient $-(x_1+x_2+x_3)$. This gives

$$x_1+x_2+x_3=\lambda^2$$

which yields a formula for x_3. Substituting the result in the equation of ℓ gives a formula for y_3. Writing out the explicit formulas gives

$$x_3=\lambda^2-x_1-x_2=\left(\frac{y_2-y_1}{x_2-x_1}\right)^2-x_1-x_2, \tag{5}$$

$$y_3=-(\lambda(x_3-x_1)+y_1)=\left(\frac{y_2-y_1}{x_2-x_1}\right)(x_1-x_3)-y_1. \tag{6}$$

Note that the equation for y_3 is ‘inverted’ to accommodate for the reflection in the horizontal axis.

Point doubling
Finding $2P=(x_3,y_3)$ is similar to point addition, only that now we use the tangent ℓ in P on the curve E, which gives (using partial differentiation of the equation for E)

$$\lambda=\frac{3x_1^2+a}{2y_1}$$

for

$$\ell : y = \lambda(x - x_1) + y_1.$$

The substituted equation of ℓ in E has exactly two distinct solutions, x_1 and x_3, because the tangent line contains the points P and $2P$. In this equation, x_1 has multiplicity 2. Hence, similar to the derivation of point addition above, we obtain

$$(x - x_1)^2(x - x_3) = x^3 - \lambda^2 x^2 + (a + 2\lambda^2 x_1 - 2\lambda y_1)x + b - \lambda^2 x_1^2 + 2\lambda x_1 y_1 - y_1^2$$

where again the coefficient of x^2 yields an expression for x_3. Analogous to the derivation of point addition, we get the explicit formulas

$$x_3 = \lambda^2 - 2x_1 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1, \tag{7}$$

$$y_3 = -(\lambda(x_3 - x_1) + y_1) = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1. \tag{8}$$

These operations together form an algebraic group for an elliptic curve E over the field F which is denoted by $[E(F), +, \mathcal{O}, -]$.

3.4 Discrete logarithm problem for elliptic curves

Similar to the discrete logarithm problem for elements from finite fields $\mathbb{F}_p$, as described in Section 3.2, we can define the discrete logarithm problem for elements of $E(\mathbb{F}_p)$ for an elliptic curve E.

Let E be an elliptic curve defined over $\mathbb{F}_p$. Let G be a point on the curve which generates the cyclic group $\langle G\rangle = \{\mathcal{O}, G, 2G, 3G, \dots, (n-1)G\}$ of order n. Given a multiple P of G, the *elliptic curve discrete logarithm problem* (ECDLP) [32] is to find $k \in [1, n-1]$ such that

$$kG = P.$$

The ECDLP gives us a way of computing public-private key pairs. First, choose a random private key $k \in [1, n-1]$. The corresponding public key is $K = kG$. When using elliptic curve cryptography, we only consider the points in $\langle G\rangle$.

The difficulty of ECDLP is directly related to n, the order of the cyclic subgroup generated by the chosen *generator point* $G \in E(\mathbb{F}_p)$. The larger n, the larger the number of possible values for k. Therefore, when we want to apply the ECDLP in an elliptic curve cryptographic system, we need to make sure n is large. We thus need to choose an appropriate curve and generator point that satisfy this condition.

Note that n is bounded by the total number of points on the curve: $n \leq \#E(\mathbb{F}_p)$. For n to be large, it is a necessary condition that we choose a curve where $\#E(\mathbb{F}_p)$ is large. Hasse's theorem provides a bound on the number of points $\#E(\mathbb{F}_p)$ [32].

Theorem 3.1 (Hasse)**.** *Let E be an elliptic curve defined over $\mathbb{F}_p$. Then*

$$p + 1 - 2\sqrt{p} \leq \#E(\mathbb{F}_p) \leq p + 1 + 2\sqrt{p}.$$

Hasse's theorem guarantees that if we choose p to be large, $\#E(\mathbb{F}_p)$ will be close to p and thus large as well. It is therefore a necessary condition for the security of ECDLP that p is chosen to be large. Usually, the elliptic curve and generator point are chosen such that $n \approx p$ [32].

The ECDLP is, like the regular discrete logarithm problem, computationally intractable when a large order n is used [32]. In general, the fastest known algorithm for solving the ECDLP is *Pollard's rho algorithm*, with an expected running time of

$$\mathcal{O}\left(\frac{\sqrt{\pi n}}{2}\right),$$

which is exponential in the number of bits of n. For the DLP, other algorithms exist, making use of index calculus, that have a sub-exponential expected running time. The structure needed for these algorithms

to work on the ECDLP is not present (in general) in the elliptic curve group, resulting in that, to date, no sub-exponential algorithms are known for solving the ECDLP [32].

Under the assumption that these algorithms mentioned above are indeed the fastest for solving the DLP and ECDLP, we can achieve the same level of security with elliptic curve cryptography with shorter keys than the key sizes necessary in public key cryptography based on DLP. Smaller key sizes mean that key operations can be executed faster, allowing for a more efficient cryptography system. In addition, the size of digital signatures can be greatly reduced. This is of particular importance for Bitcoin, as signatures are stored as part of a transaction and are stored on the blockchain. See Section 5.1.4 and Section 5.1.6 for more details on digital signatures and their usage in Bitcoin.

3.5 The Bitcoin curve

The Bitcoin protocol applies elliptic curve cryptography in its digital signature scheme ECDSA (see Section 5.1.4). In selecting the specific curve parameters, usually one must find a balance between security and efficiency. The specific elliptic curve used in Bitcoin is *secp256k1* as defined in the SEC 2 standard [21].

The curve $E : y^2 = x^3 + ax + b$ is defined over $\mathbb{F}_p$ where

$$\begin{aligned}
a &= 0, \\
b &= 7, \\
p &= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \\
&= \texttt{ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff fffffffe fffffc2f}.
\end{aligned}$$

Hence, the curve equation becomes $y^2 = x^3 + 7$. The generator point G, with coordinates in hexadecimal format, is given by

$$\begin{aligned}
G = (&\texttt{79be667e f9dcbbac 55a06295 ce870b07 029bfcdb 2dce28d9 59f2815b 16f81798}, \\
&\texttt{483ada77 26a3c465 5da4fbfc 0e1108a8 fd17b448 a6855419 9c47d08f fb10d4b8})
\end{aligned}$$

which has (prime) order

$$n = \texttt{ffffffff ffffffff ffffffff fffffffe baaedce6 af48a03b bfd25e8c d0364141}.$$

One might notice that $n \approx p$ which indicates that G is an appropriate generator point (as n is large). In fact, we know that $n = \#E(\mathbb{F}_p)$. In other words, The cyclic subgroup $\langle G \rangle$ contains all points on the curve $E(\mathbb{F}_p)$. This property is usually expressed in terms of the *cofactor*.

Definition 3.3 (Cofactor). Let E be an elliptic curve defined over $\mathbb{F}_p$. Let $\#E(\mathbb{F}_p)$ denote the total number of points on the curve and $G \in E(\mathbb{F}_p)$ a point generating a cyclic subgroup $\langle G \rangle$ of order n. The *cofactor* of E with generator point G is

$$h = \frac{\#E(\mathbb{F}_p)}{n}.$$

For *secp256k1* we thus have cofactor $h = 1$. It is not necessary that the curve has cofactor 1 for cryptographic purposes, but it is a convenient property. It ensures that every point that satisfies the curve equation indeed is in $\langle G \rangle$ without the need to check this explicitly. When, for example, a public key point K is given from an untrusted source, we merely need to verify whether K satisfies the curve equation when $h = 1$. This simplifies the implementation of cryptographic protocols since no additional checks are needed if the computed points are indeed in the proper subgroup.

Before being used in Bitcoin, the *secp256k1* curve was relatively little used in other applications [13]. A number of properties of this curve are known that favor using it, which are mainly related to efficiency of computations on the points of the curve. At the time Bitcoin was designed, the *secp256k1* curve was one of the most efficient curves available. It is also suspected that this curve was chosen because its parameters were chosen deterministically, instead of choosing a slightly more efficient curve with randomly generated parameters. The suspected reasoning behind this choice is distrust in the organisations that constructed these curves [13]. It is more likely that a curve with seemingly randomly

generated parameters was specifically designed to include 'backdoors' unknown to the general public. When the curve parameters are small and deterministically chosen, this risk is much smaller.

Currently, more efficient and secure curves have been introduced that outperform the *secp256k1* curve. These curves were not available at the time Bitcoin was designed. However, when implemented properly, *secp256k1* remains a secure choice [43].

3.6 Efficient implementation

In the Bitcoin network, a major performance bottleneck operation is signature verification, which is a necessary step when verifying incoming blocks and transactions. For more information on signature verification, see Section 5. Since Bitcoin is a time-sensitive protocol, it is desirable that this step is as efficient as possible, to allow fast synchronization of the blockchain for every node in the network, and fast propagation of new blocks and transactions.

The main elliptic curve point operation used in signature creation and verification is multiplication of a point $P \in E(\mathbb{F}_p)$ with a scalar $k \in [1, n-1]$, denoted by kP. In this section, we will look at how scalar multiplication can be implemented efficiently. Some of these methods are specifically applicable to the *secp256k1* curve. In the original Bitcoin implementation, the *OpenSSL* library was used for elliptic curve cryptography operations. In the more recent versions, the Bitcoin Core implementation switched to their own *libsecp256k1* library, which is highly optimized for the *secp256k1* curve [8]. The *libsecp256k1* library uses several of the methods described in this section.

3.6.1 Projective coordinates

The formulas for point addition and point doubling as presented in Section 3.3.1 require a field inversion operation. Since inversion in $\mathbb{F}_p$ is relatively expensive compared to multiplication, the computation of point addition and doubling might be improved significantly when we represent points on the elliptic curve in *projective coordinates*.

Until now, we have expressed elliptic curve points in *affine coordinates*. An *affine point* is of the form $(x, y) \in \mathbb{F}_p^2$. We can however also define a different coordinate system in which we express the points on the curve. Points in the projective coordinate system are represented as $(X : Y : Z)$ where $X, Y, Z \in \mathbb{F}_p$. After formalizing these projective points, we will see that computations on elliptic curve points can be implemented more efficiently when they are expressed in projective coordinates. We will also define a bijective mapping between affine points and projective points. We can use this mapping to first translate the affine points to projective coordinates, do the computations in the projective coordinate system, and finally translate back to affine coordinates.

We first define an equivalence relation on $\mathbb{F}_p^3 \setminus \{(0,0,0)\}$ by

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2) \iff X_2 \equiv \lambda^c X_1,\ Y_2 \equiv \lambda^d Y_1,\ Z_2 \equiv \lambda Z_1 \mod p$$

for some $\lambda \in \mathbb{F}_p^*$ and fixed positive integers c, d. This defines the equivalence class

$$(X : Y : Z) = \{(\lambda^c X, \lambda^d Y, \lambda Z) \mid \lambda \in \mathbb{F}_p^*\}.$$

We call $(X : Y : Z)$ a *projective point*. If $Z \neq 0$, we can write $(X : Y : Z) = (X/Z^c : Y/Z^d : 1)$. Note that $(X/Z^c, Y/Z^d, 1)$ is the only representative of $(X : Y : Z)$ with $Z = 1$. Using this representative (and the fact that it is unique under the condition $Z = 1$), we define a bijective relation between the set of projective points

$$\mathbb{P}(\mathbb{F}_p)^* = \{(X : Y : Z) \mid X, Y, Z \in \mathbb{F}_p, Z \neq 0\}$$

and the set of *affine points*

$$\mathbb{A}(\mathbb{F}_p) = \{(x, y) \mid x, y \in \mathbb{F}_p\},$$

where transformations between the coordinate systems are as follows:

- The affine point $(x, y) \in \mathbb{A}(\mathbb{F}_p)$ corresponds to the projective point $(x : y : 1)$.

- The projective point $(X : Y : Z) = (X/Z^c : Y/Z^d : 1) \in \mathbb{P}(\mathbb{F}_p)^*$ corresponds to the affine point $(X/Z^c, Y/Z^d)$.

Because of this bijective mapping, we can choose to represent points on an elliptic curve either in affine or projective coordinates. For appropriate values for the fixed c and d, we will see that if we implement the elliptic curve group operations in projective coordinates we can speed up the computation considerably.

Jacobian coordinates
When we choose $c = 2$ and $d = 3$, the projective coordinates are called *Jacobian coordinates* [32]. The projective point $(X : Y : Z)$ with $Z \neq 0$ now corresponds to the affine point $(X/Z^2, Y/Z^3)$. The projective Jacobian form of the *secp256k1* curve is given by

$$E\colon Y^2 = X^3 + 7Z^6$$

and point at infinity $\mathcal{O}$ corresponds to $(1 : 1 : 0)$.

Point doubling in Jacobian coordinates
We will now express the point doubling in terms of Jacobian coordinates. We will do this by transforming the formulas for affine points, given in Section 3.3.1, to Jacobian coordinates by the bijective mapping given above. We consider the formulas specifically for the *secp256k1* curve, so we assume $a = 0$ and $b = 7$.

Let $P = (X_1 : Y_1 : Z_1) \in E$, $P \neq \mathcal{O}$ be a point on the curve. For point doubling, we first use the affine expression in Equations 7 and 8. Since $P = (X_1/Z_1^2 : Y_1/Z_1^3 : 1)$, P can be expressed in affine coordinates as $P = (X_1/Z_1^2, Y_1/Z_1^3)$. We compute $2P = (X_3', Y_3')$ in affine coordinates with

$$X_3' = \left(\frac{3\frac{X_1^2}{Z_1^4}}{2\frac{Y_1}{Z_1^3}}\right)^2 - 2\frac{X_1}{Z_1^2}$$

and

$$Y_3' = \left(\frac{3\frac{X_1^2}{Z_1^4}}{2\frac{Y_1}{Z_1^3}}\right)\left(\frac{X_1}{Z_1^2} - X_3'\right) - \frac{Y_1}{Z_1^3}.$$

Note that $2P$ can be expressed in Jacobian coordinates as $2P = (X_3' : Y_3' : 1) = (X_3 : Y_3 : Z_3)$, where

$$X_3 = \lambda^2 X_3', \quad Y_3 = \lambda^3 Y_3', \quad Z_3 = \lambda$$

for a certain $\lambda \in \mathbb{F}_p^*$. We choose $\lambda = 2Y_1Z_1$. This results in the following equations for X_3, Y_3 and Z_3 [32]:

$$\begin{cases} X_3 = (3X_1^2)^2 - 8X_1Y_1^2 \\ Y_3 = 3X_1^2(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ Z_3 = 2Y_1Z_1. \end{cases} \tag{9}$$

With these formulas, we have defined point doubling for points represented in Jacobian coordinates. Note that the resulting equations do not require a field inversion and can be implemented by using only squaring, multiplication and addition of elements in $\mathbb{F}_p$. These operations are much cheaper to implement than a field inversion, making point doubling in Jacobian coordinates more efficient than in affine coordinates [32].

When we allow storing some intermediate results, the formulas above can be implemented using only

5 field squarings, 2 field multiplications and 6 field additions with the explicit formulas [5]:

$$
\begin{aligned}
&A = X_1^2\\
&B = Y_1^2\\
&C = B^2\\
&D = 2((X_1 + B)^2 - A - C)\\
&E = 3A\\
&F = E^2\\
&X_3 = F - 2D\\
&Y_3 = E(D - X_3) - 8C\\
&Z_3 = 2Y_1 Z_1.
\end{aligned}
$$

In the Bitcoin implementation, a slightly different algorithm is used with 4 squarings, 3 multiplications and 12 addition operations. As explained in the *libsecp256k1* source code, this approach is faster in practice (as determined experimentally) [8].

Point addition in Jacobian coordinates

Let $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : Z_2)$ be two points on the curve E with $P \neq \pm Q$. With the same approach as above, we can find explicit formulas for $P + Q = (X_3 : Y_3 : Z_3)$ that do not include field inversions by choosing an appropriate expression for λ. In the Bitcoin implementation, the chosen algorithm requires 12 multiplications, 4 squarings and 12 field addition operations [8].

However, a faster procedure exists when P is given in Jacobian coordinates and Q in affine coordinates. Then, we can compute the result $P+Q$, in Jacobian coordinates, with 8 multiplications, 3 squarings and 12 field additions. This is called *mixed addition* and is used in the Bitcoin implementation wherever it is possible to do so.

Costly field inversion operations can thus be prevented when working with projective coordinates, allowing for faster computations. Converting back to affine coordinates does require two additional squarings, a multiplication and two inversions, in order to compute $(X/Z^2, Y/Z^3)$ from $(X : Y : Z)$. However, this conversion can be delayed until all computations in Jacobian coordinates are finished.

3.6.2 Scalar point multiplication

There are multiple ways of computing the multiple $Q = kP$ of an elliptic curve point P. The most obvious method is naive repeated addition

$$Q = kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}.$$

In many cases in elliptic curve cryptography, k is large. This is for example the case when k is the private key an Q is the corresponding public key. Performing repeated addition with such a large k becomes computationally intractable since it scales exponentially in the number of bits of k. It requires even more steps than solving the elliptic curve discrete logarithm problem directly using the best-known algorithms.

It is therefore necessary to use more efficient methods of computing scalar multiples.

Double-and-add method

To compute the multiple $Q = kP$, we can use the *double-and-add method* [56]. We call P the *base point* and k the *scalar*. First, we represent k in binary

$$k = k_0 + k_1 \cdot 2 + k_2 \cdot 2^2 + \cdots + k_m \cdot 2^m$$

where $k_0, \ldots, k_m \in \{0, 1\}$. When we multiply both sides with P we see that

$$
\begin{aligned}
kP &= k_0 P + k_1 \cdot (2P) + k_2 \cdot (2^2 P) + \cdots + k_m \cdot (2^m P)\\
&= k_0 P + 2 \cdot \Big(k_1 P + 2 \cdot \big(k_2 P + \cdots + 2(k_{m-1} P + 2 k_m P) \cdots\big)\Big)
\end{aligned}
$$

This expression is the idea behind the double-and-add method. The iterative algorithm is given in Algorithm 1. The algorithm requires only $m = \lceil \log_2 k \rceil$ doublings (the number of bits of k) and $b \leq m$ additions, where b is the number of 1-bits in k. This makes it a suitable way of computing point multiples for large k. Note that the algorithm as presented here traverses the bits of k left-to-right.

Algorithm 1 Double-and-add method for elliptic curve point multiplication.

$Q \leftarrow \mathcal{O}$
for $i = m - 1$ **down to** 0 **:**
  $Q \leftarrow 2Q$
  if $k_i = 1$ **:**
    $Q \leftarrow Q + P$
return Q

We can even speed this up more if we allow precomputation of some fixed values and store these in a lookup table. When using elliptic curve cryptography, the base point P is often the fixed generator G. When P is fixed, we can precompute and store a table with $2P, 4P, 8P, \ldots, 2^m P$. We then use the modified Algorithm 2, which eliminates computing doublings entirely and only performs at most $m+1$ additions. Note that when P is not fixed, Algorithm 2 has a similar running time as the procedure in Algorithm 1, as a table with precomputed multiples $2^i P$ for $2 \leq i \leq m$ must first be computed. Also note that we traverse the bits of k right-to-left.

Algorithm 2 Modified double-and-add method for elliptic curve point multiplication.

$Q \leftarrow \mathcal{O}$
for $i = 0$ **to** m **:**
  if $k_i = 1$ **:**
    $Q \leftarrow Q + 2^i P$
return Q

Windowed double-and-add

The running time of double-and-add is bounded by m, the length of the binary representation of k. However, we can also use a different, higher base for the representation of k to reduce the size of m, the length of the higher-base representation of k. Choosing the base 2^w as a power of 2 has the advantage that we can still use the binary representation of k as we 'chop up' the binary representation into m windows of length w. To use the increased window size, we precompute a table

$$T = (\mathcal{O}, P, 2P, 3P, \ldots, (2^w - 1)P).$$

Windowed double-and-add as given in Algorithm 3 still requires $\lceil \log_2 k \rceil$ doublings. However, it requires at most only $\lceil \lceil \log_2 k \rceil / w \rceil$ additions. In general, a larger window size reduces the number of additions, but increases the precomputation table size exponentially.

Algorithm 3 Windowed double-and-add method for elliptic curve point multiplication.

$Q \leftarrow \mathcal{O}$
for $i = m - 1$ **down to** 0 **:**
  for $i = 1$ **to** w **:**
    $Q \leftarrow 2Q$
  $Q \leftarrow Q + T[k_i]$
return Q

As before, when P is fixed, we can eliminate all point doublings by precomputing and storing the tables

$$T_i = (\mathcal{O}, 2^{iw} P, 2 \cdot 2^{iw} P, 3 \cdot 2^{iw} P, \ldots, (2^w - 1) \cdot 2^{iw} P)$$

for $i = 0, 1, 2, 3, \ldots, m-1$. The procedure for performing scalar multiplication with these precomputed tables is given in Algorithm 4.

When signing a message in ECDSA, we compute a multiple zG of the fixed generator point G. Hence, the Bitcoin implementation uses Algorithm 4 with precomputed multiples of G to compute zG for signing. The window size is 4 bits [8].

Algorithm 4 Modified windowed double-and-add method for elliptic curve point multiplication.

```
Q ← 𝒪
for i = 0 to m − 1 :
    Q ← Q + T_i[k_i]
return Q
```

Non-adjacent form

We have explored representing the scalar k in different bases in order to speed up scalar multiplication. Note that on Weierstrass curves negation of points is very efficient: $-(x, y) = (x, -y)$, which means the computational cost of subtraction of points is almost equal to addition. The idea is to use a signed representation of k. The *non-adjacent form* (NAF) is such a signed representation. The NAF of an integer k is defined as the sequence $(k_0, k_1, \dots, k_m)$ of digits $k_i \in \{-1, 0, 1\}$, $k_m \neq 0$ where

$$k = k_0 + k_1 \cdot 2 + k_2 \cdot 2^2 + \cdots + k_m \cdot 2^m$$

and for all $0 \leq i < m$ we have

$$k_i \cdot k_{i+1} = 0.$$

The last condition ensures non-zero digits cannot be adjacent in the NAF representation. We will see that this condition gives the NAF representation a number of properties that make it especially useful for computing scalar multiples of elliptic curve points. We denote the NAF of an integer k as NAF(k).

The NAF representation of k can be efficiently computed by performing repeated division of k by 2. If k is odd, we can write k as

$$k = q \cdot 2 + r, \quad q \in \mathbb{N},\ r \in \{-1, 1\}.$$

We choose r such that the quotient q is even. This ensures that at the next division we get remainder 0 and that in the resulting NAF representation no digits ± 1 are adjacent. The procedure is described in Algorithm 5. Note that in the algorithm we choose the remainder $r = 2 - (k \bmod 4)$, which indeed leads to an even quotient q.

Lemma 3.1. *If k is an odd integer and $r = 2 - (k \bmod 4)$, then $q = (k - r)/2$ is even.*

Proof. Let $k = 2q + r$ be an odd integer and $r = 2 - (k \bmod 4)$. Then

$$q \cdot 2 + 2 - (k \bmod 4) = k = (k \bmod 4) + n \cdot 4$$

for a certain integer n. We obtain for q

$$\begin{aligned} q \cdot 2 &= 2 \cdot (k \bmod 4) + n \cdot 4 - 2 \\ q &= (k \bmod 4) - 1 + 2n \end{aligned}$$

Since k is odd, $k \bmod 4$ is also odd and $(k \bmod 4) - 1$ is even. Hence, q is even. □ Q. E. D.

Algorithm 5 Computing the NAF of an integer.

```
i ← 0
while k > 0 :
    if k is even :
        k_i ← 0
    else:
        k_i ← 2 − (k mod 4)
        k ← k − k_i
    k ← k/2
    i ← i + 1
return (k_{i−1}, k_{i−2}, ..., k_0)
```

The NAF representation has a number of useful properties [15]:

- The Hamming weight, the number of non-zero digits, of the NAF representation is minimal compared to other (signed) binary representations. The average number of non-zero digits in NAF(k) of length $m+1$ is approximately $(m+1)/3$.
- If the binary representation of k requires m bits, the NAF representation requires at most $m+1$ bits.

Together with the efficiency of elliptic curve point negation, these two properties make NAF(k) suitable to use for scalar multiplication of elliptic curve points. The procedure is similar to the double-and-add method and is described in Algorithm 6.

Algorithm 6 NAF method for elliptic curve point multiplication.

```
Q ← 𝒪
for i = m down to 0 :
    Q ← 2Q
    if k_i = 1 :
        Q ← Q + P
    if k_i = −1 :
        Q ← Q − P
return Q
```

There also exists a windowed version of the NAF, which is defined as follows [32].

Definition 3.4 (w-ary non-adjacent form (w-NAF))**.** Let $w \geq 2$ be an integer. A *w-ary NAF* of an integer k, denoted as NAF$_w(k)$, is a sequence of digits $(k_0, k_1, \ldots, k_m)$ where

1. $k = \sum_{i=0}^{m} k_i 2^i$;
2. Each non-zero digit k_i is odd;
3. $|k_i| < 2^{w-1}$;
4. $k_m \neq 0$;
5. At most one of any w consecutive digits is non-zero.

Note that NAF$_2(k)$ = NAF(k). Again, the w-NAF representation of k has the property that its length requires at most one more digit than the binary representation of k. The amount of non-zero digits in the w-NAF representation of length $m+1$ is approximately $(m+1)/(w+1)$.

The w-NAF of k can be efficiently computed by performing repeated division of k by 2:

$$k = q \cdot 2 + r, \quad q \in \mathbb{N},$$

allowing remainders $r \in [-2^{w-1}, 2^{w-1} - 1]$. To satisfy property 5 in the definition of w-NAF, we choose r such that q is divisible by 2^{w-1}, ensuring that the next $w-1$ digits are zero. We first show that if k is odd, the *symmetric modulo* of k, defined as

$$r = k \operatorname{mods} 2^w := \begin{cases} (k \bmod 2^w) & \text{if } k \bmod 2^w < 2^{w-1} \\ (k \bmod 2^w) - 2^w & \text{if } k \bmod 2^w \geq 2^{w-1} \end{cases}$$

is a suitable choice.

Lemma 3.2. *If k is an odd integer and $r = k \operatorname{mods} 2^w$, then $q = (k-r)/2$ is divisible by 2^{w-1}.*

Proof. Let k be an odd integer and $r = k \operatorname{mods} 2^w$. Suppose $k \bmod 2^w \geq 2^{w-1}$. Then, by the definition

of symmetric modulo,

$$\begin{aligned} q &\equiv \frac{k-r}{2} \\ &\equiv \frac{k-(k \bmod s\, 2^w)}{2} \\ &\equiv \frac{k-(k \bmod 2^w)+2^w}{2} \\ &\equiv \frac{(k \bmod 2^w)+n\cdot 2^w-(k \bmod 2^w)+2^w}{2} \\ &\equiv (n+1)2^{w-1} \\ &\equiv 0 \pmod{2^{w-1}}. \end{aligned}$$

Note that if $(k \bmod 2^w) < 2^{w-1}$ an analogous argument also gives $q \equiv 0 \pmod{2^{w-1}}$. Hence, q is divisible by 2^{w-1}. □ Q. E. D.

Computing $\text{NAF}_w(k)$, as a generalization to the procedure for computing $\text{NAF}(k)$, is given in Algorithm 7.

Algorithm 7 Computing the w-NAF of an integer.

```
i ← 0
while k > 0 :
    if k is even :
        k_i ← 0
    else:
        k_i ← k mods 2^w
        k ← k − k_i
    k ← k/2
    i ← i + 1
return (k_{i−1}, k_{i−2}, ..., k_0)
```

To use $\text{NAF}_w(k)$ in elliptic curve scalar multiplication, we again use a table with precomputed values

$$(P_0, P_1, P_3, P_5, \dots, P_{2^{w-1}-1}) = (\mathcal{O}, P, 3P, 5P, \dots, (2^{w-1}-1)P)$$

with only the odd multiples of P, since $\text{NAF}_w(k)$ only contains odd non-zero digits. Note that we also need the negatives of the values in the table. However, since we can compute negatives at almost no cost, we do this 'on the fly' to save storage space. The procedure is described in Algorithm 8.

Algorithm 8 w-NAF method for elliptic curve point multiplication.

```
Q ← 𝒪
for i = m down to 0 :
    Q ← 2Q
    if k_i ≥ 0 :
        Q ← Q + P_{k_i}
    else:
        Q ← Q − P_{k_i}
return Q
```

The Bitcoin implementation uses the w-NAF method for general elliptic curve point multiplication as described in Algorithm 8. The used window size is $w = 5$, which gives an (experimental) optimal trade-off between storage space and computation cost of 128-bit and 256-bit scalars [8]. For the intermediate computations, the points are represented in Jacobian coordinates, such that the more efficient formulas for doubling and addition can be used.

Efficiently-computable endomorphisms
The *secp256k1* has a specific structure which allows to speed up scalar multiplication even more.

The idea is that we utilize endomorphisms $\varphi : E(\mathbb{F}_p) \to E(\mathbb{F}_p)$ defined on the curve. Since the whole curve is generated by the single point G, the subgroup we work on is cyclic. In cyclic groups, every endomorphism can be interpreted as scalar multiplication. Thus, $\varphi(P) = \lambda P$ for a certain $\lambda \pmod{n}$ characteristic for φ. However, there definition of φ may be such that $\varphi(P)$ is much more efficient to compute than directly computing the scalar multiple λP. When this is the case, such an endomorphism may be used to speed up scalar multiplication kP for arbitrary scalars k as well.

Gallant, Lambert and Vanstone introduce the endomorphism φ defined by $(x, y) \mapsto (\beta x, y)$ for elliptic curves of the form

$$E : y^2 = x^3 + b$$

defined over $\mathbb{F}_p$ with $p \equiv 1 \pmod{3}$ prime and $\beta \in \mathbb{F}_p$ an element of order 3 [25]. The *secp256k1* curve satisfies these conditions, since for this curve indeed $p \equiv 1 \pmod{3}$ and $b = 7$. Note that φ is indeed an endomorphism: let $(x, y) \in E(\mathbb{F}_p)$, then

$$(\beta x)^3 + b = \beta^3 x^3 + b = x^3 + b = y^2$$

since β has order 3. Hence $(\beta x, y) \in E(\mathbb{F}_p)$. Also, note that $\varphi(p)$ is efficiently computed with a single multiplication in $\mathbb{F}_p$.

Since the subgroup $\langle G \rangle$ generated by base point G we work on is cyclic and of prime order n, every endomorphism on $\langle G \rangle$ can be interpreted as integer multiplication, hence $\varphi(P) = \lambda P$, where $\lambda \pmod{n}$ is one of the roots of the characteristic polynomial of φ [32]:

$$\lambda^2 + \lambda + 1 \equiv 0 \pmod{n}.$$

The specific values for β and λ as used in Bitcoin are [8]:

$$\lambda = \texttt{5363ad4c c05c30e0 a5261c02 8812645a 122e22ea 20816678 df02967c 1b23bd72},$$
$$\beta = \texttt{7ae96a2b 657c0710 6e64479e ac3434e9 9cf04975 12f58995 c1396c28 719501ee}.$$

Now, the basic idea is as follows. When computing kP for a given $k \in \mathbb{F}_n$, we write

$$k \equiv k_1 + k_2\lambda \pmod{n}$$

where $k_1, k_2 \in [0, \lceil \sqrt{n} \rceil]$. Then

$$\begin{aligned} kP &= (k_1 + k_2\lambda)P \\ &= k_1P + k_2\lambda P \\ &= k_1P + k_2\varphi(P). \end{aligned} \tag{10}$$

We can compute the result of Equation 10 efficiently using *Shamir's trick*: a *simultaneous multiple point multiplication algorithm* which computes $k_1P + k_2Q$ directly (where in this case $Q = \varphi(P)$). The basic idea is to consider k_1 and k_2 together as a vector (k_1, k_2) and perform the w-NAF double-and-add method. Since k_1 and k_2 have approximately half the bitlength of k, roughly half of the point doublings are needed. Determining the values of k_1 and k_2 is described by Hankerson, Menezes and Vanstone [32]. However, in the Bitcoin implementation a slightly more efficient algorithm is used which relies on precomputed estimates as descibed by Gouvea, Oliveira and Lopez [28].

This method is usually referred to as *GLV multiplication* (for Gallant, Lambert and Vanstone) [25]. It has been implemented in the *libsecp256k1* library implemented by the Bitcoin Core developers and gives an experimental speedup in signature verification of about 30 percent [8]. Although currently the endomorphism method is turned off in the Bitcoin Core implementation. Bitcoin Core developer Gregory Maxwell has indicated in developer discussions that this is due to possible patent infringements when enabled. It is however planned to enable this optimization in the future when it becomes clear patent rights are no longer applicable [41, 42].

3.7 Point compression

A public key in elliptic curve cryptography is a point P on the curve $E(F)$. Usually, points on the curve are represented as

$$\texttt{04} \mid\mid x \mid\mid y$$

where x and y are encoded in hexadecimal format. The operator $\mid\mid$ represents concatenation of the representations. The representation is prefixed with `04` to denote that the point is represented in uncompressed format [20].

However, there is a shorter way to encode points on the curve with *point compression*, where a point $P = (x, y)$ is represented as

$$c \mid\mid x$$

where

$$c = (y \bmod 2) + 2 = \begin{cases} \texttt{02} & y \text{ is even} \\ \texttt{03} & y \text{ is odd} \end{cases}$$

and x is represented in hexadecimal format. This shorter representation is useful for storing and transferring points, as less storage space is needed.

The point at infinity $\mathcal{O}$ is represented as `00`.

Completeness
Given a compressed point $c \mid\mid x$ and let $\alpha \equiv x^3 + ax + b \pmod p$. Given that there exists an $y \in \mathbb{F}_p$ for which

$$y^2 \equiv \alpha \pmod p,$$

we show that $b \mid\mid x$ uniquely represents a point on the curve. Note that there are two possible y values corresponding to a (valid) x for every point (x, y) on the curve. When (x, y) is a point on the curve, then $p - y \pmod p$ is also a valid y-coordinate for this x on the curve since

$$(p - y)^2 \equiv p^2 - 2yp + y^2 \equiv \alpha \pmod p.$$

Note that if y is even, then $p - y$ is odd and vice versa, since p is odd. Also note that the equation $y^2 \equiv \alpha \pmod p$ only has at most two solutions. Therefore, $c \mid\mid x$ uniquely represents a point on the curve.

Decompression
To decompress a compressed point $c \mid\mid x$, first compute

$$\alpha \equiv x^3 + ax + b \pmod p.$$

Then compute the square root β of α modulo p [20]. Note that there might not exist a square root of every $\alpha \in \mathbb{F}_p$. To check whether α is a square in $\mathbb{F}_p$, we use Euler's criterion [40]. The criterion states that

$$\alpha^{\frac{p-1}{2}} \equiv \begin{cases} 1 \pmod p & \text{if there exists a } \beta \text{ such that } \alpha \equiv \beta^2 \pmod p \\ -1 \pmod p & \text{otherwise.} \end{cases}$$

Thus, if $\alpha^{\frac{p-1}{2}} \equiv -1 \pmod p$, the given compressed point is invalid and therefore does not represent a point on the curve. Otherwise, we compute β with the algorithm of Cipolla or the slightly more efficient but more complicated algorithm of Tonelli and Shanks [58]. We then obtain the decompressed point

$$P = \begin{cases} (x, \beta) & \text{if } \beta \equiv c - 2 \pmod 2, \\ (x, p - \beta) & \text{if } \beta \not\equiv c - 2 \pmod 2. \end{cases}$$

Decompression when $p \equiv 3 \pmod 4$
Computing the square root can be done faster for certain fields. For instance, in $\mathbb{F}_p$ where $p \equiv 3 \pmod 4$, the square root β of α can be computed more efficiently as

$$\beta \equiv \alpha^{\frac{p+1}{4}}$$

because

$$\left(\alpha^{\frac{p+1}{4}}\right)^2 \equiv \alpha^{\frac{p+1}{2}} \equiv \alpha^{\frac{p-1}{2}+1} \equiv \alpha^{\frac{p-1}{2}}\alpha \equiv \alpha \pmod p$$

by Euler's criterion $\alpha^{\frac{p-1}{2}} \equiv 1 \pmod p$ when α is a square. The prime p used in the Bitcoin curve satisfies $p \equiv 3 \pmod 4$, speeding up point decompression.

4 Hashing

4.1 Introduction

Taxonomy of hashing methods
A hash function is defined as a function $h\colon S \to [0,m]$ that maps inputs of arbitrary size of a set S to a fixed interval $[0,m]$ [22]. There are two categories of hash functions: data-oriented hash functions and security-oriented hash functions. Data-oriented hash functions are used as a mapping function in order to speed up the data retrieval process in hash tables. In this report, we will not elaborate on this type of hash function; we will focus on security-oriented hash functions.

Again, this type of hash function can be split up into two separate categories: cryptographic hash functions, and non-cryptographic hash functions. The former type of hash function is designed to be *one-way*, which means it is computationally infeasible to invert. These hash functions have several useful properties, which we will discuss shortly. Non-cryptographic hash functions do not have the same security properties, but they are often more efficient to compute. For applications with no strong security requirements, a non-cryptographic hash function will suffice. We will not concentrate on this type of hash function, however. Instead, we will focus only on cryptographic hash functions.

Cryptographic hash functions
A cryptographic hash function is a function that maps an input bit string of arbitrary length to an output bit string of fixed size. More precisely, a cryptographic hash function h is a function

$$h\colon \{0,1\}^* \to \{0,1\}^n,$$

where $\{0,1\}^*$ denotes the set of bit strings of all sizes, and $\{0,1\}^n$ the set of all bit strings of length $n \in \mathbb{N}$ [60].

As mentioned before, cryptographic hash functions have several security properties: collision resistance, pre-image resistance, second pre-image resistance and uniform distribution.

Definition 4.1 (Collision resistant)**.** A cryptographic hash function should be *collision resistant*. This means it is computationally infeasible to find two inputs x and y, where $x \neq y$, such that $h(x) = h(y)$. Two inputs that hash to the same output value are called a *collision*.

Definition 4.2 (Pre-image resistant)**.** Cryptographic hash functions should be *pre-image resistant*. This means that if we are provided with an output bit string z of size n, it is computationally infeasible to find an input x such that $z = h(x)$. In this situation, x is called the *pre-image* of z.

Definition 4.3 (Second pre-image resistant)**.** Cryptographic hash functions should be *second pre-image resistant*. This means that if we are provided with an input x, it is computationally infeasible to find a second input $y \neq x$ such that $h(x) = h(y)$, where y is called a *second pre-image* of x.

Definition 4.4 (Uniform distribution)**.** Cryptographic hash functions should be *uniformly distributed*, which means the function has no outputs that are more likely to occur than others.

4.2 Hashing methods

In this section, we will describe an important family of hash functions: the Merkle-Damgård family. Next, we will describe examples of hash functions in this family. Lastly, we describe the hash functions used in Bitcoin.

4.2.1 Definitions

In this and upcoming sections, several different hash functions will be discussed. In these hash functions, several logical operators will be used. These operators are defined as follows:

- $\vee$ denotes the bitwise logical OR operator.
- $\wedge$ denotes the bitwise logical AND operator.
- $\oplus$ denotes the bitwise logical XOR operator.
- $\neg$ denotes the bitwise logical NOT operator.

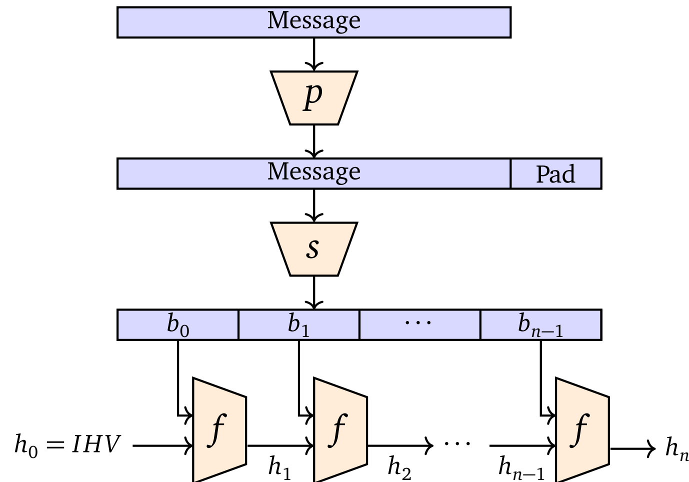

Figure 6: Basic structure of Merkle-Damgård construction.

- S_r^k denotes a right shift of k bits.
- S_l^k denotes a left shift of k bits.
- R_r^k denotes a right rotation of k bits.
- R_l^k denotes a left rotation of k bits.

Little-endian notation is used throughout this section. All these logical operators operate modulo 2^{32}, which means the output of all these functions is also a 32-bit word. The operator $||$ denotes the concatenation of two bit strings.

4.2.2 Merkle Damgård construction

In this section, we will describe the Merkle-Damgård family of hash functions [60]. The structure of a hash function in the Merkle-Damgård family is as depicted in Figure 6. First, a message padding scheme is used to pad the input message, such that it is a multiple of l bits in length, where l is a constant referred to as the *block size* of the algorithm. Next, the message is divided into n blocks, each of length l bits, $b_0, \ldots, b_{n-1}$. Now, the algorithm performs n rounds of applying a compression function f to the blocks. In round i, the compression function uses as input one of the blocks, say b_{i-1}, and an intermediate hash value h_{i-1}. This compression function then outputs a new intermediate hash value h_i, which is used as input hash value in the next compression round, together with the next block, b_i. The first compression round uses as input an initial hash value IHV, which is a constant value of the algorithm. After n rounds of applying the compression function to all n blocks, the output hash value of the n-th compression round is the output hash value.

The compression functions used in the Merkle-Damgård family also have a specific structure, as defined in Figure 7. This structure is as follows. The compression function takes as input a block b_i, which is expanded into m words $w_{i,0}, \ldots, w_{i,m-1}$, using an expansion function e. The algorithm then proceeds by performing m rounds of applying a round function r to the generated words. This round function is different for every hash function in the Merkle-Damgård family. In round j, the round function takes as input word $w_{i,j-1}$ and the previous round function output hash value. The round function then outputs a new hash value, which is used in the next round, together with word $w_{i,j}$. In the first round, the input hash value of the compression function is used as hash value. After m round function rounds, the output hash value of the m-th round is added to the input hash value of this compression round; note that this was block b_i. The result is the output value of the compression function and is used as input for the next compression round.

4.2.3 MD5

The MD5 hashing algorithm is part of the Merkle-Damgård family. The basic steps are described in Figure 6. It is designed to be fast on 32-bit machines and creates a 128-bit bit string from an input

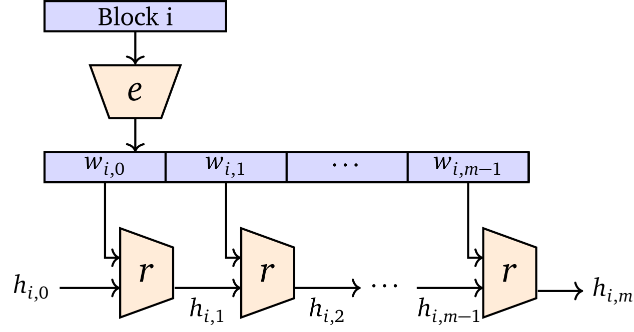

Figure 7: Compression function of Merkle-Damgård construction.

message [54].

First, the message is padded such that its length is congruent to $448 \mod 512$. This is done by adding a single 1-bit to the end of the message, followed by the minimum number of zeros required to make the length of the total message congruent to $448 \mod 512$. Next, a 64-bit representation of the length of the message is added to the end of the message. If it turns out this representation is larger than 64 bits, then only the least significant 64 bits of the length representation are used. Note that the total message length is now a multiple of 512. We split up the padded message into n 512-bit blocks, $b_0, \ldots, b_{n-1}$, which are used as input of a compression function f.

The compression function has a structure as defined in Figure 7. In the i-th compression round, block b_i is split into sixteen 32-bit words, $b_i = (w_{i,0}, \ldots, w_{i,15})$ and used by an extension function e. This message expansion function e simply outputs words $w_{i,0}, \ldots, w_{i,15}$ four times to create a total of 64 words. This message expansion function is defined as follows.

Definition 4.5 (MD5 message expansion function)**.**

$$e_k(b_i) = \begin{cases} w_k & 0 \leq k < 16 \\ w_{(1+5k \mod 16)} & 16 \leq k < 32 \\ w_{(5+3k \mod 16)} & 32 \leq k < 48 \\ w_{(7k \mod 16)} & 48 \leq k < 64. \end{cases}$$

This way, each block b_i is digested using a total of 64 rounds, where every round one of the words outputted by extension function e is used. Note that in this case, m as defined in Figure 7 is $m = 64$.

In each of the 64 steps of the compression round, a round function r is applied to four register values maintained by the algorithm. Let k denote the number of this current round function round and let A_k, B_k, C_k and D_k be the register values at step k. The registers are initialized to be the output for the previous compression round, as can be seen in Figure 6. Note that the input of compression round i is $h_{i-1} = (h_{i-1,0}, h_{i-1,1}, h_{i-1,2}, h_{i-1,3})$, which was the output of the previous compression round $i-1$. Now, the register values in compression round i are initialized as follows:

$$(A_0, B_0, C_0, D_0) = (h_{i-1,0}, h_{i-1,1}, h_{i-1,2}, h_{i-1,3})$$

The first compression round takes as input the following initial hash values,

$$A_0 = \texttt{01234567}$$
$$B_0 = \texttt{89abcdef}$$
$$C_0 = \texttt{fedcba98}$$
$$D_0 = \texttt{76543210}$$

in hexadecimal notation.

We will now define the round function used in MD5. To this end, we first define functions $\varphi_k(x, y, z)$, $q[k]$ and K_k.

Definition 4.6 ($\varphi_k(x,y,z)$).

$$\varphi_k(x,y,z) = \begin{cases} (x \wedge y) \vee (\neg x \wedge z) & 0 \leq k < 16 \\ (x \wedge z) \vee (y \wedge \neg z) & 16 \leq k < 32 \\ x \oplus y \oplus z & 32 \leq k < 48 \\ y \oplus (x \vee \neg z) & 48 \leq k < 64 \end{cases}$$

Definition 4.7 ($q[k]$).

$$\begin{aligned} q[0\dots15] &= [7,12,17,22,7,12,17,22,7,12,17,22,7,12,17,22] \\ q[16\dots31] &= [5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20] \\ q[32\dots47] &= [4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23] \\ q[48\dots63] &= [6,10,15,21,6,10,15,21,6,10,15,21,6,10,15,21]. \end{aligned} \tag{11}$$

Definition 4.8 (K_k).

$$K_k = \lfloor 2^{32} \cdot |\sin(k+1)| \rfloor$$

Now, the round function r of MD5 is defined as follows.

Definition 4.9 (MD5 round function). For compression round i and round function round $k = 0,\dots,63$, the registers are updated as follows:

$$\begin{aligned} A_{k+1} &= D_k \\ B_{k+1} &= B_k + R_l^{q[k]}(A_k + \varphi_k(B_k, C_k, D_k) + e_k(b_i) + K_k) \\ C_{k+1} &= B_k \\ D_{k+1} &= C_k \end{aligned}$$

where b_i is the input block of the i-th compression round, and $R_l^{q[k]}$ denotes a left rotation of $q[k]$ bits. Here, $q[k]$ are constants given in Equation 11 and are chosen such as to yield a fast avalanche effect [54]. K_k was introduced to further increase the uniqueness of each step. This round function is visualized in Figure 8. Note that the current round function round k uses the output values of the previous round function round $k-1$, according to the Merkle-Damgård structure.

In the i-th compression round, after $m = 64$ rounds of applying the round function using all words $w_{i,0},\dots,w_{i,63}$, we obtain values A_{64}, B_{64}, C_{64} and D_{64}. These values are added to the initial register values of compression round i, $h_{i-1} = (A_0, B_0, C_0, D_0)$, modulo 2^{32}. The output of the i-th compression round is now given by the resulting four 32-bit words.

These words are then passed to the $(i+1)$-th compression round, to be used as the initial register values of the next compression round, as described above. The MD5 output is the concatenation of the words obtained from the (final) n-th compression round.

Attacks

MD5 is no longer secure and is considered cryptographically broken, as it is proven to not be collision resistant. One of the most practical attacks thus far is the differential attack, where collisions can be found in “about 15 minutes up to an hour [of] computation time” [59]. This attack can be further exploited to create chosen-prefix collisions. That is, given two prefixes p_1 and p_2, we find two postfixes m_1 and m_2 such that $h(p_1||m_1) = h(p_2||m_2)$. It is shown that, in practice, this can be applied for multiple applications, for example to find colliding certificates or colliding executables [57].

4.2.4 SHA-256

Bitcoin uses several hashing algorithms, including SHA-256 and RIPEMD-160. In this section, we will describe SHA-256. The SHA-256 algorithm is part of the Merkle-Damgård family [60]. Figure 6 gives an overview of the steps taken in this algorithm.

First, the message is padded such that its length is congruent to $448 \mod 512$. This is done by adding a single 1-bit to the end of the message, followed by the minimum number of zeros required to make

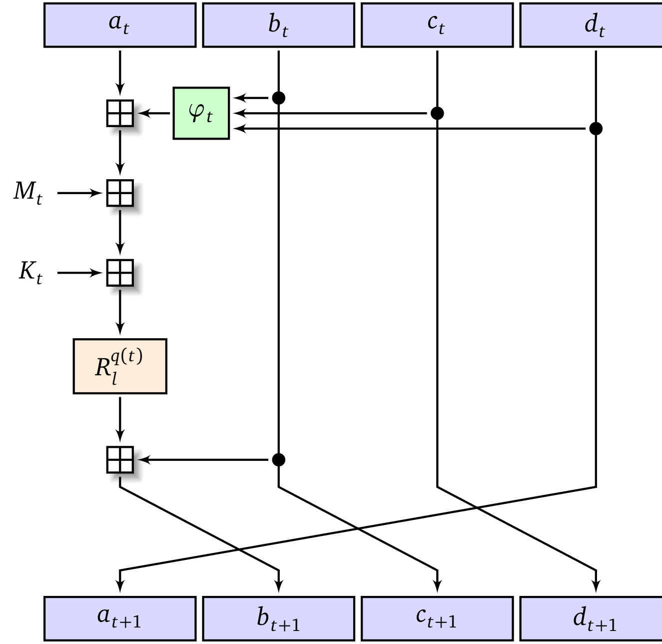

Figure 8: Round function of MD5, adapted from Tikz for Cryptographers [36].

the length of the total message congruent to 448 mod 512. Next, a 64-bit representation of the length of the message is added to the end of the message. If it turns out that this representation is larger than 64 bits, then only the least significant 64 bits of the length representation are used. Note that the total message length is now a multiple of 512. Next, the message is split up into n 512-bit message blocks, $b_0, \ldots, b_{n-1}$.

At this point, the initial hash value h_0 is set. It consists of eight 32-bit words, $h_{0,1} \ldots h_{0,8}$, which are as follows (in hexadecimal notation):

$h_{0,1} = \texttt{6a09e667}$, where $\sqrt{2} = \texttt{1.6a09e667}\ldots$

$h_{0,2} = \texttt{bb67ae85}$, where $\sqrt{3} = \texttt{1.bb67ae85}\ldots$

$h_{0,3} = \texttt{3c6ef372}$, where $\sqrt{5} = \texttt{2.3c6ef372}\ldots$

$h_{0,4} = \texttt{a54ff53a}$, where $\sqrt{7} = \texttt{2.a54ff53a}\ldots$

$h_{0,5} = \texttt{510e527f}$, where $\sqrt{11} = \texttt{3.510e527f}\ldots$

$h_{0,6} = \texttt{9b05688c}$, where $\sqrt{13} = \texttt{3.9b05688c}\ldots$

$h_{0,7} = \texttt{1f83d9ab}$, where $\sqrt{17} = \texttt{4.1f83d9ab}\ldots$

$h_{0,8} = \texttt{5be0cd19}$, where $\sqrt{19} = \texttt{4.5be0cd19}\ldots$

This initial hash value is used as input to a compression function, together with the first message block b_0. The output of this compression function are eight 32-bit words, which are then used as hash value input in the next compression round, together with the next message block b_1. This process is repeated for a total of n rounds. The eight output 32-bit words of the last compression rounds are concatenated, resulting in the output of the SHA-256 algorithm. See Figure 7 for an overview of the steps taken in the compression function.

In the SHA-256 compression function, several logical functions are used, all operating on 32-bit words. Note that all these functions output 32-bit words as well. These logical functions are defined as follows:

Definition 4.10. $\mathrm{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$.

Definition 4.11. $\mathrm{Maj}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$.

Definition 4.12. $\varphi_0(x) = R_r^2(x) \oplus R_r^{13}(x) \oplus R_r^{22}(x)$.

Definition 4.13. $\varphi_1(x) = R_r^6(x) \oplus R_r^{11}(x) \oplus R_r^{25}(x)$.

Definition 4.14. $\sigma_0(x) = R_r^7(x) \oplus R_r^{18}(x) \oplus S_r^3(x)$.

Definition 4.15. $\sigma_1(x) = R_r^{17}(x) \oplus R_r^{19}(x) \oplus S_r^{10}(x)$.

Note that the name Ch stands for *choose* and the name Maj stands for *majority* [60]. This is because the Ch function chooses as i-th output bit either the i-th bit of y as output, or the i-th bit of z, depending on whether the i-th bit of x is set to 0 or 1, respectively. This is repeated for each of the 32 bits of the inputs of the function Ch. The Maj function returns as ith output bit the majority of the ith input bits of x, y and z; if the majority of the ith input bits is set to 1, the ith output bit is 1, else it is 0.

The compression function now consists of the following steps. The i-th compression round takes as input message block b_{i-1} and the output hash of the previous compression round, h_{i-1}. Note that this hash consists of eight 32-bit words, $h_{i-1,1}, \dots, h_{i-1,8}$. Input block b_{i-1} is split into sixteen 32-bit words, $w_{i,0}, \dots, w_{i,15}$. Another 48 words $w_{i,16}, \dots, w_{i,63}$ are calculated using a message expansion function, which is defined as follows.

Definition 4.16 (SHA-256 message expansion function)**.** For $16 \leq k \leq 63$, word $w_{i,k}$ is calculated using the following equation:

$$w_{i,k} = \sigma_1(w_{i,k-2}) + w_{i,k-7} + \sigma_0(w_{i,k-15}) + w_{i,k-16}.$$

The compression function consists of a total of 64 rounds of applying a round function r to the words defined above. The round function maintains eight register values $A, \dots, H$. The registers initially contain the eight words of the input hash function $h_{i-1} = (h_{i-1,1}, \dots, h_{i-1,8})$, as follows

$$(A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0) = (h_{i-1,1}, h_{i-1,2}, h_{i-1,3}, h_{i-1,4}, h_{i-1,5}, h_{i-1,6}, h_{i-1,7}, h_{i-1,8}).$$

Every round k, register values $A_k, \dots, H_k$ are manipulated with this round function r, as shown in Figure 7. The round function used in the SHA-256 algorithm is defined as follows.

Definition 4.17 (SHA-256 round function)**.** In the k-th round of applying the round function, where $k \in \{1, \dots, 64\}$, the register values $A_{k-1}, \dots, H_{k-1}$ of the previous round, together with the word $w_{i,k-1}$ are manipulated modulo s^{32} by the round function. First, two temporary values are calculated:

$$t_1 = H_{k-1} + \varphi_1(E_{k-1}) + \text{Ch}(E_{k-1}, F_{k-1}, G_{k-1}) + K_k + w_{i,k-1}$$
$$t_2 = \varphi_0(A) + \text{Maj}(A_{k-1}, B_{k-1}, C_{k-1})$$

where K_k is a constant, equal to the first 32 bits of the fractional parts of the cube roots of the k-th prime number. This means,

$$K_1 = \texttt{428a2f98}, \text{ since } \sqrt[3]{2} = \texttt{1.428a2f98}\dots$$
$$K_2 = \texttt{71374491}, \text{ since } \sqrt[3]{3} = \texttt{1.71374491}\dots$$
$$\vdots$$
$$K_{64} = \texttt{c67178f2}, \text{ since } \sqrt[3]{311} = \texttt{1.c67178f2}\dots.$$

Now, the eight register values are manipulated as follows:

$$(A_k, B_k, C_k, D_k, E_k, F_k, G_k, H_k) = (t_1 + t_2, A_{k-1}, B_{k-1}, C_{k-1}, D_{k-1} + t_1, E_{k-1}, F_{k-1}, G_{k-1}).$$

The output register values are then used as input of the next round of applying the round function, together with the word $w_{i,k}$. This process is repeated 64 times, once for every word of $w_{i,0}, \dots, w_{i,63}$. The output of applying this round function 64 times are eight 32-bit words, which are added to the eight input hash value words of the compression round $h_{i-1} = (h_{i-1,1}, \dots, h_{i-1,8})$; note that these additions are done modulo 2^{32}. The results are then used as input in the next compression round.

The compression round described above is repeated n times, once for each input block $b_0, \dots, b_{n-1}$. The output of the last compression round is the output of the entire SHA-256 algorithm.

4.2.5 RIPEMD-160

As mentioned in Section 4.2.4, Bitcoin uses several hashing algorithms. The the RIPEMD-160 algorithm will be explained in this section. The steps of the RIPEMD-160 algorithm are as follows [16].

First, the padding of the input message is done in the same way as in the SHA-256 algorithm, explained in Section 4.2.4. Note that after the message padding, the total length of the message is a multiple of 512. The message is now split up into n 512-bit message blocks, $b_0, \ldots, b_{n-1}$.

The algorithm now involves a total of n compression rounds, where round $i \in \{1, \ldots, n\}$ consists of applying a compression function to block b_{i-1} and the output value of the previous compression round, $h_{i-1} = (h_{i-1,0}, h_{i-1,1}, h_{i-1,2}, h_{i-1,3}, h_{i-1,4})$. In the i-th compression round, block b_{i-1} is split up into sixteen 32-bit words, $w_{i,0}, \ldots, w_{i,15}$. The compression function keeps track of ten register values: five left register values, $A_l, \ldots, E_l$, and five right register values, $A_r, \ldots, E_r$. In every compression round, the left register values are manipulated independent of the right register values. Only at the end of the compression round are these values added together, in order to create the output hash of this round, $h_i = (h_{i,0}, h_{i,1}, h_{i,2}, h_{i,3}, h_{i,4})$. The initial hash values of the left register values are as follows (in hexadecimal notation):

$$
\begin{aligned}
A_l &= \texttt{67452301} \\
B_l &= \texttt{efcdab89} \\
C_l &= \texttt{98badcfe} \\
D_l &= \texttt{10325476} \\
E_l &= \texttt{c3d2e1f0},
\end{aligned}
$$

and the right register values:

$$
\begin{aligned}
A_r &= \texttt{76543210} \\
B_r &= \texttt{fedcba98} \\
C_r &= \texttt{89abcdef} \\
D_r &= \texttt{01234567} \\
E_r &= \texttt{3c2d1e0f}.
\end{aligned}
$$

These left and right register values are manipulated in the compression function.

The compression function consists of a total of 80 round function rounds, where every round, both the left and right register values are updated. Similarly, in every round one of the sixteen input words is used as input. Also, every round, the compression function uses one of five different logical functions. We continue until both the left and right register values are updated using all combinations of logical functions and input words. Note that indeed this leads to a total of $5 \cdot 16 = 80$ rounds.

In the i-th compression round, ater these 80 round function rounds, the left and right register values are added together modulo 2^{32} to form an output hash value, $h_i = (h_{i,0}, \ldots, h_{i,4})$. These values are then used as input in the next compression round, where these hash values are copied into the left and right register values of this new round. Note that this means that at the start of a new round, $A_l = A_r$, $B_l = B_r$, etcetera. The process then starts all over, until we completed a total of n compression rounds, one for every input block, $b_0, \ldots, b_{n-1}$.

Several indices will be used in the definition of the RIPEMD-160 round function. Index $i \in \{1, \ldots, n\}$ refers to the n input blocks. Index $j \in \{0, \ldots, 79\}$ refers to the current round function round. Index $k \in \{0, \ldots, 15\}$ refers to the 16 input words.

Three functions are used in the RIPEMD-160 round function definition: $s_j(k)$, $f_j(x, y, z)$, $r(j)$ and $K(j)$. These will now be defined.

Definition 4.18 ($s_j(k)$)**.** This function defines how many left rotations are to be done in the round function, depending on the current round number j and the current input word number k. It is defined in Table 1.

Definition 4.19 ($f_j(x, y, z)$)**.** This function defines which logical function is used in the round function, where. Five different logical functions are defined, operating on modulo 2^{32} on three 32-bit input words

	$0 \le j \le 15$	$16 \le j \le 31$	$32 \le j \le 47$	$48 \le j \le 63$	$64 \le j \le 79$
0	11	12	13	14	15
1	14	13	15	11	12
2	15	11	14	12	13
3	12	15	11	14	13
4	5	6	7	8	9
5	8	9	7	6	5
6	7	9	6	5	8
7	9	7	8	5	6
8	11	12	13	15	14
9	13	15	14	12	11
10	14	11	13	15	12
11	15	13	12	14	11
12	6	7	5	9	8
13	7	8	5	9	6
14	9	7	6	8	5
15	8	7	9	6	5

Table 1: RIPEMD-160 left rotation schedule $s_j(k)$.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho(k)$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8

Table 2: RIPEMD-160 ρ permutation.

x, y and z:

$$\begin{aligned}
f_1(x,y,z) &= x \oplus y \oplus z && 0 \le j \le 15 \\
f_2(x,y,z) &= (x \wedge y) \vee (\neg x \wedge z) && 16 \le j \le 31 \\
f_3(x,y,z) &= (x \vee \neg y) \oplus z && 32 \le j \le 47 \\
f_4(x,y,z) &= (x \wedge z) \vee (y \wedge \neg z) && 48 \le j \le 63 \\
f_5(x,y,z) &= x \oplus (y \vee \neg z), && 64 \le j \le 79
\end{aligned}$$

depending on the current round number j.

Definition 4.20 ($r_l(j)$ and $r_r(j)$)**.** These functions define which of the sixteen input words is being used in the current round function, depending on the current round number j and on whether we are updating left or right register values. To this end, we use permutation ρ defined in Table 2. Also define the permutation $\pi(k) = 9k + 5 \mod 16$. Now, a value $k \in \{0, \ldots, 15\}$ is the output of functions $r_l(j)$ and $r_r(j)$, according to the ordering specified in Table 3.

Definition 4.21 ($K_l(j)$ and $K_r(j)$)**.** $K_l(j)$ and $K_r(j)$ define constants used in the RIPEMD-160 round function, depending on the current round number j and whether we are updating left or right register values. The constants are defined in Table 4 in hexadecimal notation.

The round function of RIPEMD-160 is now defined as follows.

Definition 4.22 (RIPEMD-160 round functions)**.** In compression round i, the RIPEMD-160 round function updates the register values $A_l, \ldots, E_l, A_r, \ldots, E_r$ modulo 2^{32}, according to current round number j, as defined in Algorithm 9, where t_l and t_r are temporary values. This round function is repeated for all

	$0 \le j \le 15$	$16 \le j \le 31$	$32 \le j \le 47$	$48 \le j \le 63$	$64 \le j \le 79$
l	j	$\rho(j-16)$	$\rho^2(j-32)$	$\rho^3(j-48)$	$\rho^4(j-64)$
r	$\pi(j)$	$\rho\pi(j-16)$	$\rho^2\pi(j-32)$	$\rho^3\pi(j-48)$	$\rho^4\pi(j-64)$

Table 3: RIPEMD-160 ordering $r_l(j)$ and $r_r(j)$ in hexadecimal notation.

	$0 \le j \le 15$	$16 \le j \le 31$	$32 \le j \le 47$	$48 \le j \le 63$	$64 \le j \le 79$
l	00000000	5a827999	6ed9eba1	8f1bbcdc	a953fd4e
r	50a28be6	5c4dd124	6d703ef3	7a6d76e9	00000000

Table 4: RIPEMD-160 constants $K_l(j)$ and $K_r(j)$.

$j \in \{0, \ldots, 79\}$, until all left and right register values were updated using all combinations of the five logical functions and the sixteen input words.

Algorithm 9 RIPEMD-160 round function round j.

$t_l \leftarrow R_l^{s_j(r_l(j))}(A_l + f_j(B_l, C_l, D_l) + w_{i,r_l(j)} + K_l(j)) + E_l$
$t_r \leftarrow R_l^{s_j(r_r(j))}(A_r + f_{79-j}(B_r, C_r, D_r) + w_{i,r_r(j)} + K_r(j)) + E_r$

$A_l \leftarrow E_l$
$E_l \leftarrow D_l$
$D_l \leftarrow R_l^{10}(C_l)$
$C_l \leftarrow B_l$
$B_l \leftarrow t_l$

$A_r \leftarrow E_r$
$E_r \leftarrow D_r$
$D_r \leftarrow R_l^{10}(C_r)$
$C_r \leftarrow B_r$
$B_r \leftarrow t_r$

In the i-th compression round, which uses input block b_{i-1}, after a total of 80 round function rounds, we are left with ten final register values. They are added together as specified in Algorithm 10, using the input hash function of compression round i, $h_{i-1} = (h_{i-1,0}, h_{i-1,1}, h_{i-1,2}, h_{i-1,3}, h_{i-1,4})$.

Algorithm 10 RIPEMD-160 output compression round i.

$t \leftarrow h_{i-1,1} + C_l + D_r$
$h_{i,1} \leftarrow h_{i-1,2} + D_l + E_r$
$h_{i,2} \leftarrow h_{i-1,3} + E_l + A_r$
$h_{i,3} \leftarrow h_{i-1,4} + A_l + B_r$
$h_{i,4} \leftarrow h_{i-1,0} + B_l + C_r$
$h_{i,0} \leftarrow t$

The output hash value $h_i = (h_{i,0}, \ldots, h_{i,4})$, is then used as input in the next compression round, where it is split up in 5 32-bit words and copied into the left and right register values of this new round. After compressing all n blocks in n compression rounds, the outputs of the final compression round are concatenated and used as output of the RIPEMD-160 algorithm.

4.3 Applications of hashing in Bitcoin

In Bitcoin, hash functions are used for different purposes in several places. First, we will explain the phenomenon of *double hashing*, which is used by Bitcoin to prevent attacks. Next, we will discuss every use case of hashing in Bitcoin, what hash functions are used in each case and what properties of hash functions are essential for each use case.

4.3.1 Double hashing

In some applications, Bitcoin applies the SHA-256 algorithm twice, which is called *double hashing*. This is because the SHA-256 algorithm is vulnerable to so-called *length extension attacks* [24]. Such an attack is performed as follows. Remember that the SHA-256 algorithm is part of the Merkle-Damgård family, meaning it has a structure as depicted in Figure 6. Now, let m be a message, that is split into blocks $b_1, \ldots, b_k$ with hash value h. Suppose now that an attacker chooses a message m' that is split

into blocks $b_1, \ldots, b_k, b_{k+1}$. Note that the hash h of message m simply is an intermediate value in the calculation of the hash of message m'. This means the attacker can compute hashes of augmented messages he might not even know himself.

The length extension problem exists, because there is no special processing at the end of the hash function computation [24]. A solution to this problem is double hashing; now, m is not an intermediate value in the calculation of the hash of message m' and the problem is solved.

Note also that when converting public keys into addresses, as described in Section 4.3.5, Bitcoin uses both SHA-256 and RIPEMD-160. This is done due to the fact that, if one of the two hash functions breaks, this leaves the other function to sustain the system. Thus, it provides the system with an extra layer of security.

Note that in both cases, since both SHA-256 and RIPEMD-160 are cryptographic hash functions, double SHA-256 is also a cryptographic hash function, just like applying RIPEMD-160 after SHA-256. This means the security properties of collision resistance, pre-image resistance and second pre-image resistance stay intact, even when double hashing is used. This means that, in the next sections, if we state that SHA-256 is collision resistant, pre-image resistant and second pre-image resistant, and double hashing is used, double SHA-256 has these security properties as well.

4.3.2 Block and transaction identifier

A use case of hash functions in Bitcoin is generating block and transaction identifiers. The identifier of a block is calculated by hashing the block header. The identifier of a transaction is calculated by hashing the entire transaction. The hash function used here is the SHA-256 algorithm, which is applied twice.

The SHA-256 hash function has several properties that are necessary to calculate these identifiers. First of all, the hash function is uniformly distributed, which means the function has no outputs that are more likely to occur than others. This means every identifier is equally likely to occur, which is, of course, intended behavior. Indeed, in this situation, the likelihood of two objects ending up with the same identifier is the smallest, i.e. a collision is the least likely to occur in this situation.

Next to this, all outputs of the hash function have the same length, which means all calculated identifiers have the same length. This makes the identifiers recognizable and easy to compare. Note that this also makes them easy to store, as this makes it possible to allocate a fixed number of bytes to store a block identifier.

Furthermore, identifiers should be easy to calculate. This is because block and transaction identifiers need to be calculated many times in Bitcoin, for example in the proof-of-work mining algorithm. Thus, it is preferable that block and transaction identifiers can be computed quickly. Indeed, in Bitcoin it takes applying the SHA-256 algorithm twice in order to generate an identifier.

Also, the SHA-256 algorithm is collision resistant, which means it is computationally infeasible to find two inputs that hash to the same output, which is an important property. For example, if we could find two block headers that hash to the same output, these two blocks would have the same identifier. An attacker could use this flaw, for example to make sure (valid) blocks of other users are unrightfully rejected, in the following way. The attacker could generate two blocks, b_i and b_j, with the same identifier and propagate them both over the network. Some nodes in the Bitcoin network would add block b_i to their blockchain, as they received this block first. Other users, however, who received block b_j first, would add b_j to their blockchain. Note that, unlike the situation created by forks in the blockchain, this situation will never resolve itself. The attacker has created a *hard fork*. This means different users of the network will have different versions of the blockchain. This is why collision resistance is such an important property when it comes to block and transaction identifiers.

Note that in practice, this attack would probably be infeasible, because both blocks need to have the same parent block hash, and the identifier of blocks b_i and b_j needs to conform to the target value. The chance of finding such a collision before another miner has mined a new block is still very small, even when the hash function is not collision resistant. However, it is still preferable to use a hash function that is collision resistant, such that this attack can never be executed.

Lastly, it is important that the SHA-256 algorithm is second pre-image resistant, which means that if we are provided with an input x, it is computationally infeasible to find a second input y such that $h(x) = h(y)$. If the SHA-256 algorithm was not second pre-image resistant, an attacker could do the

following. After a user of the network mines a new block b_i, the attacker could create an (invalid) block b_j with the same block identifier. The attacker then propagates this block b_j across the network. Nodes in the network verify the block, and conclude that it is invalid. Now, the other user propagates his block b_i, which will also be seen as invalid. This is because nodes in the network think they just verified this block as invalid, while this was actually block b_j. So, it is important that the hash function used to calculate block identifiers is second pre-image resistant.

4.3.3 Proof-of-work

Another application where hash functions are used in Bitcoin, is in the proof-of-work algorithm. As mentioned before, this algorithm is used by miners when they are mining a new block. In this algorithm, the nonce parameter in the block header of the mined block is repeatedly changed, until the block header hash (and thus the block hash) is smaller than a specific target value. See Section 2.3 for a more detailed explanation of the mining procedure. The average work necessary to mine a block is exponential in the number of zero bits of the target value. In Bitcoin, this target value is adjusted every 2016 blocks, in order to deal with the increasing hashing power across the network [1].

A property of the proof-of-work algorithm should be that it takes an expected number of hash computations to find a valid nonce value that conforms to the target value, while it should only take one hash computation to verify that the nonce of some block is indeed valid. Note that the expected number of hash computations is adjustable through the target value; the target value defines the difficulty level of the proof-of-work algorithm. Also, note that once a block has been mined, it cannot be changed without re-doing all the mining work. A new nonce value must be found such that the block hash conforms to the target value.

As mentioned in Section 4.3.2, the SHA-256 algorithm is used to calculate a block identifier, by applying the SHA-256 algorithm twice to the block header of the block. In the proof-of-work algorithm, many block identifiers need to be calculated. The SHA-256 algorithm has several properties that are important for the proof-of-work algorithm to function properly.

First of all, as stated above, it should be easy to verify that a block conforms to the target value. This verification is done by calculating the identifier of the block, i.e. the hash of the block header, and checking that this hash is smaller than the target value specified in the block header. Indeed, it only takes one hash computation to check that a block conforms to the target value.

Next to this, in order for the proof-of-work algorithm to function properly, the SHA-256 algorithm should be pre-image resistant. Given a certain hash value z, it should be computationally infeasible to find an input x such that $h(x) = z$. Otherwise, a miner could simply choose a block hash that is smaller than the target value, and find the pre-image of this block hash. Note that the miner has now found a nonce that makes the block conform to the target value within one computation. This is why it is important that the SHA-256 algorithm is pre-image resistant.

Note that in practice, the chance that this situation occurs is small, even when the used hash function is not pre-image resistant. This is because the pre-image found actually needs to be a valid block, with the correct target value and parent hash, among others. However, it is still preferable to use a hash function that is pre-image resistant, such that this attack can never be executed.

4.3.4 Private key generation

In Bitcoin, hash functions are used in non-deterministic wallets, for generating private keys from a random number. In Bitcoin, a private key is a 256-bit number between 1 and $n-1$. Here, n is defined as the order of the subgroup being used on the elliptic curve (see Section 3 for more details):

$$n \approx 1.158 \cdot 10^{77},$$

which is slightly less than 2^{256} (see Section 3) [1]. In order to generate such a private key, we first use a secure source to generate a string of random bits. This string is hashed using the SHA-256 hash function, which indeed produces a 256-bit output hash. Next, we check whether the output hash is less than n. If so, we have found an appropriate private key. If not, we repeat the process from scratch, until we find a private key less than n. Note, however, that this almost never happens.

The SHA-256 algorithm being used here has several properties that are advantageous to the key generation process. First of all, the algorithm generates a 256-bit output, which is exactly the same length

a private key has. If we would have used another (hashing) function that produces a shorter output, there are some keys in our key space we are not able to generate. If, on the other hand, we would have used a hash function that produces a longer output, then the chance of generating a number larger than n becomes larger. Hence, the SHA-256 algorithm produces outputs with a desired output length.

Next to this, it is essential that the private keys being generated should not be predictable. Indeed, the number by which the private key was generated was random, but it did not yet have the desired length. It is therefore used as input of the SHA-256 algorithm, which is uniformly distributed, which makes every output equally likely to occur. In this situation, no private key is preferred over others. This makes the likelihood of an attacker being able to guess a private key the smallest.

4.3.5 Converting public keys to addresses

Another use for hash functions is generating a Bitcoin address from a public key. Note that bitcoin is paid to addresses instead of public keys, because this creates an additional layer of security. The reasoning behind this is discussed in more detail in Section 5.1.6. A Bitcoin address A is derived from a compressed public key K as follows [1]:

    A = RIPEMD160(SHA256(K)).

Note that double hashing is being used with two different hash functions, rather than SHA-256 being applied twice.

The hash functions used here have several properties necessary for the conversion of public keys to Bitcoin addresses. First of all, we should be able to convert a public key into a Bitcoin address quickly, as Bitcoin addresses need to be calculated often. Indeed, it only takes one step of applying SHA-256 and one of applying RIPEMD-160 to convert a public key into an address.

Next to this, in this case it is required that the hash functions used are pre-image resistant. This means we can convert a public key into a Bitcoin address, but it is computationally infeasible to derive the public key that belongs to a given Bitcoin address. This property is important, because it provides pseudo-anonymity: when somebody pays Bitcoin to a certain address, we do not know what public key the money is paid to, until the money is actually spent. It also provides an additional layer of security when the signature scheme used by Bitcoin suddenly breaks. More details on the reasoning behind this are discussed in Section 5.1.6.

Furthermore, no Bitcoin address should be preferred over others. Indeed, both hash functions used are uniformly distributed, which means all output values are equally likely to occur. In this situation, the likelihood of a collision to occur is the smallest, i.e. the chance that two Bitcoin users end up with the same Bitcoin address is the smallest.

Lastly, the RIPEMD-160 algorithm always outputs a value of exactly 160 bits, which means every Bitcoin address has exactly the same length. This makes them easy to recognize and easy to compare. It also makes them easy to store, as this makes it possible to allocate a fixed number of bytes to store a Bitcoin address. Note that Bitcoin addresses are usually converted into the Base58Check format in order to make them even more readable and comparable. Even after this conversion, all Bitcoin addresses still have the same length.

4.3.6 Base58Check encoding

Another use case for hash functions in Bitcoin is the calculation of the Base58Check encoding of data. The Base58 alphabet is a subset of the Base64 alphabet, omitting some characters that might cause confusion when displayed in certain fonts [1]. The Base58 alphabet uses all upper- and lowercase letters, and numbers, but omits the number zero (0), capital o (O), lowercase L (l), capital i (I), and the symbols '+' and '/'. Base58 encoding is used to represent Bitcoin addresses, because it provides readability and a compact representation of data.

In order to provide error detection and extra security against typing errors, in Bitcoin the Base58Check encoding format is used to represent addresses. It adds an additional 4 bytes at the end of the data, called a checksum. When a piece of data is presented in Base58Check encoding format, the checksum of the presented data is calculated and compared with the checksum included at the end of the data. If these two do not match, this means an error was introduced.

Converting data into a Base58Check format is done as follows [10]. First, the checksum of the data is calculated:

```
checksum = SHA256(SHA256(data)).
```

We see that again double hashing is applied, using the SHA-256 hashing algorithm. Note that this outputs a 32-byte hash, and the checksum should only be 4 bytes. To solve this, we only take the first 4 bytes of the output hash as the checksum of the data. This checksum is concatenated at the end of the data. The result is encoded using the Base58 alphabet. Afterwards, the number of leading zeros of the original data are counted, and the result is prefixed with as many ones. See Algorithm 11 for the pseudocode of this procedure.

Algorithm 11 Base58Check encoding.

alphabet ← `123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

z ← number of leading zero bytes of data

t ← SHA256(SHA256(data))
checksum ← $t[0..3]$ ▷ take first 4 bytes
x ← data.append(checksum)

output ← ε
while $x > 0$ **:**
    (x, remainder) = divide(x, 58)
    output.append(alphabet[remainder])
for $i = 1$ **to** z **:**
    output.append(alphabet[0])
return output.reverse()

Note that the checksum is 4 bytes, while the calculated output hash was 32 bytes. This increases the likelihood of a collision; by only taking the first 4 bytes of the output hash, we increase the chance that two different pieces of data (before and after a typing error) end up with the same checksum. However, using 32 bytes would make the Bitcoin addresses especially long, which makes the chance of typing errors larger. Also, the chance of an actual collision in the checksum of 4 bytes is still relatively small. This is why the trade-off was made to use only 4 bytes for the checksum of a Base58Check encoding, rather than 32.

The SHA-256 algorithm used to convert data into a Base58Check format has several properties necessary for this conversion. First of all, if we make one typing error when typing over a Bitcoin address, the checksum calculated from this data will be completely different compared to the original checksum. This is because the SHA-256 hash function is uniformly distributed and the hash function is collision resistant.

Next to this, every time a Bitcoin address is encountered, a checksum needs to be verified. This means we should be able to verify checksums quickly. Indeed, we only need to calculate one checksum, and compare this with the provided checksum in order to verify whether the provided checksum is correct. Indeed, the checksum of a Bitcoin address can be verified quickly, as calculating a checksum only takes one step of applying the SHA-256 algorithm twice.

4.3.7 Bloom filters

Suppose we have an element s and one of our peers has a set S. We wish to know whether s is part of set S, without having to reveal s to our peers. Bloom filters can be used for this type of situation. A Bloom filter is a data structure that represents a set S of elements in a way that is space-efficient and randomized [47]. They provide a way to check whether a certain element s is part of a set S the Bloom filter represents. However, Bloom filters do allow the possibility for false positives.

In Bitcoin, users have the possibility to run a light-weight type of node called a Simplified Payment Verification node (or SPV node for short) that verifies transactions in a block in a specific way, without having to download the entire block. These nodes do not download full blocks and thus, do not have the

transactions of the block. Instead, these nodes download only block headers. These type of nodes use Bloom filters to ask peers for specific transactions that are part of the set their Bloom filter represents. This way, these SPV nodes can request transactions without having to explicitly specify which addresses, keys or transactions they need. This allows SPV nodes to participate in the Bitcoin network, and specify a search pattern for transactions that can be tuned toward precision or privacy. A more specific Bloom filter will produce accurate results, but at the expense of revealing what patterns the SPV node is interested in, thus revealing the addresses owned by the user's wallet [1].

We will first explain the creation and verification of Bloom filters, after which we will explain how Bloom filters are used by SPV nodes.

Creation
Assume we wish to construct a Bloom filter for representing a set $S = \{s_1, s_2, \dots, s_n\}$ of n elements. To this end, we define a Bloom filter array b of m bits, initially all set to 0. The filter then defines k hash functions, $h_1, \dots, h_k$, where

$$\forall_{i \in \{1,\dots,k\}} : h_i(s) : \{0,1\}^* \to \{1, \dots, m\}.$$

Now, for each element $s \in S$, k hashes are calculated, $h_1(s), \dots, h_k(s)$. These functions each output an integer in $[1, m]$, corresponding to one of the bits of Bloom filter b. For each of the hash functions $h_i(s)$ the $h_i(s)$-th bit of the m-bit Bloom filter is set to 1. This process is repeated for all elements in set S. See Algorithm 12 for the pseudocode of this procedure.

Algorithm 12 Bloom filter construction.

```
b ← []
for i = 0 to m − 1 :
    b[i] ← 0

for s ∈ S :
    for i = 1 to k :
        b[h_i(s)] ← 1
return b
```

Verification
To check whether an element t is in S, again k hash functions are calculated, $h_1(t), \dots, h_k(t)$. If one of the bits represented by the hash outcomes $h_1(t), \dots, h_k(t)$ is 0 in the Bloom filter, we know t is not an element of S. If all of the bits represented by the hash outcomes are 1, we know t *might* be part of S. This is because, if indeed $t \in S$, all hash values will be set to 1. If, however, $t \notin S$, it might still be that all bits $h_i(t), i \in \{1, \dots, k\}$, are 1, even though t is actually not part of S. This is due to the fact that Bloom filters allow the possibility for false positives. Note that false negatives are not possible. Also, note that when one of the bits of the Bloom filter is set to 1, we do not know by which hash function this bit was set to 1.

Note that the larger we choose m, the smaller the chance for a collision to occur. On the other hand, the larger we choose k, the larger the chance for a collision to occur. These two parameters therefore define the amount of precision of the created Bloom filter. This way, the Bloom filter can be tuned to either precision or privacy, as mentioned above.

SPV nodes
The SPV nodes mentioned above use Bloom filters in the following way [1]. The SPV node constructs a Bloom filter with all addresses and transaction IDs from any UTXO controlled by its wallet. The node then sends this Bloom filter to its (full node) peers. The peer checks the Bloom filter on every incoming transaction. Each of the transaction's outputs is checked against the Bloom filter. If any of them match, the peer sends the incoming transaction to the SPV node. The SPV node can use this transaction to update its wallet UTXO set and wallet balance.

Properties
In Bitcoin, version 3 of the 32-bit Murmur hash function is used [33]. This is a non-cryptographic hash

function suitable only for general hash-based lookup. In order to get n different hash functions, the Murmur hash algorithm is initialized n times, each with a different initial value. This value is calculated using the following formula:

$$i * \texttt{FBA4C795} + \texttt{tweak},$$

where i is the number of the current hash function h_i being initialized, FBA4C795 is a constant hexadecimal value and tweak is a constant value.

Indeed, note that this hash function is not collision resistant. However, this property is not necessary for the use of Bloom filters. This is because Bloom filters themselves already create the possibility for false positives, which means there is no need for a collision resistant hash function.

Also note that the Murmur hash algorithm is not designed to be pre-image resistant. This property is not necessary in the use case of Bloom filters. This is because, if we want to find the pre-image of some bit i of Bloom filter b which was set to 1, we still do not know by which hash function $h_1, \ldots, h_k$ the bit was set to 1. This means we can only find k possibilities for the pre-image of bit i, but we cannot know for sure which of those possibilities was the actual pre-image.

The Murmur hash function has one property that is important for the application of Bloom filters. A lot of hash values need to be calculated in the creation of a Bloom filter, as well as in the usage of a Bloom filter by SPV nodes. This is why it is important that the hash function used can efficiently calculate hash values. Indeed, the Murmur hash function can be used to efficiently produce 32-bit hash values [2].

4.3.8 Merkle trees

A Merkle tree is a data structure used for easy summarization and authentication of a large set of messages [19]. They provide the ability to verify the authenticity of one of the messages, even without knowledge of the other messages represented in the Merkle tree. Merkle trees are binary trees, with the hash values of the represented messages in the leaves of the tree. The inner nodes of the Merkle tree also contain hash values.

In the Bitcoin network, Merkle trees are used to prove that a particular transaction is included in a block. Note that in this case, the messages represented by the Merkle tree are the transactions in the block. Merkle trees are used by Simplified Payment Verification nodes in the network, which were already mentioned in Section 4.3.7. These nodes only download the block headers of a block, not the transactions included in a block. These nodes use Merkle trees to prove that a transaction is part of a block. In order to verify that a transaction is included in a block, without having to download all the transactions in the block, they request an authentication path, or Merkle path, from their (full node) peers [1].

Note that Merkle roots have a property that is essential to all nodes of the Bitcoin network. The Merkle root included in the header of a block ensures that if a new transaction is added to the block, or if one of the transactions in the block changes, the header of the block changes, and thus, the block hash. This ensures the block hash changes when the transactions in the block change.

Creation

Merkle trees in Bitcoin are built up as follows. Assume we are building a Merkle tree for the set of messages $M = \{m_1, \ldots, m_n\}$. First, we define a binary tree with n leaf nodes. For every $m_i \in M$, we hash the message m_i and fill leaf node i of the tree with this hash value, where h_i is the value of leaf node i:

$$h_i = \texttt{SHA256}(\texttt{SHA256}(m_i)).$$

Next, consecutive pairs of leaf nodes are summarized in their parent node, by concatenating the hash values in the leaf nodes, and hashing this concatenation. Let the leaf nodes be h_i and h_{i+1} and let their parent node be $h_{i,i+1}$, then the value of the parent node becomes:

$$h_{i,i+1} = \texttt{SHA256}(\texttt{SHA256}(h_i \mid\mid h_{i+1})).$$

This process is continued until the whole tree is filled, including the root of the binary tree, which is called the Merkle root.

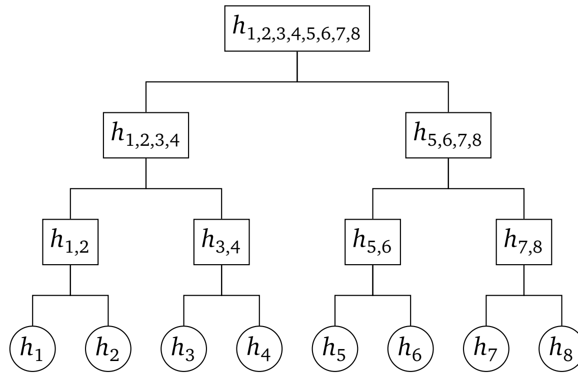

Figure 9: Example of a Merkle tree with eight leaves.

Note that this process only works if we have an even number of messages on each level. We need a solution in case we have a level with an odd number of messages. Bitcoin solves this problem by using the last transaction in the block twice, to create an even number of leaf nodes.

However, this Merkle tree implementation creates some possibilities for attacks. Suppose we just mined a block b_i with an odd number of transactions. An attacker could create a block b_j equal to this block, except it contains the last transaction twice, i.e. the last transaction of the block is duplicated. The attacker then propagates this block b_j across the network. Nodes in the network verify the block, and conclude that it is invalid, because it contains duplicate transactions. Now, note that using the Bitcoin implementation of Merkle trees, the Merkle roots of both blocks b_i and b_j are the same. This means the block hashes of the two blocks are the same as well, because a block hash is calculated using only the block header of the block, and the two block headers of blocks b_i and b_j are equal. This means if we propagate our mined block b_i across the network, this block will also be seen as invalid. This is because nodes in the network think they just verified this block as invalid, while this was actually block b_j. So, the Bitcoin implementation of Merkle trees creates some possibilities for attacks.

In order to prevent this problem, Merkle trees are implemented differently in the Brabocoin implementation. Instead of duplicating the last transaction, if a block contains an odd number of transactions, the last transaction is propagated up the Merkle tree to the first level that contains an odd number of nodes. The last transaction is added to this level, such that this level now also contains an even number of nodes. Note that this creates a legitimate binary tree where every level, except the top level, contains an even number of nodes.

Verification

As mentioned before, Merkle trees provide the ability to verify the authenticity of one of the messages, even without knowledge of the other messages represented in the Merkle tree. In Bitcoin, this means any node in the network can verify that a transaction is included in a block, using the Merkle root of the block, which included in the block header. This is done by asking other nodes in the network for a Merkle path, that proves that a certain transaction is included in a block.

Suppose node n_a asks node n_b for a proof that transaction t is included in block b. In order to prove this, node n_b provides node n_a with a Merkle path, proving a connection between the transaction and the Merkle root of the block. This Merkle path consists of $\lceil \log_2 n \rceil$ 32-byte hashes, where n is the number of transactions in the block, and thus the number of leaves of the Merkle tree. We will use an example to explain the details.

An example of a Merkle tree is provided in Figure 9. This Merkle tree represents a block of eight transactions, and node n_b wishes to prove that transaction $t = 7$ is included in the block. This transaction is represented in the Merkle tree by leaf node h_7. In order to prove that this transaction is part of the block, n_b provides n_a with the following Merkle path:

$$\{(h_8, \texttt{right}), (h_{5,6}, \texttt{left}), (h_{1,2,3,4}, \texttt{left})\}.$$

n_a can now concatenate hash h_8 right of h_7, denoted by the `right` value in the tuple $(h_8, \texttt{right})$, and hash the result to get node $h_{7,8}$. n_a concatenates $h_{5,6}$ left of $h_{7,8}$ and hashes the result to calculate $h_{5,6,7,8}$. Lastly, $h_{1,2,3,4}$ and $h_{5,6,7,8}$ are concatenated and hashed to calculate the Merkle root. If this calculated

Merkle root is equal to the Merkle root included in the block header, indeed transaction t is part of the block. If not, the provided Merkle path is not a valid Merkle proof that the transaction is included in the block. Note that the `left` and `right` values included in the tuples of the Merkle path are essential; if one were to accidentally concatenate a hash value at the wrong end, the result will be a different Merkle root.

Properties
As mentioned above, the hash function used by Bitcoin in the creation of Merkle trees is the SHA-256 algorithm. This hash function has several properties necessary for the implementation of Merkle trees. First of all, the SHA-256 algorithm always outputs 256-bit hash values. This makes its application in Merkle trees especially useful: it provides the ability to always summarize any large set of messages to one single hash value of the same length. In Bitcoin, this makes it possible to summarize any set of transactions using a 256-bit Merkle root.

Next to this, suppose we are provided with a transaction t_1 included in a block b and a hash value z. It would be problematic if we could find another transaction t_2 such that

$$z = \texttt{SHA256(SHA256(}t_1\texttt{))} = \texttt{SHA256(SHA256(}t_2\texttt{))}.$$

In the example above, transaction t_1 could be replaced by transaction t_2 in a Merkle proof, and the provided Merkle proof would still be valid, even though t_2 was not actually part of the block. This would mean that we would be able to provide Merkle proofs for transactions that are actually not part of a block. Indeed, the SHA-256 is second pre-image resistant, which means, provided with an input t_1, it is computationally infeasible to find another input t_2 such that

$$\texttt{SHA256(SHA256(}t_1\texttt{))} = \texttt{SHA256(SHA256(}t_2\texttt{))}.$$

This property prevents the ability to perform the attack described above.

Furthermore, it is essential that the used hash function is pre-image resistant. Suppose we are provided with a hash value z in a Merkle proof p and we would be able to derive a pre-image t of z:

$$z = \texttt{SHA256(SHA256(}t\texttt{))}.$$

Note that it could be that t is not actually the transaction on which proof p was based, but another transaction that was not actually included in the original block. This would mean, however, that an attacker could construct invalid Merkle proofs for transactions that are not actually included in the blocks the Merkle tree is based on. This is why it is essential that SHA-256 is pre-image resistant.

Lastly, the hash function used should be collision resistant. If not, then we could find two transactions t_1 and t_2 such that

$$\texttt{SHA256(SHA256(}t_1\texttt{))} = \texttt{SHA256(SHA256(}t_2\texttt{))}.$$

An attacker could now send a block that includes t_1 over the network, and interchange t_1 and t_2 in the Merkle proofs he provides, while only t_1 is actually included in the block. Indeed, SHA-256 is collision resistant. Note that in practice, the chance of t_1 and t_2 both being valid transaction is small. However, it is still preferable to use a hash function that is collision resistant, such that this attack can never be executed.

4.3.9 Digital signatures

Lastly, hash functions are used in Bitcoin in order to construct digital signatures. The details on what digital signatures are and how they are calculated can be found in Section 5. In summary, Bitcoin uses hash functions to create the messages that need to be signed. Raw transaction data (without signatures) is used as input and the SHA-256 algorithm is applied twice. The resulting data is used as input to calculate digital signatures.

An important requirement is that the used hash function is be second pre-image resistant. Suppose we are provided with raw transaction data t with hash value

$$m = \texttt{SHA256(SHA256(}t\texttt{))}.$$

It should not be possible to find another transaction t' with the same hash value m. Otherwise, a signature calculated for transaction t is also a valid signature for transaction t'. This means an attacker could create invalid signatures for transactions of money he does not actually own. This would enable him to steal money from other users. It is therefore essential that the used hash function is second pre-image resistant.

Also, it is important that the used hash function is pre-image resistant. Suppose we are are provided with raw transaction data t with hash value $m = h(h(t))$ and signature (r,s). Note that we can now find another message m' with the same signature (r,s) (see Section 5). This is why it is important for the hash function to be one-way; if it is not, we could derive raw transaction data t' from hash value m'. We have now derived another transaction t' for which the signature (r,s) is also valid. Again, this would allow an attacker to create signatues for money he does not actually own, and thus, steal money from other users. That's why it is essential that the SHA-256 algorithm is pre-image resistant.

4.3.10 Message authentication codes

Bitcoin wallets use *Message Authentication Codes* (MAC), which provide message integrity and message authentication, given a secret key. Hence, they share some properties with digital signatures (explained in Section 5). However, unlike digital signatures, MACs do not provide non-repudiation.

One advantage MACs have is that they are generally much faster to compute than digital signatures, as they are based on block ciphers or hash functions.

A MAC function can be interpreted as a function $\text{MAC}_k : \{0,1\}^* \to \{0,1\}^n$, where k is the secret key and n is a fixed output size for the MAC function. Given a message m, the authentication code $x = \text{MAC}_k(m)$ is computed. The tuple (m, x) is sent over a (possibly insecure) channel to the receiver. He takes the message m and uses the shared secret key k to compute its own authentication code $x' = \text{MAC}_k(m)$. The receiver can then verify that $x = x'$.

Message integrity is provided, given the MAC_k function yields an incorrect result when the message m is altered during transit over the channel, i.e when $x \neq x'$. Message authenticity is provided when the MAC_k function uses k in such a way that a valid MAC can not be constructed (or is computationally infeasible) for some message m without knowing k. That is, when some adversary changes the message m to m' during transit, the computation of a valid MAC for the altered message m' without knowing k should be infeasible.

Secret prefix and suffix MACs
A straightforward idea is to hash the key together with the message. Given a hash function h, the *secret prefix MAC* is given by

$$\text{MAC}_k(m) = h(k \,||\, m),$$

and the *secret suffix MAC* is given by

$$\text{MAC}_k(m) = h(m \,||\, k),$$

where the '||' symbol denotes concatenation.

Both are vulnerable for attacks. Take for example the secret prefix MAC and assume h is a hash function from the Merkle-Damgård family, as described in Section 4.2.2. Suppose the attacker has the tuple (m, x), where m is the message and $x = \text{MAC}_k(m) = h(k \,||\, m)$ is the MAC of the message. Note that k is unknown to the attacker. Since x is the output of a number of round functions of the given hash function h, the attacker can use x as the initial hash vector and add additional blocks (suffix message m') to yield hash x'. In this case, x' is a valid MAC for the original message m, padded to match the block size of the hash function (call this block m_{pad}) appended with the suffix message m'. That is, $x' = \text{MAC}_k(m \,||\, m_{\text{pad}} \,||\, m')$. Hence, when using the secret prefix MAC, message authenticity is broken: the attacker can change m to some altered version m' and create a new valid MAC without knowing the secret key k. This is a length extension attack, as explained in Section 4.3.1.

The secret suffix MAC does not have this vulnerability but still has other vulnerabilities and is prone to, for example, the birthday attack [53].

HMAC

A well-known MAC method is HMAC, which can be used with any cryptographic hash function [38]. The HMAC method is used in, among others, hierarchical deterministic wallets, as explained in Section 6.2. One of the main advantages of HMAC is that it is considered secure, even if the chosen hash function is not collision resistant [4]. In fact, HMAC-MD5 (HMAC using the MD5 hashing algorithm) is still considered secure even though MD5 is considered broken (see Section 4.2.3). HMAC-SHA256 (HMAC using the SHA-256 hashing algorithm) is used in the Bitcoin hierarchical deterministic wallets, as described in Section 6.

Suppose we are given a hash function h and a key k. The HMAC for some message m is computed as follows. Let l_1 be the block size (in bytes) and let l_2 be the output size (in bytes) of the hash function h. If key k is longer than l_1 bytes, it is hashed using h to yield an output of size l_2. In this case, the output will be used as the actual key k for HMAC. The minimal recommended key size is l_2.

First, two constants are defined, called the inner and outer paddings: `ipad` and `opad` respectively. The inner padding is the byte `0x36` repeated l_1 times and the outer padding is the byte `0x5C` repeated l_1 times. The HMAC of the message m is then equal to

$$\text{HMAC}_h(m, k) = h\left((k \oplus \texttt{opad}) \mid\mid \; h\left((k \oplus \texttt{ipad}) \;\mid\mid\; m\right)\right).$$

5 Digital signatures

5.1 Current digital signature schemes

5.1.1 Introduction

A special feature of cryptography is the digital signature which makes it possible to sign a message. Digital signatures can be used for three purposes [30]. First, they prove that the author indeed signed the message, so they provide authentication. Furthermore, they prove that the signed message was not altered and lastly, they make it impossible for the author to deny signing the message. In Bitcoin, all three purposes are visible: only the owner of money can spend their money. Nobody should be able to alter a transaction and nobody should be able to reclaim the money they spent. How this is achieved will be explained in the next sections.

Looking at the RSA signature scheme is not useful for the digital signatures in Bitcoin, because the RSA scheme is based on the factorization problem [60]. Signatures in Bitcoin are based on the discrete logarithm problem, which was described in Section 3. We begin with an explanation of ElGamal signatures, which was the first signature scheme that was based on the discrete logarithm problem. Then, we will explain a more advanced version of the ElGamal signature scheme: DSA. We will also cover the elliptic curve variant ECDSA and explain how Bitcoin uses ECDSA. After that, we will focus on digital signatures in the post-quantum era. We will first describe the problems that will arise and discuss one of the possible solutions: hash-based digital signatures.

Standard procedure

Every signature scheme consists of three procedures. Firstly, private and public keys must be generated. Afterwards we give a procedure to create the actual signature and a procedure to verify this signature. Next, we will describe some attacks on the scheme and comment on the signature sizes. We will use the key sizes recommended by ECRYPT-CSA in 2018 for near term protection [26]. Since we must store each signature in the blockchain, which is stored on many devices, we would like the signature size to be as small as possible. Bitcoin also stores the public key in the blockchain, so we will have a look at the public key size as well. Addresses in Bitcoin are stored in the blockchain, but those are hashes (of public keys). Hence, the storage needed for an address is simply a constant for the different signature schemes, so they are not taken into account here.

Notice that each signature requires a specific message to be signed. Since this message might be very long, it is often hashed first. Signing the hash is not only more efficient, it is also more secure against attacks, as will be explained in more detail in Section 5.1.2.

Basic arithmetic

To show the correctness of the signature algorithms that will be described in the next sections, we need some basic arithmetic. In this section we will provide a few simple lemmas that we will use in the proofs of the algorithms. These lemmas make the proofs of the algorithms easier to follow.

Lemma 5.1. *Let g, p, q, x and y be positive integers such that $g^q \equiv 1 \mod p$. Then*

$$g^{(x \bmod q)y} \equiv g^{xy} \equiv g^{(xy) \bmod q} \mod p.$$

Proof. The first equivalence follows from writing out the left hand side. For a certain integer k we have:

$$\begin{aligned} g^{(x \bmod q)y} &\equiv g^{(x+kq)y} \\ &\equiv g^{xy+kqy} \\ &\equiv g^{xy} g^{kqy} \mod p. \end{aligned}$$

As we required that $g^q \equiv 1 \mod p$ earlier, $g^{xy} g^{kqy} \mod p$ is equivalent to $g^{xy} \bmod p$. The second equivalence of the lemma follows analogously by writing out $g^{(xy) \bmod q} \mod p$. □ Q. E. D.

The previous lemma has an equivalent formulation for elliptic curves.

Lemma 5.2. *Let n, x and y be positive integers and G an elliptic curve point such that $nG = \mathcal{O}$. Then:*

$$y(x \bmod n)G = (xy)G = (xy \bmod n)G.$$

Proof. This follows from writing out the left hand side in a similar way as in the previous lemma. For a certain integer k we have:

$$\begin{aligned} y(x \bmod n)G &= y(x+kn)G \\ &= (xy+kny)G \\ &= (xy)G + (kny)G. \end{aligned}$$

As we required that $nG = \mathcal{O}$, this results in the first equivalence of the lemma. The second equivalence of the lemma follows analogously by writing out $(xy \bmod n)G$. □ Q. E. D.

5.1.2 ElGamal signatures

In 1985, Elgamal published a new signature scheme which was based on variants of the discrete logarithm problem [23]. In the next sections, we will describe the setup, signing and verification procedures of this scheme.

Setup
For this signature scheme, the following parameters are needed. These parameters may be global, i.e. they may be re-used for multiple messages by different signers.

- A sufficiently large prime number p, such that the discrete logarithm problem in $\mathbb{F}_p$ is practically unsolvable. This means that $p-1$ must have at least one large prime factor [51].
- A generator g of the field $\mathbb{F}_p$.

Furthermore, the signer has a private key $k \in [1, p-2]$. This key can also be re-used to sign more messages. Sometimes $k = 0$ is also allowed. This removes some randomness while generating the signature, since we multiply with zero in this case. Hence, we will avoid this situation. The public key is derived with $K \equiv g^k \mod p$. Due to the assumed difficulty of discrete logarithm problem, it is very hard to calculate private key k given public key K, if p is chosen as described above.

Signing
Using the defined parameters p and g, we can now sign a message m where we assume that m is an integer. The resulting signature will always consist of two integers which are called r and s. ElGamal defined a valid signature using the equation

$$g^m \equiv K^r r^s \mod p. \tag{12}$$

A valid signature for m is a pair of integers (r, s) satisfying Equation 12.

To sign a message, we must derive values for r and s using the private key k. The first step, however, is to generate an ephemeral $z \in [1, p-2]$ such that $\gcd(z, p-1) = 1$ with trial and error. This condition is necessary because we will later need to calculate the inverse of z modulo $p-1$ where $p-1$ is not prime.

The ephemeral z must be chosen randomly and can never be re-used for another message. If z is re-used, the private key k can be derived even without knowing the value of z [23].

With this ephemeral z, we can generate the signature (r, s) in the following way:

$$\begin{aligned} r &\equiv g^z \mod p, \\ s &\equiv (m - k \cdot r) \cdot z^{-1} \mod (p-1). \end{aligned}$$

Notice that the inverse of z exists, since we required that $\gcd(z, p-1) = 1$. Furthermore, notice that we generate s modulo $p-1$ and not modulo p. The reason why will become clear when we show the correctness later.

Verification
To verify a signature (r, s) of a message m, we need the public key K and global parameters p and g. We must check that $0 \le r < p$ and $0 \le s < p$ to make sure we cannot forge signatures [45]. Then we check whether Equation 12 holds. If both conditions are true, the signature is valid and otherwise it is not.

Correctness
To show the correctness of this algorithm, we fill in public key K and signature part r in the right hand side of Equation 12.

$$
\begin{aligned}
K^r r^s &\equiv (g^k \bmod p)^r (g^z \bmod p)^s \\
&\equiv (g^k)^r (g^z)^s \\
&\equiv g^{kr+zs} \mod p.
\end{aligned}
$$

We will now substitute $s = (m - kr) \cdot z^{-1} \mod (p-1)$. Fermats little theorem tells that $g^{p-1} \equiv 1 \mod p$ if $\gcd(g, p) = 1$. This implies that $g^k \mod p \equiv g^{k \bmod (p-1)}$ which allows us to simplify the formula.

$$K^r r^s \equiv g^{kr+z(m-kr)z^{-1}} \equiv g^m \mod p.$$

This is equal to the left hand side of Equation 12, which means that we indeed generated a valid signature.

Attacks
It is nice to have a procedure to generate valid signatures, but we also have to make sure that there are no attacks against this system. It must not be possible to forge a signature (r, s) without knowing private key k and it must not be possible to derive a private key k from the signature or public key. ElGamal described these kind of attacks in his original paper [23]. Most attacks are difficult, but it is possible to derive other messages and signatures, given a valid signature for a message. This can be avoided by hashing a message before signing it. Pointcheval describes the resulting security improvements in more detail [52].

Efficiency
A disadvantage of this signature scheme is the size of the signature, because both r and s are integers in the range $[0, p]$. Furthermore, in Bitcoin we also store the public keys in the blockchain, one for each signature. The public key K is within the same range, resulting in three integers with the same size as p. ECRYPT-CSA recommends a size of at least 3072 bits for prime p [26]. Since both r and s are integers in the range $[0, p]$, we would need almost 768 bytes to store a signature plus 384 bytes for the public key, giving a total of 1152 bytes. This is quite a lot of space, since every Bitcoin transaction input stores a signature and a public key. In order to make signatures smaller without making the scheme less secure, DSA was developed [34].

5.1.3 DSA

While the ElGamal signature scheme was quite intuitive, the DSA signature scheme is more complex. The idea is to work in a subgroup of $\mathbb{F}_p^*$ of prime order q [34]. As Antonopoulos describes, this is done under the assumption that the discrete logarithm problem does not become easier in this subgroup. Hence, the scheme will stay equally secure, but the size of the signature is reduced. In the next sections, we will explain the setup, signing and verification procedures. We will spend more time on the correctness now, since it is a bit less trivial than in the ElGamal signature scheme.

Setup
We need the following global parameters. Similar to ElGamal, these global parameters (and keys) may be re-used for multiple signatures.

- A sufficiently large prime p such that the discrete logarithm problem is practically unsolvable. See Section 5.1.2 for more details about this requirement.
- A prime $q < p$ such that $p \equiv 1 \mod q$.
- A generator g which has order q in $\mathbb{F}_p$. In other words, q is the smallest number larger than 0 such that $g^q \equiv 1 \mod p$.

We now generate a private key $k \in [1, q-1]$, which can be used to sign multiple messages. If we pick $k \equiv 0 \mod q$ here, we would get serious problems, because not all messages can be signed in that case. The public key K is still derived with $K \equiv g^k \mod p$. Notice that we pick a private key smaller than q, while we calculate the public key modulo p. This results in a small private key and a large public key. The public key must be calculated modulo p, because q is the order of g in $\mathbb{F}_p$. However, this also means that we can pick k smaller then q: if k would be larger than q, then $K \equiv g^k \equiv g^{k-q} \mod p$.

Signing
The basic idea is quite similar to the ElGamal scheme. Again we have a message m to sign and we have an equation which a valid signature must satisfy for given m. However, the equation where DSA is based on is more complicated than the ElGamal scheme. The signature still consists of two parts called r and s. A valid signature (r,s) for m is a signature that satisfies the equation:

$$r \equiv g^{u_1} K^{u_2} \bmod p \mod q. \tag{13}$$

In this equation, u_1 and u_2 are defined as:

$$u_1 \equiv m \cdot s^{-1} \mod q,$$
$$u_2 \equiv r \cdot s^{-1} \mod q.$$

We will now give the procedure to create a signature with the DSA scheme. In order to do that, we need an ephemeral $z \in [1, q-1]$, which cannot be re-used. If we would pick $z \equiv 0 \mod q$, we would not be able to calculate the inverse of z. The signature is generated almost similar to the ElGamal scheme:

$$r \equiv g^{z} \bmod p \mod q,$$
$$s \equiv (m + kr) \cdot z^{-1} \mod q.$$

Recall that the inverse of z modulo q will now always exist for non-zero z, since q is a prime number. Similarly, s^{-1} will always exist for non-zero s. Since s is not allowed to be 0, we must ensure that also k is not. If we would sign a message $m \equiv 0 \mod q$ with a private key $k \equiv 0 \mod q$, it will always result in s being 0. We would loop forever trying to find a value for z that would give a valid value for s. This is why the private key k must be non-zero. Usually r is not allowed to be equivalent to zero either [45], but this would not result in severe problems.

Why this procedure results in a valid signature will be explained when we analyze the correctness of DSA.

Verification
To verify a signature (r,s) for a message m, we need to fill in the signature, message, public key and global parameters in Equation 13. Furthermore, it must hold that $0 < r < q$ and $0 < s < q$. If Equation 13 holds and both r and s are in the specified range, the signature is valid.

Correctness
To show the correctness of this algorithm, we begin with rewriting $K^{u_2} \mod p$. We first fill in K and u_2 and then apply Lemma 5.1. The last one is applicable since we required that $g^q \equiv 1 \mod p$.

$$\begin{aligned} K^{u_2} &\equiv \left(g^k \bmod p\right)^{rs^{-1} \bmod q} \\ &\equiv g^{k\left(rs^{-1} \bmod q\right)} \\ &\equiv g^{krs^{-1} \bmod q} \mod p \quad \text{(after applying Lemma 5.1).} \end{aligned}$$

Hence, after filling in $u_1 \equiv m \cdot s^{-1} \mod q$:

$$g^{u_1} K^{u_2} \equiv g^{\left(s^{-1} \cdot (m+kr) \mod q\right)} \bmod p \mod q.$$

Recall that $s^{-1} \equiv z \cdot (m + kr)^{-1} \mod q$. While applying Lemma 5.1 in the last step, this results in:

$$g^{\left(s^{-1} \cdot (m+kr) \bmod q\right)} \equiv g^{z \bmod q} \equiv g^{z} \bmod p \mod q.$$

By definition, this is equal to r, meaning that this procedure is indeed correct and gives valid signatures.

Attacks
Similarly to the ElGamal signature scheme, usually the message is hashed first [60]. This will prevent any attack that could forge signatures by changing the message algebraically.

Efficiency
Compared to the ElGamal scheme, DSA gives smaller signatures. Both r and s are calculated modulo q, resulting in smaller numbers. Similar to the ElGamal scheme, ECRYPT-CSA recommends 3072 bits for the size of p, while a 256-bit prime suffices for q [26]. This makes the signature only 64 bytes long. When we only need to store signatures, this is very efficient, but in Bitcoin also the public keys are stored. Unfortunately, the public key K is calculated modulo p. To store it, we need another 384 bytes, summing up to 448 bytes in total. It is still much smaller than the ElGamal scheme, but it can be improved with ECDSA.

5.1.4 ECDSA

The elliptic curve variant of the DSA scheme introduces an elliptic curve over a finite field. We no longer look at a finite field of order p, but at an elliptic curve defined over such a finite field. The problem on which this scheme is based, is the elliptic curve discrete logarithm problem. This is a variant of the discrete logarithm problem over finite fields. To solve the problem over finite fields, subexponential algorithms are known [37]. This is not the case for the problem over elliptic curves when the parameters are well-chosen, making ECDSA a stronger algorithm. Similarly to the earlier discussed schemes, we will begin with describing the setup, signing and verification procedures. We will also discuss attacks against this scheme and look at the efficiency.

Setup
As a start, we need an elliptic curve E. We also need a prime number p defining the underlying field. Our generator is now a point G on the elliptic curve. The order of G is called n. It is advisable to make sure that n is prime: when we verify a signature, we must invert integer s in the signature modulo n. If n is prime, this is always possible. However, if we work with an order n that is not prime, the signer must make sure that this integer s is actually invertible, so $\gcd(s, n) = 1$. If this is not the case, he must pick another ephemeral to generate another signature for the same message. He keeps doing this until he finds a signature part s that is invertible. The size of n is only important for security. Johnson provides more information about conditions for these numbers [37].

The private key is a random integer $k \in [1, n-1]$. It cannot be equal to 0 for the same reason that we saw with the DSA scheme. The public key is calculated using scalar multiplication over the elliptic curve:

$$K = kG.$$

Signing
As we are now signing over an elliptic curve, the equation that signatures must satisfy is also changed compared to the DSA scheme. The signature is still a tuple (r, s) where r and s are integers between 0 and n (exclusive). The hash of the message to be signed is called m. The equation which the signature and the hash must satisfy is now:

$$r = \left((ms^{-1} \bmod n)G + (rs^{-1} \bmod n)K\right)_x \mod n. \tag{14}$$

The x subscript denotes taking the x-coordinate of an elliptic curve point. Notice the similarities with the DSA signature scheme. The generator and public key are now elliptic curve points, hence the modular exponentiation is replaced with elliptic curve (scalar) multiplication. Recall that the elliptic curve discrete logarithm problem is harder than the problem over regular finite fields, making this variant even more secure.

Also notice that this is not the exact elliptic curve equivalent of Equation 13. Such an equation would compare two elliptic curve points. However, only checking one coordinate suffices. This creates a smaller signature compared to storing the entire elliptic curve point, because we only store one of the two coordinates. This is why we take the x-coordinate of the elliptic curve point on the right hand side of Equation 14.

To calculate values for r and s, we will need an ephemeral $z \in [1, n-1]$. As usual, this ephemeral must not be re-used and we must be able to calculate the inverse of z modulo n. If n is prime, this is always possible. Else, we must require that $\gcd(z, n) = 1$.

We can now calculate a signature in the following way:

$$r = (zG)_x \mod n,$$
$$s = z^{-1}(m + kr) \mod n.$$

We need to calculate s^{-1} modulo n in the verification, so we must make sure that $\gcd(s,n) = 1$. Hence, s is not allowed to be 0 if n is prime. If n is not prime, we must check whether $\gcd(s,n) = 1$. Also, r is often not allowed to be 0 to prevent attacks [37]. If r or s would be equal to 0, or $\gcd(s,n) \neq 1$, another ephemeral should be chosen. We will later show the correctness of this procedure.

Verification
We first need to make sure that both r and s are in the range $[1, n-1]$. After that, we fill in m, r and s in Equation 14. The signature is valid if and only if Equation 14 is satisfied.

Correctness
Showing the correctness of ECDSA is quite similar to showing the correctness of DSA. We begin with rewriting the public key K. Afterwards, we apply Lemma 5.2, which is possible since we required that $nG = \mathcal{O}$:

$$\begin{aligned}(r \cdot s^{-1} \bmod n)K &= k(r \cdot s^{-1} \bmod n)G \\ &= (k \cdot r \cdot s^{-1} \bmod n)G.\end{aligned}$$

Now we can fill in $s^{-1} = z(m + kr)^{-1} \mod n$ in the right hand side of Equation 14:

$$\begin{aligned}\big((ms^{-1} \bmod n)G + (rs^{-1} \bmod n)K\big)_x \bmod n &= \big((ms^{-1} + krs^{-1} \bmod n)G\big)_x \bmod n \\ &= \Big(\big((m + kr)s^{-1} \bmod n\big)\, G\Big)_x \bmod n \\ &= ((z \bmod n)G)_x \bmod n \\ &= (zG)_x \bmod n \quad \text{(Lemma 5.2)} \\ &= r.\end{aligned}$$

This means that the generated pair (r,s) is indeed a valid signature for m.

Attacks
As stated earlier, the elliptic curve discrete logarithm problem is harder than the traditional one, which makes keys stronger per bit. Johnson describes a list of known attacks [37].

Efficiency
For ECDSA, the size of the signature depends on the order n of generator G. In Bitcoin, both n and p are 256-bit primes [1]. The reason the prime can be smaller than the (large) prime in the ElGamal and DSA schemes is a result of the more difficult discrete logarithm problem. However, now order n is close to p, while q is (on purpose) much smaller than p in the DSA scheme. Hence, we no longer have a small prime number to reduce the size of the signatures, like DSA has. Both r and s will be 256-bit numbers now, meaning that we need 64 bytes to store them. This is similar to the amount needed with the DSA scheme, but the size of the public key decreases significantly. In its compressed form, the public key K is now a 256-bit number (plus one extra bit for the sign). In total, we can store the signature and the public key in less than 100 bytes.

5.1.5 Public key recovery

The ECDSA signature scheme has a special property. If the elliptic curve is well-chosen, it is possible to recover the public key from the message m and the signature (r,s) [20]. A remarkable detail is that this public key recovery is not possible in the DSA signature scheme, although ECDSA and DSA are (almost) equivalent signature schemes that operate in different groups.

A small recap
To see why it is not possible in the DSA scheme, we recall how we calculated part r of the signature. In DSA, we needed system parameters p, q and g and an ephemeral z to calculate:

$$r \equiv (g^z \bmod p) \mod q.$$

In ECDSA, we use elliptic curve addition with generator G of order n and an ephemeral z:

$$r \equiv (zG)_x \bmod n.$$

We can see the similarities here: calculating $g^z \bmod p$ in $\mathbb{F}_p$ is similar to calculating zG over an elliptic curve (defined over finite field $\mathbb{F}_p$). In both schemes, we reduce the result modulo the order of the generator. The only difference so far is that we only take the x-coordinate of the elliptic curve point in the ECDSA scheme, but this does not really affect the key recovery.

Recovering zG from r in ECDSA
To recover the public key, we must be able to recover $g^z \bmod p$ from r in the DSA scheme or recover zG from r in the ECDSA scheme: if we know this value, or a few candidates for this value, we will see later that we can easily calculate the public key. We will first look at how this is possible in the ECDSA scheme. When choosing the parameters of the elliptic curve, one should make sure that the order of G (which is called n) is close to the amount of elements in $\mathbb{F}_p$ [37]. This is the case for the elliptic curve that Bitcoin uses, so we can assume that for example $2n > p$. Rewriting the definition of r gives that there exists an $i \geq 0$ such that

$$(zG)_x = r + in < p,$$

and since $2n > p$, we know that $i \in \{0, 1\}$. Thus, we have only two options for $(zG)_x$ (where $i = 1$ will only rarely occur, since n is very close to p). If we know $(zG)_x$, we can substitute this coordinate in the defining equation of the elliptic curve to find two possibilities for the point zG. This is similar to the conversion from a compressed to an uncompressed coordinate which is described in Section 3. Hence, we only have four candidates for the point zG using the elliptic curve that Bitcoin uses.

Recovering $g^z \bmod p$ from r in DSA
Now we try to recover $g^z \bmod p$ from r in the DSA scheme. Rewriting the definition of r gives that there exists an $i \geq 0$ such that

$$g^z \bmod p \equiv r + iq < p.$$

However, now $p \approx 2^{3072}$ is much larger than $q \approx 2^{256}$, giving us approximately 2^{2816} possibilities for i. Of course, it is infeasible to try all these possibilities for i, which is why public key recovery is not possible in the DSA scheme.

Recovering the public key in ECDSA
We will now show how to recover the public key in ECDSA, given the elliptic curve point $R = zG$. We can follow this procedure for each of the (at most) four candidates for R, and check whether we found the correct public key by verifying the signature. The procedure only consists of a simple calculation:

$$K = r^{-1}(sR - mG).$$

We can see why this is correct in the following way:

$$\begin{aligned} r^{-1}(sR - mG) &= r^{-1}(sz - m)G \quad \text{after filling in } R = zG \\ &= r^{-1}((z^{-1}(m + kr) \bmod n)z - m)G \quad \text{after filling in } s \end{aligned}$$

After applying Lemma 5.2 we find:

$$\begin{aligned} r^{-1}(sR - mG) &= r^{-1}(m + kr - m)G \\ &= kG \\ &= K. \end{aligned}$$

The main advantage of this procedure is that it is no longer necessary to store the public key, since it can always be derived from the signature. We will see how this relates to Bitcoin in Section 5.1.6.

5.1.6 Bitcoin ECDSA

In Bitcoin, digital signatures are used to prove ownership of funds. In this section we will explain how Bitcoin uses ECDSA for digital signatures.

General idea
As discussed briefly in Section 2, a transaction consists of inputs and outputs. The inputs specify which money you are spending. The outputs specify to whom you are going to spend the money.

When creating an output, you must specify two things: the amount (in bitcoin) that you want to pay and the person who receives the money. In the most simple way, this person would be identified with their public key K. Then each output would consist of an amount and a public key. This approach is a bit more simplistic than the approach Bitcoin uses, but easier to understand.

When the user would like to spend money, the user must reference an output that transferred money to them. This is done in the input of a transaction. The user must prove that he actually owns the funds. This is done with ECDSA. The message to be signed is the new transaction that is being created: all inputs and outputs, without the signatures of the inputs. This forms a message m. Bitcoin has more advanced ways of signing where only a specific part of the transaction is signed. For Brabocoin this is not very useful, so we always sign the complete transaction as explained in more detail in Section 7.11. Now we follow the procedure described in Section 5.1.4, creating a signature (r,s). The signature is added to the input of the created transaction. Anyone can now verify that this person is indeed the owner of the funds, using this signature together with public key K, which is stated in the original output. This is described in Section 5.1.4.

Notice that it is very important to make sure that the ephemeral, which is used while creating the signature, is not re-used later. Since the value r of the signature is the same when someone re-uses an ephemeral, it is not difficult to detect this. In the past, Bitcoin users have sometimes used the same ephemeral. Christin [14] investigated this. He discovered that someone already discovered this and stole money from several users that re-used an ephemeral.

A bit more advanced
As suggested earlier, this straightforward approach is a bit more simplistic than the approach Bitcoin uses. It could be possible that suddenly, ECDSA is completely broken. For example, a large quantum computer might be developed, as will be explained in more detail in the next sections. In that case, it becomes very easy to calculate a private key from a public key. As a result, someone with a quantum computer can now spend all unspent outputs, even outputs that he does not own.

Quantum computers cannot solve all hard problems. In particular, it is currently not possible to calculate the pre-image of a hash very quickly even with a quantum computer. This is why Bitcoin uses an extra layer of security. We do not pay bitcoin directly to a public key, but rather to the hash of a public key, which is called an address [1]. If ECDSA would suddenly be broken, someone cannot directly spend all bitcoin, because we cannot calculate the public key from the address. In Bitcoin, the public key used to be stated together with the signature pair, to make it possible to verify the signature. This is not necessary, as we saw in Section 5.1.5, but the creators of Bitcoin did not know about public key recovery in ECDSA at the time of creation. Public key recovery was added to the Bitcoin Core in version 0.10 [7].

Notice that after revealing the public key in a new transaction, we can again calculate the private key if ECDSA would be broken. This is a problem, but only if an address is used more than once. Otherwise, there are no funds left to claim when we calculate the private key.

Creating a signature in this advanced setting (without public key recovery) is similar to the simple case. We only state public key K with the signature (r,s) to make sure that others can verify the signature, because it is not possible to recover public key K from address A. When verifying a signature (r,s,K) for address A, we must verify two things now. Besides the normal verification, we must now also verify that A is indeed the hash of K.

Public key recovery in Bitcoin
Since public key recovery is possible in ECDSA, as explained in Section 5.1.5, storing the public key K together with the signature (r,s) is not necessary. The public keys have the same length as the signature parts, so the signature size is now reduced by a third. Bitcoin uses another small optimization: an extra byte is stored to indicate which of the four candidate public keys is the actual public key. This makes verifying signatures faster, since we know directly what the value of zG is. When we verify a signature now, we calculate the public key (which is uniquely determined by the extended signature) and check

whether the signature is valid. Furthermore, the hash of the public key must be equal to the address that is referenced by the input with this signature.

5.1.7 Issues in the post-quantum era

Development of a large quantum computer will have huge consequences for the discussed digital signatures. So far, two important algorithms for quantum computers have been discovered. Grover's algorithm makes brute-forcing faster. We can now search through n possible solutions in only $\sqrt{n}$ time [31]. This makes, for example, hashing algorithms weaker, since it is now easier to calculate the pre-image of a hash. However, hashing algorithms are not completely broken. If we switch to hashing algorithms that output a longer hash (twice the original size), the level of security is similar to the level in the pre-quantum era. Mining, on the other hand, will become unfair since someone with a quantum computer mines much faster due to Grover's algorithm. It could be replaced by other techniques [30]. The large problem for digital signatures (and other types of cryptography) is caused by Shor's algorithm. Shor's algorithm makes solving the discrete logarithm problem easy and thus breaks the discussed digital signatures [55]. Also the factorization problem used in the well-known RSA scheme can be solved easily, so this is not an alternative. In the next sections, we will discuss digital signature schemes that will not be broken by quantum computers.

5.2 Hash-based digital signature schemes

5.2.1 Introduction hash-based signatures

Hash-based signatures are one of the alternatives for the discussed digital signature schemes, because hash-based signatures do not become insecure in the post-quantum era. These are also based on public key cryptography. The public key is derived from the private key with hashing. To create a signature, a part of the private key must be revealed. This means that you can use each private key only once. That is why these signature schemes are often called 'one-time signatures'.

The security of hash-based signature schemes is well understood: it only depends on the security of the hash function that is used [30]. Unfortunately, the signature size is very large compared to the earlier discussed signature schemes. Some optimizations exist, which will be discussed in the next sections. There exist post-quantum signature schemes where the signature size is smaller. Their main disadvantage is that their security is currently not well understood.

Besides the large signature size, hash-based signature schemes have one other disadvantage: we can only use an address once. If we would use an address twice, so we sign two different message with the same private key, then others can forge signatures for many other messages as well. This means that others can try to alter data in the transaction before propagating the transaction, since it might be possible to find a valid signature. In some schemes, keys can be used more often, but still a fixed number of times. However, despite the disadvantages, it is advised to switch to these kind of signature schemes [3].

5.2.2 Notation

We use k and K for the private key and public key, respectively. The message is denoted with m and is, if not specified, written in bits (base 2). Note that this is a hash of the original message. Let m_i denote the i-th bit of the message. n is the number of bits in the message. In Bitcoin, this is typically $n = 256$, since it is a SHA-256 hash. If m is written in a different base b, we denote it with $(m)_b$. The message consists of *digits* when we write it in another base. The i-th digit is denoted with d_i which is seen as an integer from the set $\{0, 1, \ldots, b-1\}$.

Hash-based signatures have a different structure than the (r, s) pair that we saw in other schemes. A hash-based signature is denoted as a vector s where s_i denotes the i-th signature element.

So far, we have seen 'normal' hash functions that compress the input to a fixed output length. In hash-based schemes, we will use a hash-function that preserves the length of the input: a non-compressing hash function [18]. Such a function is defined as

$$f : \{0,1\}^\ell \to \{0,1\}^\ell,$$

Sometimes, we use a non-compressing hash function that is parameterized by a key z. This key has the same length as the input and output. These kind of functions are also used to calculate MACs and are defined as

$$f : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}^\ell.$$

These functions can form a chain by re-using the result of this function as input. For example, we can form a function chain where we iterate such a hash function a number of times. Such a chain of length i with starting element x (and possibly with key z) is denoted with $c_z^i(x)$. Some of the schemes define these chains a more complex manner than simply iterating the hash function, as we will see in the following sections.

5.2.3 Lamport-Diffie one-time signature

Setup

We will need two separate private keys for each of the n bits of message m. The idea is that for each bit m_i, exactly one of these two private keys will be revealed, depending on the value of the bit. Hence, we will represent the private key k and public key K as two matrices, both of dimension $n \times 2$. In the private key, each element is a random sequence of ℓ bits. The public key can be obtained from the private key by hashing each element in the private key matrix. We use a non-compressing hash function f here, so the output length is also ℓ. So:

$$K_{ij} = f(k_{ij}) \quad \text{for all } 1 \le i \le n \text{ and } j \in \{0, 1\}.$$

Signing
A signature s is going to consist of n values $s_1, \ldots, s_n$. Each signature value s_i signs a bit m_i of the message. If $m_i = 0$, we set $s_i = k_{i,0}$, and if $m_i = 1$, we set $s_i = k_{i,1}$. In a more compact form, this means that $s_i = k_{i,m_i}$ [30]. So depending on the value of the i-th bit of m, we either pick the secret value from the first or the second column of the private key matrix. Notice that this procedure will reveal half of the private key and is therefore a 'one-time signature'.

Verification
To verify a signature $s = (s_1, \ldots, s_n)$, we must check that the hash of each value s_i is in the public key matrix at the correct location. So we must check that for each $1 \leq i \leq n$, the equation

$$f(s_i) = K_{i,m_i}$$

holds.

Example 1. We will provide a small example with 8-bit values. As non-compressing hash function f, we use the first eight bits of the SHA-256 hash.

Let the message to be signed be $m = \texttt{00101010}$. We start by generating a random private key matrix k where the length of each element is eight bits. Because the message contains eight bits, we need an 8×2 matrix. Now we calculate the corresponding public key matrix K by hashing each element of the private key matrix. For example, $K_{1,0} = f(k_{1,0})$. For readability, the 8-bit integers are represented in hexadecimal notation:

$$k = \begin{pmatrix} \texttt{6f} & \texttt{7a} \\ \texttt{8c} & \texttt{4b} \\ \texttt{93} & \texttt{67} \\ \texttt{7a} & \texttt{78} \\ \texttt{55} & \texttt{7a} \\ \texttt{5b} & \texttt{6c} \\ \texttt{67} & \texttt{8e} \\ \texttt{5a} & \texttt{81} \end{pmatrix}, \quad K = \begin{pmatrix} \texttt{65} & \texttt{59} \\ \texttt{9d} & \texttt{86} \\ \texttt{d1} & \texttt{cd} \\ \texttt{59} & \texttt{2d} \\ \texttt{a2} & \texttt{59} \\ \texttt{24} & \texttt{ac} \\ \texttt{cd} & \texttt{94} \\ \texttt{bb} & \texttt{59} \end{pmatrix}.$$

We can now generate the signature vector s. Since $m = \texttt{00101010}$, the signature elements are $s_1 = k_{1,0}$, $s_2 = k_{2,0}$, $s_3 = k_{3,1}$, etc. The signature vector s for message m is now

$$s = (\texttt{6f}, \texttt{8c}, \texttt{67}, \texttt{7a}, \texttt{7a}, \texttt{5b}, \texttt{8e}, \texttt{5a}).$$

To verify the signature when the public key is provided, we must check whether the following eight equations hold:

$$f(\texttt{6f}) = \texttt{65}, \quad f(\texttt{8c}) = \texttt{9d}, \quad f(\texttt{67}) = \texttt{cd}, \quad f(\texttt{7a}) = \texttt{59},$$
$$f(\texttt{7a}) = \texttt{59}, \quad f(\texttt{5b}) = \texttt{24}, \quad f(\texttt{8e}) = \texttt{94}, \quad f(\texttt{5a}) = \texttt{bb}.$$

Notice that it is in general not possible to forge a signature for another message. If we would like to create a false signature for $m' = \texttt{00001010}$ where we flipped the third bit, we must find a pre-image of $K_{3,0} = \texttt{d1}$. When the elements of the private and public key are large enough, this is infeasible.

Notice that in this particular example, we could forge a signature for message $m' = \texttt{00101011}$ where we flipped the last bit, because the public key matrix contains collisions. A pre-image of $K_{8,1}$, which is equal to $K_{5,1}$, can be found in signature element s_5. This is also known as the 'Birthday paradox', because the chance of finding such a collision (or a collision with birthdays in a group of people) is larger than what one would expect. To prevent this, the length of the private and public key should be twice the desired bit security [35].

Correctness
The correctness follows directly from the definition of s_i. We defined s_i as:

$$s_i = k_{i,m_i},$$

so the hash of this signature part is by definition equal to the public key.

Efficiency
The value of ℓ influences the private and public key size, and also the size of the signatures. The Lamport-Diffie scheme requires a collision-resistant hash function. Thus, to ensure the recommended 128 bit security, we need $\ell = 256$ due to the birthday paradox. Thus, the public key consists of $n \cdot 2 \cdot \ell = 131\,072$ bits. The signature consists of n sequences of ℓ bits, so in total $n \cdot \ell = 65\,536$ bits. Both the public key and the signature are stored in the blockchain, meaning that we need 24 576 bytes to store one signature. For comparison, the ECDSA signature used in Bitcoin only needs 65 bytes. Fortunately, some optimizations exist, as we will see in the next sections.

Public key recovery
We could decide to still use a compressed hash (for example the SHA-256 hash) of a public key as the identifier of a recipient. Then we should store the public key together with the signature, to allow others to verify the signature. This situation is similar to the situation in Bitcoin without implementation of public key recovery, but now with a different signature scheme. However, we could save some space again by noticing that we can derive half of the public key from the signature directly. Thus, we only need to store the other half of the public key besides the signature. Now everyone is still able to construct the entire public key and everyone can verify whether the compressed hash of the public key corresponds to the address. One signature would now require 16 384 bytes storage.

Example 2. Recall Example 1 with message $m = \texttt{00101010}$. Suppose that we only know signature

$$s = (\texttt{6f}, \texttt{8c}, \texttt{67}, \texttt{7a}, \texttt{7a}, \texttt{5b}, \texttt{8e}, \texttt{5a}),$$

and address (public key hash, in this example the first eight bits of the hash of the concatenation of the public key values)

$$A = \texttt{40},$$

and partial public key (containing all values that cannot be derived from the signature)

$$K' = (\texttt{59}, \texttt{86}, \texttt{d1}, \texttt{2d}, \texttt{a2}, \texttt{ac}, \texttt{cd}, \texttt{59}).$$

We cannot verify directly whether the hash of the signature values is in the public key matrix and thus, if the signature is valid. We can, however, construct the original public key now. If bit m_i of the message is 0, element $K_{i,0} = s_i$ and $K_{i,1} = K'_i$. If bit m_i is 1, element $K_{i,1} = s_i$ and $K_{i,0} = K'_i$. Verifying the signature is now done by hashing the obtained public key matrix K (in this example, you would concatenate all values, then apply SHA-256 and lastly take the first eight bits). If this hash is equal to address A, the signature is valid.

Notice that it is still necessary to provide K', the unused parts of the public key. Address A was created by hashing the entire public key, since we did not know what the message would be at that point. If we want to verify the signature, this is now done by checking if the public key hash is indeed equal to the address. We need the complete public key to do that, so also the unused values that we cannot derive from the signature.

5.2.4 An alternative approach for the Lamport-Diffie one-time signature

The Lamport-Diffie one-time signature can be formulated in multiple ways that are equivalent. We will now provide another formulation. This formulation results in an improvement that almost halves the size of the signature. The concept is also important for slightly more difficult schemes that will be discussed in later sections.

We will only provide the general idea of this scheme together with some examples. We do not specify the exact setup, signing and verification procedures, because the underlying scheme is quite similar to the original Lamport-Diffie scheme.

Signing half the message
The idea, which does not give a secure signature scheme yet, is to only sign the 1-bits of the message. That means that we only need one of the columns of the private and public key that we defined in the earlier formulation. Thus, the public and private key are vectors with n values. The elements are denoted with k_i and K_i for $1 \le i \le n$. The signature is now created the following way. If $m_i = 0$, we leave s_i empty. If $m_i = 1$, we let $s_i = k_i$. Thus, we indeed only sign the 1-bits of the message.

However, this creates a major problem, because we can forge signatures for other messages. Suppose we know a signature s for a certain message m. If a bit m_i of the message is 1, we can change this bit to a 0 and remove the signature value s_i from the signature. Now, the signature is still valid for the changed message! We will mention two ways to fix this.

Rewriting to the Lamport-Diffie signature scheme

One obvious way to fix it, is to make sure that also the 0-bits of the message are signed. The original publication of the Lamport-Diffie signature scheme also signed all 1-bits of the 'bitwise complement message' [46]. In that message, each 0-bit is replaced by a 1-bit and each 1-bit is replaced by a 0-bit. Notice that, on the one hand, it is equivalent to use the version that was explained in Section 5.2.3. On the other hand, this is an expensive type of checksum which can be improved.

Introducing a checksum

There are smarter methods than signing all 0-bits of the message. The problem with the idea of only signing 1-bits, was that we can change a 1-bit to a 0-bit in the message. This can be prevented by adding a checksum to the message and then also signing the checksum [46]. The checksum counts the number of 0-bits in the original message (in binary). To encode that in binary, we need approximately $\log(n)$ bits. In most situations, $\lceil \log(n) \rceil$ gives the exact value of the number of digits that are needed. However, if each bit of n is 0 (and n is a power of 2), we need an extra digit. Thus, we need exactly $\lfloor \log(n) \rfloor + 1$ bits to encode the checksum.

The length of this new message m' is $n + \lfloor \log(n) \rfloor + 1$. Now we claim that we only need to sign the 1-bits in m' to get a secure signature scheme.

Suppose that we now change a 1-bit in the original message to a 0-bit, which is possible. Then, the checksum must be increased to count the number of zeros correctly. However, notice that this always means that some bit of the checksum is changed from 0 to 1: suppose that we would increase a binary number without a 0-bit becoming a 1-bit, then it must represent a number that is at most equal to the original number. We cannot add the private key value corresponding to that changed bit to the signature, so it is not possible to alter a bit in the original message anymore. Similarly, changing a 1-bit to a 0-bit in the checksum always decreases the count of the number of zeros in the original message, while it is only possible to increase them.

Example 3. We would like to sign the message $m = \texttt{00101010}$ again. We need four bits to encode the checksum of an 8-bit message, thus we need twelve private key values. The checksum equals 0101, the amount of 0-bits in m in binary. We generate the private key k, again with 8-bit values. After that we derive the public key by taking the first eight bits of the SHA-256 hash of each private key element.

$$k = (\texttt{9d}, \texttt{8f}, \texttt{8d}, \texttt{9a}, \texttt{5c}, \texttt{5f}, \texttt{60}, \texttt{95}, \texttt{5b}, \texttt{6c}, \texttt{8e}, \texttt{81}),$$
$$K = (\texttt{9d}, \texttt{5e}, \texttt{07}, \texttt{06}, \texttt{a9}, \texttt{d2}, \texttt{8d}, \texttt{5b}, \texttt{24}, \texttt{ac}, \texttt{94}, \texttt{59}).$$

Now we only sign each 1-bit of the message and the checksum. These are the third, fifth, seventh, tenth and twelfth bit. Thus, the signature becomes:

$$s = (k_3, k_5, k_7, k_{10}, k_{12}) = (\texttt{8d}, \texttt{5c}, \texttt{60}, \texttt{6c}, \texttt{81}).$$

If we would like to verify the signature, we calculate the checksum first and look at all 1-bits again. Now we check whether the signature values for these bits are correct. Hence, we must check whether the following five equations hold:

$$f(\texttt{8d}) = \texttt{07}, \quad f(\texttt{5c}) = \texttt{a9}, \quad f(\texttt{60}) = \texttt{8d}, \quad f(\texttt{6c}) = \texttt{ac}, \quad f(\texttt{81}) = \texttt{59}.$$

If these equations hold, the signature is valid.

Efficiency

After introducing the checksum, we only need $n + \lfloor \log(n) \rfloor + 1$ public and private values instead of $2n$. We again take $\ell = 256$ to obtain a bit security of 128. The public key is now $\ell \cdot (n + \lfloor \log(n) \rfloor + 1) = 67\,840$ bits long. Notice that, for the signature, we only store an ℓ-bit value for each 1-bit in the message. On average, half of the bits of the messagecei are 1-bits, so on average the signature is $\frac{1}{2} \cdot \ell \cdot (n + \lfloor \log(n) \rfloor + 1) = 33\,920$ bits long. In total, one signature with public key takes (on average) 12 720 bytes of storage now, because we must also include the public key.

Public key recovery
Just like in the original Lamport-Diffie scheme, we provide more information than necessary by storing both the entire public key and the signature. If we give a private value in a signature, we can easily calculate the corresponding public value by hashing it. Thus, we only need to store a public key value when it is not part of the signature, so when we have a 0-bit in the message. As a result, we just store an ℓ-bit value for each bit in the message. These are $\ell \cdot (n + \lfloor \log(n) \rfloor + 1) = 67\,840$ bits to store, which are 8 480 bytes.

5.2.5 Winternitz one-time signature

Signing each bit separately creates very large signatures. Winternitz developed a one-time signature scheme that is able to sign multiple bits at once. This decreases the signature size, but increases the signing and verification time. A linear decrease in signature size causes an exponential increase in signing and verification time [46].

To sign a group of bits, we write our message m in some base w where w is called the Winternitz parameter. We will now sign each digit of m in base w. Notice that m will contain $\lceil \frac{n}{\log w} \rceil$ digits, where n denotes the amount of bits in m. Typically, $w = 16$ is used, meaning that we sign groups of four bits [46].

We will first provide an overview of how this signature scheme works compared to the earlier discussed schemes. Then we will specify the setup, signing and verification procedures more precisely. We will also give an example and mention the efficiency of this scheme. Lastly, we discuss public key recovery in this situation.

Idea
The idea is that we again have a private key vector, containing $\lceil \frac{n}{\log w} \rceil$ numbers of ℓ bits each. Each private key element will sign a digit of m in base w. The public key will be a vector of the same length as the private key vector. However, each public key value is not just the hash of the private key value, but we use a function chain of iterated hash functions. We define $c^i(x)$ as the i-th hash of x using a non-compressing hash function f. More precisely, we define $c^i(x)$ recursively as

$$c^i(x) = \begin{cases} x & \text{if } i = 0 \\ f(c^{i-1}(x)) & \text{if } i > 0. \end{cases}$$

For example, if we picked $w = 16$ as Winternitz parameter, then each public key value K_i would be equal to $c^{16}(k_i)$. The reason is that for each k_i, we now have sixteen possibilities for $c^j(k_i)$ to reveal as signature, namely $j = 0, \ldots, 15$. Recall that a Winternitz parameter $w = 16$ means that each digit of the message can take sixteen different values, namely $0, \ldots, 15$. Thus, for each digit d_i in the message in base w, we reveal $c^{d_i}(k_i)$ as signature. In general, public key element K_i is $c^w(k_i)$ [46].

Unfortunately, this design has a flaw. When we receive a signature together with a message, we can increase the value of a digit by hashing the corresponding signature value once more. Now the forged signature is valid for the altered message. To prevent this, we will introduce a checksum again.

Introducing a checksum
Define the complement $\overline{d_i}$ of a digit d_i in base w as $\overline{d_i} = w - 1 - d_i$, where we see $\overline{d_i}$ again as an integer. The checksum will be the sum of the complements $\overline{d_i}$ [18]. Notice that for $w = 2$, we obtain the checksum for a binary message that we described in the previous section. Then $\overline{d_i} = 1$ if and only if $d_i = 0$, hence the checksum counts all zeros.

We saw that it is possible to increase the value of a digit, but that we cannot decrease it. Thus, we can increase the sum of the digit values, but it is not possible to decrease the sum. The checksum sums over all complement digit values. This means that it is not possible to increase the sum of the complements of the digit values. Hence, we can append the sum of the complements of the digits to the message and then sign this checksum. If we would increase a digit in a message, then the checksum decreases. That would mean that we must decrease one of the digits of the checksum which is not possible, since it is also signed. Similarly, it is also not possible anymore to increase a digit of the checksum.

Setup
Due to the checksum and the conversion to base w, the exact lengths of the public and private key are quite difficult to express. Let n_1 denote the amount of values needed to sign the message and let n_2 denote the amount of values needed to sign the checksum. Then $n_1 = \lceil \frac{n}{\log w} \rceil$, since we have this many digits. The checksum is at most

$$\sum_{\text{digits } d} \overline{d} \leq \sum_{\text{digits } d} (w-1) = n_1(w-1).$$

This means that the checksum in base w can be represented in approximately $n_2 \approx \log_w(n_1(w-1))$ digits. Similarly to the checksum for the Lamport-Diffie scheme, the precise formulation is [18]

$$n_2 = \lfloor \log_w(n_1(w-1)) \rfloor + 1.$$

This means that we have a private key vector k of length $n_1 + n_2$, where each value is a random ℓ-bit number. We also have a public key vector K of length $n_1 + n_2$ where each value K_i is defined as:

$$K_i = c^w(k_i) \quad \text{for all } 1 \leq i \leq n_1 + n_2.$$

Since we use a function chain of a non-compressing hash function, the public key elements are ℓ-bit numbers.

Signing
For each digit d_i in the message and the checksum (both in base w), we compute signature s_i as

$$s_i = c^{d_i}(k_i).$$

These values together form the signature for the message.

Verification
We must first calculate the checksum of the message. Then, to verify a signature $s = (s_1, \ldots, s_{n_1+n_2})$, we must check whether we get the public key values if we hash the signature values several times. More precisely, to verify signature s_i for digit d_i, we must hash $w - d_i$ times. So we must check that for each $1 \leq i \leq n_1 + n_2$,

$$c^{w-d_i}(s_i) = K_i$$

holds.

Correctness
The correctness follows directly from the definition of s_i. We defined s_i as:

$$s_i = c^{d_i}(k_i),$$

so if we hash this again, we find

$$c^{w-d_i}(s_i) = c^{w-d_i+d_i}(k_i) = c^w(k_i) = K_i.$$

Example 4. Once again, we would like to sign the message $m = \texttt{00101010}$ but now with a Winternitz one-time signature. We choose $w = 16$ as Winternitz parameter so we convert our message to base 16 first: $(m)_{16} = \texttt{2a}$. To encode the complement, we need another two digits. Since our message is very small, the size of the checksum is relatively large. The checksum in decimal equals $(15-2)+(15-10) = 18$, which is $\texttt{12}$ in hexadecimal.

We need to generate four private key values and the corresponding public key values: two to sign the message and two to sign the checksum. The public key values are derived by hashing the private key values repeatedly, for example $K_1 = f^{16}(\texttt{d5}) = \texttt{67}$. We find:

$$k = (\texttt{d5}, \texttt{28}, \texttt{51}, \texttt{c9}),$$
$$K = (\texttt{67}, \texttt{9d}, \texttt{3e}, \texttt{f6}).$$

w	Length message (bits)	Length checksum (bits)	Signature size (bytes)	Hash operations
2	256	9	8 480	398
4	128	5	4 256	333
8	86	4	2 880	405
16	64	3	2 144	570
32	52	3	1 760	908
64	43	2	1 440	1 463
128	37	2	1 248	2 516
256	32	2	1 088	4 369

Table 5: Signature size for different values of the Winternitz parameter.

Now we can sign our message (and the checksum). The i-th signature element is created by hashing k_i (the corresponding private key element) d_i times. So, $s_1 = c^2(\texttt{d5}) = \texttt{9a}$, $s_2 = c^{10}(\texttt{28}) = \texttt{9d}$, etcetera. This results in signature

$$s = (\texttt{9a}, \texttt{9d}, \texttt{4a}, \texttt{e1}).$$

If we would like to verify this signature for the message $(m)_{16} = \texttt{2a}$, we begin with calculating the checksum again, which was 12. Now we hash each signature value s_i for $w - d_i = 16 - d_i$ times to obtain the public key. So, the following four equations should hold:

$$c^{14}(\texttt{9a}) = \texttt{67}, \quad c^{6}(\texttt{9d}) = \texttt{9d}, \quad c^{15}(\texttt{4a}) = \texttt{3e}, \quad c^{14}(\texttt{e1}) = \texttt{f6}.$$

If these four equations hold, the signature is valid.

Efficiency

Similarly to the previous schemes, the output length of the hash function influences the size of the signature. However, now also the Winternitz parameter will influence this. Furthermore, this parameter influences the number of evaluations of the hash function needed to create and verify the signature. In Table 5, we provide an overview for different values of the Winternitz parameter. We assume a message length of 256 bits and private and public key values of $\ell = 256$ bits.

In the case of $w = 16$, we will explain the calculations in more detail. In base 16, the message contains $\frac{256}{\log(16)} = 64$ digits. To encode the checksum, we need $\log_{16}(64 \cdot 16) = 3$ digits. Thus, we have 67 signature values that each contain a 256-bit value. The signature size is therefore 2 144 bytes. The size of the public key is the same.

In this case, we need to apply the hash function 8.5 times on average for the verification of one digit. In total, we need about 570 evaluations of the hash function for the verification procedure.

Public key recovery

In this signature scheme, we can recover the entire public key from the signature. This means that it is only necessary to store the signature and the compressed public key hash (the address) in the blockchain. This halves the storage that is needed in the blockchain compared to storing both the signature and public key in the blockchain: only 2 144 bytes for the signature if $w = 16$ and the short public key hash.

5.2.6 W-OTS$^+$

Hülsing suggested an improvement of the Winternitz one-time signature scheme which he called W-OTS$^+$ [35]. The idea is very similar to the original signature scheme, but it uses some randomization elements which are included in the public key. This results in shorter signatures, because the private and public key elements can be shorter.

In this scheme, a non-compressing hash function f_z is used, which is parameterized by a key z. We again use the Winternitz parameter w to denote in which base we will sign our message. This means that we need to form a function chain to sign the digits. For each digit, we will reveal $c_z^{d_i}(x)$ for some private value x where d_i is the value of the digit.

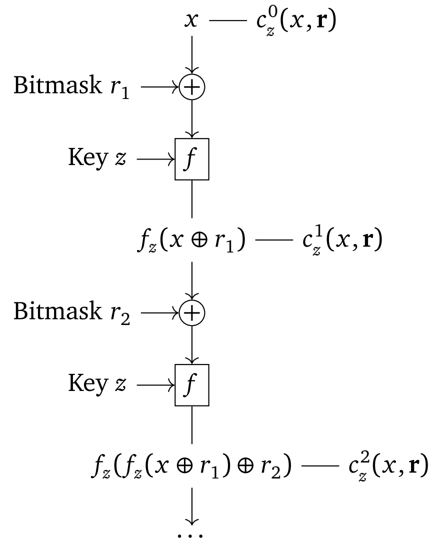

Figure 10: Function chain of W-OTS$^+$.

We could repeatedly apply f_z as chain, but W-OTS$^+$ uses a slightly more advanced technique. Before we apply f_z, we XOR the previous result with a bitmask r_i. This bitmask r_i is different for every iteration i in the chain. Thus, we need $w-1$ bitmask values, where each value is an ℓ-bit number. We denote the bitmask vector with $\mathbf{r} = (r_1, \ldots, r_{w-1})$. The bitmask vector $\mathbf{r}$ (and key z) must be revealed in the public key to make sure that anyone can verify the signature. Figure 10 shows the first iterations of this chain. Formally, this function chain is recursively defined by

$$c_z^0(x,\mathbf{r}) = x,$$
$$c_z^i(x,\mathbf{r}) = f_z(c^{i-1}(x,\mathbf{r}) \oplus r_i) \quad \text{for } i > 0.$$

At this moment we cannot see any direct improvements in the signature size, but this will follow when we analyze the bit security.

Setup

We must again use a checksum, which is defined similar to the original Winternitz scheme. The length of the message and checksum in base w are similar too. Thus, we again have a private key vector k containing $n_1 + n_2$ numbers, as defined in the previous section. Each element of the private key is an ℓ-bit integer.

The public key is now derived with function chain c_z. Each element K_i for $1 \leq i \leq n_1 + n_2$ is derived by

$$K_i = c_z^{w-1}(k_i, \mathbf{r}).$$

The public key must also include bitmask vector $\mathbf{r}$ and key z to be able to verify signatures.

Notice that we use only $w-1$ iterations here, while the original Winternitz scheme used w iterations. This is an optimization to decrease the verification time which is also possible in the original Winternitz scheme. It means that we already reveal the signatures for the digits that are equal to $w-1$. However, this does not affect the security. If a valid signature is provided, we could always increase the value of a digit by hashing the corresponding signature value. It is the checksum that makes forging a signature infeasible. Thus, it is fine to only use $w-1$ iterations: if we use this to alter the message, we get an incorrect signature for the checksum.

Signing

Let the message consist of digits d_i for $1 \leq i \leq n_1 + n_2$. The elements s_i of signature vector s are defined as

$$s_i = c_z^{d_i}(k_i, \mathbf{r}) \quad \text{for } 1 \leq i \leq n_1 + n_2.$$

Verification
To verify a signature element s_i for digit d_i, we use that $s_i = c_z^{d_i}(k_i, \mathbf{r})$. Thus, we can continue the procedure in Figure 10 until we should find the public key. This means that we must check for each digit d_i with signature element s_i and public key element K_i whether the equation

$$K_i = c_z^{w-1-d_i}(s_i)$$

holds. If, for each signature element s_i, the resulting value is equal to the corresponding public key element K_i, the signature is valid. Correctness follows directly from the definitions, similar to the correctness of the original Winternitz scheme.

Efficiency
The most interesting part of this scheme is the size of the signatures. Usually, we must choose $\ell = 256$ to achieve the advised bit security of 128. That would make this scheme even worse than the original Winternitz scheme, since we must store key z and bitmask vector $\mathbf{r}$ in the public key. Hülsing proved, however, that we can use a smaller ℓ [35]. He found the following lower bound for the bit security b:

$$b \geq \ell - \log(w^2 n + w).$$

Let $w = 16$. In Bitcoin we have $n = 256$ and want a bit security of at least 128. Thus, we need $\ell \geq 144$. This is a lot smaller than the earlier discussed schemes.

Public key recovery is still possible, so we only need to store bitmask vector r and key z together with the signature. Let $w = 16$. The size of $\mathbf{r}$ is $\ell \cdot (w-1)$ and the size of key z is ℓ. Thus, we need to store $(n_1 + n_2 + w) \cdot \ell = 11\,952$ bits in the blockchain, which are only 1 494 bytes. As comparison, the original Winternitz scheme needed 2 144 bytes to store a signature with $w = 16$. At this moment, the sixteen extra values for the mask and key are quite expensive compared to the 67 private key values. However, when we will look at hash-based signatures that can be used a few times, we will see that we can re-use this mask for other keys as well.

5.2.7 Few time signature schemes

In all earlier discussed digital signature schemes, it is important to use an address only once. Otherwise, signatures can be forged. There are also schemes that allow to re-use an address a few times. We will provide a short overview of these kind of schemes in the context of Bitcoin.

Merkle signature scheme
The Merkle signature scheme, often abbreviated to MSS, uses Merkle trees to be able to use an address multiple times [30]. Merkle trees are explained in Section 4.3.8. They are used here to create one address for a fixed amount of 'one-time signature' key pairs. The underlying scheme is not important: it could, for example, be any of the earlier discussed hash-based signature schemes. In particular, we could use W-OTS$^+$ key pairs with same bitmask without affecting the security [35].

Each leaf of the Merkle tree is the public key of such a key pair. From these leafs, we can calculate the values in the other nodes. The root of the Merkle tree is the address that we can re-use now. However, we can only re-use it a fixed amount of times: once we run out of key pairs, we must use a different address.

When we want to sign a message for a given address (which is the Merkle root), we pick a key pair that we have not used yet. We sign the message and provide the signature and public key. Notice that the public key can be replaced by the public key hash if public key recovery is possible. To prove that this signature corresponds to the address, we must also provide the Merkle path from the leaf to the root.

To verify a signature, we must first check whether the provided signature is valid for the message with the provided public key. After that, we must also check whether the public key and the Merkle path correspond to the address. If both conditions hold, the signature is valid.

Other variants
The original scheme requires a collision resistant hash function to calculate nodes in the Merkle tree. We could improve this in a similar way as in the W-OTS$^+$ scheme by making sure that a second pre-image resistant hash function suffices. The XMSS (extended Merkle signature scheme) implements this idea to create smaller signatures [17].

A major disadvantage of signature schemes like the Merkle signature scheme, is that it is necessary to keep track of which key pairs are used. For this reason, these kind of signature schemes are called *stateful* [30]. Furthermore, the amount of signatures that we can create for an address is always limited. Stateless hash-based signature schemes exist [6], but these require larger signature sizes [30]. As a result, the most feasible approach for Bitcoin would be to start using one-time signatures. A way to exchange addresses securely would be useful to support this, but that is outside the scope of this project.

6 Wallets

The term ‘wallet’ is often used in two different contexts within Bitcoin. At a high level, a wallet provides a user interface to manage keys, keep track of the user’s balance and to create or sign transactions. However, the term ‘wallet’ also refers to the data structure used to store and manage keys on a logical level.

In this section, we dive into the latter definition. We describe and compare the different techniques proposed to create, manage and securely store keys.

6.1 Wallet types

It is desirable to use a new Bitcoin address for every incoming payment as well as for receiving change for an outgoing transaction. Satoshi proposes that “a new key pair should be used for each transaction to keep them from being linked to a common owner", and supports this by stating that “the risk is that if the owner of a key is revealed, linking could reveal other transactions that belonged to the same owner" [48]. If the identity of a person is linked to an address, all transactions related to this person’s address can be observed by anyone having access to the blockchain. In fact, re-using addresses harms the privacy of all entities involved in the transaction [1]. This is one of the reasons why a cryptocurrency user may need to manage an increasing number of addresses and keys. Also, one might want to manage their funds by having different addresses for various purposes.

Before June 2016, the Bitcoin core client implemented the wallet data structure in a simple manner. This type is known as a *non-deterministic* type-0 wallet: a simple collection of keys. This means that every time a new Bitcoin address is needed, a new key is generated using a random number generator and added to the known keys of the wallet. In order to prevent the need for a backup of the wallet after every transaction, a pool of (by default) 100 keys is pre-generated and cached.

This type of wallet, also known as a JBOK (Just a Bunch of Keys) wallet, still requires frequent backup in order to store any newly generated keys after the cached keys are depleted. The Bitcoin client, which used to implement this wallet, supported encrypting the private keys inside the wallet using AES-256-CBC symmetric encryption with a user’s given passphrase. However, by doing so, the wallet loses the power of generating new keys without needing the passphrase.

In a deterministic type-1 wallet, a new key (child) is generated using a key already known to the wallet (a parent key), creating a single chain of keys. A *hierarchical* deterministic (HD) type-2 wallet differentiates itself from a deterministic wallet in that it allows for the generation of multiple child keys, given one parent. Elliptic curve arithmetic allows for the generation of new public keys without revealing the private key, as will be shown in Section 6.2.3. That is, we can generate new public keys given a parent public key (and some auxiliary data), without the need for sensitive private key data. Then, when the funds on the new public keys need to be accessed, we can derive the private keys corresponding to these public keys using the parent private key. This means we can separate the generation of public and private keys.

One of the use cases of the HD wallet is described in BIP-32, "[A deterministic wallet] permits for example a webshop business to let its webserver generate fresh addresses (public key hashes) for each order or for each customer, without giving the webserver access to the corresponding private keys (which are required for spending the received funds)" [61].

6.2 Hierarchical deterministic wallet

This section describes a scheme that allows for the generation of multiple child keys from a single parent key, as proposed in BIP-32 [61].

Arithmetic in this section is done modulo the order of the elliptic curve used in Bitcoin: secp256k1 (see Section 3.5). Whenever a byte sequence (for example, the output of a hash) is used in scalar arithmetic, it is parsed as an integer in big endian notation, most significant byte first. Whenever an elliptic curve coordinate is used as input for an operation that is not an elliptic curve operation (like appending a byte sequence), it is interpreted as a byte sequence using SEC1’s compressed form. See section 3.5 for a description on the SEC1 standard. We will denote $+_{EC}$ for elliptic curve addition, $+_S$ and $-_S$ for scalar addition and subtraction, respectively.

6.2.1 Extended keys

In order to prevent each of the derived child keys to depend solely on the parent key, BIP-32 first introduces extended keys. An extended public key (xpub) and an extended private key (xpriv) have another 256 bits of entropy appended at the end of the keys, called the *chain code*. This data is generated using a random number generator. The chain code is identical for a pair of private and public keys.

6.2.2 Obtaining the master extended private key

The generation of an HD wallet starts by obtaining a root master seed from a source of entropy. This seed can be of a chosen length between 128 and 512 bits, generated from a (pseudo) random number generator.

We now compute the HMAC-SHA512 hash of the obtained seed, using key "Bitcoin seed". For more information on the HMAC function and the SHA-256 hashing algorithm, see Section 4.3.10 and Section 4.2.4, respectively. Note that the SHA-512 hashing algorithm is an alternative to SHA-256 with a different block size and an output size of 512 bits (both belonging to the SHA-2 family of hash functions). The HMAC-SHA512 hash is considered to be the master extended private key (xpriv). We split the obtained 512-bit hash in two parts, where the first 256 bits represent the master private key and the last 256 bits represent the master chain code. This is the root node of the HD wallet.

6.2.3 Child key derivation

A description of the derivation for the extended private key of a child given a parent's extended private key will be given in this section. Also, a description for the derivation of the extended public key of a child, given only a parent's extended public key will be given. The latter is what makes HD wallets particularly useful and also what enables the webshop use case as described in Section 6.1.

xpriv → xpriv

In order to generate child keys from a given parent's extended private key, we do the following. We take the parent private key k_{par} and compute the parent public key K_{par} value using elliptic curve multiplication, $K_{\text{par}} = k_{\text{par}} \cdot G$. We now compute the HMAC-SHA512 hash H of K_{par} (interpreted as a byte sequence in SEC1 compressed form) appended with a child index value i. The key value for the hashing algorithm is the chain code c_{par} from the parent's extended private key. Again, we split the obtained 512-bit hash in two parts. The first half (256 bits), H_L, is added to the private key k_{par} of the parent, resulting in the private key k_i of the derived child. That is, $k_i = H_L +_S k_{\text{par}}$. The latter part, H_R, is the new chain code c_i for the derived child.

The child index value i is a 32 bit value, meaning we can create a total of 2^{32} children. However, by convention, we let i range between 0 and $2^{31}-1$ for this derivation. The remaining indices 2^{31} through $2^{32}-1$ are reserved for *hardened* children, as will be explained in Section 6.2.4.

The child key derivation is visualized in Figure 11.

xpub → xpub

As we will show, it is possible to generate the child's public key given *only* the parent's extended public key. Note that in the above child private key derivation, only the parent's public key K_{par} as well as the parent's chain code c_{par} was used during the generation of H. That is, the extended public key of the parent yields hash H.

Note that, for the derivation of the child's public key K_i, we have

$$\begin{aligned} K_i &= k_i \cdot G \\ &= (H_L +_S k_{\text{par}}) \cdot G \\ &= H_L \cdot G +_{EC} k_{\text{par}} \cdot G \\ &= H_L \cdot G +_{EC} K_{\text{par}} \end{aligned}$$

by the distributivity of elliptic curve multiplication. Hence, we can obtain the child's public key K_i by elliptic curve multiplication of H_L and adding the parent's public key K_{par}. Again, the chain code c_i of the child is H_R, the latter 256 bits of the hash H. The child's public key (in SEC1 compressed form) together with the child's chain code form the extended public key of the child.

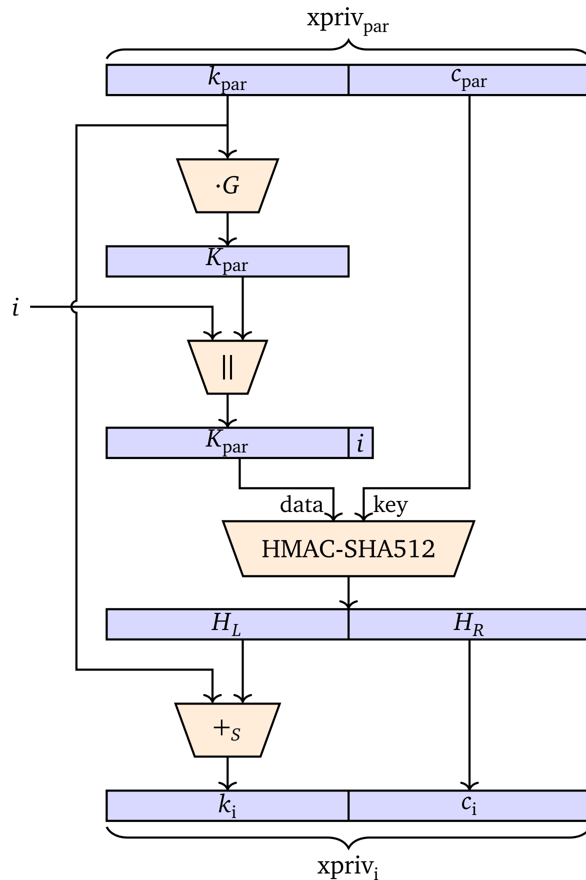

Figure 11: Child key derivation: xpriv → xpriv.

Implications
Revealing the extended public key of a node implicates the exposure of all descendants' public keys. To this end, the extended public key is considered to be more of a secret than *just* the public key of that node. Obviously, both the private key of the node as well as the extended private key are considered even more of a secret. These give access to, respectively, the funds belonging to that node's public key and the funds of all public keys in the tree rooted at that node. The latter follows from the fact that leaking the extended private key reveals the private keys of all descendants.

6.2.4 Hardened child key derivation

Motivation
A well known weakness of the above scheme is that "knowledge of a parent extended public key plus any non-hardened private key descending from it is equivalent to knowing the parent extended private key". Assume we have xpub of a parent and xpriv of a child of this parent, that is: K_{par}, c_{par}, k_i and c_i for some i. The recreation of the xpriv of the parent follows from the fact that, since $k_i = H_L +_S k_{\text{par}}$, we have $k_{\text{par}} = k_i -_S H_L$.

In order to find the parent's xpriv key, we compute H using HMAC-SHA512 on i appended to K_{par}, where the key is c_{par}, as explained in Section 6.2.3. Since k_i is known, we can compute k_{par} by subtracting H_L from k_i. Now, k_{par} together with the given c_{par} forms the extended private key of the parent.

Suppose that from a given root node, multiple children are created, each representing a separate account possibly managed by different individuals. The above implicates that if an individual's account node is compromised, i.e. its extended private key is leaked, and the root chain code is known, all other accounts are also compromised. To this end, a hardened child key derivation is proposed that removes the relationship between the parent's public key and child's chain code.

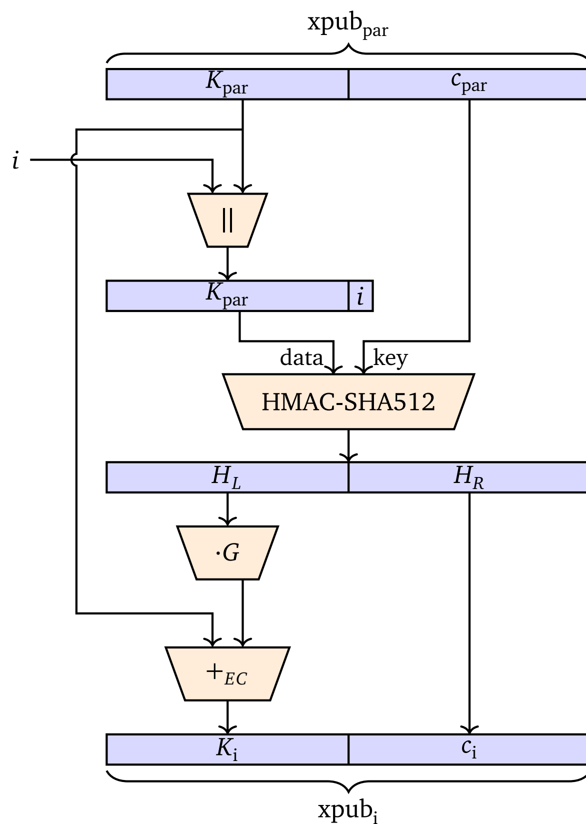

Figure 12: Child key derivation: xpub → xpub.

Hardened xpriv → xpriv
The hardened child key derivation is similar to the non-hardened child key derivation. However, instead of hashing the parent's public key K_{par} appended with the index value i, we now hash the parent's private key k_{par} appended with the index value i. The key remains the same: the chain code of the parent, c_{par}.

By convention, the indices $i = 2^{31}$ upto and including $i = 2^{32} - 1$ are reserved for hardened child key derivations.

In the hardened derivation, we require two secret values instead of only one. In the non-hardened derivation, the public key of the parent is used together with the secret chain code, while in the hardened derivation, the public key is interchanged for the secret private key of the parent. This breaks the relationship between the parent's public key and the child's chain code, since the public key is no longer used in computing the hash that yields the chain code.

Hardened xpub → xpub
This is not possible for hardened keys, as we are not able to compute the hash necessary to derive the child's public key, as done in the non-hardened version of this derivation. This is what makes hardened keys less useful.

Hierarchy
The use case of a specific level in the hierarchy of keys determines the choice between hardened or non-hardened derivation. BIP-44 [50] proposes a consensus on the levels in the hierarchy of keys in a HD wallet and can be summarized as follows:

```
m / purpose' / coin_type' / account' / change / address_index
```

where the levels containing an apostrophe indicate hardened derivation. In this notation, '`m`' is the root master seed, '`purpose`' is an indicator of the usage of this specification and equals the constant `0x8000002C`. '`coin_type`' creates a subtree for every cryptocoin. The '`account`' subtree allows for

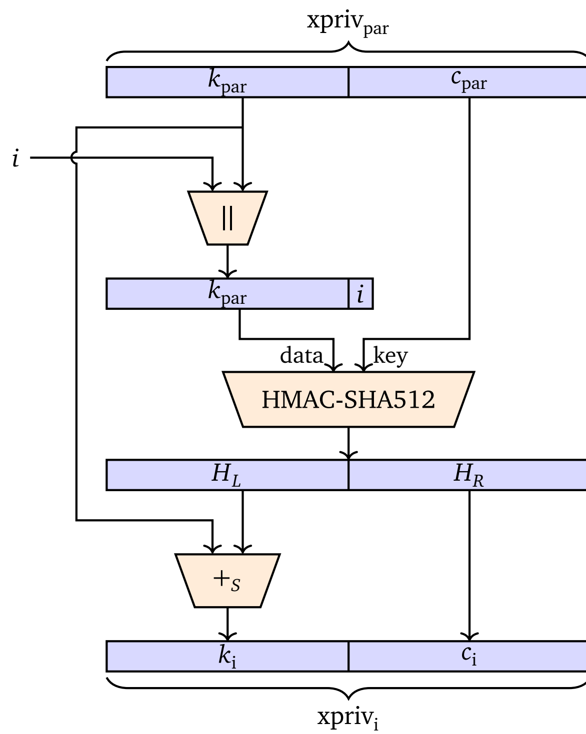

Figure 13: Hardened child key derivation: xpriv → xpriv.

different user identities. The ‘change’ level is a binary split, where the 0-th subtree is used to create addresses for external use and the other subtree is used to create addresses for returned transaction change. The ‘address_index’ level is the address index for the ‘change’ level.

7 Brabocoin

7.1 Software overview

The node software has multiple parts, where each part has a specific purpose. Users can view the state of various structures like the blockchain, transaction pool and UTXO set, but also manage their wallets, create transactions, validate blocks and transactions. The software can also be used for mining. This section gives an overview of all the functionality of a node by describing all software components of Brabocoin.

In Section 7.2 an overview is provided of Brabocoin's data model. Among others, the data structures of a transaction and a block will be described. Also, several data structures necessary for network communication and wallet serialization will be discussed.

In Section 7.3, some more advanced data structures of Brabocoin are described. These include a detailed description of the blockchain, the transaction pool and the UTXO sets. It is also discussed how orphan blocks and transactions, as well as rejected blocks and transactions are handled.

Next, Section 7.4 specifies the protocol of Brabocoin. The network design is explained and motivated, and the protocol is described.

After this, Section 7.5 describes the node environment of a node, and the way a node handles his peers. In summary, the node environment is responsible for sending request messages, and for processing incoming messages. It also maintains the set of peers of a node, and it handles orphan blocks. Lastly, it contains an initialization procedure that is used at start-up of a node. All procedures of this software component will be further described in detail in Section 7.5.

Furthermore, in Section 7.6 a list of consensus and configuration constants is provided. The constants defined in consensus are a general agreement between all the users of the network, and are thus the same for every user. The constants in configuration define the way a node is configured at start-up, according to the preferences of the user. in Section 7.6, all chosen values are discussed and motivated.

These consensus variables are mainly used in the validation of transactions and blocks, which will be discussed in Section 7.7. Transactions and blocks need to be validated in several different contexts. All validation contexts will be discussed and the validation procedures will be explained. The main deviations from Bitcoin's validation will also be pointed out.

Afterwards, the mining procedure of Brabocoin is discussed in Section 7.8. Several variables can be configured when mining a block, which will be discussed and motivated. Also, two separate mining processes are discussed: mining continuously, and mining a single block. Next, the actual mining procedure of Brabocoin is described in detail.

Next to this, the processing of transactions and blocks is discussed in Section 7.9. The processing of a block is done after a new block has been received, or a block has been mined. A transaction is processed after it is received, and again when it has been mined into a block. The procedures of processing transactions and blocks in these situations will be discussed in detail.

In Section 7.10, the wallet of Brabocoin is described. Key pair generation and management is discussed, as well as a user's balance calculation. A wallet also keeps track of a user's transaction history.

Lastly, a list of main deviations from Bitcoin is provided in Section 7.11. Every deviation is discussed and motivated. In Section 7.12 a list of possible future work is provided.

7.2 Data model

The data model of the Brabocoin software has the structure as defined in the class diagram in Figure 14.

In this section, we will first describe and discuss the data structures defined in the class diagram in Figure 14. The network message data structures used in the Brabocoin network will be defined, as well as the data structures used in the serialization of the wallet.

The Brabocoin data model consists of different data structures. Brabocoin uses Google's Protobuf for serialization, using the `proto3` syntax [27]. This allows for translating data structures into transferable or storable data.

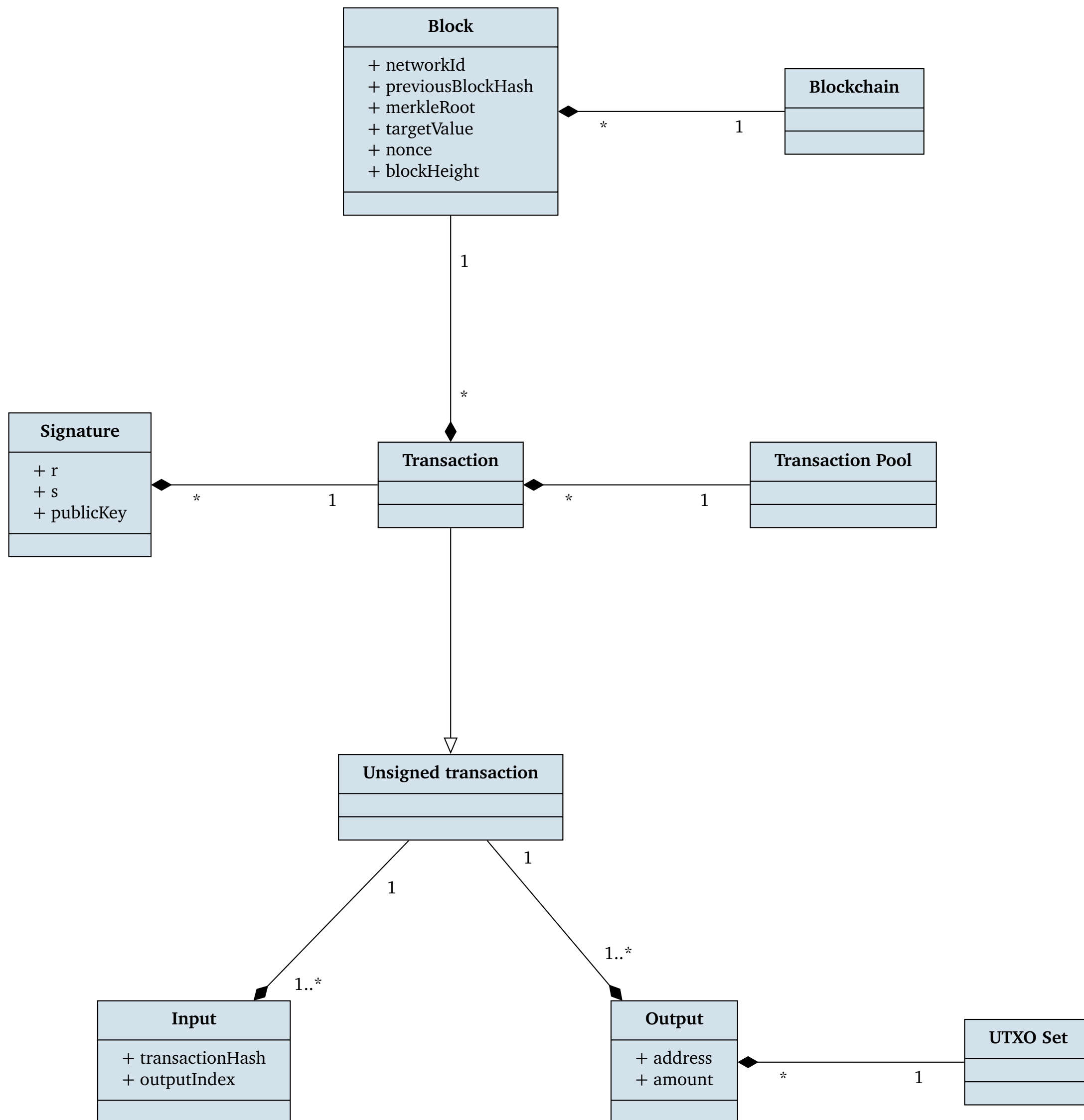

Figure 14: Class diagram of Brabocoin.

Protobuf has a number of primitive types, of which we will use a subset. The primitives `int32` and `int64` are used to describe 32 and 64 bit signed integers, respectively. The primitive `bool` is used to describe a boolean, a value that is either true or false. The primitive `bytes` is used to store an arbitrary sequence of bytes. The primitive `string` is used to represent a textual string, either UTF-8 encoded or 7-bit ASCII text.

We will now define the hash data type used in the Brabocoin software, after which the transaction and block data structures will be defined. In the definitions of these data structures, we will use bold text in the type specifier to describe a nested data structure.

7.2.1 Hash

The hash type is a wrapper for a sequence of bytes. It is used to represent transaction and block hashes, addresses and merkle roots, among others. See Table 6 for the field description of the data structure.

Field name	Type	Description
value	bytes	The bytes of the hash value.

Table 6: Hash data structure.

7.2.2 Transaction

The transaction data structure in Brabocoin consists of a number of smaller data structures, i.e. it is a nested data structure. We will first define inputs and outputs. We will then describe the unsigned transaction and signature data structures, and use these data structures to define the transaction data structure.

The main difference between the transactions defined in Bitcoin and Brabocoin is the fact that no scripting language is used in the inputs and outputs in Brabocoin, as opposed to Bitcoin [1]. This means that signatures have a fixed structure which is described in Table 10. Each input has exactly one signature now (see design decision D-10 Multi-signatures removed), which means the signatures can be removed from the the input data structure without creating difficulties linking the inputs and signatures. The list of signatures is instead a separate field in the transaction data structure. The advantage is that it is easier to sign a transaction, as we can simply build an unsigned transaction with only inputs and outputs, and sign this transaction. We can then build a real transaction using the signatures and the unsigned transaction. For more details on this, see design decisions D-9 Bitcoin's scripting language and D-11 Unsigned and signed transactions.

Input
In Brabocoin, an input is defined as the data structure described in Table 7. The main difference between the Brabocoin and the Bitcoin input data structures is the fact that there are no signatures in the Brabocoin input data structure, as mentioned before. No sequence number is present, as explained in design decision D-7 Sequence numbers removed. The Brabocoin input data structure only describes which unspent output is used, using two fields that together uniquely define the unspent output. See Table 7 for the field description of the data structure.

Field name	Type	Description
referencedTransaction	**Hash**	Hash of the referenced transaction.
referencedOutputIndex	int32	Output index of the referenced output.

Table 7: Input data structure.

Output
Outputs are defined similarly to Bitcoin as described in Table 8. Instead of a script, the recipient is identified with an address, as explained in design decision D-12 Transaction recipient addresses. This system is similar to the *Pay-to-public-key-hash* lock script used in Bitcoin. Besides recipient addresses, outputs also contain the amount (in brabocents) that is transferred to the addresses. See Table 8 for the field description of the data structure.

Field name	Type	Description
address	**Hash**	Address of the recipient.
amount	int64	Amount in brabocents to be paid to the recipient.

Table 8: Output data structure.

Unsigned transaction
The unsigned transaction is a data structure not present in Bitcoin. It was added to the Brabocoin software, in order to include signatures in the transaction data structure, instead of the input data structure (see design decision D-11 Unsigned and signed transactions for more details). The unsigned transaction contains a list of inputs and a list of outputs, as described in Table 9.

Field name	Type	Description
inputs	**Input**[]	List of inputs.
outputs	**Output**[]	List of outputs.

Table 9: Unsigned transaction data structure.

Signature
The signature data structure described in Table 10 differs from the Bitcoin implementation, since Bitcoinuses a scripting language to validate signatures (see design decision D-9 Bitcoin's scripting language). In Brabocoin, exactly one signature is calculated for every input in the transaction being signed (see design decision D-10 Multi-signatures removed). In the signature data structure, a public key is provided, which corresponds to the address defined in the unspent output, referenced in the input that is being signed. This public key is necessary to verify the signature (see Section 5 for more details on this). It also contains the actual signature, which consists of two integers. These integers are called r and s in the literature. Note that the information provided in this data structure is similar to the information provided in Bitcoin's *Pay-to-public-key-hash-script* unlock script [1].

Field name	Type	Description
s	bytes	First part of the signature.
r	bytes	Second part of the signature.
publicKey	bytes	Public key corresponding to the address in the output.

Table 10: Signature data structure.

Transaction
The transaction data structure consists of a list of inputs, a list of outputs and a list of signatures in Brabocoin, as described in Table 11. The signatures in this data structure were calculated using the unsigned transaction defined by the lists of inputs and outputs in this transaction. Also, the version number field and the nLockTime field are not present, as explained in design decision D-6 Version number and `nLockTime` fields removed.

Field name	Type	Description
inputs	**Input**[]	List of inputs.
outputs	**Output**[]	List of outputs.
signatures	**Signature**[]	Signatures corresponding to the inputs.

Table 11: Transaction data structure.

Coinbase transaction
The coinbase transaction in Bitcoin has an input with special values that indicates that the transaction is a coinbase transaction. It is not defined as a separate data structure. In Brabocoin, we made the same decision, as explained in design decision D-5 Coinbase transactions.

7.2.3 Block

The block data structure defined in Brabocoin differs slightly from the data structure used in Bitcoin. Timestamps are not present, as they are hard to validate and only useful for adjusting the target value (see design decision D-14 Timestamps). Since Brabocoin has a fixed target value, as described in design decision D-16 Target value, timestamps are not necessary in Brabocoin. Furthermore, the Merkle roots defined in the block data structure are calculated in a different way in Brabocoin, compared to Bitcoin. See design decision D-15 Merkle trees for more details on this. Next to this, the nonce field in the Brabocoin does not have a fixed size, as it does in Bitcoin. See design decision D-17 Block nonce size for more details on this. Also, Brabocoin blocks have the block height as a separate field, which Bitcoin blocks do not. See design decision D-18 Block height in header for more details on this. Finally, the Brabocoin block data structure contains a network ID field, unlke the Bitcoin block data structure. See design decision D-19 Network ID for more details on this.

7.2.4 Network messages

The transaction and block data structures can be serialized and sent over the Brabocoin network, just as the hash type. In this section, we will define some more data structures that accompany network communication between nodes. Among others, these data structures are used to establish a connection

Field name	Type	Description
previousBlockHash	**Hash**	Hash of the previous block.
merkleRoot	**Hash**	Merkle root of the transaction hashes.
targetValue	**Hash**	Target value of proof-of-work.
nonce	bytes	Nonce used in proof-of-work algorithm.
blockHeight	int64	Block height of this block.
networkId	int32	Identifier used as network ID.
transactions	**Transaction**[]	List of transactions, including a coinbase transaction.

Table 12: Block data structure.

between nodes, and to syncrhonize the blockchain of a node. See Section 7.4 for a detailed description of the protocol of the Brabocoin network and the way these data structures are used in the protocol.

HandshakeRequest
The handshake request data structure defines the handshake request message. It defines the service port and network ID of the node who requests a connection. This data structure is used as a request in a **RPC-1** Handshake message and sent over the Brabocoin network to the peer the node wishes to connect with. This **RPC-1** Handshake network message will be defined later in Section 7.4. See Table 13 for the field description of the data structure.

Field name	Type	Description
servicePort	int32	The service port of the node.
networkId	int32	The network ID of the node.

Table 13: HandshakeRequest data structure.

HandshakeResponse
The handshake response data structure defines the handshake response message. It contains the list of peers of the response node, and the network ID of the response node. This data structure is used as a response in a **RPC-1** Handshake message and sent over to the Brabocoin network to the peer who requested a connection. This **RPC-1** Handshake network message will be defined later in Section 7.4. A peer socket string is an IP address or hostname combined with a port number, separated by a colon. See Table 14 for the field description of the data structure.

Field name	Type	Description
peers	string[]	A list of peer socket strings.
networkId	int32	The network id of the client.

Table 14: HandshakeResponse data structure.

ChainCompatibility
The chain compatibility data structure is a boolean wrapper. This data structure is used as a response in an **RPC-3** CheckChainCompatible message, which will defined later in Section 7.4. See Table 15 for the field description of the data structure.

Field name	Type	Description
compatible	bool	Whether the hash is contained in the main chain of the receiver.

Table 15: ChainCompatibility data structure.

BlockHeight
The block height data structure is a wrapper for a 32-bit integer. This data structure is used as a response in an **RPC-2** DiscoverTopBlockHeight message, which will be defined later in Section 7.4. See Table 16 for the field description of the data structure.

Field name	Type	Description
height	int32	The block height value.

Table 16: BlockHeight data structure.

7.2.5 Wallet

The following data structures are used in serialization of the wallet. See Section 7.10.1 for more information on serialization of the wallet in Brabocoin.

PrivateKey
The private key data structure represents a private key (also see design decision D-30 Plain and encrypted private keys). In this data structure, the bytes are the encrypted private key bytes when the encryption flag is set to true or the plain private key bytes when encryption is set to false. See Table 17 for the field description of the data structure.

Field name	Type	Description
value	bytes	The private key bytes.
encrypted	bool	Whether the private key is encrypted or not.

Table 17: Private key data structure.

KeyPair
This data structure represents a key pair, which is a public key and a private key. See Table 18 for the field description of the data structure.

Field name	Type	Description
publicKey	bytes	The public key bytes.
privateKey	**PrivateKey**	The private key.

Table 18: Key pair data structure.

7.3 Advanced data structures

In this section, some advanced data structures of Brabocoin are described. These advanced data structures are built upon the data structures defined in Section 7.2. They include data structures of the blockchain, the transaction pool and the UTXO sets. It is also discussed how orphan blocks and transactions, as well as rejected blocks and transactions are handled.

7.3.1 Blockchain

As explained in Section 2, the blockchain is a collection of blocks organized in a tree structure. The first block in the blockchain, at block height zero, is the *genesis block*. This block has a fixed content and contains no transactions. The genesis block is distributed with the node software to ensure every node in the Brabocoin network starts with the same (genesis) block.

Other blocks in the blockchain contain a reference to a previous block in the blockchain, forming a tree of blocks. The longest chain of blocks, that is, the longest path in the blockchain starting from the genesis block, is called the *main chain*. Only the transactions that are contained in the blocks on the main chain are considered valid and can be used as inputs for subsequent transactions. If more than one chain is the longest, a tiebreaker rule is used to uniquely define the main chain (see consensus rule CS-14 `bestValidBlock`). Blocks that are recorded in the blockchain but are not part of the main chain, form a *fork* off the main chain. Note that these blocks are not considered valid and unspent outputs created in these blocks cannot be spent.

Data structure

All blocks that are recorded in the blockchain are stored in block files on disk, in serialized format. A block file contains multiple blocks, but the blocks are stored in no particular order reflecting their position in the blockchain.

Additionally, a block database is maintained, indexed by block hash, which stores for every block an information record. The format of the information record is specified in Table 19. The goal of the block database is to provide fast access to frequently used information about blocks, without the need to read the whole block from disk. All fields from the block header of a block are stored in the information record, as well as some additional information:

- Number of transactions stored in the block.
- A boolean flag indicating whether this block is valid. Initially, a block will only be stored in the block database when it passes the initial validation checks. However, additional validation is performed when an attempt is made to connect the block to the main chain. Note that this occurs after the block has already been stored. If any of these second batch of validation checks fail, the block cannot be purged from the block database and is instead marked as invalid by setting this field to `false`. See Section 7.7 for a more detailed description of the validatoin of blocks, and Section 7.9 for the processing of blocks. Also see design decision D-29 Validation stage in block database.
- Time the block was received by this node.
- File number in which the full block is stored on disk.
- Offset position and size (in bytes) of the block in the storage file.
- Offset position and size (in bytes) of the *undo data* of this block in the undo file.
- A boolean flag indicating whether this block was mined by this node.

The block database also maintains an index of the block files, indexed by file number, storing some additional information about the blocks stored in a particular file:

- Number of blocks stored in the file.
- Size of the file.
- Lowest block height of all blocks stored in the file.
- Highest block height of all blocks stored in the file.

The block files follow the naming convention `blk0000.dat`, where `0000` is an incremental counter value. The value of the counter for the most recent file is also stored in the block database.

For every block file, there exists a corresponding undo data file `und0000.dat` with the same serial number. When a block is connected to the main chain, the undo data for this block is generated, containing all the information necessary to revert the block. It therefore includes information on how to revert the chain UTXO set to the state it was before the block was connected. The undo data of a block consists of the UTXO set entries that were removed when the block was connected. Might the block be disconnected later (in a reorganizatoin of the main chain), these entries can be restored from the undo data file.

The location and maximum size of the block files, as well as the location of the block database are defined in the configuration file (see configuration constants CF-10 `DATABASE_DIRECTORY` and CF-11 `BLOCK_STORE_DIRECTORY`).

The block database contains all blocks in the blockchain, whether they are on the main chain or on a fork off the main chain. Since the blocks on the main chain need to be accessed more frequently, the information records of these blocks are stored in memory. The blocks in the main chain can be accessed by their block hash, but also by their block height.

7.3.2 Chain UTXO set

The unspent transaction output (UTXO) set maintains an index of all the transaction outputs in the blockchain that are not already spent. The chain UTXO set is a compact representation of the current

Key	Record fields
‘b’ + <block hash>	block header network ID : int previous block hash : bytes merkle root : bytes target value : bytes nonce : bytes block height : int number of transactions : int valid : bool time received : int file number: int offset in file : int size in file : int offset in undo file : int size in undo file : int mined by me : bool
‘f’ + <file number>	number of blocks : int file size : int lowest block height : int highest block height : int
‘l’	file number of last file in use : int

Table 19: Format of block database information record.

Key	Record fields
‘c’ + <transaction hash> + <output index>	coinbase : bool block height : int amount : int address : bytes
‘B’	hash of last processed block : bytes

Table 20: UTXO database structure.

state of the main chain and contains all the necessary information for validating blocks before they are connected to the main chain.

Data structure
An UTXO database is maintained on disk. For every unspent output, indexed by transaction hash and output index, we store an information record with the following fields (see also Table 20):

- Whether the output is the output of a coinbase transaction.
- Height of the block in which this output is recorded.
- Output amount.
- Address specified in the output. The owner of the private key corresponding to this address is able to spend this output.

Additionally, the hash of the block on the main chain up to which the UTXO set is up-to-date, is stored.

7.3.3 Transaction pool

The transaction pool (usually referred to as the *mempool* in Bitcoin) contains transactions that have been created but are not yet mined into a block on the blockchain. Note that the transaction pool only contains *valid* transactions, meaning that every transaction in the transaction pool can be mined into a block. The transaction pool design of Brabocoin has been greatly simplified compared to the Bitcoin implementation. In Bitcoin, the transaction pool maintains a tree-like structure of all transactions in the transaction pool, sorted by some heuristic values to maximize miner fee earnings. The optimal

selection of transactions from the pool to maximize miner profit is a complicated process. In Brabocoin, the decision was made to ignore optimal selection of transactions in blocks to simplify the transaction pool data structure (see design decision D-13 Independent and dependent transactions).

Pool UTXO set
We distinguish two types of transactions in the transaction pool:

- Independent transaction: all inputs of the transaction are already present in the chain UTXO set.
- Dependent transaction: some inputs of the transaction spend an output from a transaction still in the transaction pool.

To distinguish these types of transactions, the transaction pool maintains an internal UTXO set, called the *pool UTXO set*. The pool UTXO set maintains all unspent outputs of the transactions contained in the transaction pool. When a new transaction is added to the transaction pool, we check whether one of the inputs is present in the pool UTXO set. Whenever that is the case, the transaction is added as dependent.

Data structure
The transaction pool is stored only in memory and is not persistent, meaning that the transaction pool will be empty whenever the node starts. We maintain two separate sets of transactions, for both the independent and dependent transactions. These transactions are indexed by their hash, as well as by the transaction hash of all inputs. Because the transactions are also indexed by their inputs, it becomes possible to *promote* or *demote* transactions as independent or dependent, respectively, when the status of the parent transaction changes.

The size of the transaction pool is limited to restrict memory usage (see design decision D-25 Removing transactions). The maximum size of the transaction pool can be set in the configuration file, using constant CF-19 `MAX_TRANSACTION_POOL_SIZE`. When the maximum size of the pool is exceeded, we randomly purge transactions from the pool until the size no longer exceeds the maximum. Since all transactions in the pool need to be valid, we need to be careful to also (recursively) remove any transactions that depend on the purged transaction. We purge transactions randomly, but we make sure to first discard dependent transactions before purging independent transactions. It is important that we purge random transactions, to make sure not every node in the network purges the same transactions first. Otherwise, it becomes more likely that transactions disappear from the network and are never mined in a block (or an attack is constructed to deny a valid transaction from being mined in a block).

7.3.4 Orphan blocks and transactions

Even in a completely honest network, it might be the case that a node receives from its peers a block or transaction with an unknown parent. These blocks and transactions are marked as orphan and need to be handled properly by the receiving node.

Orphan blocks
When the *previous block hash* field in the header of the block points to a block unknown to the node, the block is considered orphan and temporarily stored in memory. As long as the parent of the block has not been processed by the node, the block remains orphan. An attempt is made to request the parent of the block from the peer that sent the orphan block, by the node environment. This procedure will be further explained in Section 7.5.7.

Since orphan blocks might be invalid, only a maximum amount of orphan blocks are stored to prevent memory exhaustion attacks. The maximum number of orphan blocks may be changed in the configuration file, in constant CF-17 `MAX_ORPHAN_BLOCKS`. When the maximum number of orphan blocks is exceeded, a random orphan is purged.

Orphan transactions
When a transaction has an input referencing a transaction output that is neither in the chain UTXO set nor the pool UTXO set, the transaction is considered orphan and temporarily stored in memory. The orphan transaction is invalid and is therefore not added to the transaction pool. Only when the missing transaction output appears in either UTXO set, the orphan transaction may be added to the pool.

Note that a malicious double-spending transaction is also considered an orphan transaction. The double spent output has already been spent by another transaction, and was therefore removed from the UTXO set. Because it is easy to deliberately create (invalid) orphan transactions, only a maximum amount of orphan transactions are stored to prevent memory exhaustion attacks. The maximum number of orphan transactions may be changed in the configuration file using constant CF-20 `MAX_ORPHAN_TRANSACTIONS`. When the maximum number of orphan blocks is exceeded, a random orphan is purged.

7.3.5 Recently rejected blocks and transactions

When validation checks fail for blocks or transactions that are being validated, these blocks and transactions are not recorded in the blockchain or transaction pool. However, they are stored temporarily in memory, together with the rule on which the validation failed. Only the most recent rejected blocks and transactions are kept and are visible to the user. The recently rejected blocks and transactions are not used by the node and merely serve an educational purpose to the user.

7.4 Protocol specification

7.4.1 Network design

For the initial version of the Brabocoin network, only one type of node will be present in the network. Every node will be a full node, with similar functionality as a full node in the Bitcoin network.

The Brabocoin network will be designed and treated as a peer-to-peer network. When starting the Brabocoin application, the only peers known to the client are the peers set in the configuration file. These are the bootstrapping peers, which we will use to bootstrap node discovery. This set of peers is later expanded with the peers of the node's bootstrapping peers, and the peers added manually by the user.

Next to this, the Brabocoin network contains a dictator node that is always running. Every node of the network is connected to this node. This node is present to ensure that every node can always connect to the network, i.e. the node always has a peer to connect to. If the Brabocoin network has grown large enough, this node can be removed.

7.4.2 Protocol description and gRPC

Every node will be able to send and receive the following messages over the network. Every message consists of a request and a response, which are data structures defined in Section 7.2. A description of the message is provided, that defines when the message is used and what requests and responses are expected.

gRPC
The Brabocoin network protocol is based on Google's gRPC framework. This framework allows for RPC's (Remote Procedure Calls) over the network. RPC is a request-response protocol, allowing a client to 'call' a procedure on a server running a service that is mutually agreed upon. In gRPC, a service is a number of named procedures, each containing one or more requests and one or more responses. These requests and responses consist of serialized Protobuf data structures, as defined in Section 7.2.

Since every node in the Brabocoin network is a full node, it implements the Brabocoin gRPC service. This allows others to use the node's service and send messages to the node, i.e allowing other to remotely call procedures defined in the service. In turn, a node uses the service of its peers to send messages into the network.

Streaming messages
In the Bitcoin protocol, a number of message types send lists of data over the network, for instance the `inv` and `getblocks` message. The replies to these messages are either a list, or a sequence of messages transferring the requested data separately, for instance the `tx` and `block` messages. Google's gRPC allows for *streaming* messages to request and respond with lists of data. Hence, the structure of these messages is slightly changed to accommodate the streaming capability. See design decision D-4 Streaming messages for more details.

We will now define the messages used in the Brabocoin network.

RPC-1 Handshake

Signature: (**HandshakeRequest**) → (**HandshakeResponse**)
Description: A node can use this message to connect to a peer in the network. The node sends a handshake request, containing its network ID and service port and receives a handshake response containing the list of peers of the peer and the network ID of the peer. The list of peers can be used to discover more peers in the network. If the peer's network ID is different from the node's network ID, a connection is not made.

RPC-2 DiscoverTopBlockHeight

Signature: () → (**BlockHeight**)
Description: With this message, a node requests from its peers the block height of their current top block on the main chain. A node can use this message to determine who has the longest main chain, and so, which peer best to synchronize his blockchain with

RPC-3 CheckChainCompatible

Signature: (**Hash**) → (**ChainCompatibility**)
Description: With this message, a node sends the hash of their current top block to its peers. If this hash is present in the main chain of a receiving peer, this peer responds with true, and false otherwise.
When the node receives the response that the block is not present in their leading fork, the node will send a new `CheckChainCompatible` message with the hash of the previous block. This process repeats until the node has received successful replies from its peer.
Thus, this message can be used to determine from what point his main chain is not up-to-date, i.e. from what block to synchronize his main chain.

RPC-4 SeekBlockchain

Signature: (**Hash**) → (stream **Hash**)
Description: With this message, a node requests from its peers any blocks that are mined on top of a specific block. The node sends a request containing the hash of the block.
A node receiving the message will look up the block corresponding to the received block hash in its main chain. It then returns all the hashes of the blocks on top of this block. Note that only blocks on the main chain are returned. If the requested block is not on the main chain, the node will return no hashes.

RPC-5 GetBlocks

Signature: (stream **Hash**) → (stream **Block**)
Description: A node can request blocks by sending a `GetBlocks` message, streaming the hashes of the requested blocks to the receiving node. The receiving node replies by sending the requested full blocks in a stream. When the receiving node does not know a block belonging to a requested hash, the response stream of blocks will not include a block with that particular hash.

RPC-6 SeekTransactionPool

Signature: () → (stream **Hash**)
Description: A node can update the transaction pool by sending the `SeekTransactionPool` message to its peers. The peers will respond by returning the transaction hashes of the transactions in their pool.

RPC-7 GetTransactions

Signature: `(stream Hash) → (stream Transaction)`

Description: A node can request the data of several transactions by sending a `GetTransactions` message, streaming the hashes of the requested transactions. The receiving node replies by retrieving the transactions from its transaction pool and sending the retrieved transactions. Only transactions from the transaction pool can be requested. When a hash of a transaction not present in the transaction pool is requested, the response stream of transactions will not include a transaction with that particular hash.

RPC-8 AnnounceTransaction

Signature: `(Hash) → ()`

Description: A node can announce a new transaction to the network with this message, which contains the hash of the new transaction.

RPC-9 AnnounceBlock

Signature: `(Hash) → ()`

Description: A node can announce a new block to the network with this message, which contains the hash of the block.

7.5 Node environment and peers

Each node creates a node environment at startup. This node environment acts as an interface for sending request messages and for processing incoming messages. It contains the higher level logic for processing messages received by other peers and contains methods to request other information from peers. It has access to the blocks and transactions processors of the node and uses these to process incoming blocks and transactions received by peers. For example, on receival of a block, the node environment invokes the block processor to process the block and decides what to do when the block has been processed. It also handles peer management by calling the peer processor of the node. All tasks of the node environment are now specified in more detail.

7.5.1 Message queue

The node environment of a node contains a message queue where pending outgoing messages are stored. This message queue is used to send outgoing messages in response to received messages. The node environment has a thread running asynchronously that processes the messages in the queue and sends them to the appropriate peer(s).

As explained in Section 7.4.2, Brabocoin uses the gRPC protocol to send and receive messages. Because incoming messages are added to the gRPC message queue and handled by the gRPC `ExecutorService` threads, we do not want to use these threads to process any logic in response to the received message. Doing so would block the gRPC thread and render the thread unusable for handling other received messages. This is why the node environment needs a message queue to handle the responses to incoming messages. If any message ought to be sent to peer(s) in response, we add this message to the message queue and finalize the received message stream, after which the thread continues to process other received messages in the queue. The response messages will be dispatched by the message queue thread in an asynchronous manner.

7.5.2 Initialization

At startup, the node environment performs a number of steps to properly setup the node. First, it initializes the set of peers used to announce messages to the network. Next, the blockchain is updated with the blockchains of the acquired peers. Lastly, the transaction pool is initialized. We will now specify these tasks individually in more detail.

7.5.3 Maintaining a set of peers

The node environment is responsible for maintaining the set of peers of the node. At startup, the node software loads the bootstrap peer socket strings specified in configuration (see configuration constant CF-8 BOOTSTRAP_PEERS). Using these sockets, the node tries to handshake with these acquired peers by sending a **HandshakeRequest** to each peer, using the **RPC-1** Handshake message. This message contains the node's own service port and network ID.

If the peer has the same network ID, it responds with his own list ofpeers, by sending a list of peer sockets contained in a **HandshakeResponse**. The peer sockets received by the node are then used to handshake and expand the known peer set further, until the desired amount of peers (as set in configuration constant CF-3 TARGET_PEER_COUNT) is reached. If the network ID specified in the **HandshakeRequest** is not equal to the peer's own network ID, the message is ignored and no response is sent. This process, starting from zero peers, is called the peer bootstrapping process.

When a node receives a **HandshakeRequest**, the peer specified in this request is also added to the set of known peers of the node. Of course, the peer is only added if the network ID in the received **HandshakeRequest** matches the node's network ID. This means that peer connections are symmetric. It should be noted that this means the target peer count specified in configuration constant CF-3 TARGET_PEER_COUNT is *not* a maximum number of peers. It specifies a target number of peers the node wishes to reach before the search for new peers is stopped.

However, it might be the case that some peers go offline or stop responding from time to time. This is why the node environment runs a thread which handshakes with each peer on a predetermined interval as set in configuration constant CF-4 UPDATE_PEER_INTERVAL UPDATE _PEER_INTERVAL. When the peer does not respond, it is removed from the set of known peers. If a peer does respond with a list of peer sockets and we have less peers than the target peer count, we handshake with those new peers and add the peers until we reach the target number of peers specified in configuration.

7.5.4 Initializing the transaction pool

After the node finished the peer bootstrapping process, the node sends a **RPC-6** SeekTransactionPool message to its peers. The peers repond with a list of transaction hashes, the hashes of the transactions that are currently in their transaction pool. The node uses the received hashes to request the actual transactions, using the **RPC-7** GetTransactions message.

7.5.5 Updating the blockchain

The node updates its blockchain by means of the **RPC-2** DiscoverTopBlockHeight, **RPC-3** CheckChain-Compatible and **RPC-4** SeekBlockchain messages, and possibly requesting missing blocks using the **RPC-5** GetBlocks message. This process is initiated for every peer that is added to the set of known peers.

Suppose node A initiates the update blockchain process with one or more peers. Node A uses the **RPC-2** DiscoverTopBlockHeight message to discover the top block height of the peers. These peers respond with the height of their current top block. The node then selects the peer with the highest top block, let this peer be node B. Node A then uses the **RPC-3** CheckChainCompatible message to pinpoint a point on the main chain from where synchronization can start. This message consists of a request containing a block hash, and the **RPC-3** CheckChainCompatible message response returns whether or not the given hash is contained in the main chain of the receiver. Node A starts by sending the hash of it's top block. If node B's chain is incompatible, node A moves further back in the chain and keeps sending block hashes until a block is found that is also on node B's main chain. The step size is doubled on each fail, resulting in an exponential backoff. When a compatible block is found, the node will send a **RPC-4** SeekBlockchain message. This message's request contains the hash of the compatible block, and the response contains the block hashes of the blocks on top of the compatible block. The node then sends a **RPC-5** GetBlocks message containing the received hashes, in order to acquire the actual blocks. These blocks are then processed, and added to the blockchain when valid.

7.5.6 Message propagation

When a node receives an **RPC-9** AnnounceBlock or **RPC-8** AnnounceTransaction message, it should propagate the message to all its peers. However, blocks should only be propagated if they are indeed

valid and a transaction should only be propagated if it is valid and it is not an orphan transaction. Therefore, a node first requests the announced block or transaction using the **RPC-5** GetBlocks and **RPC-7** GetTransactions messages. The node propagates the block or transaction only after the receival and proper processing of the block or transaction.

Note that this also guarantees that a node always has the block or transaction it announces. It could never be the case that a node receives an **RPC-9** AnnounceBlock message from one of his peers, requests the block from the same peer, and than gets the response that the peer does not have this block in his blockchain.

Note that this is a safe way to handle the announce messages. Suppose a malicious node announces a random transaction or block hashes to the network for which it does, in fact, not have the actual block or transaction. Then each peer will try to acquire the data using the appropriate 'Get' messages, but the node does not respond. The peers then simply cease to try to acquire the data.

Also, if a node tries to exclude one peer from acquiring a legitimate block or transaction, but does propagate the block or transaction to other peers, then the excluded peer will eventually receive announce messages from other peers in the network. The peer can then acquire the data from one of those peers.

Obviously, when the **RPC-5** GetBlocks and **RPC-7** GetTransactions messages are used after the **RPC-4** SeekBlockchain and **RPC-6** SeekTransactionPool messages, we do not propagate the received blocks and transactions. This prevents excessively announcing blocks and transactions that have already propagated throughout the network.

Lastly, note that orphan transactions are not propagated to peers. These transactions are only propagated after they loose their 'orphan' status, i.e. after the transactions this transaction depends on are received. Orphan blocks are also handled in a special manner, which is explained in the next section.

7.5.7 Handling orphan blocks

After receiving an orphan block from a peer, a node immediately requests the parent block from the same peer. It could be the peer already has this parent block, in which case he returns this block to the node. It could also happen that the peer does not have this block yet either. However, he will have already requested the block to one of his peers and will therefore announce (i.e. propagate) this block to all of his peers as soon as the block has been acquired. Either way, the node eventually receives the parent block of the orphan block.

It could be the case that a peer announces a block that starts a long chain of parent block retrieval messages until a block is found that is known to the peer. This is undesirable, since it loads all intermediate blocks in the the orphan block storage until they can be connected to the blockchain. When the number of orphan blocks exceeds the maximum number of orphan blocks that can be stored in memory (as defined in constant CF-17 `MAX_ORPHAN_BLOCKS`), random orphan blocks are discarded. This means the chain of blocks the node is trying to acquire is broken.

Therefore, it is preferable to update the blockchain starting at a block that is known to the node, as described in Section 7.5.5. That is why, after sequentially receiving a number of orphan blocks, retrieving a chain of parent blocks is ceased. Instead, the node updates the blockchain using the procedure described in Section 7.5.5 using all known peers. The maximum number of sequential orphan blocks before updating the blockchain is set in configuration constant CF-6 `MAX_SEQUENTIAL_ORPHAN_BLOCKS`.

7.6 Consensus and configuration

7.6.1 Consensus

This section describes the constants defined in Brabocoin's consensus, as well as tiebreak rules. The constants defined in consensus are a general agreement between all the users of the network, and are thus the same for every user. In this section, we discuss and motivate every consensus constant.

We have based these design decisions on a few requirements:

- On average, every user is able to mine one block every fifteen minutes.

- We assume that each user is able to compute 400 000 hashes per second[1]. As a result, an expected number of 360 000 000 nonce values should be tried in order to mine a block. Thus, we will use $N = 360\,000\,000$ in other calculations.
- The network should allow at least 200 students to use Brabocoin at the same time.
- The size of the blockchain must be less than 2.5 GB after two weeks, even if all students are continuously mining blocks of maximum size. This is a worst-case scenario where each student is continuously mining maximum-sized blocks. In practice, the size of the blockchain will most likely be much smaller. It is not a problem if this limit is exceeded after these two weeks, because the lectures will cover other topics at that point and students will not have to use the application anymore.

We will now define all consensus constants.

CS-1 `MAX_BLOCK_SIZE`

Description: Maximum size of a block in bytes.

Value: 10 000 bytes (10 kB)

Motivation: The maximum block size in Brabocoin is not as high as the maximum block size in Bitcoin. Bitcoin originally used a maximum block size of 1 000 000 bytes which was later increased to 4 000 000 bytes [11]. However, every node in Brabocoin maintains a copy of the complete blockchain and laptops of students often do not have a lot of available storage. Furthermore, in Brabocoin blocks are mined more frequently compared to Bitcoin. Notice that we assume that even a block without transactions can have the size of a maximum block: users could pay the coinbase transaction to an enormous, invalid address. They could also create many outputs in a transaction that each spend only a small amount of brabocoin.

We base our calculations on the requirements stated above. Each user is able to create $4 \cdot 24 \cdot 14 = 1\,344$ blocks in two weeks, so all users together could create 268 800 blocks in two weeks. As a result, keeping in mind the requirements above, the maximum block size should be approximately 10 kB. A block of this size would be able to store approximately 40 *basic* transactions, i.e. 40 transactions with one input and two outputs.

CS-2 `TARGET_VALUE`

Description: The proof-of-work target value. A newly mined block's hash should be smaller than this value.

Value: $3216 \cdot 10^{65}$

Movitation: The target value should be set according on the average amount of nonce hashes that must be calculated until we find a block hash smaller than the target value. The chance that such a hash is found should be $\frac{1}{N}$, since this results in N as the expected amount of tries to find a valid block hash. The solution space has 2^{256} possible hash results, so we only allow the lowest

$$\frac{2^{256}}{N} \approx 3216 \cdot 10^{65}$$

hash values. Thus, we set the target value to $3216 \cdot 10^{65}$.

CS-3 `MAX_NONCE_SIZE`

Description: Maximum nonce size in a block in bytes.

Value: 5

Movitation: The maximum nonce size depends on the target value defined as consensus constant CS-2 `TARGET_VALUE`. If the target value is set to a smaller value (which makes it more difficult to mine a block), the nonce size should become larger. This is a result of not including the timestamp: we cannot reuse the entire nonce space every second, as Bitcoin does. It should be highly unlikely that someone tries all possible nonces without finding a nonce

[1] This is experimentally determined by running the miner in Brabocoin on a computer with an Intel Core i7-4710MQ Quad Core CPU (2,5 GHz, 6Mb Cache, Turbo 3,5GHz) and 8 GB (1× 8GB) 1600Mhz DDR3 memory.

that results block hash smaller than the target value. The chance that we do not find such a hash in x tries equals:

$$\mathbb{P}(\text{all } x \text{ hashes invalid}) = (\mathbb{P}(\text{one hash invalid}))^x = \left(\frac{N-1}{N}\right)^x.$$

If we would have 4 bytes of nonce space just like Bitcoin, we can try $x = 2^{32}$ different values, resulting in a chance of

$$\mathbb{P}\big(\text{all } 2^{32} \text{ hashes invalid}\big) = 6.59 \cdot 10^{-6}.$$

As the performance of Brabocoin is not very important and it is not desirable to run out of possibilities for the nonce, we add one extra byte for the nonce size. This gives 5 bytes as nonce size and 2^{40} different values for the nonce. This results in a chance of

$$\mathbb{P}\big(\text{all } 2^{40} \text{ hashes invalid}\big) \approx 10^{-1328}.$$

This chance is definitely small enough.

Notice the importance of changing the nonce size if the target value changes. For example, if the target value is divided by 1 000 (which gradually happened in Bitcoin over the last five years), the calculated chance is not so small anymore [12]:

$$\mathbb{P}\big(\text{all } 2^{40} \text{ hashes invalid}\big) = \left(\frac{1000 \cdot N - 1}{1000 \cdot N}\right)^{2^{40}} = 0.0471$$

CS-4 `MAX_NONCE`

Description: The amount of nonces that can be created with CS-3 `MAX_NONCE_SIZE` bytes.

Value: 2^{8x}, where x equals CS-3 `MAX_NONCE_SIZE`.

Movitation: This value follows directly from CS-3 `MAX_NONCE_SIZE`. It is useful for mining, where we for example need to increase the nonce modulo the maximum nonce value.

CS-5 `COINBASE_MATURITY_DEPTH`

Description: The number of blocks that should be on top of a block containing a coinbase, before it can be spent.

Value: 10

Movitation: When a reorganization occurs, it is possible that validated transactions disappear from the main chain. Especially in the Bitcoin network, this is not a problem, as these transactions will usually get placed in another block not much later. However, coinbase transactions completely disappear and they will not come back (unless we would have another reorganization). That is why Bitcoin requires to wait 100 blocks until a coinbase transaction can be spent. This constant of 100 is called the coinbase maturity.

In Brabocoin, a smaller coinbase maturity is desired as students should not have to wait a long time until they can create their first transaction. We set the coinbase maturity to 10 blocks. If ten students would be mining during an instruction, it would take approximately fifteen minutes to be able to spend coinbase money. Since this value is much lower than in Bitcoin, and since we are able to mine much faster, this could mean that transactions that spend coinbase transactions get lost during a reorganization. We do not expect that this is a problem for Brabocoin due to its educational purpose.

CS-6 `MAX_MONEY_VALUE`

Description: The maximum amount of brabocents that can occur in any arithmetic within Brabocoin.

Value: $3 \cdot 10^{18}$

Movitation: The maximum value of a long is approximately $9.22 \cdot 10^{18}$. We should always prevent that this value overflows. As we only add money pairwise, the maximum amount of brabocents must be less than half of the maximum long value. To avoid off-by-one errors, we divide the maximum long value by 3 and round down. This gives a maximum amount of brabocents of $3 \cdot 10^{18}$.

CS-7 `COIN`

Description: The amount of brabocents that equal one brabocoin.

Value: 100

Movitation: To make the currency a bit more user-friendly, we use brabocoins and brabocents. Similar to most normal currencies, 100 brabocents equals one brabocoin. The very detailed scale of Bitcoin, where 10^8 satoshis are equal to one bitcoin, is not needed in Brabocoin as it will not have any real value.

CS-8 `BLOCK_REWARD`

Description: The amount of brabocents that a miner receives for mining a block.

Value: 10· `COIN`, as defined in CS-7 `COIN`

Movitation: As we do not have to create a currency that represents anything of actual value, we can use a constant as the block reward. Brabocoin uses a block reward of 10 brabocoin. The exact amount is not important for the functionality of the application.

CS-9 `MINIMUM_TRANSACTION_FEE`

Description: The minimum transaction fee required for a transaction to be valid.

Value: 1

Movitation: When we construct a block to mine in Brabocoin, we choose random transactions and do not look at the transaction fee. This means that users could simply make transactions without a transaction fee, because the transaction fee has no additional value anymore. As the transaction fee is useful for educational purposes, we require a minimum transaction fee. The value does not really matter, as long as there is a fee present in a transaction. We choose a minimum transaction fee of one brabocent.

CS-10 `MAX_BLOCK_HEADER_SIZE`

Description: The maximum block header size in bytes, excluding the nonce.

Value: 139

Movitation: This value was only needed for calculating the maximum transaction size. It is calculated experimentally by creating many blocks with random parameters.

CS-11 `MAX_COINBASE_TRANSACTION_SIZE`

Description: The maximum coinbase transaction size in bytes.

Value: 36

Movitation: This value is only needed for calculating the maximum transaction size. It was calculated experimentally by creating many blocks with random parameters.

CS-12 `CURVE`

Description: The elliptic curve used in Brabocoin.

Value: The *secp256k1* standard

Movitation: Brabocoin uses the same elliptic curve as in Bitcoin. The curve is suitable for many reasons which is further explained in Section 3.5.

CS-13 `GENESIS_BLOCK`

Description: The initial block (at height 0) of the blockchain.

Value: A block with empty previous block hash, empty Merkle root hash, empty target value hash, a zero nonce, blockheight 0, network id 0 and no transactions.

Movitation: The genesis block is specifically chosen not to depend on consensus values. We could, for example, use the target value specified in CS-2 `TARGET_VALUE`. However, consensus values can change in Bitcoin and theoretically they can also change in Brabocoin. This would mean that if our target value changes, the hash of the genesis block becomes different! Our entire blockchain would not be connected to the genesis block anymore. That is why the genesis block contains only empty hashes, zeros and no transactions. This way, it is not dependent on consensus values. Note that as a result, the genesis block is not a valid block. When verifying incoming blocks, the genesis block is a special case which is always valid.

CS-14 `bestValidBlock` rule

Description: Determines the best valid block, given a set of blocks.

Value: Filters the blocks for valid blocks, and compares their blockheight, using a block hash comparison tiebreaker (where a smaller hash value is preferred).

Movitation: This tiebreaker rule is used by the block processor when two or more forks of the blockchain are the longest chain (see Section 7.9.2 for more details). Which tiebreaker is used does not matter since they are resolved automatically. However, we must use some tiebreaker because there must be a main chain, even if there are multiple chains of maximum length. It is more difficult to keep track of which block arrived first, which is used in Bitcoin, so we pick the block with the smallest hash. Intuitively, this would be the block with the most 'proof-of-work'. Any other tiebreaker would also have been fine, as long as we always have a main chain.

CS-15 `merkleTreeHashFunction` rule

Description: The hash function used to compute Merkle trees.

Value: Double SHA-256

Movitation: We use the same hashing algorithm as Bitcoin to compute Merkle trees. Details about why double SHA-256 is used, can be found in Section 4.

CS-16 `MAX_TRANSACTION_SIZE`

Description: The number of bytes available for regular transactions within a block.

Value: `MAX_BLOCK_SIZE − MAX_BLOCK_HEADER_SIZE − MAX_COINBASE_TRANSACTION_SIZE −` x, where x is the number of bytes used in a `long` variable in Java. These variables are obtained from:

- CS-1 `MAX_BLOCK_SIZE`;
- CS-10 `MAX_BLOCK_HEADER_SIZE`; and
- CS-11 `MAX_COINBASE_TRANSACTION_SIZE`.

Movitation: We only accept transactions that can actually be placed in a block. Else, the transaction pool could get spammed easily by transactions that are too large and can never be mined. The maximum transaction size is the minimum size that is available in an empty (valid) block. We find this by subtracting the maximum size of the header and the maximum size of the coinbase from the maximum block size. Protobuf also stores the length of the transactions. That is why we also subtract the amount of bytes needed to store a `long` variable in Java, which is 8.

7.6.2 Configuration

This section describes the constants defined in Brabocoin's configuration. The constants in configuration define the way a node is configured at start-up, according to the preferences of the user. In this section, we discuss and motivate the standard configuration settings.

CF-1 `NETWORK_ID`

Description: Network ID of the network the node has joined.

Value: 1

Movitation: The main network uses a network ID of value 1.

CF-2 `SERVICE_PORT`

Description: Service port the node uses to connect to peers and listen for incoming messages.

Value: 56129

Movitation: This value is beyond the range of registered port numbers by the ICANN. This value can, however, be any port number within the range of valid port numbers (positive and less than 65535).

On a side note, the postal code of the Eindhoven University of Technology is 5612.

CF-3 `TARGET_PEER_COUNT`

Description: The target number of peers the node should be connected with.

Value: 25

Movitation: This value should be large enough to ensure fast outgoing message propagation in the network and small enough such that gRPC can handle all incoming and outgoing messages.

CF-4 `UPDATE_PEER_INTERVAL`

Description: The interval at which the node updates his set of peers, in seconds.

Value: 45

Movitation: This should be small enough to ensure the set of peers is up-to-date and large enough to ensure the gRPC threads are not blocked by processing handshake messages.

CF-5 `ALLOW_LOCAL_PEERS`

Description: Whether or not it is allowed to add `localhost` or `127.0.0.1` peers to the set of peers, allowing the user to run multiple instances of Brabocoin on one device.

Value: `false`

Movitation: By default, this behavior is not desired.

CF-6 `MAX_SEQUENTIAL_ORPHAN_BLOCKS`

Description: The number of sequentially received orphan blocks before the node starts updating his blockchain.

Value: 10

Movitation: This should at least be less than the constant CF-17 `MAX_ORPHAN_BLOCKS`, as explained in Section 7.5.7. This value should be small enough such that a node does not request a long chain of parent blocks before syncing, and it should be large enough such that a node does not unnecessarily start syncing.

CF-7 `LOOP_INTERVAL`

Description: The interval at which the node checks whether there were new incoming messages to process, in milliseconds.

Value: 500

Movitation: The value should be large enough to prevent the CPU from spinning in this thread and small enough to ensure incoming messages are processed quickly.

CF-8 `BOOTSTRAP_PEERS`

Description: Consists of the list of bootstrap peers (socket strings) the node will connect with at start-up.

Value: [brabocoin.org:56129]

Movitation: This is a server running at the Eindhoven University of Technology that can be used to bootstrap the node software.

CF-9 `DATA_DIRECTORY`

Description: The name of the directory where the user data is stored.

Value: data

Movitation: N/A

CF-10 `DATABASE_DIRECTORY`

Description: The name of the directory where the node's database is stored.

Value: db

Movitation: N/A

CF-11 `BLOCK_STORE_DIRECTORY`

Description: The name of the directory where the node's blockchain blocks are stored.

Value: blocks

Movitation: N/A

CF-12 `UTXO_STORE_DIRECTORY`

Description: The name of the directory where the node's UTXO set is stored.

Value: utxo

Movitation: N/A

CF-13 `WALLET_STORE_DIRECTORY`

Description: The name of the directory where the node's wallet is stored.

Value: wallet

Movitation: N/A

CF-14 `WALLET_FILE`

Description: The name of the file that contains the node's wallet.

Value: wallet.dat

Movitation: N/A

CF-15 `TRANSACTION_HISTORY_FILE`

Description: The name of the file that contains the node's transaction history.

Value: txhist.dat

Movitation: N/A

CF-16 `MAX_BLOCK_FILE_SIZE`

Description: The maximum size of a block storage file, in bytes.

Value: 128000000

Movitation: This is 128 MB, a file size that is small enough to allow common file systems to store and large enough to prevent creating a large amount of block storage files.

CF-17 `MAX_ORPHAN_BLOCKS`

Description: The maximum number of orphan blocks being stored.

Value: 100

Movitation: This value should be large enough to prevent prematurely dropping orphan blocks because of insufficient storage and small enough to prevent significant memory usage.

CF-18 `MAX_RECENT_REJECTED_BLOCKS`

Description: The maximum number of recently rejected blocks being stored.

Value: 20

Movitation: This value should be large enough to allow students to experiment with a significant amount of rejected blocks and small enough to prevent significant memory usage.

CF-19 `MAX_TRANSACTION_POOL_SIZE`

Description: The maximum number of transactions in the transaction pool.

Value: 300

Movitation: This value should be large enough to prevent prematurely dropping transactions because of insufficient storage and small enough to prevent significant memory usage.

CF-20 `MAX_ORPHAN_TRANSACTIONS`

Description: The maximum number of orphan transactions being stored.

Value: 100

Movitation: This value should be large enough to prevent prematurely dropping orphan transactions because of insufficient storage and small enough to prevent significant memory usage.

CF-21 `MAX_RECENT_REJECTED_TRANSACTIONS`

Description: The maximum number of recently rejected transactions being stored.

Value: 20

Movitation: This value should be large enough to prevent prematurely dropping recently rejected transactions because of insufficient storage and small enough to prevent significant memory usage.

7.7 Validation and verification

In the upcoming sections, the transaction and block validators will be described. Each validator has a list of rules which are used to validate incoming transactions and blocks. For each rule, the deviations from Bitcoin will be discussed and motivated.

The validators execute the rules one after the other. If one rule fails, the validation immediately stops, and the transaction or block is marked as invalid. This is because, for some validation rules, we need the assumption that earlier validation rules were passed, so if one rule fails, validation must be terminated.

7.7.1 Transaction validator

In this section, the transaction validator will be discussed. The rules described in this section form the validation of transactions in the Brabocoin network. These validation rules are executed on every new transaction a node receives. Only if the transaction is valid, i.e. passes all the validation rules discussed below, the transaction is added to the transaction pool. Transactions in blocks are validated with a different rule set. Validation of these transactions will be specified in the next section.

Now, we will provide a detailed description of all transaction validation rules. Brabocoin executes some rules in a different way compared to Bitcoin. Deviations with respect to Bitcoin and design decisions that were made are motivated. The parameters necessary to execute each rule are also provided.

Next, we will mention some rule that were implemented by Bitcoin, but were not implemented by Brabocoin. These rules are indicated with a special 'unimplemented' tag. A motivation for why these rules were skipped is provided.

Lastly, one transaction rule is mentioned that was implemented by Brabocoin, but not by Bitcoin. This rule is indicated with a special 'additional' tag. A motivation for adding this rule is provided as well.

After this, we will detail the order in which these rules are executed, and compare this with Bitcoin's validation ordering.

Transaction validation rules

TxRule-1 Duplicate present in transaction pool

Parameters: `Transaction, Consensus, TransactionPool`
Description: Reject when the transaction is already present in either the orphan pool or transaction pool or the recent rejects, or when one of the first two outputs exist in the chain UTXO set.
Deviations and decisions: For Brabocoin, we only check if the transaction is already present in either the orphan pool or transaction pool. The latter two are best-effort checks, which are not necessary and skipped in the Brabocoin network. Brabocoin does not focus on performance or efficiency but instead focusses on transparency for educational purposes.

TxRule-2 Input or output list not empty

Parameters: `Transaction, Consensus`
Description: Reject if either the list of inputs or the list of outputs is empty.
Deviations and decisions: None.

TxRule-3 Maximum size

Parameters: `Transaction, Consensus`
Description: Reject if the serialized transaction size exceeds the maximum transaction size.
Deviations and decisions: The maximum transaction size is defined in consensus as constant CS-1.

TxRule-4 Output value

Parameters: `Transaction, Consensus`
Description: Reject when any of the individual outputs of the transaction is not within the legal money range. Also reject when the total sum of outputs is not within the legal money range.
Deviations and decisions: Legal money range is defined to be any positive number less than consensus constant CS-6.

TxRule-5 Duplicate inputs

Parameters: `Transaction, Consensus`
Description: Reject if there are multiple inputs in the transaction that spend the same output.
Deviations and decisions: None

TxRule-6 Coinbase rejection

Parameters: `Transaction, Consensus`
Description: Reject if the transaction is a coinbase transaction.
Deviations and decisions: Coinbase transactions are only valid within a block and should not be present in the transaction pool.

TxRule-7 Transaction pool double spending

Parameters: `Transaction, Consensus, TransactionPool`
Description: Reject if any other transaction in the transaction pool spends the same referenced output as any of the inputs of the transaction.
Deviations and decisions: None

TxRule-8 Valid inputs

Parameters: `Transaction, Consensus, chainUTXOSet`

Description: Reject if any of the referenced outputs of the inputs are neither present in the chain UTXO set, nor the transaction pool UTXO set.

Deviations and decisions: This rule is used for the validation of transactions in blocks, as well as the validation of incoming individual transactions. When we validate transactions in blocks, we need to use the chain UTXO set in this rule. However, when we validate individual transactions, we need to use both the transaction pool UTXO set and the chain UTXO set. The UTXO set provided as a parameter of this rule therefore varies, depending on the context of this validation rule.

TxRule-9 Coinbase maturity

Parameters: `Transaction, Consensus`

Description: When any of the inputs spends a coinbase output, reject when the coinbase maturity condition is not met.

Deviations and decisions: The coinbase maturity is defined by consensus constant CS-5.

TxRule-10 Input values

Parameters: `Transaction, Consensus, chainUTXOSet`

Description: Reject if any of the input amounts is not within the legal money range, or the sum of input amounts overflows.

Deviations and decisions: Legal money range is defined to be any positive number less than consensus constant CS-6.

TxRule-11 Sufficient input

Parameters: `Transaction, Consensus`

Description: Reject when the sum of input amounts is no larger than the sum of output amounts plus the minimum transaction fee.

Deviations and decisions: Bitcoin enforces a minimum transaction fee. Brabocoin only enforces that the fee should be greater than zero, which means we reject if the sum of input amounts is less than the sum of output amount.

TxRule-12 Signature public key

Parameters: `Transaction, Consensus, chainUTXOSet`

Description: Reject if the any of the public keys of the signatures in the transaction is invalid. The signature's public key is invalid if the address in the referenced output is not equal to the hash of the public key in the signature.

Deviations and decisions: None

TxRule-13 Signature

Parameters: `Transaction, Consensus`

Description: Reject if any of the signatures is invalid, as explained in Section 5.

Deviations and decisions: In this rule, we do not necessarily deviate from Bitcoin, other than the fact that we do not use scripts to generate signatures. See design decision D-9 Bitcoin's scripting language.

TxRule-14 Syntactic correctness — Unimplemented

Description: The structure of the transaction message should be according to consensus, see Section 7.2.2.

Motivation: The Protobuf library handles malformed messages. These messages are considered invalid and are already discarded before the validation procedure starts. Hence, this transaction validation rule is not necessary for Brabocoin.

TxRule-15 scriptSig length in coinbase transaction — Unimplemented

---

Description: If the transaction is a coinbase transaction, reject if the `scriptSig` length is not in $[2, 100]$.

Motivation: Scripts are not used in Brabocoin, which is why this rule is not implemented by Brabocoin. See design decision D-9 Bitcoin's scripting language.

TxRule-16 Input reference null — Unimplemented

Description: Reject if for any input its referenced transaction hash or output index is `null`.

Motivation: This is enforced by the data structure of the transaction in Brabocoin, which is why this rule is not implemented.

TxRule-17 Standard transaction — Unimplemented

Description: Reject if non-standard, that is either:

- Invalid version number
- size > `MAX_STANDARD_TX_WEIGHT`
- `scriptSig` > 1650 or not push-only
- `scriptPubKey` not standard
- Bare multi-signature
- Dust output
- Multiple `OP_RETURN`s
- size < `MIN_STANDARD_TX_NONWITNESS_SIZE` = 82
- Not final (`nLockTime`)

Motivation: Brabocoin does not use scripts and the other features mentioned are not relevant for Brabocoin. See design decision D-9 Bitcoin's scripting language. This is why this rule is not implemented by Brabocoin.

TxRule-18 Sequence lock check — Unimplemented

Description: Check sequence locks.

Motivation: Time locks are not used in Brabocoin, which is why this rule is not implemented.

TxRule-19 Fee value — Unimplemented

Description: Reject if the transaction fee is not within the legal money range.

Motivation: This follows from **TxRule-10** Input values, **TxRule-4** Output value and **TxRule-11** Sufficient input. This means this rule does not need to be implemented in Brabocoin.

TxRule-20 scriptPubKey — Unimplemented

Description: Reject if `scriptPubKey` is not standard.

Motivation: Scripts are not used in Brabocoin, which is why this rule is not implemented. See design decision D-9 Bitcoin's scripting language.

TxRule-21 Signature operation count — Unimplemented

Description: Bitcoin rejects if the total number of signature operations done for the entire block is more than `MAX_SIGOPS_COUNT/5`.

Motivation: Multisignatures are not used in Brabocoin, which means the amount of signatures should be exactly equal to the input count. This is checked in another transaction rule, see **TxRule-24** Signature count. This is why this rule is not implemented.

TxRule-22 Sufficient transaction fee — Unimplemented

Description: Reject if the transaction fee is not sufficient to be added to the transaction pool.

Motivation: Brabocoin enforces a static minimum transaction fee of one brabocent. This is checked in another transaction rule, see rule **TxRule-11** Sufficient input. This is why this rule is not implemented.

TxRule-23 Dependent transaction ancestors — Unimplemented

Description: Reject if a dependent transaction has too many ancestors in the transaction pool.

Motivation: Brabocoin distinguishes independent and dependent transactions. Since Brabocoin only allows independent transactions in a block, the limit on the number of ancestors is not a problem. See design decision D-13 Independent and dependent transactions.

TxRule-24 Signature count — Additional

Parameters: `Transaction, Consensus`

Description: Reject if the number of signatures is not equal to the number of inputs of the transaction.

Motivation: In the Brabocoin network, scripts are not used and transactions are only *P2PKH* transactions (in Bitcoin terms) without multisignatures. See design decision D-13 Independent and dependent transactions and D-9 Bitcoin's scripting language. Therefore, each input belongs to exactly one signature. For the block to be valid, the input count should be equal to the amount of signatures in the transaction.

Transaction rule ordering

Brabocoin contains two lists of transaction validation rules, that execute the rules in a specific order. We work with two separate rule lists to be able to validate new transactions in a different way, depending on the context we are in. In Section 7.9, we explain which rule list is used in which context in more detail.

The first rule list *incoming transaction* is executed when a new incoming transaction has been received. See Figure 21 for the rules ordering in this list.

The second rule list *after-orphan* is only used when a transaction needs to be validated after it has been marked as an orphan transaction. See Figure 22 for the rules ordering in this list.

Bitcoin, however, executes their validation in a slightly different order. This validation can be compared to the validation of rule list *incoming transaction*. See Figure 23 for the rules ordering in this list.

The main difference between Bitcoin's validation and the rule list *incoming transaction* is that Bitcoin performs a lot of checks that are not implemented by Brabocoin. The reasons for skipping these checks were motivated in the detailed rule descriptions above. Note, however, that the order in which the rules that were implemented by both Brabocoin and Bitcoin are executed, is also different (see design decision D-27 Validation order). This is because Bitcoin's main focus when executing these validation checks is performance, which is why Bitcoin's validation order is based on efficiency. Brabocoin, on the other hand, focusses mainly on clarity for educational purposes. This is why Brabocoin's rules are ordered in such a way that they are grouped by subject and purpose.

Incoming transaction	
1.	**TxRule-1** Duplicate present in transaction pool
2.	**TxRule-3** Maximum size
3.	**TxRule-6** Coinbase rejection
4.	**TxRule-2** Input or output list not empty
5.	**TxRule-5** Duplicate inputs
6.	**TxRule-7** Transaction pool double spending
7.	**TxRule-8** Valid inputs
8.	**TxRule-9** Coinbase maturity
9.	**TxRule-4** Output value
10.	**TxRule-10** Input values
11.	**TxRule-11** Sufficient input
12.	**TxRule-24** Signature count
13.	**TxRule-12** Signature public key
14.	**TxRule-13** Signature

Table 21: Incoming transaction.

After orphan	
1.	**TxRule-7** Transaction pool double spending
2.	**TxRule-8** Valid inputs
3.	**TxRule-9** Coinbase maturity
4.	**TxRule-4** Output value
5.	**TxRule-10** Input values
6.	**TxRule-11** Sufficient input
7.	**TxRule-24** Signature count
8.	**TxRule-12** Signature public key
9.	**TxRule-13** Signature

Table 22: After orphan transaction.

Bitcoin’s transaction validaiton	
1.	**TxRule-14** Syntactic correctness
2.	**TxRule-1** Duplicate present in transaction pool
3.	**TxRule-2** Input or output list not empty
4.	**TxRule-3** Maximum size
5.	**TxRule-4** Output value
6.	**TxRule-5** Duplicate inputs
7.	**TxRule-15** scriptSig length in coinbase transaction
8.	**TxRule-16** Input reference null
9.	**TxRule-6** Coinbase rejection
10.	**TxRule-17** Standard transaction
11.	**TxRule-7** Transaction pool double spending
12.	**TxRule-8** Valid inputs
13.	**TxRule-18** Sequence lock check
14.	**TxRule-9** Coinbase maturity
15.	**TxRule-10** Input values
16.	**TxRule-11** Sufficient input
17.	**TxRule-19** Fee value
18.	**TxRule-20** scriptPubKey
19.	**TxRule-21** Signature operation count
20.	**TxRule-22** Sufficient transaction fee
21.	**TxRule-23** Dependent transaction ancestors
22.	**TxRule-12** Signature public key
23.	**TxRule-13** Signature

Table 23: Bitcoin’s transaction validation.

7.7.2 Block validator

The following rules describe the validation of blocks in the Bitcoin network. For each rule, deviations made in Brabocoin will be described.

In this section, the block validator will be discussed. The rules described in this section form the validation of blocks in the Brabocoin network. These validation rules are executed on every new block a node receives. Only if the block is valid, i.e. passes all the validation rules discussed below, the block is added to the blockchain. Note that the block validator contains separate validation rules used on the transactions in the block, which were not discussed in the previous section.

Now, we will provide a detailed description of all block validation rules. Brabocoin executes some rules in a different way compared to Bitcoin. Deviations with respect to Bitcoin and design decisions that were made are motivated. The parameters necessary to execute every rule are also provided.

Next, we will mention some block validation rule that were implemented by Bitcoin, but were not implemented by Brabocoin. These rules are indicated with a special 'unimplemented' tag. A motivation for why these rules were skipped is provided.

Lastly, some block validation rules will be discussed that were implemented by Brabocoin, but not by Bitcoin. These rules are indicated with a special 'additional' tag. A motivation for adding these rules is provided as well.

After this, we will detail the order in which these rules are executed, and compare this with Bitcoin's validation ordering.

Block validation rules

BlkRule-1 Block hash satisfies target value

Parameters: `Block, Consensus`
Description: Reject if the block hash is greater than the target value.
Deviations and decisions: None

BlkRule-2 Valid Merkle root

Parameters: `Block, Consensus`
Description: Reject if the Merkle root is not equal to the computed Merkle root using the transactions in the block.
Deviations and decisions: In Brabocoin, the Merkle root computation is altered compared to Bitcoin, see design decision D-15 Merkle trees.

BlkRule-3 Non-empty transaction list

Parameters: `Block, Consensus`
Description: Reject if the transaction list is empty.
Deviations and decisions: None

BlkRule-4 Maximum size

Parameters: `Block, Consensus`
Description: Reject if the serialized block size is greater than the maximum block size.
Deviations and decisions: The maximum block size is defined in consensus, by constant CS-1.

BlkRule-5 First transaction is coinbase transaction

Parameters: `Block, Consensus`
Description: Reject if the first transaction is not a coinbase transaction.
Deviations and decisions: None

BlkRule-6 Single coinbase transaction

Parameters: `Block, Consensus`
Description: Reject if any transaction besides the first transaction is a coinbase transaction.
Deviations and decisions: None

BlkRule-7 Non-contextual transaction validation

Parameters: `Block, Consensus`
Description: For each transaction in the block, the following validation steps should pass:

1. **TxRule-2** Input or output list not empty
2. **TxRule-4** Output value
3. **TxRule-24** Signature count

Deviations and decisions: Some transaction validation steps used in this rule in Bitcoin were omitted in Brabocoin. First, we do not check if the transactions are larger than the maximum block size, since we already check if the block is larger than maximum block size in rule **BlkRule-4** Maximum size. If this rule passes, all transactions must also be smaller than the maximum block size. Next, Bitcoin also executes rule **TxRule-15** scriptSig length in coinbase transaction here, but since we do not use scripts in Brabocoin, we discard this rule. See design decision D-9 Bitcoin's scripting language. Lastly, Bitcoin executes rule **TxRule-16** Input reference null, but this rule was discarded in Brabocoin.

BlkRule-8 Duplicate storage

Parameters: `Block, Consensus, Blockchain`
Description: Reject if the block was already stored.
Deviations and decisions: None

BlkRule-9 Known parent

Parameters: `Block, Consensus, Blockchain`
Description: Reject if the parent of this block is not known.
Deviations and decisions: None

BlkRule-10 Valid parent

Parameters: `Block, Consensus, Blockchain`
Description: Reject if the parent is not valid.
Deviations and decisions: None

BlkRule-11 Valid target value

Parameters: `Block, Consensus`
Description: Reject if the target value in the block is not according to consensus.
Deviations and decisions: For Brabocoin, the target value in consensus is defined by constant CS-2.

BlkRule-12 Valid network ID

Parameters: `Block, Consensus, Config`
Description: Reject if the network ID in a block is not according to configuration.
Deviations and decisions: The network ID in Brabocoin can be set in the configuration file in constant CF-1. This is useful for educational purposes, e.g. running multiple Brabocoin networks using different network IDs.

BlkRule-13 Valid coinbase block height

Parameters: `Block, Consensus`
Description: Reject if the block height in the coinbase transaction is incorrect.
Deviations and decisions: None

BlkRule-14 Unique unspent coinbase

Parameters: `Block, Consensus, chainUTXOSet`
Description: Reject if the hash of the coinbase transaction its output is already present in the chain UTXO set.
Deviations and decisions: None

BlkRule-15 Contextual transaction validation

Parameters: `Block, Consensus`
Description: For each transaction in the block, the following validation steps should pass:
1. **TxRule-8** Valid inputs
2. **TxRule-9** Coinbase maturity
3. **TxRule-10** Input values
4. **TxRule-11** Sufficient input
5. **TxRule-12** Signature public key
6. **TxRule-13** Signature

Deviations and decisions: None

BlkRule-16 Legal transaction fees

Parameters: `Block, Consensus, chainUTXOSet`
Description: Reject if the sum of transaction fees is not within the legal money range or overflows.
Deviations and decisions: Legal money range is defined to be any positive number less than consensus constant CS-6.

BlkRule-17 Valid coinbase output amount

Parameters: `Block, Consensus, chainUTXOSet`
Description: Reject if the coinbase output amount is larger than the block reward plus the sum of transaction fees.
Deviations and decisions: The block reward is defined in consensus, by constant CS-8.

BlkRule-18 Syntactic correctness — Unimplemented

Description: The structure of the block should be according to consensus. See Section 7.2.3
Motivation: The Protobuf library handles malformed messages. These messages are considered invalid and are already discarded before the validation procedure starts. Hence, this block validation rule is not necessary for Brabocoin.

BlkRule-19 Target value sanity check — Unimplemented

Description: Reject if the target value is negative, zero, overflows or is larger than the constant `PROOF_OF_WORK_LIMIT` in Bitcoin.
Motivation: This is a sanity check, as the target value is also checked in validation rule **BlkRule-11** Valid target value. This is therefore a performance improvement. Brabocoin does not focus on performance or efficiency, but instead focusses on clarity for educational purposes, which is why this rule was not implemented.

BlkRule-20 Signature operation count Unimplemented

Description: Reject if the number of signature operations done is more than `MAX_SIGOPS_COUNT`.
Motivation: Multisignatures are not used, and therefore the amount of signatures in a transaction should be exactly equal to the amount of inputs in the transaction. We do not need to check this separately for each block, as it is already checked for each transaction in the block in rule **BlkRule-7** Non-contextual transaction validation.

BlkRule-21 Checkpoint Unimplemented

Description: Reject if the block forks before the last known checkpoint.
Motivation: Brabocoin does not use checkpoints, as Brabocoin does not focus on performance. This rule was therefore not implemented.

BlkRule-22 Timestamp median time past Unimplemented

Description: Reject when the timestamp of the block is less or equal to the median time past.
Motivation: Brabocoin does not have timestamps in the block header, which is why this rule is not implemented. See design decision D-14 Timestamps.

BlkRule-23 Timestamp future Unimplemented

Description: Reject if the timestamp is more than 2 hours in the future.
Motivation: Brabocoin does not have timestamps in the block header, which is why this rule is not implemented. See design decision D-14 Timestamps.

BlkRule-24 Final transactions Unimplemented

Description: Reject if any of the transactions in the block are non-final.
Motivation: Brabocoin does not implement the lock times feature, as this is unnecessary for educational purposes, see design decision D-6 Version number and `nLockTime` fields removed. This block rule was therefore not implemented.

BlkRule-25 Check sequence locks Unimplemented

Description: Check sequence locks.
Motivation: Time locks are not used in Brabocoin.

BlkRule-26 Maximum nonce Additional

Parameters: `Block, Consensus`
Description: Reject if the nonce is greater than the maximum nonce defined in consensus, by constant CS-3.
Motivation: Brabocoin has a maximum nonce determined by consensus, which should be adhered to for a block to be valid. Bitcoin has a fixed nonce size, which is why Brabocoin needed an additional validation rule to check the nonce size of blocks.

BlkRule-27 Valid block height Additional

Parameters: `Block, Consensus, Blockchain`
Description: Reject if the block height is not equal to the block height of the parent plus one.
Motivation: None

	Incoming block
1.	**BlkRule-26** Maximum nonce
2.	**BlkRule-4** Maximum size
3.	**BlkRule-12** Valid network ID
4.	**BlkRule-8** Duplicate storage
5.	**BlkRule-1** Block hash satisfies target value
6.	**BlkRule-11** Valid target value
7.	**BlkRule-3** Non-empty transaction list
8.	**BlkRule-5** First transaction is coinbase transaction
9.	**BlkRule-6** Single coinbase transaction
10.	**BlkRule-2** Valid Merkle root
11.	**BlkRule-7** Non-contextual transaction validation
12.	**BlkRule-28** Duplicate input
13.	**BlkRule-9** Known parent
14.	**BlkRule-10** Valid parent
15.	**BlkRule-27** Valid block height
16.	**BlkRule-13** Valid coinbase block height

Table 24: Incoming block.

	After orphan
1.	**BlkRule-9** Known parent
2.	**BlkRule-10** Valid parent
3.	**BlkRule-27** Valid block height
4.	**BlkRule-13** Valid coinbase block height

Table 25: After orphan block.

BlkRule-28 Duplicate input Additional

Parameters: `Block, Consensus`
Description: Reject if any of the inputs in transactions within the block occur more than once.
Motivation: Bitcoin processes transactions while verifying. This causes the state to update and hence, this rule is not necessary for Bitcoin. Brabocoin validates the transactions first and then processes all transactions in one batch. See design decision D-21 Processing transactions in a block.

Block rule ordering

Brabocoin contains three lists of block validation rules, that execute the rules in a specific order. We work with three separate rule lists to be able to validate new blocks in a different way, depending on the context we are in. In Section 7.9, we explain which rule list is used in which context in more detail.

The first rule list *incoming block* is used when a new incoming block has been received. See Figure 24 for the rules ordering in this list.

The second rule list *after-orphan* is only used when a block needs to be validated after it has been marked as an orphan block. See Figure 25 for the rules ordering in this list.

The third rule list *connect-to-chain* is used when a block is connected to the main chain. See Figure 26 for the rules ordering in this list.

	Connect to chain
1.	**BlkRule-14** Unique unspent coinbase
2.	**BlkRule-15** Contextual transaction validation
3.	**BlkRule-16** Legal transaction fees
4.	**BlkRule-17** Valid coinbase output amount

Table 26: Connect to chain.

	Bitcoin's block validation
1.	**BlkRule-18** Syntactic correctness
2.	**BlkRule-19** Target value sanity check
3.	**BlkRule-1** Block hash satisfies target value
4.	**BlkRule-2** Valid Merkle root
5.	**BlkRule-3** Non-empty transaction list
6.	**BlkRule-4** Maximum size
7.	**BlkRule-5** First transaction is coinbase transaction
8.	**BlkRule-6** Single coinbase transaction
9.	**BlkRule-7** Non-contextual transaction validation
10.	**BlkRule-20** Signature operation count
11.	**BlkRule-8** Duplicate storage
12.	**BlkRule-9** Known parent
13.	**BlkRule-10** Valid parent
14.	**BlkRule-27** Valid block height
15.	**BlkRule-11** Valid target value
16.	**BlkRule-21** Checkpoint
17.	**BlkRule-22** Timestamp median time past
18.	**BlkRule-23** Timestamp future
19.	**BlkRule-12** Valid network ID
20.	**BlkRule-24** Final transactions
21.	**BlkRule-13** Valid coinbase block height
22.	**BlkRule-14** Unique unspent coinbase
23.	**BlkRule-25** Check sequence locks
24.	**BlkRule-15** Contextual transaction validation
25.	**BlkRule-16** Legal transaction fees
26.	**BlkRule-17** Valid coinbase output amount

Table 27: Bitcoin's block validation.

As mentioned above, it is the case that Bitcoin executes their block validation in a slightly different order. This validation can be compared to the validation of rule list *incoming block*, together with the *connect-to-chain* rule list. See Figure 27 for the rules ordering in this list.

Just as in the transaction validation, it holds that Bitcoin performs a lot of checks in the block validation that are not implemented by Brabocoin. Again, the reasons for skipping these checks were motivated in the detailed rule descriptions above. The order in which the rules that were implemented by both Brabocoin and Bitcoin are executed, is also different, just like in the transaction validation (see design decision D-27 Validation order).

7.8 Mining

Each node in Brabocoin contains a miner, wich is responsible for mining new blocks on the blockchain. The implementation of the miner will be discussed in this section.

7.8.1 Configuration

The mining process of the node's miner can be configured through a configuration window in Brabocoin. Among other things, the mining reward address can be configured. This is the address in the user's wallet where the coinbase output will be sent to. The user can also choose whether or not to mine on the top block. If the user decides *not* to mine on the top block, the parent block hash field is enabled. The user can then enter a block hash value that will be used as a parent hash when mining new blocks.

7.8.2 Mining refresh

It might be the case that while a user is mining a block on top of some block B, another peer finishes mining a valid block on top of block B. In this case, the user has 'lost' the mining competition and should restart the mining process, to mine on top of block B.

The miner is notified by the node environment of any valid blocks that are processed. This means that when the user is mining on the top block and a new block is received that has the same parent block hash as the parent block hash in the block that is currently being mined, the mining process is restarted in order to start mining on the new top block. The same holds for any valid blocks with the same parent block hash as the hash set in the custom parent block hash when mining on the top block is disabled. This allows for multiple entities to mine on the same fork and perform, for example, a 51% attack (see Section 2.9).

7.8.3 Continuously mine vs. Mine single block

When starting the mining process, the user can choose to either continuously mine or to mine a single block. Mining a single block will start the mining process until one block is mined, even when the miner needs to refresh and restart the mining process when new top blocks are received. Mining continuously will start the mining process until the miner is stopped.

7.8.4 Mining procedure

The following mining procedure is executed each time the miner attempts to mine a new block.

1. First, a number of transactions are collected from the transaction pool, such that their total size is still smaller than a size x. Here, x is the maximum block size (Consensus constant CS-1 `MAX_BLOCK_SIZE`), minus all the data in the block that do not include the mined transaction data. This means that x is the maximum block size minus the maximum header size of a block excluding the nonce (Consensus constant CS-10 `MAX_BLOCK_HEADER_SIZE`), the maximum size of a coinbase transaction (Consensus constant CS-11 `MAX_COINBASE_TRANSACTION_SIZE`), the maximum nonce size (Consensus constant CS-3 `MAX_NONCE_SIZE`) and a number of bytes to allow Google's protobuf to serialize the message.

   In this procedure, dependent transactions, i.e. transactions that depend on other transactions in the pool, are not taken into account (see design decision D-13 Independent and dependent transactions). These transactions can only be mined after the transactions they depend on are included in a block.

2. Next, a coinbase transaction is created. To this end, the fees of all collected transactions are summed up, and the block reward value is added as well. This total amount is the only output of this coinbase transaction, paid to the mining address of the miner, set in the configuration window of the miner (see Section 7.8.1).

3. The Merkle root of the Merkle tree of the transactions is calculated. See Section 4.3.8 for a description of Merkle trees. See design decision D-15 Merkle trees on the deviations from Bitcoin. We use the hash function as given in Consensus constant CS-15 `merkleTreeHashFunction`.

4. A random starting nonce is selected, using Java's `java.util.Random` class.

5. A new block is created. The parent block hash of this block is set to the hash of the current top block of the main chain. The Merkle root is set to the Merkle root calculated in step 3. The target value is set to the constant CS-2 `TARGET_VALUE` in consensus. The nonce is set to the random nonce selected in step 4. The block height is set to the current height of the main chain plus one. The network ID of the block is set to the constant CF-1 `NETWORK_ID` defined in configuration. Lastly, the collected transactions are added, together with the created coinbase transaction.

6. Next, the block hash is calculated and it is checked whether this hash is smaller than the specified target value. If not, the nonce is increased by one (modulo the max nonce, specified in consensus constant CS-4 `MAX_NONCE`), and we start from step 5. This step is repeated until the block hash is smaller than the target value.

7. The newly mined block is processed and propagated to the node's peers.

7.9 Processors

7.9.1 Starting the node

When the software is started, a number of tasks are performed to initialize the node. In addition to the initialization procedure defined in the node environment (see Section 7.5.2), the persistent data must be processed to restore the current state of the node.

Synchronizing the main chain with the UTXO set
When the node is started, a number of initialization tasks need to be performed to restore the correct state of the blockchain in memory. In particular, the block headers of the blocks on the main chain need to be loaded from disk into memory, as these blocks will need to be accessed more frequently. It is important that the state of the main chain is in sync with the state of the UTXO set. The UTXO set maintains the hash of the *last processed block*, which is the hash of the block in the main chain up to which the UTXO set is up-to-date (as described in Section 7.3.2). Normally, when the UTXO set is correctly maintained, this is the main chain top block. On startup, the main chain is thus initialized with all blocks up to the last processed block in the UTXO set. The block headers of the blocks on the main chain are read from disk and put in memory.

7.9.2 Block processor

Since Brabocoin is a purely decentralized network, no other node can be trusted in sending a completely valid block. Therefore, a new block must be validated against the consensus parameters and the current state of the blockchain from the perspective of the node itself, before processing the block and recording it in the blockchain and propagating the block to its peers.

When the blockchain has been initialized, new blocks can be processed and, if they are valid, they can be added to the blockchain. New blocks can either be received from one of the peers of the node, or can be mined by the node itself. Every new block needs to be processed, which is done by the block processor of a node. This procedure consists of the following steps:

1. The validation rules for an *incoming block* are executed, which are defined in Table 24. These rules are *non-contextual*: they validate the block structure, independent from the node's current state. As explained in the rule descriptions in Section 7.7.2, we use different UTXO sets to execute these validation rules depending on the context we are in. For all block validation, we use only the chain UTXO set.
2. Depending on the outcome of the *incoming block* validation:
   - If the block is invalid, the block is rejected and processing is terminated.
   - If the block is orphan, the block is stored as orphan and processing is terminated. For more details on orphan blocks, see Section 7.3.4.
   - If the block is valid, processing continues.
3. The block is stored in the block database (see Section 7.3.1).
4. Add the block to a new set of *top candidates* T. This set is defined as a set of blocks currently not on the main chain, which are valid and can now be connected to the main chain.
5. Search in the block orphan pool for blocks that descend from the newly processed block. These orphans now have a known parent. For every discovered orphan, the validation rules *after orphan* are executed, which are defined in Table 25:
   - When the orphan is invalid, it is removed from the orphan pool.
   - When the orphan is valid, it is removed from the orphan pool and added to T.
6. The current top block of the main chain is added to T.
7. From T, the best candidate c is selected and removed from T. The best candidate is the block with the highest block height. When there are multiple blocks with the highest block height, the block with the lowest hash value is selected. This tie break rule is defined in consensus value CS-14 `bestValidBlock`. Note that T cannot be empty, since T always contains at least the current top of the main chain.

8. If c is the current top of the main chain, the reorganization is completed successfully and processing stops.
9. If c turns out not to be the current top of the main chain, we need to perform a *main chain reogranization*. Find a *fork path* $f = (b_1, b_2, \ldots, b_n, c)$ of valid blocks connecting block c to the main chain, where the parent of b_1 is the first and only block both in f on the current main chain.
10. If any block in f is marked as *invalid*, the fork path f is invalid: go back to step 7.
11. Keep disconnecing the top blocks of the main chain until b_1 is the top block. For every block that is disconnected:
    - Remove the outputs of the transactions in the block from the UTXO set.
    - Apply the block undo data to re-add unspent outputs to the UTXO set.
    - Demote any transactions in the transaction pool that depend on the coinbase transaction in the block to orphan.
    - Add the transactions (exceptfor the coinbase transactions) in the block to the transaction pool.
    - Demote independent transactions in the transaction pool to dependent when necessary.
    - Promote orphan transactions to dependent transactions in the transaction pool when necessary.
    - Remove the top block from the main chain.
12. Now, we need to connect all blocks in f to the main chain. For every block b_i in f (except for block b_1, which is already on the main chain):
    - Validate b_i with the *connect to chain* validation rule list, defined in Table 26.
    - When b_i is invalid, the block is stored as *invalid*. This makes f invalid and we go back to step 7.
    - The block undo data is collected and stored on disk.
    - Remove any transactions recorded in b_i from the transaction pool. Promote dependent transactions to independent where necessary.
    - Connect b_i to the top of the main chain.

7.9.3 Transaction processor

When a new transaction is received, it must be validated before it is placed in the transaction pool, marked as orphan, or is rejected. A new transaction that is created by the node itself must also be validated before it is processed and added to the transaction pool, and propagated to its peers. The procedure that processes incoming and newly created transactions is done by the transaction processor of a node. It consists of the following steps:

1. The validation rules for an *incoming transaction* are executed, which are defined in Table 21. As explained in the rule descriptions in Section 7.7.1, we use different UTXO sets to execute these validation rules depending on the context we are in. For the current validation, we use a combined UTXO set from the chain and the pool.
2. Depending on the outcome of the *incoming transaction* validation:
    - If the transaction is invalid, the transaction is rejected and processing is terminated.
    - If the transaction is orphan, the transaction is stored as orphan and processing is terminated. For more details on orphan transactions, see Section 7.3.4.
    - If the block is valid, processing continues.
3. Check whether the transaction is independent or dependent, by executing validation rule **TxRule-8** Valid inputs on just the chain UTXO set.
    - When the rule passes, the transaction is added to the transaction pool as independent.

- When the rule fails, the transaction is added to the transaction pool as dependent.

4. Search the orphan transactions for any transactions that depend on the processed transaction. For every discovered orphan transaction t:

   - Check whether the transaction is no longer orphan by executing validation rule **TxRule-8** Valid inputs on t.
   - If the validation rule fails, leave t in the orphan set and continue with the next discovered orphan.
   - If the validation rule does not fail, execute the validation rules *after orphan*, which are defined in Table 22, using both UTXO sets:
     - When t is invalid, it is rejected.
     - When t is valid, it is stored in the transaction pool as dependent.

7.10 Wallet

The wallet implemented by Brabocoin is a JBOK wallet, as described in Section 6.1. Hierarchical deterministic (HD) wallets come with great benefits, as explained in Section 6.1. HD wallets were added to the Bitcoin core in version 0.13. However, since Brabocoin will be used for educational purposes only, a JBOK wallet suffices (see design decision D-31 Wallet type).

7.10.1 Key pair management

As explained in Section 6, a wallet consists of a collection of key pairs. Remember that a key pair is a combination of a private and public key. The wallet contains a number of **KeyPair** data structures, as can be found in Section 7.2. On wallet creation, an initial key pair is generated and the user is prompted to enter a password for their wallet. This password is used for encryption.

Serialization and encryption
When the user saves the wallet, the wallet is flushed to disk after encryption with AES in CBC mode with PKCS5 padding. The encrypted data consists of serialized **KeyPair** objects, concatenated by using Protobuf's delimiters. The secret key for the AES algorithm is generated by using PBKDF2 with HMAC-SHA256 (see Section 4.3.10). We use 10 000 iterations for PBKDF2, as recommended by NIST [29].

On application exit, the wallet is automatically saved.

Key pair generation
When a new key pair is generated, a random number less than the order of the Elliptical Curve used in Brabocoin (see consensus constant CS-12 `CURVE`) should be generated. The result now is our private key This is done using the `java.security.SecureRandom` class in Java. The implementation of this class is platform dependent. The analysis of the implementation is beyond scope of this document.

The public key is now calculated by multiplying the Elliptical Curve generator point with the private key, as explained in Section 3. Lastly, the Brabocoin address is calculated by applying the SHA-256 algorithm and the RIPEMD-160 algorithm to the compressed public key (see Section 4.3.5).

7.10.2 Balance calculation

The wallet in Brabocoin maintains four values that together describe the user's balance. The *confirmed balance* describes the user's balance considering only unspent outputs in the main chain. It might be the case that transactions are created that are still in the transaction pool, and are therefore not yet mined and contained in the main chain. This is where the *pending balance* comes in. This value describes the change in confirmed balance considering the transactions in the transaction pool. Obviously, this value can be positive or negative, depending on how much brabocoin the user has spent or received within the transaction pool.

Together, these values form the *spendable balance*. This is the user's balance, based on both the main chain *and* the transaction pool. In other words, it is the confirmed balance considering the pending balance change.

The last value is the *immature mining reward*. When a user mines a block, the mining reward and mining fees (specified in the coinbase transaction) is only considered spendable when a number of blocks are mined on top of the user's mined block. This number is described in consensus constant CS-5 `COINBASE_MATURITY_DEPTH`. The immature mining reward in the wallet is the total mining reward, i.e. the sum of coinbase transaction outputs, that will be spendable when sufficient blocks are mined on top of the user's mined blocks.

Brabocoin also allows the user to view the confirmed balance, pending balance and immature mining reward for each individual key pair in the wallet.

7.10.3 Transaction history

The Brabocoin wallet also keeps track of a user's transaction history. Whenever a transaction is received that has an output referencing one of the keys in the wallet, the transaction is saved to the transaction history. Each transaction that is created by the user and sent into the network is also added to the transaction history.

The transaction history file is written to disk when the wallet is saved. The data is not encrypted. In practical applications, it is best practice to encrypt the transaction history. However, due to the educational purpose of Brabocoin, this is not of significance.

The transaction history file location can be configured in constant CF-15 `TRANSACTION_HISTORY_FILE`.

7.11 Deviations from Bitcoin

In this section, all Brabocoin's deviations from Bitcoin are described and motivated. The deviations are grouped by subject.

7.11.1 General

D-1 Re-use Bitcoin core or build from scratch
When developing Brabocoin, we had a choice of re-writing the Bitcoin core implementation, or to develop a new system from scratch. The Bitcoin core was an open source project where optimization was a main priority. This caused a significant part of the software to not be reusable. We therefore decided it was easier to build a new system, with education and clarity as its main priorities.

D-2 All nodes are full nodes
Bitcoin includes so-called SPV nodes in their network, which are nodes that only download the header of a block, not the transactions included in the block. See Section 4.3.7 for more details on these nodes. In Brabocoin, we decided that all nodes in the network are full nodes; we did not include SPV nodes. This is because we wanted the system to be as comprehensive and clean as possible, which is why all nodes in the network should have the same functionality.

D-3 Hash functions used in Brabocoin
We decided to use the same hash functions as Bitcoin, and to apply them in the same manner as they have been in Bitcoin. This is because Brabocoin is meant to be an implementation of a cryptocurrency based on Bitcoin, which is why we wanted to keep its main features the same as much as possible. See Section 4.3 for more details on how hash functions are used in the system.

7.11.2 Network

D-4 Streaming messages
In the Bitcoin protocol, a number of message types send lists of data over the network, for instance the `inv` and `getblocks` message. The replies to these messages are either a list, or a sequence of messages transferring the requested data separately, for instance the `tx` and `block` messages. In Brabocoin, the protocol is built on the Protobuf system, which allows for *streaming* messages to request and respond with lists of data. Hence, the structure of these messages is slightly changed to accommodate the streaming capability.

7.11.3 Transactions

D-5 Coinbase transactions
The coinbase transaction in Bitcoin has a special input that indicates that the transaction is a coinbase transaction. A transaction input references a previous transaction output, but in a coinbase transaction this is not the case. Therefore, these fields are filled with a special value to mark it as a coinbase transaction input.

One could reason that in Brabocoin, to avoid the need to fill certain fields with special values, we could specify a coinbase transaction as a transaction with no inputs, instead of a transaction with special-value inputs. However, as specified in design decision D-18 Block height in header, this would cause some coinbase transactions to have the same transaction hash, which leads to several issues. This is why we decided to still fill the input of the coinbase transaction with special values. See Table 7 for a specification of the input data structure. The referenced transaction field of a coinbase transaction is left empty. The referenced output index is filled with the blockheight of the block the coinbase transaction is a part of.

D-6 Version number and `nLockTime` fields removed
Blocks and transactions in Bitcoin all have a version number. Full nodes check the version numbers of incoming blocks and transactions. If the version number of the full node is lower than that of the block or transaction, this means the node probably does not use the most recent consensus rules. In Brabocoin, we do not need such a version number. Instead, a network ID field was introduced, see design decision D-19 Network ID.

Transactions also include an `nLockTime` field, which indicates the earliest time the transactions are ready to be added to the blockchain. This allows signers to change their minds before their transaction is added to the blockchain. Brabocoin will be used for educational purposes, which is why we want the system to be structured and comprehensive. We therefore did not include the `locktime` feature in Brabocoin.

D-7 Sequence numbers removed
The sequence number of a transaction in Bitcoin is used to update time-locked transactions before being added to the blockchain. As we did not include the `timelock` feature in Brabocoin (see design decision D-6 Version number and `nLockTime` fields removed), the sequence number field was not included either.

D-8 Copying previous transaction output `scriptPubKey` before signing
Before a transaction is signed in Bitcoin, the signature script (the `scriptSig` and `scriptSigLen` fields) of the input that is signed is replaced with the public key script of the output that is referenced [49]. Since the transaction already references the used output, with the previous transaction hash and output index fields, no additional information is signed by this copy step.

The only useful reason as to why redundant data from the previous transaction is included in the signature is related to dedicated signing devices that are not present on the Bitcoin network as full nodes. When the additional data is included in the signature, the signing device might retrieve the previous transaction data from an untrusted source. When the received data is incorrect, the signature will be invalid, providing an extra layer of security for the signing device. This is why in BIP-143 a new signing algorithm for the segregated witness program is proposed to also include the previous transaction output amount in the signature [39].

Since in the initial Brabocoin proposal every node is online and no dedicated signing devices exist, this extra signing step is not important to include. The copy step makes the signing algorithm more difficult to understand and is unnecessary from a strictly cryptographic validation point-of-view. Therefore, we omit this from the Brabocoin signing algorithm.

D-9 Bitcoin's scripting language
Bitcoin provides a scripting language, which is used for digital signatures [1]. Using these scripts, it is possible to create more complex types of transactions. In Brabocoin, these complex transaction types are not needed. This allows us to remove the scripting language entirely and only allow *Pay-to-public-key-hash* transactions. An output in a transaction now needs an address instead of the script. Similarly, there is no need for the `SIGHASH` flags in Bitcoin that allow other more complex types of transactions.

D-10 Multi-signatures removed

Bitcoin includes the possibility to create a `pubkey script` that provides $n \in \mathbb{N}$ number of public keys, and requires the corresponding `signatures script` to provide a minimum of $m \in \mathbb{N}$ signatures, using the n provided public keys. This phenomenon is referred to as an m-of-n multi-signature. In Brabocoin, we want to keep the system as comprehensive as possible, we have not included the possibility for multi-signatures. Exactly one signature should be included for every input of a transaction.

D-11 Unsigned and signed transactions
We deviate from Bitcoin in the way we define and calculate signatures over a transaction. In Bitcoin, signatures are included in the input data structure. In Brabocoin, we moved the signatures to the transaction data structure. We calculate signatures by specifying two types of transactions: unsigned and signed transactions. When a new transaction is created, first an unsigned transaction is created, which does not include any signatures. The signatures of the transaction are calculated over this unsigned transaction, using the address defined in an unspent output referenced by one of the inputs in this unsigned transaction. A signed transaction is then created using the unsigned transaction and the calculated signatures.

D-12 Transaction recipient addresses
In the output data structure of Bitcoin, the recipient is defined in a script that is included in the output. In Brabocoin, however, we have removed the scripting language entirely, which is explained in design decision D-9 Bitcoin's scripting language. In Brabocoin, recipients are identified by their address.

D-13 Independent and dependent transactions
Bitcoin uses advanced sorting mechanisms to keep track of all transactions in the transaction pool. These sorting mechanisms take into account the transaction fees and the transactions the inputs of this transaction reference, among others. These mechanisms were introduced to make Bitcoin as efficient as possible. Brabocoin does not focus on performance, however, which is why these sorting mechanisms were not included. For simplicity, we do not sort the transactions on transaction fee rate. This also means that we do not try to maximize the transaction fees when we mine a new block.

Instead, Brabocoin only distinguishes dependent from independent transactions. Dependent transactions are defined as transactions in the transaction pool that reference outputs from another transaction in the pool, independent transactions do not. In order to include a dependent transaction in a block, the transactions this transaction depends on would have to be included as well. Note that this could be a long chain of dependencies, which might not even fit into a block entirely. This is why we decided that only independent transactions can be included in a block. Also note that every dependent transaction will eventually become independent, because transactions cannot depend on each other in a circular manner.

7.11.4 Blocks

D-14 Timestamps
Bitcoin specifies a timestamp in the block header. These are used to allow variations in the target value over time. As the target value will be kept constant in Brabocoin, the timestamp field has no purpose anymore and is not included. Note that this simplifies validation of blocks, as validating a timestamp is quite a tedious procedure.

In Brabocoin, we do save the time a block or transaction was received. However, this *time received* variable does not have any use case in the validation of blocks and transactions. It is simply a point of reference for the Brabocoin user to see when their blocks and transactions were received.

D-15 Merkle trees
The hashes of all transactions in a block form the leaves of the Merkle tree of the block. This might be an odd number of transactions. Bitcoin duplicates the last leave to make sure that every intermediate node has exactly two children. However, this Merkle tree implementation creates some possibilities for attacks, which are explained in Section 4.3.8. To avoid this problem in Brabocoin, we do not duplicate the last transaction when we calculate the Merkle root. Instead, the last transaction is propagated up the Merkle tree to the next level that contains an odd number of nodes. The last transaction is added to this level, such that this level now also contains an even number of nodes.

D-16 Target value
Bitcoin adjusts the target value every time another 2016 blocks have been mined, to adjust the difficulty of the proof-of-work algorithm [1]. However, since the Brabocoin network is relatively small and used

for educational purposes only, this adjustment is not necessary in Brabocoin. Therefore, in Brabocoin, the target value is constant. It is defined in consensus value CS-2 `TARGET_VALUE`.

D-17 Block nonce size
In the block data structure of Bitcoin, the nonce field allows miners to solve the proof-of-work challenge by allowing variations in the block header data. The nonce field used in Bitcoin is only four bytes long [49]. This allows for only so much variation in the block header, which might not be sufficient when the target value of the block hash becomes lower as computing power increases. Therefore, the space left in the description of the coinbase transaction input is used as additional nonce space to allow more variations, since altering the input description alters the Merkle root in the header, and thus, the hash of the block.

This solution is chosen to maintain backwards compatibility on the Bitcoin network, but this is not a problem for a new system such as Brabocoin. Therefore, Brabocoin will use a variable-length nonce field in the header instead, such that enough variation is always possible in the block header. The nonce size is determined in consensus variable CS-3 `MAX_NONCE_SIZE`, and can be updated at any time to allow larger nonces. If the nonce length in a block is larger than the consensus value, the block is invalid.

D-18 Block height in header
In Bitcoin, the block height is included in the coinbase transaction of a block, as explained in design decision D-5 Coinbase transactions. Since it is more logical to place the block height in the header of the block, the block height is moved to the block header in Brabocoin. This gives one problem: coinbase transactions are all quite similar. As a result, it is reasonable to assume that they could be exactly the same if someone mines multiple blocks. This gives two transactions with exactly the same hash, which creates issues when storing the transactions in the UTXO database, among others. In design decision D-5 Coinbase transactions, we specify how we solved this problem.

D-19 Network ID
In Brabocoin, a network ID field is included in the block header. This field is not present in Bitcoin. We added this field, because Brabocoin will be used for educational purposes. Adding a network ID field in the block header allows for experimentation with multiple Brabocoin networks with different target values and it creates the possibility of building several blockchains at the same time.

D-20 Calculating a block hash
In Bitcoin, a block hash is calculated by hashing the header of the block. This is because Bitcoin contains so-called SPV nodes (see Section 4.3.7), which only download the header of a block, not the transactions included in the block. These nodes should still be able to calculate block hashes, which is why a block hash is calculated using only the header of a block. In Brabocoin, we decided to calculate block hashes in the same way, even though we do not include SPV nodes. This is because Brabocoin is supposed to be an implementation of a cryptocurrency based on Bitcoin, which is why we wanted to keep its main features the same as much as possible.

D-21 Processing transactions in a block
Bitcoin processes transactions while validating the block. This was done, because Bitcoin focusses on performance, and processing the transactions of a block while validating is very efficient. Brabocoin, however, focusses on clarity and transparancy. This is why, in Brabocoin, the transactions in a block are validated first and processed all in one batch, once the block has been declared valid.

7.11.5 Blockchain

D-22 Tiebreaker rule when blocks have the same height
It is possible that we have multiple choices for a main chain, if we have multiple forks of the same length. In this situation, we must choose one. Bitcoin chooses the chain where the last block was received first. However, this option requires keeping track of arrival times. In Brabocoin, we follow the main chain where the last block has the smallest hash; this block would have more *proof-of-work*.

D-23 Block storage
Bitcoin stores the full block data on files on disk; these files are called block files. In addition, in the block database, a record is created for every block, indicating in which file the block is stored, and at which position in the file. The block files store blocks in network format. Every Bitcoin network message starts

with the same four bytes, which is a *magic* constant identifying it belonging to the Bitcoin protocol. The blocks stored in the block files are separated by these four bytes.

Bitcoin uses these separators to determine where to stop reading the block data from the file, to avoid storing the serialized block size in the block database record. However, it is possible that the four-byte constant is contained in the raw data of a certain block. Reading the full block from the file will then not be possible anymore, since the reading is terminated when the byte sequence is discovered prematurely.

Therefore, we propose to store the size of the serialized block in the block database record, to remove the need of a magic separator in the block files. This extra storage is low overhead for a relatively small system compared to Bitcoin, and makes reading blocks from disk more robust.

7.11.6 Transaction pool

D-24 Persistent transaction pool
In Bitcoin, the transaction pool is persistent, meaning it is flushed to disk when the application is closed such that it can be reloaded when the application is re-started. In Brabocoin, we decided not to make our transaction pool data structure persistent. This is because in Brabocoin, persistant storage of the transaction pool is not essential, and keeping track of persistant storage is unnecessary overhead we wish to avoid.

D-25 Removing transactions
The size of the transaction pool is limited. Also, the amount of stored orphan transactions that can be stored is fixed. Bitcoin removes the oldest transactions during an overflow of the transaction pool. However, this requires the tracking of the arrival times of transactions. Therefore, in Brabocoin, we pick random transactions to be removed in case of an overflow, and also remove all transactions that depend on this one.

Notice that with this strategy, it is also not possible for a transaction to expire. Bitcoin removes orphan transactions after twenty hours and normal transactions after two weeks, to deal with expiring transactions. In Brabocoin, these transactions will be removed randomly if the pool size becomes too large.

7.11.7 UTXO sets

D-26 UTXO database obfuscation
In Bitcoin, to avoid a false-positive problem with antivirus scanners, the records stored in the UTXO database in Bitcoin are first obfuscated with an obfuscation key [9]. Since this is an extra step that just complicates the database system, this will be omitted in Brabocoin. The probability that the non-obfuscated data will trigger an antivirus response is the same as with obfuscated data, hence it is unnecessary to obfuscate the UTXO database in the Brabocoin implementation.

7.11.8 Validation

D-27 Validation order
The different aspects of validating a block or transaction in Bitcoin are ordered in a way that is most efficient for the system. In Brabocoin, however, we focus on comprehension and clarity, which is why we changed the validation ordering to match this purpose. Instead of focussing on efficiency, we grouped the different aspects of the validation by subject and purpose, in order to make the validation as comprehensive as possible. See Section 7.7 for more details on Brabocoin's validation.

D-28 Creating invalid transactions
In Brabocoin, we included a feature where users can create and validate invalid transactions, and send them to their peers in the network. This gives users a chance to experiment with the validation of transactions and thus, learn more about the inner workings of this validation. For obvious reasons, in Bitcoin these kind of features are not made available to the user.

If a node receives an invalid transaction, however, he does not propagate it to its peers automatically. A user can only do this manually. This prevents an invalid transaction from propagating through the entire network.

D-29 Validation stage in block database
Bitcoin slaat op in hoeverre een block is gevalidate, tot welke stap, zodat voorkomen kan worden

dat blokken twee keer gevalideert worden met dezelfde regels. wij houden dat niet bij, want geen performance. wij valideren gewoon wanneer nodig. wel een boolean

When a block is saved in the blockchain of Bitcoin, i.e. in the block database, a special field keeps track up to which step the block has been validated. This prevents the block from being validated with the same validation steps twice. In Brabocoin, we do not keep track of this, as this field was only implemented by Bitcoin for performance reasons. In Brabocoin, we simply validate a block when necessary.

We do, however, keep track of a special boolean flag indicating whether the block is valid. A block is first stored in the block database when it has passed the validation rules in the *incoming block* rule list, defined in Table 24. At this point, the boolean flag is set to `true`. Note that additional validation is performed when the block is being added to the main chain. These validation rules are specified in the *connect to chain* rule list, defined in Table 26. If the block passes these rules, the boolean flag remains `true`. If, however, the block does not pass a rule in this rule list, the boolean flag is set to `false`. The block is now marked as invalid.

7.11.9 Wallet

D-30 Plain and encrypted private keys
In Brabocoin, we only include plain private keys; private keys that are encrypted with a password were not included. This is because Brabocoin does not focus on security. Instead its main purpose is clarity, which means encrypted keys were not necessary in this system. Therefore, encrypted keys were not included.

D-31 Wallet type
In Brabocoin, we use a JBOK wallet, while the Bitcoin core implements an HD wallet. These wallet types are described in Section 6.1. The additional benefits of an HD wallet are of no importance for the educational purpose of Brabocoin. Hence, a JBOK wallet suffices.

7.12 Future work

Due to time constraints, we were not able to implement all features that we would like to see in the Brabocoin software. In this section, we will describe the features that are left as future work.

Address book
It is hard to remember which addresses belongs to which user. Therefore, an address book would be a nice addition to Brabocoin. It would allow users to store addresses together with a name or a description. When a user is creating a transaction, in order to transfer money to somebody in their address book, he would be able to select name instead of entering an address manually.

Import a key
The current wallet is only able to create new key pairs. For these keys, all transactions concerning these keys are stored in a transaction history and a balance is computed. However, it is not possible for a user to include a private key in his wallet that was not created by him. A typical use case of a cryptocurrency is such a *joint account* where several users share the same private key. This way, they are all able to see, for example, the transactions concerning the corresponding address of the key. They are also all able to spend money with this key. In Brabocoin, it is possible for a user to spend money with a key that he does not own. However, he must manually calculate the signatures and find the corresponding unspent outputs. It would be nice to add a feature to Brabocoin that enables users to import keys from other users, in order to enable these joint accounts.

Fork visualization
Only the main chain of the blockchain is displayed in the Brabocoin user interface. However, the blockchain also contains forks. These forks are stored, but they are not displayed to the user. For users to get a better understanding of the blockchain, it would be nice to visualize these forks.

Merkle tree visualization
In every block, the root of the Merkle tree of the hashes of the included transactions is stored in the

block header. For educational purposes, the application can be improved when all intermediate nodes in the Merkle tree structure can be visualized.

Mine a custom block
When a user wants to mine a block, the entire block is constructed automatically. For educational purposes, it is nice to be able to do this construction manually. This would enable a user to, for example, construct invalid blocks, or choose the transactions in the pool they want to mine. This manual construction window could also contain, for example, an explanation of the construction of the Merkle root of a block.

Progress of syncing process
It is not possible to use the Brabocoin application during the initialization process. Currently, that is not clearly displayed; the application is simply disabled. It would be better to see a progress bar that indicates the status of the initialization process.

Blocked peers
It is currently not possible to block peers. A user could remove a peer from his list of peers, but when the removed peer sends a handshake message, he would be added to the list again. Blocking a peer would be desirable if someone spams the network with invalid blocks or transactions.

Protect wallet with asymmetric crypto
The wallet of a user is now protected with AES, which is a symmetric cryptosystem. Thus, when a user wants to save his wallet, he also needs his password. This requires either saving his password when the application is running or asking him for his password when he closes the application (since the wallet needs to be saved when the application is closed). A better approach would be to use asymmetric cryptography to secure the wallet.

Use MAC for storing keys
When a user opens the application, he must unlock his wallet with a password. When the password is incorrect, the decryption process usually fails. However, sometimes it is possible to decrypt a file with an incorrect password, resulting in a corrupted file. To avoid this, Brabocoin always adds a prefix before encrypting. This prefix is the same for every file. If the prefix is different after decryption, the password was incorrect. This approach is, however, not optimal. A better approach would be to use a message authentication code, as explained in Section 4.3.10.

Limit notifications
The amount of notifications is currently only limited by the height of the screen of the user. It would be desirable to be able to limit the amount of notifications to a number chosen by the user.

8 Conclusion

In this report, we have seen an overview of the Bitcoin network. We have introduced the mathematical foundations on which the Bitcoin network is built. In particular, we have explored the various applications of cryptography that are essential for Bitcoin to function as a cryptocurrency. In order to show the inner workings of the Bitcoin network more clearly, we have developed the Brabocoin application.

The Brabocoin application was designed for educational purposes, providing a simple yet accurate implementation of a cryptocurrency similar to Bitcoin. We provide only the essential features of the Bitcoin network, to make the Brabocoin system comprehensible. The result is a more transparent system, making the internal behavior of the software visible to the user. We especially focus on the applications of cryptography, and give the user the ability to manually replicate the cryptographic operations performed by the software. This gives the user the opportunity to experiment with the main features that Bitcoin provides in a constrained, educational environment. We also provide a list of future work of features we deem important in improving the Brabocoin system in the future.

The source code of the Brabocoin application is available on GitHub, accessible via the URL `https://github.com/brabocoin/brabocoin`. Built executables of the software are available for download at `https://brabocoin.org`. On this website, we also provide a user manual.

References

[1] A. M. Antonopoulos. *Mastering Bitcoin*. 2nd ed. O'Reilly Media, Inc., 2017. ISBN: 1491954388.
[2] A. Appleby. *SMHasher*. 2016. URL: https://github.com/aappleby/smhasher (visited on 2019-05-10).
[3] D. Augot et al. *Initial recommendations of long-term secure post-quantum systems*. URL: https://pqcrypto.eu.org/docs/initial-recommendations.pdf (visited on 2019-05-10).
[4] M. Bellare. "New Proofs for NMAC and HMAC: Security without Collision Resistance". In: *Journal of Cryptology* 28.4 (2015), pp. 844–878. DOI: 10.1007/s00145-014-9185-x.
[5] D. J. Bernstein and T. Lange. *Explicit-Formulas Database*. 2009. URL: https://hyperelliptic.org/EFD/ (visited on 2019-05-10).
[6] D. J. Bernstein et al. "SPHINCS: Practical Stateless Hash-Based Signatures". In: *Advances in Cryptology*. Ed. by E. Oswald. Vol. 9056. Lectore Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 368–397. DOI: 10.1007/978-3-662-46800-5_15.
[7] Bitcoin Core. *Bitcoin Core version 0.10.0 released*. 2015. URL: https://bitcoin.org/en/release/v0.10.0 (visited on 2019-05-10).
[8] Bitcoin Core. *libsecp256k1*. 2017. URL: https://github.com/bitcoin-core/secp256k1 (visited on 2019-05-10).
[9] Bitcoin Core. *Obfuscate database files*. 2015. URL: https://github.com/bitcoin/bitcoin/issues/6613 (visited on 2019-05-10).
[10] Bitcoin Wiki. *Base58Check encoding*. 2017. URL: https://en.bitcoin.it/wiki/Base58Check_encoding (visited on 2019-05-10).
[11] Bitcoin Wiki. *Block size limit controversy*. 2019. URL: https://en.bitcoin.it/wiki/Block_size_limit_controversy (visited on 2019-05-10).
[12] BitcoinWiki.org. *Difficulty in Mining*. 2018. URL: https://en.bitcoinwiki.org/wiki/Difficulty_in_Mining (visited on 2019-05-10).
[13] K. Bjoernsen. "Koblitz Curves and its practical uses in Bitcoin security". 2009.
[14] J. W. Bos et al. "Elliptic Curve Cryptography in Practice". In: *Financial Cryptography and Data Security*. 18th International Conference, FC 2014. Ed. by N. Christin and R. Safavi-Naini. Vol. 8437. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 157–175. DOI: 10.1007/978-3-662-45472-5_11.
[15] W. Bosma. "Signed bits and fast exponentiation". In: *Journal de Théorie des Nombres de Bordeaux* 13.1 (2001), pp. 27–41. DOI: 10.5802/jtnb.301.
[16] A. Bosselaers. "RIPEMD Family". In: *Encyclopedia of Cryptography and Security*. Ed. by H. C. A. van Tilborg. Boston, MA: Springer, 2005, pp. 524–527. DOI: 10.1007/0-387-23483-7_360.
[17] J. Buchmann, E. Dahmen, and A. Hülsing. "XMSS – A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions". In: *Post-Quantum Cryptography*. Ed. by B.-Y. Yang. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Berlin, Heidelberg: Springer, 2011, pp. 117–129. DOI: 10.1007/978-3-642-25405-5_8.
[18] J. Buchmann et al. "On the Security of the Winternitz One-Time Signature Scheme". In: *Progress in Cryptology*. AFRICACRYPT 2011. Ed. by A. Nitaj and D. Pointcheval. Vol. 6737. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 363–378. DOI: 10.1007/978-3-642-21969-6_23.
[19] B. Carminati. "Merkle Trees". In: *Encyclopedia of Database Systems*. Ed. by L. Liu and M. T. Özsu. New York, NY: Springer, 2016. DOI: 10.1007/978-1-4899-7993-3.
[20] Certicom Research. *Standards for Efficient Cryptography 1: Elliptic Curve Cryptography*. SEC 1. Certicom Corp., 2009.
[21] Certicom Research. *Standards for Efficient Cryptography 2: Recommended Elliptic Curve Domain Parameters*. SEC 2. Certicom Corp., 2000.
[22] L. Chi and X. Zhu. "Hashing Techniques: A Survey and Taxonomy". In: *ACM Comput. Surv.* 50.1 (2017), 11:1–11:36. DOI: 10.1145/3047307.
[23] T. ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *Advances in Cryptology*. CRYPTO '84. Ed. by G. R. Blakely and D. Chaum. Vol. 196. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1985, pp. 10–18. DOI: 10.1007/3-540-39568-7_2.
[24] N. Ferguson, B. Scheier, and T. Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010. DOI: 10.1002/9781118722367.

[25] R. P. Gallant, R. J. Lambert, and S. A. Vanstone. “Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms”. In: *Advances in Cryptology*. CRYPTO 2001. Ed. by J. Kilian. Vol. 2139. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 190–200. DOI: 10.1007/3-540-44647-8_11.

[26] D. Giry. *Keylength - ECRYPT-CSA Report on Key Sizes*. 2018. URL: https://www.keylength.com/en/3/ (visited on 2019-05-10).

[27] Google Developers. *Protocol Buffers*. 2018. URL: https://developers.google.com/protocol-buffers/ (visited on 2019-05-10).

[28] C. P. L. Gouvêa, L. B. Oliveira, and J. López. “Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller”. In: *J. Cryptogr. Eng.* 2.1 (2012), pp. 19–29. DOI: 10.1007/s13389-012-0029-z.

[29] P. A. Grassi et al. *Digital Identity Guidelines: Authentication and Lifecycle Management*. NIST SP 800-63B. NIST, 2017. DOI: 10.6028/NIST.SP.800-63b.

[30] L. Groot Bruinderink. “Towards Post-Quantum Bitcoin: side-channel analysis of bimodal lattice signatures”. Master’s Thesis. Eindhoven University of Technology, 2016.

[31] L. K. Grover. “A fast quantum mechanical algorithm for database search”. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC ’96. New York, NY: ACM, 1996, pp. 212–219. DOI: 10.1145/237814.237866.

[32] D. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Berlin, Heidelberg: Springer, 2003. DOI: 10.1007/b97644.

[33] M. Hearn and M. Corallo. *Connection Bloom filtering*. BIP 37. Bitcoin Core, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki (visited on 2019-05-10).

[34] J. Hoffstein, J. Pipher, and J. H. Silverman. *An Introduction to Mathematical Cryptography*. 2nd ed. New York, NY: Springer, 2008. DOI: 10.1007/978-0-387-77993-5.

[35] A. Hülsing. “W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes”. In: *Progress in Cryptology*. AFRICACRYPT 2013. Ed. by A. Youssef, A. Nitaj, and A. E. Hassanien. Vol. 7918. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 173–188. DOI: 10.1007/978-3-642-38553-7_10.

[36] J. Jean. *TikZ for Cryptographers*. 2016. URL: https://www.iacr.org/authors/tikz/ (visited on 2019-05-10).

[37] D. Johnson, A. J. Menezes, and S. A. Vanstone. “The Elliptic Curve Digital Signature Algorithm (ECDSA)”. In: *International Journal of Information Security* 1.1 (2001), pp. 36–63. DOI: 10.1007/s102070100002.

[38] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Internet Requests for Comments, 1997. DOI: 10.17487/RFC2104. URL: http://www.rfc-editor.org/rfc/rfc2104.txt (visited on 2019-05-10).

[39] J. Lau and P. Wuille. *Transaction Signature Verification for Version 0 Witness Program*. BIP 143. Bitcoin Core, 2016. URL: https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki (visited on 2019-05-10).

[40] F. Lemmermeyer. *Reciprocity laws: from Euler to Eisenstein*. Berlin, Heidelberg: Springer, 2000. DOI: 10.1007/978-3-662-12893-0.

[41] G. Maxwell. *Core switched to libsecp256k1 for verification. Speedboost! Yeah!* Bitcoin Reddit page. 2015. URL: https://www.reddit.com/r/Bitcoin/comments/3t0kff/core_switched_to_libsecp256k1_for_verification/cx2ljbo/ (visited on 2019-05-10).

[42] G. Maxwell. *secp256k1 library and Intel cpu*. Bitcoin Forum. 2017. URL: https://bitcointalk.org/index.php?topic=1740973.msg17449704#msg17449704 (visited on 2019-05-10).

[43] Hartwig Mayer. “ECDSA security in Bitcoin and Ethereum: a research survey”. 2016.

[44] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Norwell, MA: Kluwer Academic Publishers, 1994. DOI: 10.1007/978-1-4615-3198-2.

[45] A. J. Menezes, S. A. Vanstone, and P. C. van Oorschot. *Handbook of applied cryptography*. Boca Raton, FL: CRC Press, Inc., 1996. ISBN: 0849385237.

[46] R. C. Merkle. “A Certified Digital Signature”. In: *Advances in Cryptology*. CRYPTO ’89. Ed. by G. Brassard. Vol. 435. Lecture Notes in Computer Science. New York, NY: Springer, 1990, pp. 218–238. DOI: 10.1007/0-387-34805-0_21.

[47] M. Mitzenmacher. “Bloom Filters”. In: *Encyclopedia of Database Systems*. Ed. by L. Liu and M. T. Özsu. New York, NY: Springer, 2016. DOI: 10.1007/978-1-4899-7993-3.

[48] S. Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 2019-05-10).

[49] K. S. Okupski. “(Ab)using Bitcoin for an Anti-Censorship Tool”. Master’s Thesis. Eindhoven University of Technology, 2014.
[50] M. Palatinus and P. Rusnak. *Multi-Account Hierarchy for Deterministic Wallets*. BIP 44. Bitcoin Core, 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki (visited on 2019-05-10).
[51] S. Pohlig and M. Hellman. “An Improved Algorithm for Computing Logarithms over and Its Cryptographic Significance (Corresp.)” In: *IEEE Trans. Inf. Theor.* 24.1 (2006), pp. 106–110. DOI: 10.1109/TIT.1978.1055817.
[52] D. Pointcheval and J. Stern. “Security Arguments for Digital Signatures and Blind Signatures”. In: *Journal of Cryptology* 13.3 (2000), pp. 361–396. DOI: 10.1007/S001450010003.
[53] B. Preneel and P. C. van Oorschot. “On the Security of Two MAC Algorithms”. In: *Advances in Cryptology*. EUROCRYPT 1996. Ed. by U. Maurer. Vol. 1070. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 19–32. DOI: 10.1007/3-540-68339-9_3.
[54] R. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321. Internet Requests for Comments, 1992. DOI: 10.17487/RFC1321. URL: http://www.rfc-editor.org/rfc/rfc1321.txt (visited on 2019-05-10).
[55] P. W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: *SIAM J. Comput.* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/S0097539795293172.
[56] J. H. Silverman. *The Arithmetic of Elliptic Curves*. 2nd ed. New York, NY: Springer, 2009. DOI: 10.1007/978-0-387-09494-6.
[57] M. Stevens et al. “Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate”. In: *Advances in Cryptology*. CRYPTO 2009. Ed. by S. Halevi. Vol. 5677. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 55–69. DOI: 10.1007/978-3-642-03356-8_4.
[58] G. Tornaría. “Square Roots Modulo p”. In: *LATIN 2002: Theoretical Informatics*. LATIN 2002. Ed. by S. Rajsbaum. Vol. 2286. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 430–434. DOI: 10.1007/3-540-45995-2_38.
[59] X. Wang and H. Yu. “How to Break MD5 and Other Hash Functions”. In: *Advances in Cryptology*. EUROCRYPT 2005. Ed. by R. Cramer. Vol. 3494. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 19–35. DOI: 10.1007/11426639_2.
[60] B. de Weger. “ISTS 3USU0 Cryptography”. 2017.
[61] P. Wuille. *Hierarchical Deterministic Wallets*. BIP 32. Bitcoin Core, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki (visited on 2019-05-10).