Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Zero-Knowledge Proofs of Innocence From Deterministic Wallets

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Cordian Alexander Daniluk
geboren am: 21.04.2000
geboren in: Berlin

Gutachter/innen: Prof. Dr. Scheuermann
Prof. Dr.-Ing. Graß

eingereicht am: ..... verteidigt am: .....

Contents

1. Introduction

Cryptocurrencies are currencies in digital form that allow their users to transact money without relying on a third party such as a bank. The first cryptocurrency is Bitcoin, which was first presented in 2009.
From its first presentation till now, Bitcoin has made its way into mainstream. As the first cryptocurrency, it has spawned an abundance of other ones: there are about eleven thousand cryptocurrencies with a market capitalization of two trillion dollars in total (see the website CoinMarketCap [1]).
Bitcoin still dominates the market by a large margin: 44% of the market is in Bitcoin followed by Ethereum, which represents 18% (see website CoinMarketCap [1]). However, the gain in anonymity by using cryptocurrencies and especially Bitcoin attracts all sorts of criminals: according to Foley et al. [2], till 2017, 46% of all Bitcoin transactions were due to illegal activity.
The inner workings of Bitcoin allow its users to stay anonymous to a certain degree. This poses a problem for law enforcement, which would like to trace back illegal activity and identify the perpetrators.
Besides the lack of a third party, there exists a mechanism to merge multiple transactions into one. This renders the association between senders and receivers non-obvious; a participant can thereby conceal to whom a transaction is sent. An important subcase of this is transactions to onself. Since Bitcoin is based on pseudonymity—a user usually has many pseudonyms—sending one user's money from one to another pseudonym is a way to further cover the user's tracks. In fact, these "wash trades" as Foley et al. [2] call them have been used as a characteristic of illegal activities in their paper.
To prosecute criminals despite this method, Keller et al. [3] have come up with "collaborative deanonymization": well-meaning participants in a wash trade prove their innocence in order to single out the criminal. However, since a proof of innocence involves proving possession of a sending and receiving pseudonym, stealing one of these from a well-intentioned user allows criminals to counterfeit such proofs. To remedy this problem, some refinement of the notion of pseudonyms is defined: the sending and receiving ones must be from the same deterministic wallet, i.e., they need to share a common root. Proving possession of two pseudonyms now amounts to proving possession of the root and some data identifying and distinguishing the two pseudonyms. Since the root is supposed to be kept secure at all costs, possessing the root is the ultimate proof for possessing the pseudonyms. Therefore, proving possession of the root by revealing it is no option: anyone could repeat this "proof" and knowledge of the root implies knowledge of all pseudonyms, which is unacceptable. But how to prove that two such pseudonyms belong to the same deterministic wallet without revealing any sensitive information?

1.1. Contribution

The question of how to perform such a proof of innocence without revelation of sensitive information has been posed by Keller et al. [3]. This thesis attempts to answer it using

a type of proof called zero-knowledge proofs: this technique allows proofs of innocence of willing participants without revealing any sensitive information such as the common root.
The implementation of such zero-knowledge proofs is the main contribution of this thesis. Two different approaches have been investigated: one using multi-party computation, another one directly using a type of zero-knowledge proof called zk-SNARK. Their implementations are compared against each other and discussed. In addition, applicability in the real world is examined.
Other than this, much effort has been put into vulgarizing the basic concepts such as zero-knowledge and multi-party computation. This also involved intensive study of literature.

1.2. Outline

This thesis has the following structure: Section 2 explains the basics needed to understand what follows. Readers with a sufficient level of expertise can skip this part. Section 3 describes the zero-knowledge proof implementations. It is the central contribution of this thesis. Section 4 evaluates how the implementations have performed. Section 5 summarizes what this thesis has accomplished and describes possible future work. Appendix A and Appendix B contain the relevant code of the MPC and zk-SNARK implementations, respectively.

2. Background

This section shall give sufficient background to enable any reader to understand this thesis. Cryptographic hash functions are introduced first. Bitcoin and its relevant technical aspects follow. Next come arithmetic circuits, a similar model to boolean circuits. This is followed by the two foundations leveraged to construct solutions to the problem in this thesis: zero-knowledge proofs and multi-party computation. In the context of the former, special attention is payed to zk-SNARKs, the type of zero-knowledge proof used in this thesis.

2.1. Cryptographic Hash Functions

A ubiquitous component of cryptographic applications is the *cryptographic hash function* (CHF). CHFs serve purposes as different as verification of data integrity, password verification, or as part of the proof-of-work scheme employed by Bitcoin (see Nakamoto [4]). Their properties will prove useful when formulating what this thesis attempts to solve.
First comes a very basic definition of CHFs. This is followed by a particular construction of CHFs notably used for the SHA-256 function (see FIPS 180-2 [5]).

2.1.1. Definition

The following definition of CHFs is from NIST Special Publication 800-106 [6]:

Definition 1. *A cryptographic hash function is a function* $H\colon \{0,1\}^* \to \{0,1\}^n$ *for some* $n \in \mathbb{N}$ *that fulfills the following properties:*

1. *Collision resistance: it is computationally infeasible to find* $x_1, x_2 \in \{0,1\}^*$ *such that* $x_1 \neq x_2$ *and* $H(x_1) = H(x_2)$.

2. *Preimage resistance: given a digest* $y \in \{0,1\}^n$*, it is computationally infeasible to find* $x \in \{0,1\}^*$ *such that* $y = H(x)$.

3. 2^{nd}*-preimage resistance: given* $x_1 \in \{0,1\}^*$ *and* $y \in \{0,1\}^n$ *such that* $y = H(x_1)$*, it is computationally infeasible to find* $x_2 \in \{0,1\}^*$ *such that* $x_2 \neq x_1$ *and* $y = H(x_2)$.

Another desirable property of CHFs is the *avalanche effect* (see Al-Kuwari et al. [7]): if a single input bit is changed, the output should change drastically. There should be no statistical correlation between input and output.
The groundwork for talking about CHFs has been laid. Next comes a certain framework frequently used to construct CHFs.

2.1.2. Merkle-Dåmgard Construction

This construction was independently described by Merkle [8] and Dåmgard [9], hence the name. Its description closely follows Menezes et al. [10].
Let there be an input $x \in \{0,1\}^*$. It will be split into blocks $x_i \in \{0,1\}^r$ for some $r \in \mathbb{N}$. To this end, x needs to be padded such that its length becomes a multiple of r. For security reasons, a length block containing the length of x is also appended.
The result is a padded $x = x_1 x_2 \cdots x_t$ where $t \in \mathbb{N}$ is the number of blocks. The length of x therefore equals $r \cdot t$.
Afterwards, the blocks are fed one by one into $f\colon \{0,1\}^s \times \{0,1\}^r \to \{0,1\}^s$ for some $s \in \mathbb{N}$, the *compression function* of h. In the i^{th} iteration, f takes the previous result $H_{i-1} \in \{0,1\}^s$ and the next block $x_i \in \{0,1\}^r$ to compress these into a new result $H_i \in \{0,1\}^s$. Concisely, this is described by

$$H_i = f(H_{i-1}, x_i) \qquad\qquad 1 \leq i \leq t$$

In the very first iteration, $H_0 \in \{0,1\}^s$ is some value called the *initializing value* (IV). Finally, a *finalisation step* is performed on H_t, modeled by the function $g\colon \{0,1\}^s \to \{0,1\}^n$ where n is the digest length as in Definition 1. If this step does nothing, simply set $s = n$ and $g(H_t) = H_t$.
The entire process can be seen in Figure 1.

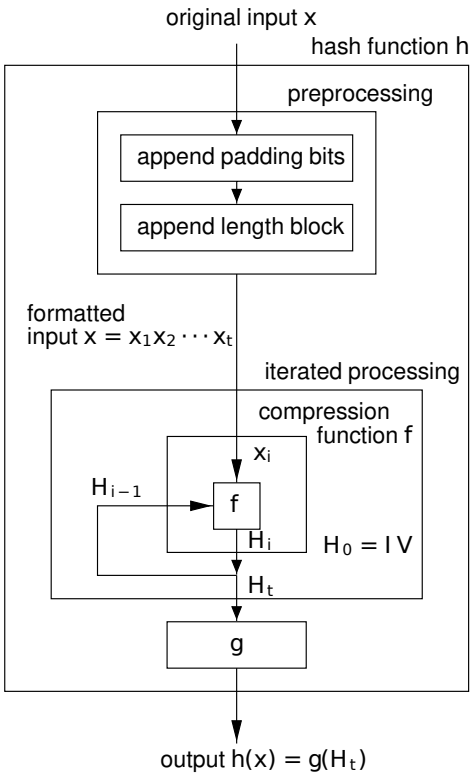

Figure 1: Merkle-Dåmgard construction [10].

2.2. Bitcoin

Bitcoin is an immensely popular cryptocurrency. The central feature of Bitcoin is the blockchain, which is explained in the beginning to gain a rough understanding of how Bitcoin works in general. This is followed by how transactions work. Related to transactions is CoinJoin, a way to merge many transactions into one. It will be introduced afterwards. Finally, the technique of collaborative deanonymization is described to trace back perpetrators via Bitcoin transactions.

2.2.1. The Blockchain

Bitcoin is supposed to eliminate the need for trusting in a third party to exchange money. The basics of Bitcoin are described by Nakamoto [4]. While usually there is a third party such as a bank, which must be trusted in order to transact money, Bitcoin attempts to eliminate this need via cryptographic proof. Two parties can exchange transactions directly by means of cryptography.
However, if parties exchange money directly, the problem of *double-spending* emerges. Imagine that one party sends the same money, the same coins so to speak to two different parties. This is of course problematic and not even possible with physical money. Even if there is no central bank or other third party to keep track of all money, double-spending must be made impossible.

The solution consists in changing from transactions kept track of by a trusted third party to transactions in a central data structure that everybody knows. Since all transactions are public in this data structure, the so-called *blockchain*, everybody can determine double-spending.
If there is no central party, though, someone needs to decide on what the "correct" blockchain is. This is done collaboratively by all users via a *consensus protocol*.
The blockchain is literally a chain of blocks, each of which contains a multitude of transactions. Every block contains a hash digest of the preceding one, which chains the blocks to one another. New blocks are added by means of a proof-of-work system where users perform work in order to solve a computational problem related to the new block. It is incentivized by an amount of Bitcoin money a user receives in return for his work.
This is a rough overview how the double-spending problem is mitigated using the blockchain. At the heart of this thesis lies the concept of transactions and their implementation by Bitcoin, though.

2.2.2. Transactions

Transactions in Bitcoin have inputs and outputs. The former refer to the various sources of the money, the latter to its destinations. Each input references an output from another transaction. Each output can be referenced later on by the input of a new transaction. In the meantime, though, it is an *unspent output*, waiting to be spent by the person to which this output was destined.
In Bitcoin, transactions happen directly without a third party. Unlike physical money, it should be proved that money really belongs to the person who wants to spend it. This proof happens cryptographically using a digital signature scheme: the elliptic curve digital signature algorithm (ECDSA) [11].
Every output contains a public key, which belongs to some user. This user is capable of spending the output because he knows the corresponding private key. He spends it by signing the output with ECDSA and specifying a recipient, i.e., a public key. This is merged into a new transaction: the signature goes into an input, the recipient's public key goes into the output. Anyone can check that the public key belonged to the spender by verifying the signature with that public key. The transaction is then broadcoast to all other users such that the transaction is finally included in the blockchain.
A transaction is not required to contain only a single input and a single output. A common scheme is that of two outputs: one to pay, another for the change routed back to oneself. There can be even more, though. Similarly, multiple inputs can be used for one transaction since users may have many unspent outputs that they want to add.

2.2.3. Wallets

To spend and receive money, every user needs a pair of private and public keys. If this key pair was used for every transaction, though, there would be a one-to-one mapping between user and public key. Since the blockchain is public, anyone could search all

transactions to find out how much money a user currently has. If anyone happened to learn the identity of the user behind one public key he could instantly see all money spent and received by that user. Instead, users use many different key pairs, ideally one key pair per transaction. Connecting transactions by common public keys is therefore no longer possible.
The set of all key pairs that belong to a user is stored in a *wallet*. Similarly to a physical wallet that contains physical money, a Bitcoin wallet contains the keys to spend and receive money.
It is possible to store the key pairs just like that, as a loose set of keys. These wallets are called *just-a-bunch-of-keys* (JBOK) wallets [12]. However, backing these up means to store all of them elsewhere. Furthermore, losing a key is final, it cannot be easily recuperated without backup.
To solve these problems, *deterministic wallets* are used. All private keys are generated in a deterministic manner from a *seed*. This generation is called *key derivation*. The corresponding public keys can be generated from the private keys using the ECDSA-specific key generation routine (see Johnson et al. [11] for details).
A special type of deterministic wallets are *hierarchical deterministic* (HD) wallets. Keys can not only be derived from the seed, but also from other keys. This recursive mechanism allows for a tree-like, hierarchical derivation structure. Furthermore, public keys can not only be derived from private keys, but also from other public keys.
HD wallets are implemented in Bitcoin Improvement Proposal (BIP) 32 [13]. BIP 32 defines the functions *CKDpriv* and *CKDpub*. CKDpriv takes a parent private key and a four-byte index and produces a child private key. CKDpub takes a parent public key and a four-byte index and produces a child public key. The seed is converted to a *master private key* m, which has a corresponding *master public key* M. Keys are derived from m and M, that is, indirectly from the seed. A derived private key could be priv = CKDpriv(CKDpriv($m, 1$), 3). The corresponding public key would be pub = CKDpub(CKDpub($M, 1$), 3). These form a key pair, so given the ECDSA-specific key generation routine [11], call it pubgen, pubgen(priv) = pub. Therefore, public keys can be derived separately from M or generated from private keys.
Note that details and irrelevant information have been omitted. For full details, see BIP 32 [13].
With deterministic wallets, backing up all keys is reduced to backing up the seed (or master private key for BIP 32). All keys can be generated deterministically from that. Hence, losing a key is not possible as long as the seed is stored. Using deterministic wallets therefore facilitates following the guideline of using one key pair per transaction.

2.2.4. CoinJoin

Even using one key pair per transaction suffers from shortcomings, however, since key pairs can be traced back to the owner by other means (see Goldfeder et al. [14]). Therefore, a scheme called *CoinJoin* [15] was invented. It leverages the fact that the set of inputs is totally unrelated to the set of outputs in a transaction. There is only the sum of inputs (the money available) and the sum of outputs (the money spent).

For this reason, different users can merge their transactions into one to conceal their traces. Now, an outsider can no longer tell which input belonged to which output due to the independence of inputs and outputs. This renders the task of following money trails non-trivial. However, this added untraceability also allows malicious users to abuse the system for their vicious needs and hardens the task of legal prosecution.

2.2.5. Collaborative Deanonymization

To combat malicious actors who take advantage of this increased degree of anonymity via CoinJoin, Keller et al. [3] propose a scheme that they call *collaborative denanonymization*. This scheme counteracts anonymity and, in particular, untraceability gained by using CoinJoin transactions.
CoinJoin can be used to send money between different keys of the same owner: in a CoinJoin transaction, for every input there exists an output such that both are controlled by the same person. Assuming every participant does this, now it is not clear which output belongs to which input. The more users participate, the more possibilities there are.
If in such a CoinJoin transaction any output's public key has somehow been identified as "bad", law enforcement would like to trace back that money to its origin. This involves finding the corresponding input to the "bad" output. Collaborative deanonymization consists of *witnesses*, that is, users who control inputs of the respective transaction. They testify using their private keys to affirm control over outputs different from the "bad" one and respective inputs. This narrows down the transaction input from where the malicious actor procured his money, even uniquely identifying him if everyone participates. Traceability is thereby regained to a certain degree.

2.3. Arithmetic Circuits

Arithmetic circuits are a model of computation. They serve to describe computational statements that can be proved using a subtype of zero-knowledge proofs, the zk-SNARK. The following definition describes arithmetic circuits such that it will suit the remainder of this thesis.

Definition 2. *An arithmetic circuit C over a field $\mathbb{F}$ is a directed acyclic graph. Its vertices are called gates, its edges are called wires. The gates are multiplication, addition, or multiplication-by-scalar gates.*
Let there be $n \in \mathbb{N}$ wires. An input wire has no gate, which it leaves, but is incident to a gate. $n = n_{primary} + n_{aux}$ where $n_{primary} \in \mathbb{N}$ is the number of primary input wires and $n_{aux} \in \mathbb{N}$ the number of auxiliary wires. One auxiliary wire is an input wire that always carries the constant value 1 to be able to deal with constants. Using 1 as an input wire to a multiplication-by-scalar gate, any constant in $\mathbb{F}$ can be represented.
A is satisfiable with an input $x \in \mathbb{F}^{n_{primary}}$ if and only if there exists a wire assignment $w \in \mathbb{F}^n$ such that

1. $x = (w_1, \ldots, w_{n_{primary}})$

2. *for every arithmetic gate g, the input to g matches its output: for an addition gate, the summands (inputs) must yield the correct sum (output); for a multiplication gate, the factors (inputs) must yield the correct product (output); for a multiplication-by-scalar gate, the scalar is hard-coded and, when multiplied with the input, must yield the correct product (output).*

The scalars hard-coded in the multiplication-by-scalar gates are elements of $\mathbb{F}$. All computation is performed over $\mathbb{F}$.
This case is described as $C(x, w) = 1$, i.e., C is satisfied.

Figure 2 shows an example for an arithmetic circuit. The "subtraction gate" is constructed by combining a multiplication-by-scalar gate with scalar -1 and an addition gate.
Note that computing if $C(x, w) = 1$ for some $x \in \mathbb{F}^{n_{primary}}$ and $w \in \mathbb{F}^n$ is feasible in polynomial time.
Let A be an arithmetic circuit and n and $n_{primary}$ be defined as in Definition 2. Define

$$\text{ARITH-SAT} = \left\{ (C, x) \;\middle|\; \begin{array}{l} C \text{ is an arithmetic circuit and } x \in \mathbb{F}^{n_{primary}}, \\ \exists w \in \mathbb{F}^n \colon C(x, w) = 1 \end{array} \right\}$$

In short, $(C, x) \in$ ARITH-SAT if and only if C can be satisfied for the input x. ARITH-SAT $\in$ NPC as mentioned in Groth16 [16].
Let C be an arithmetic circuit. Define

$$L_C = \{x \mid \exists w \colon C(x, w) = 1\}$$
$$R_C = \{(x, w) \mid C(x, w) = 1\}$$

$L_C \in$ NP, obviously.

2.4. Zero-Knowledge Proofs

This section will introduce the notion of zero-knowledge proofs, a type of proof that gives away no information but the truth of the statement to be proved.
A toy example will be used to introduce the subject. Afterwards, an overview of interactive zero-knowledge proofs will be given with an effort to formalize the toy example. Non-interactive zero-knowledge proofs in the common reference string model will be outlined, which is used by the next section that introduces and defines zk-SNARKs.

2.4.1. A Toy Example

Consider the the following scenario described by Goldreich [18]: a person is trying to prove to another color-blind person that two balls differ in color. The balls are

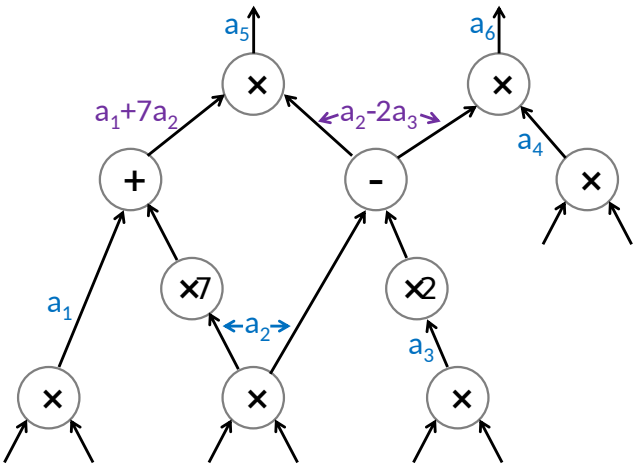

Figure 2: A sample arithmetic circuit [17]. Subtraction is possible by using a multiplication-by-scalar gate with the scalar -1.

indistinguishable for any person except with regard to their colors. The proving person can therefore distinguish them, the color-blind person cannot, however.
The proving person A now wants to prove to the color-blind person B that the balls actually have different colors.
Let $k \in \mathbb{N}$. The two follow the following protocol:

1. B flips a coin.
2. If it shows heads, B swaps positions of the balls. If it shows tails, B does nothing. A has their eyes closed and is prevented from hearing during this process.
3. A needs to deduce whether B flipped heads or tails.
4. If he deduces the wrong side of the coin, which B can verify, the proof has failed. If he deduces the correct side of the coin, go back to step 1. Repeat at most k times, however.
5. The proof has succeeded.

This protocol's correctness needs to be shown, formalized as

$$A \text{ succeeds with his proof} \iff \text{The balls differ in color} \tag{1}$$

This way, the protocol always conveys the truth, A cannot prove a wrong statement, nor can A fail proving a correct one. B is assumed to be honest here: the color-blind person does not want to trick A and follows the protocol exactly.
If the balls differ in color, A will always see whether B swapped the balls or not. A will therefore always know if B flipped heads or tails, so after k times, the proof has

succeeded. This was the $\Longleftarrow$ direction. Now, look at the contrapositive of the $\Longrightarrow$ direction, which is

$$\text{The balls have the same color} \implies A \text{ fails with his proof}$$

If the balls have the same color, A can impossibly know if the balls were swapped or not. The color is the only feature, from which A could deduce an exchange. Nor can A know if B flipped heads or tails for that reason. All A can do is guess the side of the coin. A needs to guess correctly k times in a row, though, which only has a probability of $\frac{1}{2^k}$. For this reason, A probably fails to prove that the balls differ in color.
Note that (1) is only proved probabilistically. There is still a slim chance that it does not hold for the given protocol.
Last but not least, this protocol is zero-knowledge. A has not conveyed any information apart from the fact that the balls differ in color such as which of the balls has which color. Put differently, B knows nothing new after the proof aside from the (probable) truth of the statement. Because of this, B is still unable to prove to anyone else that the balls differ in color.

2.4.2. Interactive Zero-Knowledge Proofs

What was described in Section 2.4.1 is an *interactive proof system* (A, B) for the language

$$L_{color} = \{(b_1, b_2) \mid b_1 \text{ and } b_2 \text{ are balls that differ in color}\}$$

An interactive proof system *for a language* $L \subseteq \{0,1\}^*$ is an interactive proof system, which guarantees completeness and soundness for L as (A, B) does for L_{color}. An *interactive proof*, i.e., interaction between two parties, the *prover* and the *verifier*, leads to a statement $(b_1, b_2) \in L_{color}$ being proven to the verifier. The prover A has unlimited computational power ("A can see colors"), while the verifier B has limited, polynomial computational power ("B is color-blind").
The correctness of the proof system (A, B) has been shown in Section 2.4.1. The $\Longrightarrow$ direction is called *completeness*, the $\Longleftarrow$ direction (or, equivalently, its contrapositive) *soundness*.
What is still missing is the zero-knowledge property. An *interactive zero-knowledge proof* reveals no knowledge to the verifier but the truth of the statement. Knowledge is defined to be anything *infeasible to compute* for the party in question (the verifier). Given a language $L \in \mathsf{NP}$, an interactive proof system (P, V) for a language L, and the input $x \in L$, an example of knowledge would be the witness w of the input x, which fulfills the equation $A(x, w) = 1 \iff x \in L$ for a polynomial-time algorithm A (the definition of NP). Assuming that $\mathsf{P} \neq \mathsf{NP}$, w is infeasible to compute for the polynomial-time verifier V and thus V must not learn the value of w. Similarly, for (A, B), the verifier B must not learn any knowledge.

Let (P, V) now be an interactive proof system for any language $L \subseteq \{0,1\}^*$. Let $(P, V)[x]$ be the set containing the probabilities of what messages are exchanged between P and V on input x. Put differently, $(P, V)[x]$ is the probability distribution of what the conversation between P and V looks like on input x. The zero-knowledge property is formalized by a *simulator argument*: (P, V) is zero-knowledge, i.e., communicates no knowledge if there exists a probabilistic polynomial-time simulator S, which can *simulate* $(P, V)[x]$ for all inputs $x \in L$.
Informally, simulating $(P, V)[x]$ means to reproduce the probability distribution $(P, V)[x]$, which is why S is probabilistic. Denote the probability distribution of the output of S on input x as $S[x]$. For each $x \in L$, S simulates $(P, V)[x]$ if $S[x]$ is indistinguishable from $(P, V)[x]$. In other words, the probability that S reproduces a certain conversation is indistinguishable from the probability that $(P, V)[x]$. "indistinguishable" informally means that no probabilistic polynomial-time algorithm can tell a difference between the two distributions.
Now, consider a proof system (P, V) that leaks knowledge to the verifier V. A simulator S would have to reproduce $(P, V)[x]$ for all $x \in L$, i.e., output the same conversations with indistinguishable probabilities. S cannot compute the leaked knowledge himself because it is only as powerful as the verifier V. Hence, S is not able to simulate $(P, V)[x]$.
To show informally that the toy example from Section 2.4.1 is zero-knowledge, let S be a simulator. Since S is probabilistic it can reproduce all of B's coin tosses. Note that this does not mean that the literal toss results, heads or tails, are reproduced, only the probabilities that heads or tails occur. Based on the coin tosses, S can simulate B's messages, that is, switch balls or not: S and B run in polynomial time, they have the same capabilities so to speak. Since S knows the result of each coin toss, S can also reproduce A's messages: given $(b_1, b_2) \in L_{color}$, S can always give the correct answer as it knows the coin tosses.
In summary, S can reproduce $(A, B)[(b_1, b_2)]$ for all $(b_1, b_2) \in L_{color}$.
For the full formal definition of interactive zero-knowledge proofs, see Goldwasser et al. [19]. For a comprehensive discussion on zero knowledge in general, see Goldreich and Oren [20].
Interactive zero-knowledge proofs serve well to understand the notion of zero knowledge. In practice, they have little use, though. Especially interaction is often not desired. Fortunately, there exists a non-interactive alternative. Most of the concepts treated in this section will remain relevant, though.

2.4.3. The Common Reference String Model

A way to leverage the zero-knowledge property of proof systems without interaction are *non-interactive zero-knowledge* (NIZK) proofs. The prover creates the NIZK proof on his own and passes it to the verifier, which verifies it on his own afterwards.
To enable NIZK proofs for all languages $L \in \mathsf{NP}$, a model was constructed by Blum et al. [21], in which both prover and verifier share a common string of bits. This common string is called the *common reference string* (CRS) and the model is the CRS model.

It has to be generated prior to creating the proofs themselves. NIZK proofs in the CRS model exist for any language $L \in \mathsf{NP}$.
One major problem with the CRS model is that the CRS has to be trusted. Anyone who can decide the CRS on their own would be able to create a NIZK proof that attests to the truth of a false statement. Therefore, the CRS must be generated by a *trusted setup*. If there is no perfect trustworthy party, and there usually is none, no single party can be responsible for generating because no single party is trustworthy enough. Zcash, for instance, used an MPC "ceremony" between close to a hundred people to generate its CRS (or "public parameters" as they are called by Zcash; see Zcash Parameter Generation [22] for the ceremony and Section 2.6 for what MPC is). The CRS model is relevant as it is used for the type of proof introduced in the next section.

2.4.4. Preprocessing zk-SNARKs

To make zero-knowledge proofs fit for practice, *zero-knowledge succinct non-interactive arguments of knowledge* (zk-SNARKs) have been developed. They were first described by Bitansky et al. [23]. The cryptocurrency Zcash [24] notably uses zk-SNARKs to establish security. zk-SNARKs are NIZK proofs with additional properties, which have rendered them usable in practice. The central one is that they require proofs to be small and verification to take a short time. zk-SNARKs and its properties will be defined first, some explanation will follow.
The following (rather informal) definition of preprocessing zk-SNARKs, a special type of zk-SNARKs, is a combination of GGPR13 and BCTV14 [25]. The former defines zk-SNARKs while the latter extends this to prepocessing zk-SNARKs.

Definition 3 (Preprocessing zk-SNARK)**.** *A (preprocessing) zk-SNARK for arithmetic circuit satisfiability over a field $\mathbb{F}$ is a triple of polynomial-time algorithms (G, P, V) called key generator, prover, and verifier.*
The key generator G, given a security parameter κ and an arithmetic circuit C over $\mathbb{F}$ creates a proving key $\boldsymbol{pk}$ and a verification key $\boldsymbol{vk}$. The security parameter denotes the level of security achieved by the zk-SNARK.
The prover P generates proofs using $\boldsymbol{pk}$. That is, the proof $\pi = P(\boldsymbol{pk}, x, w)$ is created, attesting that $(x, w) \in R_C$.
The verifier V verifies the proof using $\boldsymbol{vk}$. That is, $V(\boldsymbol{vk}, x, \pi)$ equals 1 if π has been deemed a valid proof attesting that $x \in L_C$. Otherwise, it equals 0.
A (preprocessing) zk-SNARK (G, P, V) satisfies the following properties:

1. ***Completeness**: being given $(x, w) \in R_C$ and*

$$(\boldsymbol{pk}, \boldsymbol{vk}) = G(1^\kappa, C)$$
$$\pi = P(\boldsymbol{pk}, x, w)$$

the probability that $V(\boldsymbol{vk}, x, \pi) = 0$ is very close to 0.

2. ***Soundness**: being given $x \notin L_C$, the keys*

$$(\boldsymbol{pk}, \boldsymbol{vk}) = G(1^\kappa, C)$$

and a (malevolent) polynomial-time prover who provides a proof π, the probability that $V(\boldsymbol{vk}, x, \pi) = 1$ is very close to 0.

3. ***Succinctness**: the proof π's size is a polynomial in the security parameter κ and V runs in time proportional to the length of x in bits and polynomial in κ.*

4. ***Knowledge of Witness**: the prover knows the witness w to the input x such that $(x, w) \in R_C$. More formally: for any $x \in \{0,1\}^*$ there exists a polynomial-time extractor $\mathcal{E}_x$ such that for any $\pi = P(\boldsymbol{pk}, x, w)$, $\mathcal{E}_x$ retrieves the witness w: $w = \mathcal{E}_x(\pi)$.*

5. ***Zero-Knowledge**: the proof π reveals nothing but the fact that $x \in L_C$.*

Note that it is conventional to encode the security parameter κ in unary as 1^κ.

Define an NP *statement* to be any statement of the form $x \in L$ for $L \in \mathsf{NP}$. By Definition 3, preprocessing zk-SNARKs prove NP statements. Since Definition 3 defines a "zk-SNARK for arithmetic circuit satisfiability", these must be of the form $(C, x) \in$ ARITH-SAT for some arithmetic circuit C and input x. This is no restriction, though: ARITH-SAT is NP-complete (see Section 2.3), so any NP statement $x \in L$ can be proven by first reducing L to ARITH-SAT using Karp reduction. How efficient this reduction is depends on L. If L is rather arithmetically oriented, e.g., proving knowledge of an integer factorization, the reduction is more efficient than for a logically oriented L.
Next, notice the analogy between Definition 3 and interactive zero-knowledge proofs from Section 2.4.2. Completeness, soundness, and zero-knowledge are essential components of both of them. However, note that the zk-SNARK only guarantees *computational soundness*: given a prover that runs in polynomial time, soundness is given. Security against a superpolynomial prover is not guaranteed. This is why a zk-SNARK is not a proof, but an *argument*.
Furthermore, the zero-knowledge property is a little different than that of interactive zero-knowledge proofs. Once again, a simulator argument is used. The simulator only simulates the creation of the proof by the prover based on `pk` and x; there is no "conversation" to be simulated. Prior to that, the simulator must generate `pk` and `vk`. The full formalism is still a little more complex, see GGPR13 [17] for more information.
There are also two new properties: succinctness and knowledge of witness. The former renders proofs feasible in reality: proof sizes are independent of all parameters except the security parameter. Hence, for a given security parameter, proof sizes are constant. They cannot grow with the size of the arithmetic circuit, for example. Nor can the

verifier take too much time: for a fixed security parameter, he can only take time linear in the size of x.
Knowledge of witness means that the prover P knows the witness to some input. Showing that $x \in L_C$ per se only requires the mere existence of a witness w such that $(x, w) \in R_C$. A zk-SNARK enforces that w not only exists but that P knows w. This is why a zk-SNARK is an argument of knowledge.
Preprocessing zk-SNARKs work in the CRS model. The CRS in this case consists of the proving and verifying keys `pk` and `vk`. The arithmetic circuit C is encoded in `pk` and `vk`. The keys are not identical, though.
Numerous approaches to zk-SNARKs making use of different cryptographic tools have been proposed and some of them are even implemented in software. Amongst those are PGHR13 [26], BCTV14 [25], and Groth16 [16], which will be referenced throughout this thesis.

2.5. Creating zk-SNARKs

The goal of the next few sections will be to show in some more detail how this thesis will create a preprocessing zk-SNARK for any NP statement.
According to Definition 3 in Section 2.4.4, the following operations must be implemented to create a zk-SNARK:

1. $(\texttt{pk}, \texttt{vk}) = G(C, 1^\kappa)$
2. $\pi = P(\texttt{pk}, x, w)$
3. $V(\texttt{vk}, x, \pi)$

All identifiers are defined as in Definition 3. Note that C is an arithmetic circuit.
The operations used in this thesis are identical except that no arithmetic circuit is passed to G:

1. $(\texttt{pk}, \texttt{vk}) = G(S, 1^\kappa)$
2. $\pi = P(\texttt{pk}, x, w)$
3. $V(\texttt{vk}, x, \pi)$

S is a so-called R1CS. It is a structure very similar to arithmetic circuits. In fact, there is the problem of R1CS satisfiability (R1CS-SAT), which is NP-complete like ARITH-SAT (see Section 2.3). Hence, R1CS-SAT and ARITH-SAT are equivalent and using R1CS or arithmetic circuits to describe NP statements is non-fundamental.
The following sections will explain the necessary parts to create a zk-SNARK. R1CS will be explained first. As part of G, the R1CS S is first converted to yet another format called QAP. How this conversion happens will come next. Finally, G creates the keys based on that QAP and P and V do their respective work. These last steps are described in the last section.

2.5.1. R1CS

To talk about R1CS, some mathematical notation is needed.

Definition 4. *Let $\mathbb{F}$ be a field. For two vectors $U, V \in \mathbb{F}^n$ for some $n \in \mathbb{N}$, the dot product $\langle U, V \rangle$ is defined as*

$$\langle U, V \rangle = \sum_{i=1}^{n} U_i V_i$$

Definition 5. *Let $\mathbb{F}$ be a field. For a vector $v \in \mathbb{F}^n$, define*

$$(1, v) = (1, v_1, \ldots, v_n)$$

The vector $(1, v)$ starts with the index 0 such that $(1, v)_0 = 1$.

A *rank-one constraint system* (R1CS) is a model of computation, just like arithmetic circuits (see Section 2.3) are. It describes a set of linear algebraic *constraints*. They are the equivalent of an arithmetic gate and describe an equation that defines the output of the gate.
The following definition is from BCGTV13 [27]:

Definition 6. *A rank-one constraint system or R1CS S over a field $\mathbb{F}$ is a tuple $S = (\{(A_i, B_i, C_i) \mid 1 \leq i \leq n_c\}), n_{primary})$ where $A_i, B_i, C_i \in \mathbb{F}^{n+1}$.*
The system is called satisfiable for an input $x \in \mathbb{F}^{n_{primary}}$ if there exists a witness vector $w \in \mathbb{F}^n$ such that

$$x = (w_1, \ldots, w_{n_{primary}}) \tag{2}$$

and

$$\langle A_i, (1, w) \rangle \cdot \langle B_i, (1, w) \rangle = \langle C_i, (1, w) \rangle \qquad \forall i \in \{1, \ldots, n_c\} \tag{3}$$

What wires were in arithmetic circuits are *variables* in R1CS. n_c is the number of constraints, $n_{primary}$ the number of input variables, and n the total number of variables. Therefore, it must also hold that $n_{primary} \leq n$. w can be thought of as the variable assignment. (2) states that the input variables contained in x are a part of w as w contains all variables. (3) expresses that every single constraint needs to be satisfied by w in order for the entire system to be satisfied and therefore satisfiable. If both conditions are fulfilled, we write concisely $S(x, w) = 1$, i.e., S is satisfied. Note that computing if $S(x, w) = 1$ is feasible in polynomial time.
Take a look at Figure 3 for some rather simple examples. It is always assumed that $w = (w_1, w_2, w_3)$. Note the constant 1 in $(1, w)$. It serves to express constants other

$$
\begin{aligned}
w_1 \cdot w_2 = w_3 \qquad & A = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, B = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, C = \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \\
w_1 + w_2 = w_3 \qquad & A = \begin{pmatrix}0\\1\\1\\0\end{pmatrix}, B = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, C = \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \\
1 - w_1 = w_2 \qquad & A = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, B = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, C = \begin{pmatrix}0\\1\\1\\0\end{pmatrix}
\end{aligned}
$$

Figure 3: Simple R1CS constraints. $w = (w_1, w_2, w_3)$.

than 0, just like in arithmetic circuits. That is not possible with the variables in w alone. Consider $w_1 + w_2 = w_3$ in the figure, which exemplifies this construction. $w_1 + w_2$ is expressed by $\langle A, (1, w)\rangle$, so $\langle B, (1, w)\rangle$ has to be eliminated by setting it to 1. Another example is $1 - w_1 = w_2$, which can be obtained by noting the equivalence to $1 = w_1 + w_2$ and setting the vectors as shown. Again, the constant element 1 from $(1, w)$ must be utilized.
These are rather simple constraints, however. There are also more complex ones such as $w_3^3 = w_1 + w_2$ or even SHA-256$(w_1) = w_2$ [5]. To obtain these, the constraints need to be broken down, similarly to how it would be done in an arithmetic circuit. Cubes cannot be directly expressed in one constraint, only squares would work, so another approach must be taken. We note that $w_1^3 = w_1 + w_2 \iff w_3 w_3 w_3 = w_1 + w_2$, it essentially boils down to addition and multiplication. Now, internal variables are introduced, just like internal wires would be in a circuit. The original constraint can now be flattened to something like

$$
\begin{aligned}
w_3^2 &= w_4 \\
w_4 \cdot w_3 &= w_5 \\
w_3 &= w_1 + w_2
\end{aligned}
$$

which can then be converted into the formal vector representation. SHA-256 would be modeled analagously.
Define the language

$$\text{R1CS-SAT} = \{(S, x) \mid S \text{ is an R1CS and } x \in \mathbb{F}^{n_{primary}}, \exists w \colon S(x, w) = 1\}$$

$n_{primary}$ belongs to S (see Definition 6). As Ben-Sasson et al. [28] mention, R1CS-SAT

is NP-complete.
The first step of creating the proving and verifying keys by G involves the conversion from R1CS to QAP. That conversion to a QAP will be the subject of the next section.

2.5.2. From R1CS to QAPs

R1CS is still not compact enough to create a zk-SNARK. There is yet another format to encode computation, however, called *quadratic arithmetic program* (QAP). Checking all constraints of an R1CS is replaced by checking polynomial divisibility once for some QAP. Finding a QAP from an R1CS such that they are satisfied by the same inputs is the task of this section.
Every constraint is of the form given in (3):

$$\langle A_i, (1, w)\rangle \cdot \langle B_i, (1, w)\rangle = \langle C_i, (1, w)\rangle$$

and therefore equivalent to

$$\langle A_i, (1, w)\rangle \cdot \langle B_i, (1, w)\rangle - \langle C_i, (1, w)\rangle = 0 \tag{4}$$

This constraint shall now be encoded by means of the zero of a polynomial. For a fixed random value $r_i \in \mathbb{F}$, define a polynomial $P \in \mathbb{F}[x]$ such that

$$P(r_i) = \langle A_i, (1, w)\rangle \cdot \langle B_i, (1, w)\rangle - \langle C_i, (1, w)\rangle \tag{5}$$

For the moment, all other values $P(x)$ for $x \neq r_i$ play no role. If r_i is a zero, i.e., $P(r_i) = 0$, then 4 holds. If w satisfies 4, then r_i is a zero of P. Hence,

$$\langle A_i, (1, w)\rangle \cdot \langle B_i, (1, w)\rangle - \langle C_i, (1, w)\rangle = 0 \iff P(r_i) = 0 \tag{6}$$

Now, consider the following theorem:

Theorem 1 (Factor theorem)**.** *Let $P \in \mathbb{F}[x]$. Let $r \in \mathbb{F}$. It holds that*

$$(x - r) \mid P(x) \iff P(r) = 0$$

That is, $(x - r)$ divides $P(x)$ if and only if r is a zero of P. $(x - r)$ divides $P(x)$ means that there exists a polynomial $Q \in \mathbb{F}[x]$ such that $(x - r) \cdot Q(x) = P(x)$.

For a proof of Theorem 1, see Larson [29].
According to Theorem 1 and (6), (4) is equivalent to

$$t_i(x) = (x - r_i) \mid P(x)$$

The equation has been reduced to divisibility. Converting every constraint into an equivalent form is not very useful, though. However, as will be described in the next lines, multiple constraints can be merged into exactly one divisibility condition. Not one per constraint, but a single one. Checking many constraints is thereby reduced to checking divisibility exactly once. What this will look like is that any i^{th} constraint will be assigned one such random value $r_i \in \mathbb{F}$. Let there be n_c constraints. The polynomial P is extended to not only contain one constraint at r_i, but n_c constraints at $r_1, \ldots, r_{n_c}$:

$$\begin{aligned} P(r_1) &= \langle A_1, (1,w)\rangle \cdot \langle B_1, (1,w)\rangle - \langle C_1, (1,w)\rangle \\ &\vdots \\ P(r_{n_c}) &= \langle A_{n_c}, (1,w)\rangle \cdot \langle B_{n_c}, (1,w)\rangle - \langle C_{n_c}, (1,w)\rangle \end{aligned} \tag{7}$$

Now, (5) holds for all r_i, $1 \leq i \leq n_c$. Accordingly, divisibility is also extended to all constraints:

$$t(x) = \prod_{i=1}^{n_c} t_i(x) = (x - r_1) \cdots (x - r_{n_c}) \mid P(x)$$

$P(x)$ is now a very compact form to encode an entire computation and well suited for zk-SNARKs. The divisibility by $t(x)$ ensures that all constraints are satisfied, which amounts to choosing w correctly.
While $P(x)$ itself is pretty compact now, the question of how to obtain $P(x)$ is still open. (7) describes how it should behave, but a closed expression, which achieves this behavior is desirable. Here, *Lagrange interpolation* comes into play. It is a means of polynomial interpolation, which is precisely what is needed.

Definition 7. *Let there be k points $(x_1, y_1), \ldots, (x_k, y_k) \in \mathbb{F}^2$. The process of Lagrange interpolation yields an interpolation polynomial in Lagrange form $L \in \mathbb{F}[x]$ of degree at most $k-1$ that has the property*

$$L(x_i) = y_i \qquad \forall i \colon 1 \leq i \leq k$$

That is, it coincides with each of the points given.

For a definition of Lagrange interpolation and explanations on how it works, see Burden et al. [30].
While P could be interpolated to conform to (7), this would prevent the option to plug in values for w. w is contained in the points' definitions used for interpolation, but these need to be constant. Therefore, the value of w would have to be fixed and plugging in witness values to test whether they are satisfying or not is impossible. Instead, note what changes for the lines in (7): it is the A_i, B_i, C_i vectors for $1 \leq i \leq n_c$. If just these were interpolated and their polynomials stored, one could plug in w as one

wishes and then compute $P(x)$. Interpolating the vectors would mean that polynomials $a(x)$, $b(x)$, and $c(x)$ were to be found, such that

$$\begin{aligned} a(r_i) &= A_i \\ b(r_i) &= B_i \\ c(r_i) &= C_i \end{aligned}$$

(7) is thus fulfilled:

$$\begin{aligned} P(r_i) &= \langle a(r_i), (1, w)\rangle \cdot \langle b(r_i), (1, w)\rangle - \langle c(r_i), (1, w)\rangle \\ &= \langle A_i, (1, w)\rangle \cdot \langle B_i, (1, w)\rangle - \langle C_i, (1, w)\rangle \end{aligned}$$

There is one last hurdle to overcome: Lagrange interpolation interpolates points of the form (x_i, y_i), where x_i and y_i are scalar. y_i is not a vector. However, this can be easily fixed by breaking down the vectors into their elements. Let n be the number of variables in the original R1CS. Note that $(1, w)$ contains $n + 1$ elements. Instead of interpolating $a(x)$, $b(x)$, and $c(x)$, their elements $a_j(x)$, $b_j(x)$, and $c_j(x)$ for $0 \leq j \leq n$ are interpolated. These elements are obtained according to Definition 4 of the dot product

$$\begin{aligned} P(r_i) &= \langle a(r_i), (1, w)\rangle \cdot \langle b(r_i), (1, w)\rangle - \langle c(r_i), (1, w)\rangle \\ &= \sum_{j=0}^{n} a_j(r_i)(1, w)_j \cdot \sum_{j=0}^{n} b_j(r_i)(1, w)_j - \sum_{j=0}^{n} c_j(r_i)(1, w)_j \end{aligned}$$

this also fulfills (7). Finally, the following closed expression for $P(x)$ is obtained:

$$P(x) = \sum_{j=0}^{n} a_j(x)(1, w)_j \cdot \sum_{j=0}^{n} b_j(x)(1, w)_j - \sum_{j=0}^{n} c_j(x)(1, w)_j \tag{8}$$

Since the first element of $(1, w)$ is 1, (8) can be simplified to

$$P(x) = (a_0(x) + \sum_{j=1}^{n} a_j(x)w_j) \cdot (b_0(x) + \sum_{j=1}^{n} b_j(x)w_j) - (c_0(x) + \sum_{j=1}^{n} c_j(x)w_j)$$

The following definition is due to GGPR13 [17], which has introduced QAPs in the first place. It has been slightly adjusted to R1CS, which has not been directly addressed in the original definition.

Definition 8. *A quadratic arithmetic program (QAP) over a field $\mathbb{F}$ contains three sets of polynomials*

$$A = \{a_j(x) \in \mathbb{F}[x] \mid j \in \{0, \ldots, n\}\}$$
$$B = \{b_j(x) \in \mathbb{F}[x] \mid j \in \{0, \ldots, n\}\}$$
$$C = \{c_j(x) \in \mathbb{F}[x] \mid j \in \{0, \ldots, n\}\}$$

and a target polynomial $t(x) \in \mathbb{F}[x]$. *As such, a QAP* Q *is formally defined as* $Q = (t(x), A, B, C)$.
Let S *be an R1CS with* n_c *constraints,* $n_{primary}$ *input variables, and* n *variables in total (see Definition 6). It is said that* Q *is a QAP that computes* S *if the following is true: for an input* x, *there exists a* w *such that* $S(x, w) = 1$ *if and only if*

$$t(x) \mid (a_0(x) + \sum_{j=1}^{n} a_j(x)w_j) \cdot (b_0(x) + \sum_{j=1}^{n} b_j(x)w_j) - (c_0(x) + \sum_{j=1}^{n} c_j(x)w_j) \tag{9}$$

The size of Q *is* n. *The degree of* Q *is the degree of* $t(x)$.

The QAP is the final representation of computation. It is directly used to create the final zk-SNARK.
As per Virza [31], satisfiability of QAPs is NP-complete. This should come as no surprise, since R1CS-SAT is NP-complete and this section described an efficient reduction from R1CS to QAP.

2.5.3. From QAPs to zk-SNARKs

Having obtained the QAP, the proving and verifying keys need to be generated. Using these, zk-SNARKs can be created and verified, conforming to Definition 3.
Let $Q = (t(x), A, B, C)$ be a QAP of size n and degree d. Q was obtained from the R1CS S. Q is encoded in two different ways in the `pk` and `vk` such that the needs of proving and verifying are met. In addition, `pk` and `vk` contain random information to ensure the security of the zk-SNARK scheme.
Once G is done, P can create a proof. Given an input $x \in \mathbb{F}^{n_{primary}}$ ($n_{primary}$ is obtained from S) and a witness $w \in \mathbb{F}^n$, the division by $t(x)$ from Definition 8 is carried out. Assuming that $S(x, w) = 1$, divisibility is ensured by Definition 8 and the quotient $h(x) \in \mathbb{F}[x]$ is obtained. Along with other encoded information on Q, $h(x)$ is encoded in the proof π.
V verifies π by testing if $h(x)$ really is the quotient of the division performed by P and performing other tests to make sure the prover did not cheat. All tests are performed using a so-called *bilinear map* or *pairing*. This prevents the verifier from learning more about the QAP than it needs to, thus ensuring the zero-knowledge property.
The scheme is proven to conform to Definition 3 by relying on certain cryptographic assumptions.

This section only contained a very rough and informal overview. The avid reader is gently referred to PGHR13 [26] and the references in there for detailed explanations of the creation of zk-SNARKs from QAPs and security proofs.

2.6. Multi-Party Computation

The cryptographic field of secure multi-party computation (MPC) was first investigated by Yao [32]. It has been proposed to serve purposes as varied as secure voting, spam filtering on encrypted email [33], or secure computation of satellite trajectories [34].
A sample problem solved using an MPC method is described. Based on these, the concept of MPC is formalized.
All sections will heavily rely on *A Pragmatic Introduction to Secure Multi-Party Computation* by Evans et al. [33] and Lindell [35].

2.6.1. An Example

This part on MPC shall start with a brief example of a problem that is solved by means of multi-party computing.
Suppose that there are n parties $P_1, \ldots, P_n$. P_i knows a number $m_i \in \mathbb{N}$ for $1 \leq i \leq n$. Now, they want to compute the function $f(m_1, \ldots, m_n) = \sum_{i=1}^{n} m_i$. It should happen without any P_i learning the value of any x_j for $j \neq i$, though; no party wants anybody else to know their private value. This might be useful in a voting system (see Evans et al. [33]), for instance, where the sum of votes for a candidate is computed, but each vote should remain secret.
The above problem shall be solved by following Shamir [36] who provided a very elegant way to implement MPC. It is commonly called *Shamir secret sharing* (SSS).
What it does is divide all private data $m_1, \ldots, m_n$ into pieces, perform the addition on these pieces, and reassemble the obtained sums into the final sum $\sum_{i=1}^{n} m_i$.
The following theorem will be of crucial importance. Here is how it was formulated by Shamir [36]:

Theorem 2 (Interpolation theorem)**.** *Let $\mathbb{F}_p$ be the finite field of integers modulo a prime number p. Let $(x_i, y_i) \in \mathbb{F}_p^2$ for $1 \leq i \leq k$ be k points with pairwise different x_i. It must hold that $k < p$. There exists a unique polynomial $Q \in \mathbb{F}_p[x]$ of degree $k-1$ such that $Q(x_i) = y_i$.*

As Shamir [36] says, given any subset of k points, a unique $k-1$-polynomial can be constructed by interpolation due to Theorem 2. Given less than k points, that polynomial cannot be uniquely constructed, however.
To divide any one m_i into pieces, let $Q_i \in \mathbb{F}_p[x]$ with degree $n-1$, that is,

$$Q_i(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

Set the prime p such that $m_i < p$ and $n < p$. The party P_i defines $Q_i(x)$, i.e., P_i sets $a_0 = m_i$ such that $Q_i(0) = m_i$ and assigns random values from $\mathbb{F}_{\text{|}}$ to $a_1, \ldots, a_{n-1}$. Afterwards, P_i hands the value $Q_i(j)$ to P_j for $1 \leq j \leq n$ (P_i obtains $Q_i(i)$ himself).
Note that according to Theorem 2, if all n values $Q_i(j)$ are known, then Q_i can be reconstructed and $Q_i(0)$ can be retrieved.
$Q_i(j)$ for $1 \leq i, j \leq n$ represents the piece of m_i belonging to P_j. Every party P_j now receives $Q_1(j), \ldots, Q_n(j)$, so P_j knows one piece of each $m_1, \ldots, m_n$.
If P_j calculates $\sum_{i=1}^{n} Q_i(j)$ now, P_j obtains the value $Q(j)$ on the polynomial $Q \in \mathbb{F}[x]$ for $Q(x) = \sum_{i=1}^{n} Q_i(x)$. Note the degree of $Q(x)$, which is, again, $n-1$. This means that if for $1 \leq j \leq n$, party P_j properly computed their $Q(j)$, $Q(x)$ can be reconstructed from these values by Theorem 2. $Q(0) = \sum_{i=1}^{n} Q_i(0) = \sum_{i=1}^{n} m_i = f(m_1, \ldots, m_n)$ can be retrieved.
The goal was achieved: $f(m_1, \ldots, m_n)$ was computed without any party learning others' private values. No P_j can compute any m_i with $j \neq i$ on his own. All n pieces of m_i are needed to reconstruct m_i, so all parties would need to collude together to reconstruct m_i.
Note the trade-off made here: the degree of the Q_i for $1 \leq i \leq n$ could have been some $k < n$. Less pieces would have been required, decreasing the risk of errors and improving efficiency of the reconstruction. However, collusion between a subset of parties would be made possible: if $k < n$ parties colluded together, they could obtain the m_i of all other parties. Since this is not desirable, $k = n$ has been chosen.
Of course, any party P_i can just hand wrong pieces to the other parties such that some other value different from m_i is fed into the addition. SSS therefore depends on the fact that every party follows the protocol.

2.6.2. Formalization

These possible security problems in an MPC protocol such as SSS are subsumed under an *adversary*. The adversary controls the *corrupt* parties that collude with each other. All other parties are called *honest*.
Given some adversary that controls a subset of the parties, an MPC protocol is secure if, despite the adversary's efforts, the security properties of the protocol persist.
It was argued that with SSS and $k = n$, no party P_i can learn the other parties' private values (see Section 2.6.1). Assuming that every party follows the scheme, everyone also learns the correct value of $f(m_1, \ldots, m_n)$.
These are already two properties of SSS and MPC protocols in general: privacy and correctness. But instead of defining secure MPC as a list of properties that are maintained under the attack of an adversary, another approach is chosen called the *real-ideal paradigm* (see Evans et al. [33]).
Imagine there is a trusted, incorruptible party T. If T existed, all parties $P_1, \ldots, P_n$ could simply send their input $m_1, \ldots, m_n$ to T. T then computes $f(m_1, \ldots, m_n)$ and sends the output to each party.
This is the *ideal world*. Note that privacy and correctness are fulfilled in this ideal world: any party sends their private input only to T, which is incorruptible, so m_i

is not leaked to others. Furthermore, every party obtains the output from T, so the output certainly is correct.
In the *real world*, no trusted party exists and an MPC protocol is executed, i.e., the parties send messages between each other. There is also an adversary, which controls certain parties.
An MPC protocol is considered secure in the real world if any attack by an adversary can also be achieved in the ideal world. Stated differently, the security of a secure MPC protocol is the same in the real and ideal world.
As in Section 2.4.2 on interactive zero-knowledge proofs, this is a simulator argument: given a successful attack by an adversary A in the real-world, there must exist a successful attack by an adversary B with the same effect in the ideal world. B is the simulator, which simulates A's behavior in the ideal world. Since the ideal world is ideally secure, B cannot simulate A's sucessful attack, hence A's attack cannot be performed in a secure MPC protocol.
What an attack is needs some elaboration, though. In this thesis, only the model of a *malicious* adversary will be of interest. In this model, the adversary is not obliged at all to follow the protocol. He can send and do whatever he wishes.
One other notable model is that of the *semi-honest* or *honest-but-curious* adversary. Such an adversary follows the protocol, but extrapolates as much information as he can get during the execution of the protocol. That is, if data leaks, the adversary will know. SSS is only secure in the semi-honest model, for example, as every party has to follow the protocol.
Another interesting parameter is how many parties have been corrupted by the adversary. If at least half of the parties are corrupt, one speaks of a *dishonest majority*. Otherwise, there is an *honest majority*. For instance, SSS with $k = n$ is secure against a dishonest majority in the semi-honest model: corrupted parties are semi-honest, so it does not matter how many are corrupted. As long as all parties follow protocol, no knowledge is leaked due to how SSS works.
For the full formal definitions of the concepts introduced in this section, see Evans et al. [33].

3. Collaborative Deanonymization Using Zero-Knowledge Proofs

This section constitutes the main part of this thesis. It will introduce the problem that this thesis attempts to solve. Two solutions that implement zero-knowledge proofs will then be described: one using MPC (see Section 2.6) the other one using zk-SNARKs (see Section 2.4.4).

3.1. The Problem

Consider a Bitcoin (see Section 2.2) user who uses many key pairs (see Section 2.2.2) stored in his wallet (see Section 2.2.3). He regularly helps the police find criminals by

testifying as a witness for collaborative deanonymization (see Section 2.2.5). If the private keys used for testimony are compromised for one reason or another and used by a malicious actor, this actor can manipulate the collaborative deanonymization scheme. As an alternative, one could replace the notion of "control" over inputs and outputs with proof that the relevant public keys belong to the same wallet. Proving common membership in the same wallet aims to set the bar a little higher, requiring even more private knowledge than just some private keys for testimony. This method is infeasible for JBOK wallets, since there is nothing that intrinsically binds a public key to the wallet. However, it certainly is possible for deterministic wallets (see Section 2.2.3) by revealing necessary derivation information for both keys, so anyone can verify their membership. Revealing this information can therefore be considered an individual testimony.

Consider hierarchical deterministic wallets as specified by BIP 32 (see Section 2.2.3 for the concept and terminology). Any public key k in a BIP 32 wallet is defined by

$$k = \mathrm{CKDpub}(\ldots \mathrm{CKDpub}(\mathrm{CKDpub}(M, i_1), i_2) \ldots, i_n)$$

for some $n \in \mathbb{N}$, where CKDpub is the public key derivation function from BIP 32. As such, the private knowledge required to derive a key is the master public key M and the indices $i_1, i_2, \ldots, i_n$.

Formalizing the goal of proving common membership in a wallet, the language

$$L_{commonBIP32} = \left\{ (k_1, k_2) \;\middle|\; \begin{array}{l} \exists M, n, m, i_1, \ldots, i_n, j_1, \ldots, j_m : \\ k_1 = \mathrm{CKDpub}(\ldots \mathrm{CKDpub}(\mathrm{CKDpub}(M, i_1), i_2) \ldots, i_n), \\ k_2 = \mathrm{CKDpub}(\ldots \mathrm{CKDpub}(\mathrm{CKDpub}(M, j_1), j_2) \ldots, j_m) \end{array} \right\}$$

is obtained. The master public key M captures the notion of "common membership in a wallet", since, with very high probability, M uniquely identifies a wallet.

Of course, simply revealing M and the indices is not an option as this would allow anyone to list all public keys in a wallet. Moreover, anyone with that information could reproduce the proof that the public keys belong to the wallet. Instead, it should be proved that $(k_1, k_2) \in L_{commonBIP32}$ without revealing any such sensitive information. This idea is captured by zero-knowledge proofs (see Section 2.4).

It is important to note that given (k_1, k_2), none of M, i, and j can be computed in polynomial time.

Public key derivation in BIP 32 is rather complex, so some simplification has been applied. The master public key M is replaced with a plain seed (or root) value r. Instead of the hierarchies of indices $i_1, \ldots, i_n$ and $j_1, \ldots, j_m$, only single indices i and j are used. Key derivation now happens as follows: the seed value r is concatenated bitwise with the respective index. The binary operator of bitwise concatenation is denoted by the symbol $\circ$. To the resulting value, a cryptographic hash function h (see Section 2.1) is applied. The final result is the derived public key. By using a CHF, the

security properties of CKDpub are modeled: due to preimage resistance, none of r, i, and j can be recovered from the keys alone. Collision and 2^{nd}-preimage resistance ensure that a key originates from exactly one wallet: it is not easily possible to find seeds and indices that differ but yield the same public key.
Note that this is a deterministic key derivation scheme for public keys. It will serve to model BIP 32, the function CKDpub in particular, for a proof of concept in this thesis. For real-world key derivation, corresponding private keys have to be computable, so this scheme itself is not fit for real-world use.
r is defined to be an integer of 16 bits. i and j are defined to be integers of each 8 bits. k_1 and k_2 are defined to be integers of each 256 bits. Let $h\colon \{0,1\}^* \to \{0,1\}^{256}$ be a cryptographic hash function. The following definitions emerge:

Definition 9.

$$L_{common} = \{(k_1, k_2) \mid \exists r, i, j\colon h(r \circ i) = k_1 \wedge h(r \circ j) = k_2\}$$
$$R_{L_{common}} = \{(k_1, k_2, r, i, j) \mid h(r \circ i) = k_1 \wedge h(r \circ j) = k_2\}$$

The question that is addressed in this thesis takes shape: how to prove in zero-knowledge that, given two public keys, both stem from the same deterministic wallet?
Equivalently, the statement to prove is

Definition 10. *Being given (k_1, k_2), I know r, i, j such that*

$$(k_1, k_2, r, i, j) \in R_{L_{common}}$$

3.2. An MPC Solution

The first approach to prove the statement in Definition 10 for some input was created using multi-party computation (MPC). Adequate technical background on MPC is provided in Section 2.6.
This section will explain how MPC was used in software to prove the statement. First, it will be explained why MPC is suitable for the creation of zero-knowledge proofs. Then, some contemporary MPC frameworks will be described and MP-SPDZ will then be further investigated. Finally, the software solution will be outlined.

3.2.1. Zero-Knowledge as MPC

Since the statement shall be proved in zero-knowledge, it must be established how MPC qualifies for this purpose.
As mentioned by Ishai et al. [37], interactive zero-knowledge proof systems are a special case of MPC with two parties. In fact, an interactive zero-knowledge proof system can be built using MPC for the language L_{common}.
Let a party P_1 be the prover and another party P_2 be the verifier. Given $(k_1, k_2, r, i, j) \in R_{L_{common}}$, P_1 knows (k_1, k_2), r, i, and j. P_2 only knows (k_1, k_2). Since $L_{common} \in \mathsf{NP}$,

there exists an algorithm A such that $A(k_1, k_2, r, i, j) = 1 \iff (k_1, k_2, r, i, j) \in R_{L_{common}}$. This algorithm A is evaluated via MPC, where r, i, and j are P_1's private input and (k_1, k_2) is publicly known. Finally, P_2 accepts if $A(k_1, k_2, r, i, j) = 1$ and otherwise does not.
Note that the MPC protocol in use must be secure against malicious adversaries and a dishonest majority: the verifier should under no circumstances learn any of r, i, and j, even if he does not follow protocol. Since only two parties are involved, a malicious verifier constitutes a dishonest majority and the MPC protocol must ensure security in this case.

3.2.2. MPC in Practice

There are many frameworks that implement MPC (for a list, see awesome-mpc [38]). One of, if not the most comprehensive one is MP-SPDZ [39], which implements about thirty protocols. They all exhibit different properties, covering a wide breadth of use cases. MP-SPDZ is documented in a paper by Keller [40] and online [41]. It also offers a large variety of protocols and is easily usable. Moreover, it is actively maintained: when the author of this thesis discovered a bug in the implementation [42], it was fixed on the following day.
For these reasons, MP-SPDZ has been made the final choice.

3.2.3. Using MP-SPDZ

Writing a program using MP-SPDZ is basically writing Python code. The code is compiled into *bytecode* and run on a virtual machine. This implies that every protocol offers the same interface to the programmer. The bytecode is very similar to assembly language. This thesis only deals with the high-level Python code, though.
To understand the basics of MP-SPDZ, consider the following example. Let $\mathbb{F}$ be a finite field. The dot product of two vectors $a, b \in \mathbb{F}^n$ is computed, i.e., $\sum_{i=1}^{n} a_i b_i$:

```
# input
a = sint.Array(n)
b = sint.Array(n)
a.input_from(0)
b.input_from(1)

# computations
res = sint.dot_product(a, b)

# output
print_ln('%s', res.reveal())
```

There are three essential components in this program: input, computation, and output. First, consider the input. `a` and `b` are defined to be arrays of length `n` of secret integers (that is what `sint` stands for). This is the primary distinction that MP-SPDZ makes

between data types: data is either *secret* or *clear*. Secret data is known to no one but the party where the data originates from. Clear data is known to everyone. Next, `a` is filled with input from party 0, `b` is filled with input from party 1. Each party is denoted with such a non-negative integer. Since the arrays have been defined as secret, party 0 does not know `b` and party 1 does not know `a`.
All input has been gathered. The computation is now effectuated, which conists in calculating the dot product of `a` and `b`. This boils down to exactly one line, `res = sint.dot_product(a, b)`, which calculates the dot product over some secret data. `res` contains the result and is secret as well. This shows another natural feature: computation on secret data yields secret data. Similarly, computation on clear data yields clear data.
What remains is to notify the parties of the output. This amounts to two operations: revealing `res` and printing it to the screen. While the latter primarily fulfills convenience purposes, the former is essential: `res` is a secret value and since nobody knows the entire input, nobody knows `res`. `res.reveal()` yields a clear value, which any party knows.
All computation that involves `sint`s is performed over the finite field $\mathbb{F}$. How large an `sint` is can be set at compile time. Its default value is 64 bits, which is the value that will be used.

3.2.4. The Problem Written for MP-SPDZ

The solution to Definition 10 is modeled as two parties 0 and 1. Party 0 is the prover and party 1 is the verifier. Party 0 receives r, i, and j as input. (k_1, k_2) is publicly known, i.e., to both party 0 and party 1. Collaboratively, both parties now compute if $(k_1, k_2, r, i, j) \in R_L$, i.e., if $(k_1, k_2) \in L$. This lives up to the conditions defined in Section 3.1 as described in Section 3.2.1 (party 0 is P_1, party 1 is P_2).
Since splitting MPC into input, computation, and output is natural, the following sections will deal each with one of these phases.
The entire code is shown again in Appendix A.

3.2.5. MP-SPDZ: Input

The input consists of the public input (k_1, k_2) and the private input r, i, j from party 0. Unfortunately, no efficient way was found to pass the 256-bit values as public input into the program. There surely is a way, but this would have required much code for little gain in content in return. Therefore, k_1 and k_2 are hardcoded into the program. This requires recompilation for every change of k_1 and k_2, though. The definition of k_1 and k_2 is done like this:

```
h1_str = "37ad9e5625d7e5fae209db84a28dc45d" \
         "3e8e4631f41827a837db8073de4629eb"
h2_str = "53937f3b7fb58e104f113264695d5da3" \
         "4e4bf43da3dc3652ac7d868b4c122ce9"
```

`h1_str` and `h2_str` are string literals that contain the hexadecimal 256-bit values of k_1 and k_2, respectively. The values given here are only examples (they correspond to $m = 1$, $i = 2$, and $j = 3$). To perform computations later on, they need to be integers, though. This conversion is accomplished by the following two lines:

```
h1 = sbits.get_type(256)(int(h1_str, 16))
h2 = sbits.get_type(256)(int(h2_str, 16))
```

`h1` and `h2` now contain the integral values of k_1 and k_2, respectively. Take the first line as an example. `int(h1_str, 16)` interprets `h1_str` as a string of hexadecimal (base 16) digits and yields an `int` representing this value. `sbits.get_type(256)(...)` then converts this `int` to a secret 256-bit value.
For the private input, see the following snippet:

```
root_t = sbits.get_type(16)
index_t = sbits.get_type(8)

r = root_t.get_input_from(0)
i = index_t.get_input_from(0)
j = index_t.get_input_from(0)
```

`r`, `i`, and `j` contain their respective values afterwards. The function `some_data_type.get_input_from(...)` was already mentioned in Section 3.2.3. It fills the integers with their input values. The most interesting thing here is the definition of new types. `root_t` and `index_t` are two new types, secret integers of 16 and 8 bits, respectively. `r` is therefore 16 bits long, `i` and `j` are each 8 bits long. This conforms to the bit lengths defined in Section 3.1.

3.2.6. MP-SPDZ: Computation

All input has been collected and stored appropriately. The actual computation happens now. It basically consists of the following steps:

1. Bitwise concatenation ($\circ$)
2. Cryptographic Hash Function (h)
3. Bitwise comparison ($=$)
4. Logical AND ($\wedge$)

Concatenation happens like this:

```
preimage_t = sbits.get_type(root_t.n + index_t.n)
ri = preimage_t((r << index_t.n) + i)
rj = preimage_t((r << index_t.n) + j)
```

`ri` and `rj` now contain $r \circ i$ and $r \circ j$, respectively. `preimage_t` is a secret integral data type, which contains the pre-image of the hash function that will be applied. Its bit length is that of `root_t` and `index_t` added together, i.e., 24. `(r << index_t.n) + i` shifts `r` to the left by the length of `index_t.n` bits, i.e., 8, and inserts `i` in this 8 clear bits. The same is done for `rj`.
For the cryptographic hash function h, SHA3-256 [5] was chosen. It is natively supported by MP-SPDZ and therefore easy to use. Its output is 256 bits long, which conforms to h's definition in Section 3.1. MP-SPDZ provides its usage for inputs of up to 1080 bits. Since $r \circ i$ and $r \circ j$ are both 24 bits long, this poses no problem. However, SHA3-256 relies on the Keccak sponge function, which is not implemented in MP-SPDZ. Instead, it is passed to MP-SPDZ in the so-called Bristol Fashion format. This format describes a circuit, which implements the desired function. The sponge function in this format is offered online by cryptographer Nigel Smart [43].
Executing SHA3-256 on $r \circ i$ and $r \circ j$ looks as follows:

```
result = sha3_256(sbitvec([ri, rj])).elements()
h_ri = result[0]
h_rj = result[1]
```

`h_mi` now contains SHA3-256($r \circ i$), `h_rj` contains SHA3-256($r \circ j$).
Interestingly, the SHA3-256 implementation of MP-SPDZ initially malfunctioned when it was tested. The first 64 bits were correct, the rest was not. Upon contacting the maintainers on GitHub [42], a fix was provided the day after and further tests succeeded.
Bitwise comparison is very straightforward:

```
h1_equal = h1.equal(h_ri)
h2_equal = h2.equal(h_rj)
```

`h1_equal` and `h2_equal` contain 1 or 0 depending on whether SHA3-256($r \circ i$) $= k_1$ or SHA3-256($r \circ j$) $= k_2$, respectively.
The bitwise AND is also made easy by MP-SPDZ:

```
equal = h1_equal.bit_and(h2_equal)
```

`equal` now contains the bitwise AND of `h1_equal` and `h2_equal`, i.e., the end result. Note that throughout the entire series of computations, the results have been kept secret.

3.2.7. MP-SPDZ: Output

The last part involves printing the output, i.e., the value of `equal`:

```
print_ln('result: %s', equal.reveal())
```

`equal` is first revealed as it is a secret integer. Then it is printed and both parties learn the final result.

3.3. A zk-SNARK Solution

The concrete way zk-SNARKs (see Section 2.4.4) were used to obtain a zero-knowledge proof of the NP statement (see Section 2.4.4) in Definition 10 will now be presented. Background on zero-knowledge proofs and zk-SNARKs in particular is given in Section 2.4.
First, a brief overview of current zk-SNARK frameworks, including libsnark [44], will be given. Afterwards, it will be described how libsnark was used to program a solution, which consists of expressing L_{common} (see Section 3.1) in the linear algebraic system R1CS (see Section 2.5.1) and then creating a zk-SNARK based on that.

3.3.1. zk-SNARKs in Practice

Probably the most popular framework for creating zk-SNARKs is libsnark [44]. It is written in the programming language C++ and was notably used in a prior version of Zcash [45], a cryptocurrency that makes use of zk-SNARKs for its privacy needs. There exist two other zk-SNARK libraries, jsnark [46] and snarky [47], both of which use libsnark as a backend, adding to its popularity.
Another interesting library is bellman [48]. It is written in Rust and has replaced libsnark in Zcash. However, bellman currently only implements Groth16 [16], while libsnark implements the zk-SNARK approaches from PGHR13 [26], BCTV14 [25], and Groth16 [16].
Even a web-based implementation of zk-SNARKs is available under the name of snarkjs [49]. The zk-SNARK approaches PLONK [50] and Groth16 [16] are supported. NP statements can be described in a specially designed programming language called Circom [51]. snarkjs is very modern in its use of JavaScript. Furthermore, documentation for both snarkjs and Circom is quite comprehensive.
libsnark is a mature library that is reasonably well documented in the shape of tutorials and a Ph.D. thesis [31]. For these reasons and due to the author's familiarity with C++, libsnark has been made the framework of choice.

3.3.2. Using libsnark

libsnark offers a variety of workflows to go about creating a zk-SNARK. The zk-SNARK from BCTV14 [25] is implemented, for example, as is Groth16 [16]. In this work, the preprocessing zk-SNARK from PGHR13 [26] is used with optimizations from BCTV14 [25]. Furthermore, unlike PGHR13 [26] itself, libsnark offers a slightly more convenient system to describe the NP statement to be proved: the rank-one constraint system (see Section 2.5.1). The statement from Definition 10 is expressed by means of this system and based on that a zk-SNARK is created (see Section 2.5.2 and 2.5.3). While there exist other systems libsnark understands, R1CS was chosen right from the start due to libsnark's support of a gadget library for this system. This library offers gadgets, that is, building blocks, from which an R1CS can be composed. Combined with the option to create custom gadgets, this facility allows for building zk-SNARK applications in a

straightforward way that is coherent with the state-of-the-art programming paradigm of object-oriented programming.

Once the statement from Definition 10 is expressed in R1CS, the zk-SNARK is created by libsnark. Since a preprocessing zk-SNARK is used, the workflow of generating the keys, proving, and verifying the proof has to be followed (see Section 2.4.4). While the key generation only needs to happen once, the proof creation is necessary for every new tuple $(k_1, k_2, r, i, j) \in R_{L_{common}}$. Correspondingly, the verifier must be run anew for every proof.

Following this recipe, the first step is writing the R1CS. While the formal definition of R1CS has been covered in 2.5.1, one needs to deal with libsnark's intricacies when writing a R1CS too.

For reference, all code that is shown here plus some more for context can be found in Appendix B.

3.3.3. R1CS in libsnark

One could write a list of constraints, i.e., a list of vectors A_i, B_i, C_i and make the R1CS work. Just like with respect to programming languages, though, spaghetti code is frowned upon. The components described before should not vanish in a big blob of constraints. Instead, some structure is desired, for better human readability and better maintenance. As a remedy libsnark offers an abstraction called a `gadget`. It describes a reusable component of a bigger system. `gadget`s describe essentially just sets of raw R1CS constraints also known as `r1cs_constraint`s in libsnark. While `r1cs_constraint`s are part of a `r1cs_constraint_system`, i.e., an R1CS, `gadget`s are part of a `protoboard`. These can be seen as similar to the protoboards or so-called breadboards used for electronic prototyping. Every `protoboard` has an underlying `r1cs_constraint_system` that it fills with `r1cs_constraint`s by means of `gadget`s. `gadget`s cannot be added to `r1cs_constraint_system`s, though, they should be understood as something more abstract. In contrast, `r1cs_constraint`s can still be added to `protoboard`s. `protoboard`s are essentially an abstraction of `r1cs_constraint_system`s, which support the concept of `gadget`s.

Consider the compound constraint $w_3^3 = w_1 + w_2$. Similarly to the actual language L_{common}, a language is associated to this problem:

$$L_{example} = \{(w_1, w_2) \mid \exists w_3 \colon w_3^3 = w_1 + w_2\}$$

This corresponds to the problem of finding the cube root of $w_1 + w_2$. For illustrative purposes, a `gadget` will be implemented for this constraint. A list of constraints equivalent to $w_3^3 = w_1 + w_2$ is

$$w_3 \cdot w_3 = w_4$$
$$w_4 \cdot w_3 = w_5$$
$$w_5 = w_1 + w_2$$

Every `gadget` must implement the functions `generate_r1cs_constraints` and `generate_r1cs_witness`. `generate_r1cs_constraints` adds the R1CS constraints related to the `gadget` to a chosen `protoboard`. A `gadget` is initialized with whatever `protoboard` it is supposed to be part of. For this example, the added constraints are the ones given above for $w_3^3 = w_1 + w_2$. `generate_r1cs_constraints` would look something like this in libsnark:

```
void generate_r1cs_constraints()
{
  // w_3 * w_3 = w_4
  this->pb.add_r1cs_constraint(
    r1cs_constraint<FieldT>({{w3}}, {{w3}}, {{w4}})
  );

  // w_4 * w_3 = w_5
  this->pb.add_r1cs_constraint(
    r1cs_constraint<FieldT>({{w4}}, {{w3}}, {{w5}})
  );

  // w_5 * 1 = w_1 + w_2
  this->pb.add_r1cs_constraint(
        r1cs_constraint<FieldT>({{w5}}, 1, {{w1 + w2}})
  );
}
```

Here, `pb` is the `protoboard`. `w1` through `w5` make up the witness vector w in the underlying R1CS (see Definition 6). They are variables that have been defined for `pb`. That is, libsnark does not deal with the vector w, but directly with the variables in there. Also note that, as the function name `add_r1cs_constraint` says, the `gadget` adds the `r1cs_constraints` to `pb`. Strictly speaking, `gadgets` only contribute to a `protoboard` by extending the R1CS, they are no autonomous entities that exist within a `protoboard`.

`generate_r1cs_witness` populates the witness vector w with values. Let S denote the created R1CS. Given $L_{example}$, w_3, and the input (x_1, x_2), `generate_r1cs_witness` then calculates w_4 and w_5 such that $S(x_1, x_2, w) = 1 \iff (x_1, x_2) \in L$. Defining x_1, x_2 to be the input variables, `generate_r1cs_witness` fills w_1 and w_2 to meet Definition 6 condition (2) and w_4 and w_5 to meet condition (3). A possible implementation in libsnark is the following:

```
void generate_r1cs_witness(FieldT x1_in, FieldT x2_in,
  FieldT w3_in)
{
  // first, initialize the input variables
  this->pb.val(w1) = x1_in;
  this->pb.val(w2) = x2_in;
  this->pb.val(w3) = w3_in;

  // second, deduce the other variables' values
  this->pb.val(w4) = this->pb.val(w3) * this->pb.val(w3);
  this->pb.val(w5) = this->pb.val(w4) * this->pb.val(w3);
}
```

For instance, given $x_1 = 4$, $x_2 = 4$, and $w_3 = 2$, after running `generate_r1cs_witness`, the R1CS described by the `protoboard`, is satisfied. Given $x_1 = 1$, $x_2 = 2$, and $w_3 = 3$, after running `generate_r1cs_witness` it is not, however.
Lastly, note the recurrent use of the data type `FieldT`. In Definition 6, R1CS was defined over a field $\mathbb{F}$. The witness vector w contains elements from $\mathbb{F}$ as do all other vectors that make up an R1CS. `FieldT` represents a finite field and fulfills the role of $\mathbb{F}$. As such, when evaluating whether an R1CS is satisfied or not according to Definition 6, finite-field arithmetic is performed.
As can be seen, some abstractions are made by libsnark for better ease of use. In the end, it all boils down to the formal description in Definition 6, though.

3.3.4. Formulating the Problem in R1CS

Now that some basics of libsnark have been covered, the actual problem will be addressed. For L_{common} and $R_{L_{common}}$ from Definition 9, an R1CS S is to be found such that for an input (k_1, k_2) and values r, i, and j,

$$(k_1, k_2, r, i, j) \in R_L \iff S\,(k_1, k_2, r, i, j) = 1 \tag{10}$$

The CRF h is chosen to be SHA-256 [5]. SHA-256 outputs a 256-bit value, which conforms to h's definition in Section 3.1.
To define S accordingly, the description of L_{common} from Definition 9 needs to be modeled in R1CS. Breaking the description down into pieces, the following essential components emerge (the same as for the MPC approach in Section 3.2.6):

1. Bitwise concatenation ($\circ$)
2. Cryptographic Hash Function (SHA-256)
3. Bitwise comparison ($=$)

4. Logical AND ($\wedge$)

These need to be captured in R1CS. Once this is done, they can be glued together in order to obtain the full R1CS for L_{common}, so that (10) holds. In the next few sections, it will be described how each component is implemented.

3.3.5. libsnark: Bitwise concatenation

$r \circ i$ and $r \circ j$ are the input to SHA-256. They describe the bitwise concatenations of r and i and of r and j, respectively. Instead of passing r, i, and j separately into the R1CS, two values ri and rj are passed as input. This eliminates the need for constraints to enforce concatenation.
Obviously, it is expected that $ri = r \circ i$ and $rj = r \circ j$. However, it needs to be enforced by the R1CS that ri and rj start with the same r. The following code fulfills this purpose:

```
for (size_t i = 0; i < sizeof(root_t) * CHAR_BIT; ++i) {
  this->pb.add_r1cs_constraint(
    libsnark::r1cs_constraint<FieldT>(
      1, {{ri_block.bits[i]}}, {{rj_block.bits[i]}}
    )
  );
}
```

`ri_block` and `rj_block` are 64-byte blocks that contain ri and rj right at their start, respectively. Why ri and rj are put in such blocks will become clear in the next section, Section 3.3.6. They are arrays of bits and introduce one new variable in the R1CS's witness vector per bit. For each pair of bits of ri and rj, one R1CS equality constraint is added to the protobord `pb`. `root_t` is the data type of r and `sizeof(root_t) * CHAR_BIT` yields r's length in bits.

3.3.6. libsnark: SHA-256

With SHA-256 things are a lot more complicated. It needs to be implemented in R1CS such that given $r \circ i$ and h for instance, $h = \text{SHA-256}(r \circ i)$ is ensured. Fortunately, this has already been implemented in libsnark as a `gadget` called `sha256_compression_function_gadget`. The reason for this lengthy name is that the `gadget` only implements a part of SHA-256. SHA-256 is based on the Merkle–Damgård construction (see Section 2.1.2). `sha256_compression_function_gadget` implements only the compression function, so it is insufficient for larger inputs. However, $r \circ i$ and $r \circ j$ are both exactly three bytes, so they easily fit into one SHA-256 block (64 bytes for SHA-256 (see FIPS 180-2 [5]), respectively. Therefore, only one iteration of the compression function is necessary with the IV as the previous output. For instance, denoting the SHA-256 block containing $r \circ i$ as b, we only calculate

$$\text{digest} = f(\text{IV}, b)$$

where f is the commpression function. It should be clear now why `ri_block` and `rj_block` were used in Section 3.3.5.
Because this is not a rare use case, though, this very behavior is already built into libsnark in the shape of the `sha256_two_to_one_hash_gadget`. It uses the `sha256_compression_function_gadget` internally, but assumes the use of the IV. In other words, it implements the very first iteration of the Merkle-Dåmgard construction in SHA-256. Using it looks like

```
sha256_two_to_one_hash_gadget<FieldT> gadget{
  pb, block_size, block, output
};
gadget.generate_r1cs_constraints();
gadget.generate_r1cs_witness();
```

where `pb` is the associated protoboard, `block` the SHA-256 block to be compressed such as `ri_block`, `block_size` the size of `block` (always 64 bytes) for debugging purposes, and `output` the result of the compression function f.
In summary, what happens is that $r \circ i$ and $r \circ j$ are put into SHA-256 blocks that follow a certain structure. For this reason, what is actually passed into the R1CS are the two SHA-256 blocks, which contain $r \circ i$ and $r \circ j$, respectively.
Once the hash values have been calculated, they need to be compared to the input (k_1, k_2).

3.3.7. libsnark: Bitwise Comparison

While the internals of how SHA-256 was modeled in R1CS have not been discussed in detail due to its complexity, comparison will be explained deeper. Bitwise comparison is provided by the `comparison_gadget` in libsnark. Given two integral values A and B of bit length n, their relation to one another are to be determined, i.e., exactly one of $A < B$, $A \leq B$, $A = B$, $A > B$, or $A \geq B$. libsnark solves this by only implementing $A \leq B$ and $A < B$ directly. This yields all the above relations, though, as depicted in the following table:

Desired result	$<$	$\leq$	$=$	$>$	$\geq$
Equivalent expression	$<$	$\leq$	$\leq \wedge \neg <$	$\neg \leq$	$\neg <$

One central expression is used to deduce the values of $A < B$ and $A \leq B$:

$$2^n + B - A \tag{11}$$

2^n adds a bit with the value 1 before the value $B - A$. This bit shall be denoted msb (most significant bit). It shall also be distinguished between whether (11) equals 2^n or not, i.e., whether $A = B$. This shall be represented by a bit called eq.
First, note that $A \leq B \iff \text{msb} = 1$. If $A > B$, msb is cleared via subtraction. If $A \leq B$, it remains set. Second, note that $A < B \iff msb = 1 \wedge eq = 0$. Using these equivalences, definitions for appropriate constraints emerge. Defining the binary variables less and less_or_eq and stating that $A < B \iff \text{less} = 1$ and $A \leq B \iff \text{less\_or\_eq} = 1$, these can be defined in R1CS as

$$\begin{aligned} msb &= \text{less\_or\_eq} \\ msb \cdot eq &= \text{less} \end{aligned}$$

And from these, all other relations can be obtained in accordance with the table.
While this was still a rather high-level explanation, it captures the main ideas. In the final R1CS, (11) is represented bitwise. Therefore, eq actually refers to the fact that bits 0 through $n - 1$ of (11) are cleared. Since the `comparison_gadget` only exposes less_or_eq and less, the needed equality condition is built from these.

3.3.8. libsnark: Logical AND

The last component is that of logical conjunction provided by the `conjunction_gadget`. Given n bits $b_0, \ldots, b_{n-1}$, constraints for an output bit

$$o = \bigwedge_{i=0}^{n-1} b_i \tag{12}$$

are to be found. The basic idea is to compute $\Delta = n - \sum_{i=0}^{n-1} b_i$. Note that $\Delta \geq 0$. If $\Delta = 0$, then $o = 1$, otherwise $o = 0$. However, this "otherwise" poses a problem. A way to map the case that $\Delta > 0$ to the equality $o = 0$ needs to be found. If $\Delta > 0$ could be made equivalent to $\Delta = 1$, o could be very simply described as $\Delta = 1 - o$. This is accomplished by determining the multiplicative inverse Δ^{-1} of Δ. By definition, it fulfills $\Delta \cdot \Delta^{-1} = 1$. All constraints are defined over elements of a (finite) field (see Definition 6 of R1CS and Section 3.3.3). By the definition of a field, each one of these elements has a multiplicative inverse except 0. The following constraint defines o:

$$\Delta^{-1} \cdot \Delta = 1 - o \tag{13}$$

If $\Delta = 0$, $\Delta \cdot \Delta^{-1} = 0$, so the (mathematically non-existent) value of Δ^{-1} can be set arbitrarily. As can be seen, this multiplication by the inverse effectuates the mapping from $\Delta > 0$ to $o = 0$ that was demanded above.
However, a malicious party still could choose Δ^{-1} and o such that (13) is satisfied, but (12) is not. Just consider $\Delta^{-1} = 0$ and $o = 1$, for example. There are plenty more,

though, for specific values of Δ, since arithmetic is performed over a finite field. To eliminate this last problem, the constraint

$$o \cdot \Delta = 0$$

is introduced. It says that any of o and Δ must be equal to 0. If $o = 0$, (13) becomes $\Delta^{-1} \cdot \Delta = 1$ and Δ^{-1} must have the correct value to fulfill the equation. If $\Delta = 0$, (13) becomes $0 = 1 - o$, so $o = 1$, which is the correct output for $\Delta = 0$. In conclusion, this constraint visibly enforces the correct values for Δ^{-1} and o while avoiding sidestepping.

3.3.9. From R1CS to zk-SNARK

The main components have now been described. Some details were not considered worth presenting. The components can be assembled more or less easily to an R1CS, which fulfills (10). This assembly has been merged into a single custom gadget called `common_root_gadget`. It implements `generate_r1cs_constraints` and `generate_r1cs_witness` just like any other `gadget`. This gadget is the basis, which the remainder of this section will build on.
Writing the code to construct the R1CS and describing it was rather lengthy, while the theoretical content was not much. Regarding the step from R1CS to a full-fledged zk-SNARK, it is the other way around. As will be seen, writing the code to generate, prove, and verify (see Section 2.4.4) amounts to approximately thirty lines. However, there is a bulk of theory behind the scenes, which was explained in some detail in Section 2.5.2 and Section 2.5.3.

3.3.10. Generating Proving and Verifying Keys

Generation of the keys involves three steps:

1. Generating the R1CS
2. Creating the keys
3. Writing the keys to files

Generation is achieved by using the `common_root_gadget`:

```
common_root_gadget<FieldT> common_root_gadget{pb};
common_root_gadget.generate_r1cs_constraints();
```

`pb` is the protoboard as usual. After this, the components outlined in the preceding sections have been added to `pb`.
The key creation involves exactly one statement:

```
r1cs_ppzksnark_keypair<default_r1cs_ppzksnark_pp> keypair =
  r1cs_ppzksnark_generator<default_r1cs_ppzksnark_pp>(
    pb.get_constraint_system()
  );
```

After executing this, `keypair.pk` contains the proving key and `keypair.vk` contains the verifying key. `r1cs_ppzksnark_generator` generates the keys from the underlying R1CS of `pb`, which is the `protoboard`.
The final step is to export the keys. This is made very simple by libsnark:

```
std::fstream pk_file{pk_path, std::ios_base::out};
std::fstream vk_file{vk_path, std::ios_base::out};

pk_file << keypair.pk;
vk_file << keypair.vk;
```

Given the path to the proving key file `pk_path` and the path to the verifying key file `vk_path`, the keys end up serialized in these files. They can be imported, i.e., de-serialized afterwards to use them for proving and verifying.

3.3.11. Proving the Statement

Five steps are involved to prove a statement:

1. Generate the R1CS constraints
2. Generate the witness vector for the R1CS (see Definition 6 for details)
3. Read the proving key
4. Create the zk-SNARK
5. Write the zk-SNARK to a file

Generating the constraints and the witness relies on the `common_root_gadget`:

```
common_root_gadget<FieldT> gadget{pb};
gadget.generate_r1cs_constraints();
gadget.generate_r1cs_witness(h1, h2, ri_block, rj_block);
```

where `pb` is the protoboard, `h1` is k_1, `h2` is k_2, `ri_block` is the SHA-256 block containing $r \circ i$ and `rj_block` the SHA-256 block containing $r \circ j$.
Reading the proving key is as easy as writing it to a file:

```
std::fstream pk_file{pk_path, std::ios_base::in};
r1cs_ppzksnark_proving_key<default_r1cs_ppzksnark_pp> pk;
pk_file >> pk;
```

`pk` now contains the de-serialized proving key and is ready to be used. To create the proof, once again, a single statement is needed:

```
r1cs_ppzksnark_proof<default_r1cs_ppzksnark_pp> proof =
  r1cs_ppzksnark_prover<default_r1cs_ppzksnark_pp>(
    pk, pb.primary_input(), pb.auxiliary_input()
  );
```

`pb.primary_input()` contains the R1CS variables' values that describe the input (k_1, k_2). `pb.auxiliary_input()` contains the values of all other variables in the R1CS. Using these and `pk`, the proof is generated. `proof` now contains the zk-SNARK. It is exported to a file with

```
std::fstream proof_file{proof_path, std::ios_base::out};
proof_file << proof;
```

where `proof_path` is the path to the proof file.

3.3.12. Verifying the Statement

To finally verify the proof, the following steps are required:

1. Read the verifying key
2. Read the proof
3. Verify the proof

Reading the verifying key is similar to reading the proving key:

```
std::fstream vk_file{vk_path, std::ios_base::in};
r1cs_ppzksnark_verification_key<default_r1cs_ppzksnark_pp>
  vk;
vk_file >> vk;
```

`vk` now contains the verifying key.
Reading the proof happens in the same way:

```
std::fstream proof_file{proof_path, std::ios_base::in};
r1cs_ppzksnark_proof<default_r1cs_ppzksnark_pp> proof;
proof_file >> proof;
```

`proof` now contains the proof.
To verify the proof, the following is done:

```
r1cs_primary_input<FieldT> primary_input;
for (auto bit : hex_to_bit_vector(h1_str)) {
  primary_input.push_back(bit ?
    FieldT::one() : FieldT::zero());
```

```
}

for (auto bit : hex_to_bit_vector(h2_str)) {
  primary_input.push_back(bit ?
    FieldT::one() : FieldT::zero());
}

return !libsnark::r1cs_ppzksnark_verifier_strong_IC
 <libsnark::default_r1cs_ppzksnark_pp>(
 vk, primary_input, proof
)
```

The input (k_1, k_2) is first obtained from the two strings `h1_str` and `h2_str`. Then, `proof` is verified for the statement $(k_1, k_2) \in L_{common}$. If the verification succeeds, 0 and otherwise 1 is returned.

4. Evaluation

In this section, the obtained solutions will be analyzed and evaluated. First, it is shown and explained how both solutions are used. Both will be compared with regard to their performance. Their usability is discussed and compared afterwards.

4.1. Usage

The crafted solutions are real programs, so this section will showcase their usage. This is supposed to give a feeling of how the solutions are used in practice.

4.1.1. The MPC Solution

Once the MPC program has been build using MP-SPDZ, it can be executed using one of the protocols supplied by MP-SPDZ. The program runs on a virtual machine such that any protocol for two parties can be used interchangeably (see Section 3.2.3). However, a protocol that is secure against a malicious adversary with a dishonest majority is needed (see Section 3.2.1). Various protocols implemented in MP-SPDZ fit these needs and will be examined in Section 4.2. For showing the program's usage, this section will restrict itself to the MASCOT protocol by Keller et al. [52].
The following listing shows what is printed once both parties have launched the program:

```
party 0> ./bin/Linux-amd64/mascot-party.x -N 2 -I -p 0 \
         common_root
No modulus found in Player-Data//2-p-128/Params-Data,
generating 128-bit prime
Please enter 3 numbers:
1 2 3
```

```
Thank you
result: 1
Time = 21.0164 seconds
Data sent = 637.702 MB
Global data sent = 1275.23 MB

party 1> ./bin/Linux-amd64/mascot-party.x -N 2 -I -p 1 \
         common_root
No modulus found in Player-Data//2-p-128/Params-Data,
generating 128-bit prime
result: 1
Time = 21.0157 seconds
Data sent = 637.53 MB
Global data sent = 1275.23 MB
```

`-N 2` means that two parties are involved. `-I` renders the program interactive, i.e., I/O happens via the terminal. `-p x` indicates that party x is launched. `common_root` refers to the program name.
Computation is performed over a finite field (see Section 3.2.3). For MASCOT, that is the field of integers modulo a prime p. Since no user-supplied modulus is found, p is generated first. p must be greater than the specified field size, which is 64 bits by default.
Now, party 0 needs the three values r, i, and j. Since it is interactive, the protocol asks for the three values. (k_1, k_2) is set corresponding to $r = 1$, $i = 2$, $j = 3$ by default (recall that (k_1, k_2) is hardcoded in the program), so these values are entered.
After some time, the result 1 is displayed to both parties 0 and 1. This means that the proof has succeeded and $(k_1, k_2, r, i, j) \in R_{L_{common}}$. Some statistics are also printed to output.
Choosing values different from $r = 1$, $i = 2$, and $j = 3$ makes the program print a result of 0.

4.1.2. The zk-SNARK Solution

Once the zk-SNARK solution has been built, the resulting executable can be run. The three steps of generating, proving, and verifying (see Section 2.4.4) are involved:

```
> ./common_root -g
> ./common_root -p -r 1 -i 2 -j 3 \
  -k1 ae4b3280e56e2faf83f414a6e3dabe9d\
      5fbe18976544c05fed121accb85b53fc \
  -k2 b744d600fbe3853702978ec726c166d2\
      6274fe7b09b2c600ddf2d7d895667b24
> ./common_root -v \
  -k1 ae4b3280e56e2faf83f414a6e3dabe9d\
      5fbe18976544c05fed121accb85b53fc \
```

```
  -k2 b744d600fbe3853702978ec726c166d2\
      6274fe7b09b2c600ddf2d7d895667b24
> echo $?
0
```

`-g` generates the proving and verifying keys. This is alright for this proof-of-concept implementation; in reality, it should be ensured that the generation is performed by a trusted setup, possibly involving MPC (see Section 2.4.3). Two files `proving_key` and `verifying_key` containing the respective keys are created.
`-p` creates a proof. The input (k_1, k_2) is passed along with r, i, and j. The proving key in file `proving_key` is implicitly used. A new file `proof` that contains the proof is created.
`-v` verifies the proof. The input (k_1, k_2) is passed. The verifying key in file `verifying_key` is implicitly used. The program returns 0 if verification succeeded, i.e., $(k_1, k_2) \in L_{common}$. Otherwise, it returns 1.
`echo $?` prints the return code of the verification step.

4.2. Performance

The performance of both solutions will now be evaluated. For this, the setup used to measure will be described first. Then, performance is measured. This is subdivided into data and speed considerations. How much data is exchanged and otherwise needed as well as the execution speed of both implementations is measured, presented, and compared.

4.2.1. General Setup and Methodology

All measurements were performed on a 3.50 GHz Intel Core i5-6600K CPU with 16 GiB RAM. The operating system in use was Linux Fedora 30.
All measurements were performed without any other user-enabled software running.
All measurements are of successful proofs, that is, for keys $(k_1, k_2) \in L_{common}$. The tool is supposed to be used by honest users, so a non-matching pair of keys should not be the rule. Nevertheless, the performance for such non-successful proofs has been measured to be comparable.

4.2.2. Data

The MPC solution is interactive: data is exchanged between the two parties. No other data than that is required. The data sent for one protocol for the MPC implementation remains constant. The following table shows the total data sent between the two parties using MPC protocols that are secure against a malicious adversary with a dishonest majority (see Section 3.2.1):

Protocol	MASCOT	SPDZ2k	FKOS15	LowGear	HighGear
Total Data in MB	1275.23	1275.24	1317.622	1275.21	1275.21

The zk-SNARK solution is non-interactive. The proof is sent one time from the prover to the verifier. Other used data are the proving and verifying keys. These are public and must be procured once from a trusted source. The proof-of-concept implementation described in this thesis generates the keys on its own, which can then be shared. In reality, the keys would have to be generated in a more trustworthy manner using MPC, for example. This could be very data-intensive. Since it is not part of this implementation, it will be left out of consideration. The relevant data is shown in the following table:

Proof	Proving Key	Verifying Key
312 Bytes	22 MB	20 KB

4.2.3. Speed

Both solutions were tested on only one machine, so no actual network connection was established. Therefore, the latencies caused by a true network connection have not been taken into account. The times measured here tell how the solutions perform under optimal external conditions.
Several protocols can be used to run the MPC solution. For this reason, measurements were carried out for all these protocols. They are the same as in Section 4.2.2 and fulfill the conditions demanded in Section 3.2.1.
For each protocol, a hundred measurements were performed. For each measurement, a shell script sampled random values for m, i, and j and created the corresponding keys (k_1, k_2). The script substituted the keys into the source code, compiled the code, and ran it for the respective protocol, passing m, i, and j as input.
For each measurement of the MPC solution, the real time reported by the `time` command [53] was saved. Since the execution times of both parties differ only negligibly for one measurement, the times of party 0 are the ones effectively shown in this thesis.
Figure 4 shows that execution speed was roughly 12 seconds for all protocols. Since all measurements lie at most two seconds apart, the small differences between protocols will not be further discussed. While the measured times are sufficient for the task that the program fulfills, network delay plays a major role due to the immense size of transferred data. Each protocol transfers over one gigabyte of data. If both parties can exchange data at 100 MBit per second, which is 12.5 MB per second, only data transmission still takes more than a minute.
For the zk-SNARK solution, a hundred measurements were performed per component. The components are generating, proving, and verifying. The real time reported by the `time` command [53] was saved for each measurement. Similarly to the MPC solution, for each measurement of the proving and verifying steps, random values for m, i, and j were created and based on these (k_1, k_2) was computed. These values were passed to the proving component, so that a proof was created and then passed to the verifying component.
As is visible in Figure 5, generating took about 15 seconds, proving about 19.5 seconds, and verifying about 50 milliseconds. Since the generating step is a one-time operation,

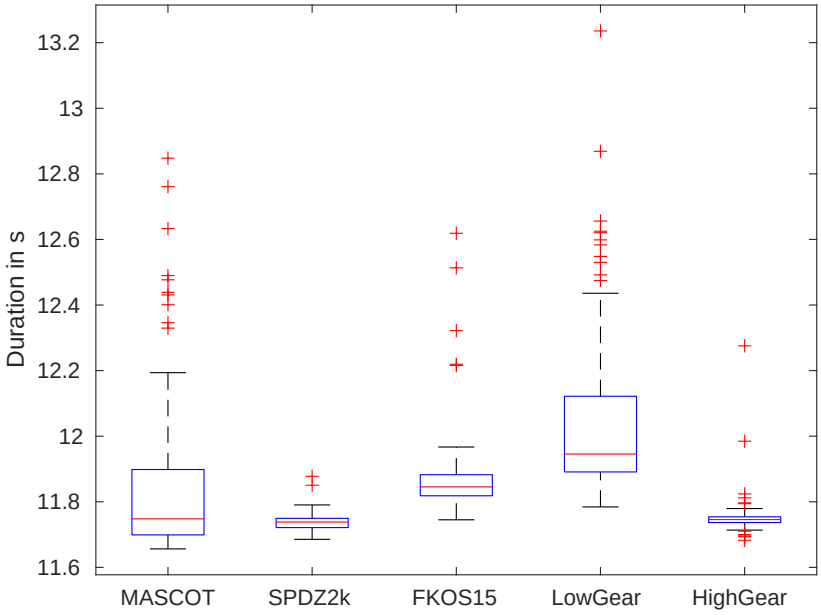

Figure 4: Execution speeds per MPC protocol.

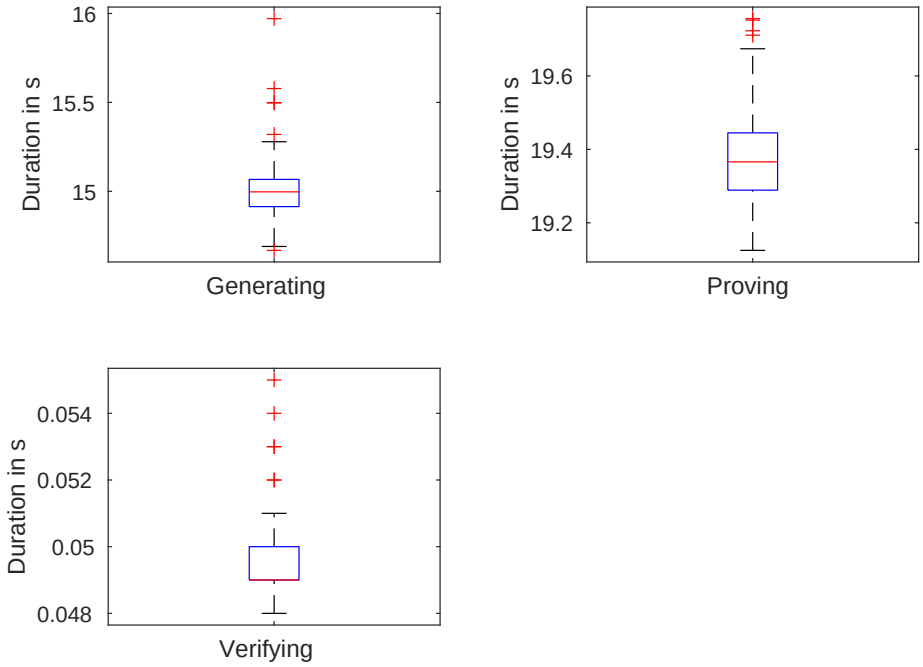

Figure 5: Execution speeds of zk-SNARK components.

15 seconds are perfectly fine. The proving part takes a little longer than MPC, but is independent of network speeds, which MPC is not. Verifying is very fast. All in all, speed is not an issue regarding the zk-SNARK implementation.

4.3. Usability

The most obvious difference that affects usability is the fact that the MPC solution is interactive while the zk-SNARK solution is non-interactive. The former requires that both parties are active at the same time. With the latter, the proof is put in an accessible place from where the verifier can retrieve it at any time. Hence, less planning overhead is needed for the non-interactive variant, which increases its ease of use.
As the data shows, execution speed is not a real problem for any solution. While the computer used for measurements was perhaps above average, even speeds crossing the one-minute mark should be acceptable. Proving membership of two public keys in the same deterministic wallet is not envisioned to be a routine task that needs to be carried out at high frequencies.
The amounts of data exchanged differ significantly, however. The zk-SNARK itself is only 312 bytes in size. This is negligibly small compared with the whopping 1.2 GB transferred over the network using MPC. Such amounts of data are obviously terrible and require fast connections. Otherwise, performing the proof via MPC could become impossibly slow.
The one great drawback of the zk-SNARK solution is the need for trusted generation of the keys (see Section 2.4.3). The proof of concept described in this thesis does not do this to a great extent. For real-world use, it is absolutely necessary, though. Law enforcement (see Section 2.2.5) could possibly be assumed to be a trusted party: they could fake proofs, but that is not in their interest. Hence, the problem of establishing a trusted setup would be solved. Otherwise, one would have to resort to MPC.
This prior generation step is not necessary for the MPC solution.
Lastly, let it be said that the MPC solution needs recompilation for every new pair of keys. It is not the case for the zk-SNARK solution. Since it is only a proof of concept and there probably is a way to fix it, this argument does not matter much, however.

5. Conclusion and Future Work

In this last section, it shall be concluded what solutions have been constructed to prove in zero-knowledge that two public keys belong to the same deterministic wallet. Furthermore, possible avenues for future work will be presented.

5.1. Conclusion

This thesis has proposed two ways to solve the problem introduced in Section 3.1: given a pair of public keys, prove in zero-knowledge that both keys were derived by the

same deterministic wallet. Two working proof-of-concept implementations, one using MPC, the other one using a zk-SNARK, have been constructed and tested.
Benchmarks have shown that both implementations execute fast. The MPC solution is highly dependent on a fast network connection, though.
In conclusion, the zk-SNARK solution will surely perform better in practice. The amounts of data transferred using MPC are prohibitively large and the non-interactive nature of zk-SNARKs offers better ease of use. However, the used zk-SNARK scheme relies on a trusted setup to generate its keys. Depending on its exact implementation, this trusted setup could be a costly endeavor.

5.2. Future Work

While the two solutions have been shown to work conceptually, improvements need to be made to yield products viable for the real world. The MPC solution must fix the problem of recompilation for new inputs. Both should strive to fully implement BIP 32 (see Section 2.2.3) to be compatible with current software. Furthermore, trustworthy generation of the keys should either be done by law enforcement or be enabled via MPC.
In general, MPC and zk-SNARKs are in steady development. Any advances like new protocols or optimizations to existing protocols can yield better solutions. In fact, more modern zk-SNARK approaches than the used in this thesis exist such as Groth16 [16]. zk-STARKs that do not require a trusted setup, implemented in libSTARK [44] might also yield a potential solution. These are a lot more experimental than what libsnark has to offer, though.

Appendices

A. MP-SPDZ Code

```
# Player 0 is the prover, player 1 is the verifier.
#
# Primary input (public): (h1, h2)
# Auxiliary input (private, only known to prover): m, i, j
# Pseudocode (H is a cryptographic hash function):
# return h1 == H(m || i) && h2 == H(m || j)

# The Keccak circuit necessary for the sha3_256 function
# comes from
# https://homes.esat.kuleuven.be/~nsmart/MPC/Keccak_f.txt

from circuit import sha3_256

# Custom data types.
root_t = sbits.get_type(16)
index_t = sbits.get_type(8)
preimage_t = sbits.get_type(root_t.n + index_t.n)

# The hash values as strings.
h1_str = "37ad9e5625d7e5fae209db84a28dc45d" \
         "3e8e4631f41827a837db8073de4629eb"
h2_str = "53937f3b7fb58e104f113264695d5da3" \
         "4e4bf43da3dc3652ac7d868b4c122ce9"

# Store string representation of hash values in
# 256-bit integers.
h1 = sbits.get_type(256)(int(h1_str, 16))
h2 = sbits.get_type(256)(int(h2_str, 16))

# Read root key and indices from player 0.
m = root_t.get_input_from(0)
i = index_t.get_input_from(0)
j = index_t.get_input_from(0)

# Concatenate root key and indices bitwise to obtain the
# preimages plugged into the hash function.
mi = preimage_t((m << index_t.n) + i)
mj = preimage_t((m << index_t.n) + j)
```

```
# Compute SHA3-256 values of preimages and store the
# results.
result = sha3_256(sbitvec([mi, mj])).elements()
h_mi = result[0]
h_mj = result[1]

# Check if the computed hash values match the given ones.
# If so, 1 is printed, if not, 0 is printed.
h1_good = h1.equal(h_mi)
h2_good = h2.equal(h_mj)
good = h1_good.bit_and(h2_good)

print_ln('result: %s', good.reveal())
```

B. libsnark Code

For clarity, only the relevant parts of the libsnark implementation are shown. Omitted parts are marked by three vertical dots.

```
// Necessary #includes

                              .
                              .
                              .

using namespace std;
using namespace libff;
using namespace libsnark;

                              .
                              .
                              .

/*
 * This gadget provides hashing and comparison of the hash.
 * It encapsulates the functionality described in
 * Section 3.3.6 and Section 3.3.7.
 */
template<typename FieldT>
class hash_cmp_gadget : public gadget<FieldT> {
    pb_variable_array<FieldT> supplied_bit_array;
    block_variable<FieldT> block;
```

```
digest_variable<FieldT> computed_hash;
sha256_two_to_one_hash_gadget<FieldT> hash_gadget;
dual_variable_gadget<FieldT>
    computed_packer1, computed_packer2;
dual_variable_gadget<FieldT>
    supplied_packer1, supplied_packer2;
pb_variable<FieldT> less1, less2,
    less_or_eq1, less_or_eq2;
comparison_gadget<FieldT> cmp1, cmp2;
pb_variable_array<FieldT> conjunction_inputs;
conjunction_gadget<FieldT> conjunction;

public:
pb_variable<FieldT> equal;

hash_cmp_gadget(
    protoboard<FieldT> &pb,
    const pb_variable_array<FieldT>&
        supplied_bit_array,
    const block_variable<FieldT> &block,
    const string &annotation_prefix = ""
) :

// member variable initializations
                            .
                            .
                            .

{ }

    void generate_r1cs_constraints()
{
    computed_hash.generate_r1cs_constraints();
    hash_gadget.generate_r1cs_constraints();

    computed_packer1.generate_r1cs_constraints(true);
    computed_packer2.generate_r1cs_constraints(true);

    supplied_packer1.generate_r1cs_constraints(true);
    supplied_packer2.generate_r1cs_constraints(true);

    cmp1.generate_r1cs_constraints();
    cmp2.generate_r1cs_constraints();
```

```
        // ensures negation
        this->pb.add_r1cs_constraint(
            r1cs_constraint<FieldT>(
            1, {{less1, conjunction_inputs[0]}}, 1),
            "");
        // ensures negation
        this->pb.add_r1cs_constraint(
            r1cs_constraint<FieldT>(
            1, {{less2, conjunction_inputs[1]}}, 1),
            "");
        // ensures equality
        this->pb.add_r1cs_constraint(
            r1cs_constraint<FieldT>(
            1, {{less_or_eq1}}, {{conjunction_inputs[2]}}),
            "");
        // ensures equality
        this->pb.add_r1cs_constraint(
            r1cs_constraint<FieldT>(
            1, {{less_or_eq2}}, {{conjunction_inputs[3]}}),
            "");

        conjunction.generate_r1cs_constraints();
    }

    void generate_r1cs_witness()
    {
        hash_gadget.generate_r1cs_witness();

        computed_packer1.generate_r1cs_witness_from_bits();
        computed_packer2.generate_r1cs_witness_from_bits();
        supplied_packer1.generate_r1cs_witness_from_bits();
        supplied_packer2.generate_r1cs_witness_from_bits();

        cmp1.generate_r1cs_witness();
        cmp2.generate_r1cs_witness();

        this->pb.val(conjunction_inputs[0]) =
            this->pb.val(less1) == FieldT::zero() ?
            FieldT::one() : FieldT::zero();
        this->pb.val(conjunction_inputs[1]) =
            this->pb.val(less2) == FieldT::zero() ?
            FieldT::one() : FieldT::zero();
        this->pb.val(conjunction_inputs[2]) =
            this->pb.val(less_or_eq1);
```

```
        this->pb.val(conjunction_inputs[3]) =
            this->pb.val(less_or_eq2);

        conjunction.generate_r1cs_witness();
    }
};

                                .
                                .
                                .

/*
 * This gadget uses 'hash_cmp_gadget' to compute the two
 * hashes and compare them. The results are connected
 * via a binary AND as described in Section 3.3.8.
 * The SHA-256 blocks are passed as input to this gadget,
 * so it is ensured that both start with the same r as
 * described in Section 3.3.5.
 */
template<typename FieldT>
class common_master_gadget : public gadget<FieldT> {
    block_variable<FieldT> ri_block;
    block_variable<FieldT> rj_block;
    hash_cmp_gadget<FieldT> h1_gadget, h2_gadget;
    pb_variable_array<FieldT> conjunction_inputs;
    conjunction_gadget<FieldT> conjunction;

    public:
    pb_variable<FieldT> output;

    common_master_gadget(
        protoboard<FieldT> &pb,
        const pb_variable_array<FieldT> &h1_array,
        const pb_variable_array<FieldT> &h2_array,
        const string &annotation_prefix = ""
    ) :

    // member variable initializations
                                .
                                .
                                .

    { }
```

```
    void generate_r1cs_constraints()
{
    for (size_t i = 0; i < sizeof(root_t) * CHAR_BIT;
        ++i) {
        this->pb.add_r1cs_constraint(
            r1cs_constraint<FieldT>{
            1, {{ri_block.bits[i]}},
            {{rj_block.bits[i]}}}
        );
    }

    h1_gadget.generate_r1cs_constraints();
    h2_gadget.generate_r1cs_constraints();

    // ensures equality
    this->pb.add_r1cs_constraint(
        r1cs_constraint<FieldT>(1,
        {{conjunction_inputs[0]}}, {{h1_gadget.equal}}
        ), ""
    );

    // ensures equality
    this->pb.add_r1cs_constraint(
        r1cs_constraint<FieldT>(1,
        {{conjunction_inputs[1]}}, {{h2_gadget.equal}}
        ), ""
    );

    conjunction.generate_r1cs_constraints();

    this->pb.add_r1cs_constraint(
        r1cs_constraint<FieldT>{1, {{output}}, 1}
    );
}

void generate_r1cs_witness(
    const bit_vector &ri_block_bits,
    const bit_vector &rj_block_bits
)
{
    ri_block.generate_r1cs_witness(ri_block_bits);
    rj_block.generate_r1cs_witness(rj_block_bits);

    h1_gadget.generate_r1cs_witness();
```

```
        h2_gadget.generate_r1cs_witness();

        this->pb.val(conjunction_inputs[0]) =
            this->pb.val(h1_gadget.equal);
        this->pb.val(conjunction_inputs[1]) =
            this->pb.val(h2_gadget.equal);

        conjunction.generate_r1cs_witness();

    }
};

                                .
                                .
                                .

// Wrapper for the generating algorithm.
int generate(const string &pk_path, const string &vk_path)
{
    protoboard<FieldT> pb{};

    pb_variable_array<FieldT> h1_array, h2_array;
    h1_array.allocate(pb, 256, "h1_array");
    h2_array.allocate(pb, 256, "h2_array");

    pb.set_input_sizes(2 * 256);

    common_master_gadget<FieldT> common_master_gadget
        {pb,h1_array, h2_array};
    common_master_gadget.generate_r1cs_constraints();

    r1cs_ppzksnark_keypair<default_r1cs_ppzksnark_pp>
        keypair = r1cs_ppzksnark_generator
        <default_r1cs_ppzksnark_pp>
        (pb.get_constraint_system());

    fstream proving_key_file{pk_path, ios_base::out};
    if (!proving_key_file) {
        cerr << "Opening␣file␣" << pk_path << "␣failed.\n";
        return EXIT_FAILURE;
    }

    fstream verifying_key_file{vk_path, ios_base::out};
    if (!verifying_key_file) {
```

```
        cerr << "Opening␣file␣" << vk_path << "␣failed.\n";
        return EXIT_FAILURE;
    }

    proving_key_file << keypair.pk;
    verifying_key_file << keypair.vk;

    return EXIT_SUCCESS;
}

// Wrapper for the proving algorithm.
int prove(
    const string &h1_str, const string &h2_str,
    root_t r, index_t i, index_t j, const string &pk_path,
    const string &proof_path
)
{
    auto m_bits =
        int_list_to_bits({r}, sizeof(r) * CHAR_BIT);
    auto i_bits =
        int_list_to_bits({i}, sizeof(i) * CHAR_BIT);
    auto j_bits =
        int_list_to_bits({j}, sizeof(j) * CHAR_BIT);

    protoboard<FieldT> pb{};

    pb_variable_array<FieldT> h1_array, h2_array;
    h1_array.allocate(pb, 256, "h1_array");
    h2_array.allocate(pb, 256, "h2_array");

    pb.set_input_sizes(2 * 256);

    h1_array.fill_with_bits(pb, hex_to_bit_vector(h1_str));
    h2_array.fill_with_bits(pb, hex_to_bit_vector(h2_str));

    common_master_gadget<FieldT> common_master_gadget
        {pb, h1_array, h2_array};
    common_master_gadget.generate_r1cs_constraints();
    common_master_gadget.generate_r1cs_witness(
        to_block(m_bits, i_bits),
        to_block(m_bits, j_bits)
    );

    fstream proving_key_file{pk_path, ios_base::in};
```

```
    if (!proving_key_file) {
        cerr << "Opening␣file␣" << pk_path << "␣failed.\n";
        return EXIT_FAILURE;
    }

    r1cs_ppzksnark_proving_key<default_r1cs_ppzksnark_pp>
        proving_key;
    proving_key_file >> proving_key;

    r1cs_ppzksnark_proof<default_r1cs_ppzksnark_pp> proof =
        r1cs_ppzksnark_prover<default_r1cs_ppzksnark_pp>(
        proving_key, pb.primary_input(),
        pb.auxiliary_input());

    fstream proof_file{proof_path, ios_base::out};
    if (!proof_file) {
        cerr << "Opening␣file␣" << proof_path <<
            "␣failed.\n";
        return EXIT_FAILURE;
    }

    proof_file << proof;

    return EXIT_SUCCESS;
}

// Wrapper for the verifying algorithm.
int verify(
    const string &h1_str, const string &h2_str,
    const string &vk_path,
    const string &proof_path
)
{
    fstream verifying_key_file{vk_path, ios_base::in};
    if (!verifying_key_file) {
        cerr << "Opening␣file␣" << vk_path << "␣failed.\n";
        return EXIT_FAILURE;
    }

    fstream proof_file{proof_path, ios_base::in};
    if (!proof_file) {
        cerr << "Opening␣file␣" << proof_path <<
            "␣failed.\n";
        return EXIT_FAILURE;
```

```
    }

    r1cs_ppzksnark_proof<default_r1cs_ppzksnark_pp> proof;
    r1cs_ppzksnark_verification_key
        <default_r1cs_ppzksnark_pp> verifying_key;
    r1cs_primary_input<FieldT> primary_input;

    proof_file >> proof;
    verifying_key_file >> verifying_key;

    for (auto bit : hex_to_bit_vector(h1_str))
        primary_input.push_back(bit ?
            FieldT::one() : FieldT::zero());
    for (auto bit : hex_to_bit_vector(h2_str))
        primary_input.push_back(bit ?
            FieldT::one() : FieldT::zero());

    return !r1cs_ppzksnark_verifier_strong_IC
        <default_r1cs_ppzksnark_pp>(
            verifying_key, primary_input, proof
    );
}

                                .
                                .
                                .

// Entry point of the program.
int main(int argc, char **argv)
{
    string h1_str = "";
    string h2_str = "";
    uint16_t m;
    uint8_t i, j;
    string pk_path = "proving_key";
    string vk_path = "verifying_key";
    string proof_path = "proof";
    task_type task;

    parse_options(
        argc, argv, h1_str, h2_str, m, i, j,
        pk_path, vk_path, proof_path, task
    );
```

```
    default_r1cs_ppzksnark_pp::init_public_params();

    switch (task) {
        case task_type::none:
            usage(argv[0]);
            return EXIT_SUCCESS;
        case task_type::generate:
            return generate(pk_path, vk_path);
        case task_type::prove:
            return prove(
                h1_str, h2_str, m, i, j,
                pk_path, proof_path
            );
        case task_type::verify:
            return verify
                (h1_str, h2_str, vk_path, proof_path);
    }
}
```

References

[1] CoinMarketCap. https://coinmarketcap.com/all/views/all/. Accessed: 2021-08-15.

[2] Sean Foley, Jonathan R. Karlsen, and Tālis J. Putniņš. Sex, Drugs, and Bitcoin: How Much Illegal Activity Is Financed through Cryptocurrencies? *The Review of Financial Studies*, 32(5):1798–1853, 04 2019.

[3] Patrik Keller, Martin Florian, and Rainer Böhme. Collaborative deanonymization. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *Financial Cryptography and Data Security. FC 2021 International Workshops*, pages 39–46, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[5] FIPS 180-2, Secure Hash Standard. https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf. Accessed: 2021-08-20.

[6] NIST SP 800-106, Randomized Hasing for Digital Signatures. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-106.pdf. Accessed: 2021-08-20.

[7] Saif Al-Kuwari, James H. Davenport, and Russell J. Bradford. Cryptographic hash functions: Recent design trends and security notions. Cryptology ePrint Archive, Report 2011/565, 2011. https://ia.cr/2011/565.

[8] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford, CA, USA, 1979. AAI8001972.

[9] Ivan Damgård. A design principle for hash functions. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '89, page 416–427, Berlin, Heidelberg, 1989. Springer-Verlag.

[10] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 2001.

[11] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Secur.*, 1(1):36–63, August 2001.

[12] Bitcoin Developer: Wallets. https://developer.bitcoin.org/devguide/wallets.html. Accessed: 2021-08-15.

[13] BIP 32: Hierarchical Deterministic Wallets. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki. Accessed: 2021-08-15.

[14] Steven Goldfeder, Harry A. Kalodner, Dillon Reisman, and Arvind Narayanan. When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies. *Proc. Priv. Enhancing Technol.*, 2018(4):179–199, 2018.

[15] CoinJoin. `https://en.bitcoin.it/wiki/CoinJoin`. Accessed: 2021-08-20.

[16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[17] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[18] Showing without Telling. `http://www.wisdom.weizmann.ac.il/~/oded/poster03.html`. Accessed: 2021-08-20.

[19] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[20] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *J. Cryptol.*, 7(1):1–32, December 1994.

[21] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 103–112, New York, NY, USA, 1988. Association for Computing Machinery.

[22] Parameter Generation. `https://z.cash/technology/paramgen/`. Accessed: 2021-08-20.

[23] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 326–349, New York, NY, USA, 2012. Association for Computing Machinery.

[24] Zcash Protocol Specification. `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`. Accessed: 2021-07-30.

[25] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, San Diego, CA, August 2014. USENIX Association.

[26] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[27] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 90–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[28] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for r1cs. Cryptology ePrint Archive, Report 2018/828, 2018. `https://ia.cr/2018/828`.

[29] Ron Larson. *College Algebra, Hybrid Edition*. Cengage Learning, 2014.

[30] Annette Burden, Richard Burden, and J. Faires. *Numerical Analysis, 9th Revised ed.* Cengage Learning, 2016.

[31] Madars Virza. *On deploying succinct zero-knowledge proofs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA, 2017.

[32] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.

[33] David Evans, Vladimir Kolesnikov, and Mike Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation*. now, 2018.

[34] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser Iv. High-precision secure computation of satellite collision probabilities. In *Proceedings of the 10th International Conference on Security and Cryptography for Networks - Volume 9841*, page 169–187, Berlin, Heidelberg, 2016. Springer-Verlag.

[35] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, December 2020.

[36] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[37] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[38] awesome-mpc. `https://github.com/rdragos/awesome-mpc`. Accessed: 2021-08-15.

[39] Multi-Protocol SPDZ. `https://github.com/data61/MP-SPDZ`. Accessed: 2021-08-15.

[40] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[41] MP-SPDZ Documentation. `https://mp-spdz.readthedocs.io/en/latest/`. Accessed: 2021-08-15.

[42] SHA3-256: only first 64 bits are correct. `https://github.com/data61/MP-SPDZ/issues/247`. Accessed: 2021-08-15.

[43] Keccak sponge function. `https://github.com/zkcrypto/bellman/`. Accessed: 2021-07-30.

[44] libsnark: a C++ library for zkSNARK proofs. `https://github.com/scipr-lab/libsnark`. Accessed: 2021-07-30.

[45] libsnark: a C++ library for zkSNARK proofs. `https://github.com/zcash/zcash/tree/v2.0.7-3/src/snark`. Accessed: 2021-07-30.

[46] jsnark. `https://github.com/akosba/jsnark`. Accessed: 2021-07-30.

[47] snarky. `https://github.com/o1-labs/snarky`. Accessed: 2021-07-30.

[48] bellman. `https://github.com/zkcrypto/bellman/`. Accessed: 2021-07-30.

[49] snarkjs. `https://github.com/iden3/snarkjs`. Accessed: 2021-08-15.

[50] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. `https://ia.cr/2019/953`.

[51] Circom. `https://github.com/iden3/circom`. Accessed: 2021-08-15.

[52] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 830–842, New York, NY, USA, 2016. Association for Computing Machinery.

[53] Linux User's Manual: time. `https://man7.org/linux/man-pages/man1/time.1.html`. Accessed: 2021-08-15.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den October 20, 2021 .....