Dominik Schreilechner, BSc

Extended Functionality of Cryptographic Protocols in Bitcoin

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Christian Rechberger

Institute of Applied Information Processing and Communications
Head: O.Univ.-Prof. Dipl-Ing. Dr.techn. Reinhard Posch

Graz, February 2019

Affidavit

Date	Signature

Abstract

As Bitcoin, and cryptocurrencies in general, are gaining in popularity it becomes more viable to use them for various purposes. The rise in adoption, however, impacts the computational load on the cryptocurrency network and therefore the time it takes until transactions are processed. This makes it difficult to deploy them in scenarios, where a customer expects to instantly receive purchased goods. In this work a payment protocol is presented, which makes fast and secure offline payments with cryptocurrencies possible. In order to enable offline transactions, the protocol is built around a double authentication preventing signature (DAPS) scheme, with which fraudulent behavior can be disincentivized. An other approach for faster payments is to use a payment network on top of the cryptocurrency. Payment networks allow for bidirectional transactions within a channel of two participants and to route payments through an arbitrary amount of channels. However, they require an online connection to function properly. Even though the presented protocol only supports unidirectional transactions, they can be performed completely offline in a fast and secure manner.

Kurzfassung

Bitcoin und generell Kryptowährungen gewinnen immer mehr an Popularität. Dies ermöglicht es verschiedenste Anwendungen dafür umzusetzen. Der vermehrte Einsatz von Kryptowährung führt jedoch dazu, dass das Netzwerk der jeweiligen Währung stärker ausgelastet wird, was wiederum die Zeit beeinträchtigt, die eine Transaktion braucht um durchgeführt zu werden. Das macht es schwierig Kryptowährungen in Bereichen einzusetzen, in denen ein Kunde erwartet seine bezahlten Güter sofort zu erhalten. In dieser Arbeit wird ein Bezahlprotokoll vorgestellt, mit welchem man schnelle und sichere Offline-Transaktionen in Kryptowährungen ausführen kann. Um Transaktionen offline zu ermöglichen, ist das Protokoll um ein so gennantes Double Authentication Preventing Signature (DAPS) Schema aufgebaut, mit dem es möglich ist Betrugsversuche zu entmutigen. Ein anderer Ansatz um schnelle Transaktionen in Kryptowährungen zu ermöglichen ist es, ein Bezahlnetzwerk aufbauend auf der Kryptowährung zu verwenden. In einem Bezahlnetzwerk wird ein Kanal zwischen zwei Teilnehmern hergestellt und Transaktionen können bidirektional zwischen ihnen durchgeführt werden. Des weiteren kann eine Zahlung auch durch eine beliebige Anzahl von Kanälen geführt werden. Diese Netzwerke haben jedoch den Nachteil, dass für ihre Funktion eine Online-Verbindung benötigt wird. Das in dieser Arbeit vorgestellte Protokoll ermöglicht nur unidirektionale Transaktionen, diese können jedoch schnell und sicher offline durchgeführt werden.

Contents

1 Introduction

In the digital realm it is easy to exactly replicate or modify any kind of data. This has on one side the obvious advantages, that it is easy to distribute and work on the data. On the other side, however, these properties make it difficult to verify, if the data was altered by some party or if their exist different versions of it. To solve this issue, the data can be signed with a digital signature scheme. Whenever signed data is altered, the verification process of the signature will fail and the tampering can be detected. Therefore, the origin and integrity of the data can be validated with a signature scheme. This is sufficient for most use cases, like the public key infrastructure (PKI) of the web or code signing for software. However, a digital signature scheme cannot hinder a party to publish an arbitrary amount of distinct versions of some data. This might not sound as a problem at first, but there are several use cases for which it is desirable that a party cannot make contradicting statements or publish contradicting data.

One area where contradiction is inherently a problem is digital currencies or cryptocurrencies. Here a party should not be able to spend a digital coin multiple times, which is also called double spending. In other words, in a cryptocurrency it should not be possible to make the contradicting statement, that some coin belongs to party A and simultaneously party B. Modern cryptocurrencies, like Bitcoin [Nak09], solve this problem by using a blockchain as a ledger for the transactions. Every transaction made is stored on the blockchain and it is publicly visible for everyone. Doing so makes a consensus finding process possible, where everybody can participate in determining, which transactions are valid and should be incorporated into the blockchain. This stands in stark contrast to conventional payment methods, where a central authority, like a bank, has to be trusted to take care of each transaction. Following the approach with the blockchain, however, comes with the downside, that the time a transaction takes to be executed is limited by the duration of the consensus finding process. In Bitcoin for example, this process takes about 10 minutes on average[1].

[1] https://en.bitcoin.it/wiki/Confirmation#Confirmation_Times

Because it is not guaranteed, that a transaction will be included in the next round of the consensus finding, it can take longer than these 10 minutes until a transaction is added to the blockchain.

In recent years cryptocurrencies are gaining more and more in public awareness and find wider adoption. For example Bitcoin has become an officially recognized payment method in Japan[2], and Venezuela[3] and Iran[4] are launching their own national cryptocurrency. This rise in popularity further increases the computational load on the network of the cryptocurrency and therefore the time it takes until a transaction is processed and added to the blockchain. This high transaction times may not be a problem for use-cases like online stores, where the ordered goods still have to be shipped to the costumer, but there are several scenarios, for which a fast payment process is desirable. These scenarios include on-demand and streaming services, digital goods, which can be downloaded right away and buying goods at a store in person.

In this thesis a payment protocol is presented, with which it is possible to make fast transactions in a cryptocurrency. The confirmation time is basically only bound by the time it takes to sign and verify a transaction. In this protocol an interaction with the blockchain is only needed to establish and close a channel. Therefore, no transaction in this channel is added to the blockchain, which removes the delay from the consensus finding process. Besides being off-chain, the transactions can also be performed completely offline within certain constraints. The payment protocol is based on the application presented in [RKS15] and is built around a double authentication preventing signature (DAPS) scheme. A DAPS scheme acts like a conventional digital signature scheme, as long as only one statement is signed per context. If two distinct statements are ever signed for the same context, it is possible to extract the secret signing key from the signatures. Thus, a DAPS does not directly prevent publishing two contradicting statements, but it acts as a disincentive. It can, however, only do so, if the signing key itself possesses some value for its owner. A cryptocurrency has a monetary value and its funds are usually bound to a cryptographic key. Therefore, if the DAPS key used to sign a transaction is additionally related to a deposit of a cryptocurrency, this can suffice as an disincentive. By using a DAPS scheme, the presented protocol achieves fast transactions for cryptocurrencies and, even so

[2]https://www.cnbc.com/2017/04/12/bitcoin-price-rises-japan-russia-regulation.html

[3]https://www.nytimes.com/2017/12/03/world/americas/venezuela-cryptocurrency-maduro.html

[4]https://www.aljazeera.com/news/2019/01/iran-inches-closer-unveiling-state-backed-cryptocurrency-190127060320571.html

it can be executed completely offline, it still provides a protection against double spending.

Other solutions to the problem of increasing transaction times are to use a faster consensus finding process, like for example Ripple [SYB14] or payment networks on top of the cryptocurrency. The consensus finding process of Ripple is fast enough for the aforementioned scenarios, but its throughput might not be high enough for a widespread adoption. Payment networks, like Lightning [PD16] for Bitcoin, also operate on an off-chain channel, similar to the one in the presented protocol. In payment networks, a bidirectional channel is established between two parties and an arbitrary amount of transactions can be exchanged between them, until the channel is closed. To be able to make payments to anyone on the cryptocurrency network, a path is routed through several channels from payer to payee. In contrast, the presented protocol only allows for unidirectional transactions to partners of a provider. However, transactions can be made completely offline, whereas in payment networks an online connection is required to find a payment path.

The remainder of this thesis starts with an overview of all relevant cryptographic building blocks in Chapter 2. Thereafter the basic functionality of a cryptocurrency will be described, followed by a comparison of the three cryptocurrencies Bitcoin, Ethereum and Ripple as well as a more in-depth description of Bitcoin in Chapter 3. Then Chapter 4 gives a description of DAPS schemes, a comparison with other self-enforcing signatures, an overview of all currently known DAPS and the concrete scheme utilized in this thesis. In Chapter 5 applications for DAPS are presented, including a more detailed description of the payment protocol for Bitcoin and a comparison with other payment methods for cryptocurrencies. Implementation details of this payment protocol are then given in Chapter 6, where additionally the execution of the implemented Bitcoin scripts will be presented. After that a conclusion of the work is given in Chapter 7.

2 Cryptographic Building Blocks

In the following an overview on the cryptographic primitives used in this thesis is presented. Cryptocurrencies make use of several of them to enable a secure operation in the digital world. They utilize hash functions, to provide integrity and digital signatures, to provide authenticity for their data. The double authentication preventing signature (DAPS) scheme, which was focused on in this thesis, employs an encryption scheme, secret sharing and zero-knowledge proofs, which will also be covered here. Additionally, in Section 2.4 an overview of the discrete logarithm setting and its use in cryptography is presented, including a description of two concrete schemes, namely ECDSA and elliptic curve ElGamal. The descriptions in this chapter follow mostly the work from [Kat10], [KL14] and [Sma15].

A definition used throughout this work is, that the probability Pr that a certain event E occurs, is negligible. This means that the probability is smaller than a negligible function $\varepsilon(\kappa)$, where κ is the security parameter and thus,

$$\mathsf{Pr}[E] \leq \varepsilon(\kappa). \tag{2.1}$$

A function is negligible, if it approaches zero faster than the inverse of any polynomial. In other words, it is negligible if there is an integer κ_c so that

$$\varepsilon(\kappa) \leq \kappa^{-c}, \tag{2.2}$$

for every $\kappa > \kappa_c$ and every $c \in \mathbb{N}$.

2.1 Hash Functions

A hash function is a one-way function, which compresses its input. Therefore, it takes an input of various length and produces an output of fixed length. Hash functions have various applications either on their own, as for example modification detection codes (MDC) and password hashes or as part of other schemes, like digital signature schemes or message authentication code schemes (MAC). A hash function H with an output length of l, an input $x \in \{0,1\}^*$ and an output $y \in \{0,1\}^l$ can be described as follows,

$$H(x) = y. \tag{2.3}$$

The output or hash y should be unique for its input x, but hence data of an arbitrary length is mapped to data of fixed length, there are inevitably multiple sets of input data, that produce the same output. The event, that two different inputs x and x' produce the same hash is called a collision. Because it is impossible by its definition to have no collisions, a cryptographic hash function has to be designed in a way, that a collision only occurs with negligible probability. The probability, that a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ is able to produce a collision is

$$\Pr\left[(x,x') \leftarrow \mathcal{A}(1^\kappa) : x \neq x' \wedge H(x) = H(x')\right], \tag{2.4}$$

and should be negligible. The straight forward way to find a collision is to try out $2^n + 1$ different inputs for a hash function with an n-bit output. This will guarantee to find a collision. However, when the input is chosen randomly, a collision can be found with a probability of more than 50% with only about $2^{n/2}$ inputs, due to the birthday paradox. Therefore, the upper boundary of the collision resistance of a hash function is about the square root of the number of all possible outputs, or $2^{n/2}$ for an output size of n-bit. The actual collision resistance does not necessarily comply to this number, because there might be a more efficient way to find a collision, due to the design of the function. Either way, for cryptographic hash functions, like the SHA2-family [01a] or SHA3 [15], which are considered secure at the moment, there is no other, efficient method known yet. For example, SHA-256 has an output size of 256-bit and therefore a collision resistance of 2^{128}. That means, if every person on earth would cooperatively calculate hashes with a computing power of 10^9 hashes/s each, it would take more than 10^{12} years to find a collision.

Another important property of hash functions, which is similar to collision resistance, is called 2^{nd} pre-image resistance or universal one-way. A hash function is 2^{nd} pre-image resistant, if the probability, that a PPT adversary can find an input x', which produces a collision for an given input x, is negligible. In other words, given x it should be hard to find x', so that $H(x) = H(x')$. Formally, the following probability has to be negligible,

$$\mathsf{Pr}\left[x \leftarrow D, x' \leftarrow \mathcal{A}(1^\kappa, x) \,:\, x \neq x' \wedge H(x) = H(x')\right], \tag{2.5}$$

where D is the input domain of H and x is sampled uniformly at random from D. 2^{nd} pre-image resistance is therefore a weaker property than collision resistance and accordingly a hash function, which is collision resistant, is implicitly 2^{nd} pre-image resistant. When building a construction utilizing a hash function, it is advantageous to rely, if possible, only on the weaker 2^{nd} pre-image resistance, because it makes it harder for an adversary to attack the construction. For example there are no efficient attacks against the 2^{nd} pre-image resistance of the hash function MD5 [Riv92] known yet, even though attacks against its collision resistance were discovered in 2005 [WY05] and have been improved since then. Therefore, constructions, which only rely on its 2^{nd} pre-image resistance, are still secure. However, it is not advisable to use MD5 anymore and instead hash functions like the SHA2-family or SHA3 should be used.

A cryptographic hash function should also be pre-image resistant, which means that it is hard to get knowledge or make conclusions about the input, by only seeing the hash or output. This property is useful, when storing the hashes of passwords or making commitments, without revealing to what one has committed to. A hash function is pre-image resistant, if a PPT adversary has only a negligible chance to find an input x for a given output y. Formally the probability

$$\mathsf{Pr}\left[x \leftarrow D, y := H(x), x' \leftarrow \mathcal{A}(1^\kappa, y) \,:\, H(x') = y\right], \tag{2.6}$$

has to be negligible. If a hash function is 2^{nd} pre-image resistant, it is also pre-image resistant, because if it were possible to find a pre-image, it is also possible to compute a 2^{nd} pre-image with high probability.

Besides the commonly required properties of collision and (2^{nd}) pre-image resistance, a property especially interesting for an application in cryptocurrencies is the puzzle friendliness. The puzzle is usually to find an x, that produces a desired output y, when x is concatenated with a random number r and fed into a hash function.

$$H(r||x) = y, \tag{2.7}$$

where || denotes concatenation. The output y might not only be valid for a single value, but also for a set of values. The smaller this set is, the more difficult the puzzle. For example, if y has to have three leading zeros, the puzzle would be relatively easy and if y has to consist of all zeros, it would be maximum hard. A hash function is considered puzzle friendly if the probability to produce the correct output is the same for all possible inputs. Thus, no one can gain an advantage in solving the puzzle, by knowing that one input will be more likely the correct one.

2.2 Digital Signatures

Digital signatures should act like handwritten signatures in the digital world and provide additionally stronger security guarantees. Accordingly the signature should be bound to the signed data, everyone should be able to verify the correctness of the signature and no one, except oneself should be able to create one. Therefore, a valid signature ensures, that the signed data has not been modified, the data was sent by the signer and the signer cannot deny, that she did sent the data. To be able to create a signature, the signer has to generate a public-private keypair first. The signer can then sign data with her private key. The public key can be made public, so that every potential verifier is able to verify the signatures of the signer. A signature scheme for a message space M_k can be described formally with the following three algorithms:

$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(1^\kappa)$: The key generation algorithm takes the security parameter κ as input and outputs a private key sk to sign messages and a public key pk to verify signatures.

$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$: This method takes a message m and a private key sk as input and outputs a signature σ if $m \in M_k$ and $\perp$ otherwise.

$\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{pk}, m, \sigma)$: This algorithm takes a public key pk, a message m and a signature σ as input and outputs 0 if the verification failed or if the message $m \notin M_k$ and 1 otherwise.

Simply put, signatures can be created with a private key and verified with the corresponding public key. A digital signature scheme is generally required to be correct. It is correct, when

$$\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1, \tag{2.8}$$

for all possible keypairs generated from KGen, for all messages in the message space and all signatures outputted from Sign. To define the security of a digital signature scheme, there are several different notions. Existential Unforgeability under Chosen Message Attack (EUF-CMA) is a common one, which is also required for the DAPS scheme used in this thesis. A scheme is EUF-CMA secure, when an adversary has only a negligible chance to win the game shown in Figure 2.1 with an honest challenger. In short, even if an adversary knows the public key and a reasonable amount of valid message - signature pairs, she is only able to create herself a valid signature for a new message with a negligible probability. There is also a stronger version of this security notion titled Strong Existential Unforgeability under Chosen Message Attack (SUF-CMA). For a signature scheme to be SUF-CMA secure, an adversary is only allowed to win the game in Figure 2.2 with negligible probability. The difference to EUF-CMA is, that in SUF-CMA an adversary wins the game, when she is able to forge a signature for any message, new or old, as long as she does not reuse an old signature.

```
$\mathbf{Exp}_{\mathcal{A}}^{\mathsf{EUF-CMA}}(\kappa)$ :
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(1^\kappa)$
    $\mathcal{Q} \leftarrow \emptyset$
    $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}'(\mathsf{sk}, \cdot)}(\mathsf{pk})$
        where oracle $\mathsf{Sign}'$ on input $m$ :
            let $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$
            set $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{m\}$
            return $\sigma$
    return 1, if $\mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1 \land m^* \notin \mathcal{Q}$
    return 0
```

Figure 2.1: EUF-CMA security

In practical applications a signature is usually not created from the message itself, but form the hash of the message. The reason for this is, that on one hand hashing large messages is usually faster than signing them and on the other hand most signature schemes have a limited message space. The output space of the hash function, of course, has to match the message space of the signature scheme. If the employed hash function is collision resistant, it can be shown, that the influence of the hash function on the security of the signature scheme is negligible. Signature schemes used in

```
$\mathbf{Exp}_{\mathcal{A}}^{\mathsf{SUF-CMA}}(\kappa)$ :
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(1^{\kappa})$
    $\mathcal{Q} \leftarrow \emptyset$
    $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}'(\mathsf{sk},\cdot)}(\mathsf{pk})$
        where oracle $\mathsf{Sign}'$ on input $m$ :
            let $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$
            set $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(m, \sigma)\}$
            return $\sigma$
    return 1, if $\mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1 \wedge (m^*, \sigma^*) \notin \mathcal{Q}$
    return 0
```

Figure 2.2: SUF-CMA security

practice usually rely on a mathematical hard problem to achieve their security. Two of the most popular types utilized at the moment are RSA-based signature schemes, which rely on the difficulty to factor large prime numbers and discrete logarithm based signature schemes, which rely on the difficulty to find the discrete logarithm. Commonly used schemes include RSA-PSS [BR96] for the RSA-based ones and the Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] or Edwards-curve Digital Signature Algorithm (EdDSA) [Ber+12] for the discrete logarithm ones. Discrete logarithm based schemes are gradually replacing RSA-based schemes, because the former provide smaller signatures and a lower computational cost for the same security level.

2.3 Encryption Schemes

An encryption scheme transforms its input, also referred to as plaintext, into a so called ciphertext with the help of a key. The plaintext can be retrieved from the ciphertext by applying the reverse function, which again involves a key. The ciphertext should show no correlation with the input data and it should only be possible to retrieve the input data from the ciphertext with the knowledge of the correct decryption key. An encryption scheme can be described with the following three algorithms:

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\kappa)$: The key generation algorithm takes the security parameter κ as input and outputs a public key pk used to encrypt messages and a private key sk used to decrypt ciphertexts.

$ct \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$: This method takes a message m and a public key pk as input and outputs a ciphertext ct, which is dependent on the key.

$m \leftarrow \mathsf{Dec}(\mathsf{sk}, ct)$: This algorithm takes a private key sk and a ciphertext ct as input and outputs the decrypted ciphertext.

The above algorithms actually describe an asymmetric encryption scheme. Asymmetric means, that there is a separate key for encryption (pk) and decryption (sk). A symmetric encryption scheme can be described with these three algorithms as well, except that the encryption and decryption keys are the same. Accordingly, in the case of a symmetric scheme, party A uses a secret key sk to encrypt data and sends it to party B. Party B then uses the same key sk to decrypt the data. The fact that A and B have to use the same key makes it harder to share the key, because they have to find an other, secure way to exchange it. In an asymmetric scheme on the other hand each party has its own private and public key. Therefore, B publishes its public key, A encrypts the data with B's public key and sends it to B. B can now decrypt the data with its private key. This makes the key exchanging process easier, as the key needed for encryption is public knowledge. Either way, there is still a mechanism needed to prevent an adversary from tempering with the key during transport. An encryption scheme is generally required to be correct, and it is correct if

$$\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = m, \tag{2.9}$$

for all messages in the message space, all keys, which can be generated by the key generation algorithm KGen and all ciphertexts produced by Enc. An important security notion to define the security for encryption schemes is the Indistinguishability under Chosen Plaintext Attack (IND-CPA). The security game for IND-CPA is depicted in Figure 2.3 and for an IND-CPA secure encryption scheme an adversary can only do better than random guessing in this game with negligible probability. In essence, the adversary selects two messages, one of them is encrypted by the challenger and the adversary has to find out which one was encrypted. This is hard, if the ciphertext reveals nothing about the plaintext.

Some popular symmetric encryption schemes used in practice include AES [01b] and ChaCha [Ber08]. As the encryption schemes used in this thesis are asymmetric, the remainder of this section will focus on these schemes. Like digital signatures, most

```
$\mathbf{Exp}^{\mathsf{IND-CPA}}_{\mathcal{A},\Omega}(\kappa)$ :
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(1^\kappa)$
    $\mathsf{b} \leftarrow \{0, 1\}$
    $(m_0, m_1, \mathsf{state}_\mathcal{A}) \leftarrow \mathcal{A}(\mathsf{pk})$
    if $m_0 \notin \mathcal{M} \vee m_1 \notin \mathcal{M}$, let $\mathrm{C} \leftarrow \perp$
    else, let $\mathrm{C}^* \leftarrow \mathsf{Enc}(\mathsf{pk}, m_b)$
    $\mathsf{b}^* \leftarrow \mathcal{A}(\mathrm{C}^*, \mathsf{state}_\mathcal{A})$
    return 1, if $\mathsf{b}^* = \mathsf{b}$
    return 0
```

Figure 2.3: IND-CPA security

asymmetric encryption schemes rely on a mathematical hard problem to achieve their security. Again there are two popular groups among them, namely schemes based on the RSA and schemes based on the discrete logarithm problem. RSA [RSA78] would be an example for former ones and ElGamal [Elg85] an example for the latter ones. A useful property, which some asymmetric encryption schemes hold is homomorphism. With homomorphic encryption schemes it is possible to operate directly on encrypted data. For example multiplying two ciphertexts results in the product of the two corresponding plaintexts upon decryption. However, the number of possible operations is usually limited. The aforementioned schemes RSA and ElGamal are multiplicative homomorphic, as long as they are not padded. Homomorphic encryption, in the form of the ElGamal cryptosystem, is also leveraged in the DAPS scheme covered in Section 4.4.

2.4 Discrete Logarithm Based Cryptography

Many cryptographic algorithms are based on a mathematical hard problem to assure their security. The discrete logarithm problem (DLP) is one of these problems. The hardness of calculating the discrete logarithm in certain groups is utilized in schemes, like for example the signature algorithms (EC)DSA [98] [JMV01] and EdDSA [Ber+12], the encryption scheme ElGamal [Elg85] and the Diffie-Hellman key exchange [DH76]. The DLP can be described as follows: given a finite abelian group

$\mathbb{G}$ with a generator g and of prime order $q \in \mathbb{P}$, find an integer x to satisfy the equation

$$g^x = y, \tag{2.10}$$

where y is a random group element of $\mathbb{G}$. Therefore, solve $x = \mathrm{dlog}_g(y)$. For certain groups this problem is believed to be hard. Groups used in cryptographic schemes include multiplicative groups of a finite field or elliptic curve groups. A popular signature scheme in the discrete logarithm setting, which uses an elliptic curve group is ECDSA (Elliptic Curve Digital Signature Algorithm). It finds various applications, which include cryptocurrencies and will therefore be discussed here. The three algorithms describing ECDSA are depicted in Figure 2.4, where P_x represents the

```
$\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{KGen}_{ECDSA}(1^\kappa)$ :
    Let $\mathbb{G}$ be an elliptic curve group of prime order $q$ and with generator $g$ :
    $x \xleftarrow{R} \mathbb{Z}_q^*$
    $\mathsf{sk} = x$
    $\mathsf{pk} = g^x$
$\sigma \leftarrow \mathsf{Sign}_{ECDSA}(\mathsf{sk}, m)$ : parse sk as $x$
1 :  $k \xleftarrow{R} \mathbb{Z}_q^*$
2 :  $R \leftarrow g^k$
3 :  $r \leftarrow R_x \mod q$, if $r = 0$ goto step 1
4 :  $s \leftarrow k^{-1}(H(m) + rx) \mod q$ and if $s = 0$ goto step 1
5 :  $\sigma \leftarrow (r, s)$
$\{0, 1\} \leftarrow \mathsf{Verify}_{ECDSA}(\mathsf{pk}, m, \sigma)$ : parse $\sigma$ as $(r, s)$
1 :  if $r = 0 \lor s = 0$ return 0
2 :  $z \leftarrow H(m) \mod q$
3 :  $w \leftarrow s^{-1} \mod q$
4 :  $u_1 \leftarrow zw \mod q$
5 :  $u_2 \leftarrow rw \mod q$
6 :  $R \leftarrow g^{u_1} \cdot \mathsf{pk}^{u_2}$
7 :  if $R_x = r \mod q$ return 1 and return 0 otherwise
```

Figure 2.4: ECDSA scheme

x-coordinate of a curve point P and H is a hash function, which maps exactly to the order of the group. For the group operations a multiplicative notion has been used and will be used henceforth in this thesis. It is believed that ECDSA is EUF-CMA secure, but its security cannot be proved in the standard or random oracle model. However, EUF-CMA security can be proved in the generic group model due to [Bro05] and through recent work from Fersch et al. [FKP16] in the bijective random oracle model as well.

In addition to the DLP there are two other problems related to it, which are also important for cryptographic schemes. These are the Computational and the Decision Diffie-Hellman problem (CDH and DDH). The CDH is to find a group element h, such that $h = g^{xy}$ for a given $a = g^x$ and $b = g^y$ as well as a finite abelian group $\mathbb{G}$ with a generator g and of prime order q. Formally, the game to solve the CDH is depicted in Figure 2.5. It can be shown that the CDH is not harder than the DLP

```
$\mathbf{Exp}_{\mathcal{A}}^{\mathsf{CDH}}$ :
    $g \leftarrow \mathbb{G}$
    $x, y \overset{R}{\longleftarrow} \mathbb{Z}_q^*$
    $h \leftarrow \mathcal{A}(g, a \leftarrow g^x, b \leftarrow g^y)$
    return $1$, if $h = g^{xy}$
    return $0$
```

Figure 2.5: Computational Diffie-Hellman problem

and indeed, if the discrete logarithm can be solved for either x or y, the CDH can be solved trivially. For the DDH again a finite abelian group $\mathbb{G}$ with a generator g and of prime order q is given with three elements $a = g^x$, $b = g^y$ and $c = g^z$. The goal is to find out if either z was chosen uniform at random or set to $z = x \cdot y$. The game for the DDH is presented in Figure 2.6. The DDH can be reduced to the CDH and thus if it is possible to solve the CDH, the decision in the DDH is trivial. Therefore, the DDH is not harder than the CDH, however there are groups in which solving the DDH takes polynomial time, but solving the CDH takes sub-exponential time.

An asymmetric encryption scheme in the discrete logarithm setting is the ElGamal encryption [Elg85] and its algorithms are shown in Figure 2.7. The security of the scheme relies on the hardness of the DDH in the group it is instantiated in. In other words, ElGamal encryption is IND-CPA secure in any group, where the DDH

```
$\mathbf{Exp}^{\mathsf{DDH}}_{\mathcal{A}}$ :
    $d \leftarrow \{0, 1\}$
    $g \leftarrow \mathbb{G}$
    $x, y \xleftarrow{R} \mathbb{Z}^*_q$
    if $d = 1$ then $z \xleftarrow{R} \mathbb{Z}^*_q$
    if $d = 0$ then $z \leftarrow x \cdot y$
    $d' \leftarrow \mathcal{A}(g, a \leftarrow g^x, b \leftarrow g^y, c \leftarrow g^z)$
    return $1$, if $d' = d$
    return $0$
```

Figure 2.6: Decision Diffie-Hellman problem

assumption holds and can therefore also be instantiated in for example elliptic curve groups. The DDH, however, is for example easy in symmetric pairing based groups. Pairings find on one hand use in cryptanalysis to reduce hard problems, but on the other hand they are also used in cryptographic schemes and in particular in identity and attribute based encryption. Following the work from [MJ16] a pairing or bilinear map for groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ of order q can be defined as

$$\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_3. \tag{2.11}$$

Additionally a pairing has to be bilinear, non-degenerate and efficiently computable. It is bilinear, if for $u \in \mathbb{G}_1$, $v \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}$

$$e(u^a, v^b) = e(u, v)^{ab}. \tag{2.12}$$

It is non-degenerate if $e(g_1, g_2) \neq 1$, where g_1 and g_2 are generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. If it is the case that $\mathbb{G}_1 = \mathbb{G}_2$, then the DDH is easy, because $e(g^a, g^b) = e(g, g^{ab})$ and thus it can be verified if $e(g^x, g^y) = e(g, c)$ or not (see Figure 2.6).

Even though ElGamal is not IND-CPA secure in groups in which the DDH is easy, it may also be instantiated in groups in which the Decision Linear Problem (DLIN) is hard. In this case a modified version of ElGamal, titled Linear encryption [BBS04], has to be used. The DLIN problem is hard, if the probability to do better than random guessing in the game depicted in Figure 2.8 is negligible. In other words, given u, v, h, u^a, u^b and u^c, it is hard to distinguish if c was sampled uniform at

```
$\underline{\{\mathsf{sk},\mathsf{pk}\} \leftarrow \mathsf{KGen}_{ElGamal}(1^\kappa)}$ :
    Let $\mathbb{G}$ be a group of prime order $q$, where DDH is hard and with generator $g$ :
    $x \xleftarrow{R} \mathbb{Z}_q^*$
    $\mathsf{sk} = x$
    $\mathsf{pk} = g^x$
$\underline{ct \leftarrow \mathsf{Enc}_{ElGamal}(\mathsf{pk}, m)}$ :
1 :  $r \xleftarrow{R} \mathbb{Z}_q^*$
2 :  $C_1 = g^r \mod q$
3 :  $C_2 = m \cdot \mathsf{pk}^r \mod q$
4 :  $ct = (C_1, C_2)$
$\underline{pt \leftarrow \mathsf{Dec}_{ElGamal}(\mathsf{sk}, ct)}$ : parse $ct$ as $(C_1, C_2)$
    $pt = C_2 \cdot C_1^{-\mathsf{sk}}$
```

Figure 2.7: ElGamal encryption scheme

```
$\mathbf{Exp}_{\mathcal{A}}^{\mathsf{DLIN}}$ :
    $d \leftarrow \{0,1\}$
    $u, v, h \leftarrow \mathbb{G}$
    $a, b \xleftarrow{R} \mathbb{Z}_q^*$
    if $d = 1$ then $c \xleftarrow{R} \mathbb{Z}_q^*$
    if $d = 0$ then $c \leftarrow a + b$
    $d' \leftarrow \mathcal{A}(u, v, h, u^a, v^b, h^c)$
    return 1, if $d' = d$
    return 0
```

Figure 2.8: Decision Linear problem

random or $c = a + b$. The DLIN assumption is believed to hold in groups, where the DDH assumption does not hold. Therefore, it is possible to use linear ElGamal in conjunction with some pairing-based elliptic curve or Schnorr groups. The traditional ElGamal is multiplicative and linear ElGamal additive homomorphic.

2.5 Zero-Knowledge Proofs

With a zero-knowledge proof (ZKP), first introduced by [GMR85], a prover can convince a verifier with a certain probability that a fact is true. The fact might be the solution to an NP-hard problem. A ZKP protocol has to be complete, sound and zero-knowledge. It is complete, if a prover, that knows the fact and follows the protocol, can convince the verifier with probability 1. It is sound, if a prover, that does not know the fact, can only convince the verifier with negligible probability. The zero-knowledge property can be described by the similarity between the set of all valid transcripts of protocol runs $\mathcal{V}$ and the set of all simulations $\mathcal{S}$. If the sets are equal, the ZKP is perfect zero-knowledge and if they are indistinguishable by a computationally bound adversary, it is said to be computational zero-knowledge. However, these ZKP are in general not efficient enough for a practical use. Instead proofs of knowledge, with weaker requirements, like Σ-protocols are used.

A Σ-protocol is a three move protocol, where the prover starts with a commitment T, then the verifier sends a challenge c and the prover answers with a response r. In Σ-protocols the verifier has to be honest and must follow the protocol correctly and thus they are honest verifier zero-knowledge protocols. The protocol is usually executed with an $\mathcal{NP}$ language $\mathcal{L}$ and the prover has to prove, that he knows the witness w for a statement $X \in \mathcal{L}$. Σ-protocols are required to be complete, special sound and honest verifier zero-knowledge. A protocol is complete, if like above the verifier always accepts, if the prover and verifier follow the protocol correctly. It is special sound, if there exists a PPT extractor for two accepting transcripts (T, c, r) and (T, c', r'), which will always compute the witness w from these transcripts. It is honest verifier zero-knowledge, if again, like above the sets $\mathcal{V}$ and $\mathcal{S}$ are indistinguishable, but the verifier has to act honestly.

Schnorr's identification protocol [Sch90] as shown in Figure 2.9 is such a Σ-protocol. In a finite abelian group of prime order q and with generator g, the prover knows the discrete logarithm x of a $A = g^x$ and wants to convince the verifier accordingly.

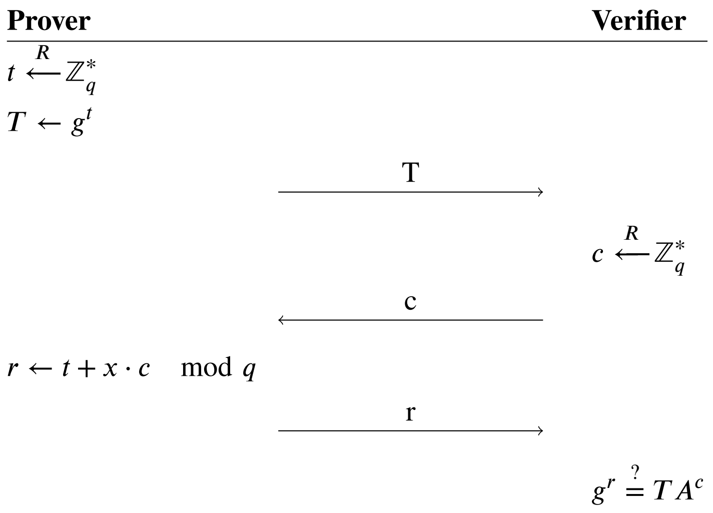

Prover
Verifier
$t \xleftarrow{R} \mathbb{Z}_q^*$
$T \leftarrow g^t$
T
$c \xleftarrow{R} \mathbb{Z}_q^*$
c
$r \leftarrow t + x \cdot c \mod q$
r
$g^r \overset{?}{=} TA^c$

Figure 2.9: Schnorr's identification protocol

Therefore, the prover samples t uniform at random from $\mathbb{Z}_q$ and sends $T = g^t$ as the commitment to the verifier. The verifier samples c, again uniform at random from $\mathbb{Z}_q$, as the challenge and sends it to the prover. Then, the prover calculates $r = t + x \cdot c \mod q$ and sends r as the response to the verifier. The verifier calculates g^r and $T \cdot A^c$ and, if both values are equal, the verifier can be sure, with a high probability, that the prover knows the discrete logarithm of A. This protocol can be extended, so that it guarantees, that the prover knows x and given $A = g^x$ and $B = h^x$, that (g, h, A, B) form a DDH-tuple. This can be achieved by using a commitment value for each publicly known value (here A and B). The extended version of this protocol is depicted in Figure 2.10.

Additionally, a Σ-protocol can be turned into a non-interactive zero-knowledge proof (NIZK), by applying for example the Fiat-Shamir transform [FS87]. Here the challenge c of the verifier is basically replaced by a hash function, modeled as a random oracle. This bypasses the requirement for an honest verifier and additionally everyone is able to verify the proof. A NIZK proof system can be described with the following three algorithms:

$\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$: The setup takes a security parameter κ as input and generates a common reference string crs.

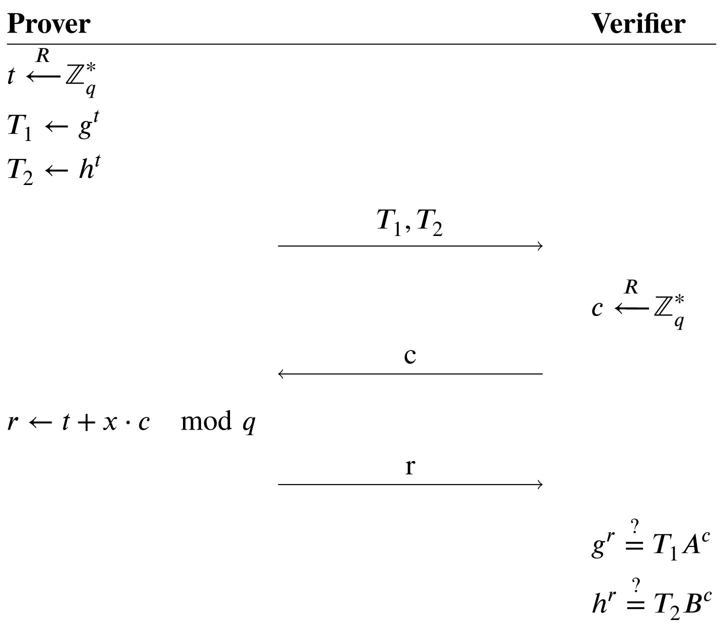

Prover
Verifier
$t \xleftarrow{R} \mathbb{Z}_q^*$
$T_1 \leftarrow g^t$
$T_2 \leftarrow h^t$
T_1, T_2
$c \xleftarrow{R} \mathbb{Z}_q^*$
c
$r \leftarrow t + x \cdot c \mod q$
r
$g^r \stackrel{?}{=} T_1 A^c$
$h^r \stackrel{?}{=} T_2 B^c$

Figure 2.10: Σ-protocol to show that (g, h, A, B) forms a DDH-tuple

$\pi \leftarrow \mathsf{Proof}(\mathsf{crs}, S, w)$: This algorithm takes a common reference string crs, a statement S and a witness w as input and outputs a proof π.

$\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{crs}, S, \pi)$: This algorithm takes a common reference string crs, a statement S and a proof π as input and outputs 1 if the proof was correct and 0 otherwise.

Applying the Fiat-Shamir transform to Schnorr's identification protocol from Figure 2.9 leads to the NIZK proof system in Figure 2.11. Like before, in a finite abelian group $\mathbb{G}$ of prime order q and with generator g, the prover wants to prove, that she knows the witness $w = x$ for a statement S, where S is $\exists x : A = g^x$. For the prove, at first the commitment T is calculated and then hashed with the statement A to create the challenge. Thereafter the response r is calculated and the proof π is outputted as a tuple consisting of the commitment and the response. To verify the proof, the challenge is again created by hashing the commitment and the statement and it has to be checked if $g^r \stackrel{?}{=} T \cdot A^c$.

```
$\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$ :
    fix a hash function $H$,
    which maps exactly to $\mathbb{Z}_q^*$
    $\mathsf{crs} \leftarrow (\kappa, H)$
```

```
$\pi \leftarrow \mathsf{Proof}(\mathsf{crs}, A, x)$ :
1 :   $t \xleftarrow{R} \mathbb{Z}_q^*$
2 :   $T \leftarrow g^t$
3 :   $c \leftarrow H(T, g, A)$
4 :   $r \leftarrow t + x \cdot c$
5 :   $\pi \leftarrow (r, T)$
```

```
$\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{crs}, A, \pi)$ :  Parse $\pi$ as $(r, T)$
1 :   $c \leftarrow H(T, g, A)$
2 :   if $g^r = T \cdot A^c$ return 1 and return 0 otherwise
```

Figure 2.11: NIZK proof system for the Fiat-Shamir transform of Schnorr's identification protocol

2.6 Secret Sharing

Secret sharing is used to share a secret s with a set $\mathcal{P}$ of n participants. All qualifying subsets of $\mathcal{P}$, which should be able to recover s, are contained within the monotone access structure Γ. Therefore, the participants in each subset of Γ are able to recombine their shares and retrieve the secret. However, all other sets, not contained in Γ, should not be able to do so and ideally learn nothing about the secret. A secret sharing scheme can be described with the following two algorithms

$\mathsf{Share}(s, \Gamma)$: This algorithm takes the secret s and an access structure Γ as input and outputs a share s_A for each participant $A \in \mathcal{P}$.

$\mathsf{Recombine}(H_B)$: This algorithm takes a set of shares H_B as input, where $B \subset \mathcal{P}$. It outputs the secret if $B \in \Gamma$ or $\perp$ otherwise.

Desirable properties of a secret sharing scheme are perfect and ideal. A secret sharing scheme is perfect, if each subset of participants $B \notin \Gamma$, has no additional information about the secret in comparison to someone who has no shares at all and therefore they cannot do better than guessing all possible values for the secret. A secret sharing scheme is ideal, if it is perfect and the size of the shares equals the size of the secret. A well known ideal secret sharing scheme is Shamir's secret sharing [Sha79]. It is a (k, n)-threshold secret sharing scheme, which means that at least k out of n shares are needed to reconstruct the secret. To create the shares a random polynomial $f(X)$

of degree $k-1$ is chosen over a prime field $\mathbb{Z}_q$ and the secret s is set as the constant term of this polynomial

$$f(X) = s + f_1 \cdot X + \cdots + f_{k-1} \cdot X^{k-1}. \tag{2.13}$$

A share s_A is then an evaluation of this polynomial for an arbitrary, but distinct $x_A \in \mathbb{Z}_q \setminus \{0\}$ for every participant A. To recombine the shares of a qualifying set of participants $B \in \Gamma$, Lagrange interpolation can be used as

$$s \leftarrow \sum_{x_i \in H_B} \left(s_i \cdot \prod_{x_j \in H_B, x_j \neq x_i} \frac{-x_j}{x_i - x_j} \right). \tag{2.14}$$

In instances, where a participant cannot be trusted to output a valid share of the secret, a verifiable secret sharing scheme can be used. In particular Shamir's secret sharing scheme can be made verifiable due to a technique from [Fel87]. This comes at the cost of an reduced security from information-theoretic to computational, which is however still secure against any PPT adversaries. To achieve verifiability a one-way homomorphism is used. Therefore, the sequence $(g^s, g^{f_1}, \ldots, g^{f_{k-1}})$ has to be published for a group $\mathbb{G}$ of prime order q and with generator g and where s is the secret and $f_1 \ldots f_{k-1}$ are the coefficients of the polynomial in Equation 2.13. A share s_A can then be verified by checking

$$g^{s_A} \overset{?}{=} \prod_{j=0}^{k-1} \left(g^{f_j}\right)^{x_A^j}, \tag{2.15}$$

where $g^{f_0} = g^s$.

3 Cryptocurrencies

In this chapter an overview on cryptocurrencies is presented, starting with the basic working principles of a cryptocurrency, and followed by a brief overview and comparison of three of the most popular ones and the differences between them. The cryptocurrency mainly used in this thesis, Bitcoin, will be explained in more detail at the end of this chapter. The explanations follow mostly the work from [Nar+16].

3.1 Cryptocurrency Basics

In this section a brief explanation on how a cryptocurrency works is presented. This basic functionality is roughly the same for all cryptocurrencies, but there are slight deviations. As the name implies a cryptocurrency relies on cryptography to ensure its functionality and security.

A user or identity is usually represented by the public key of the underlying digital signature scheme as shown in Figure 3.1. Therefore, all signed data, which verifies under a certain public key, can be associated with an identity.

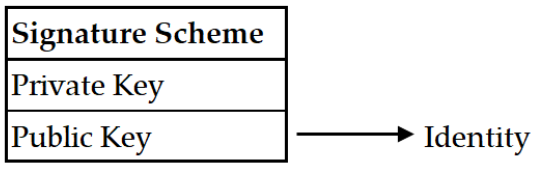


Figure 3.1: The public key of a signature scheme serves usually as an identity in a cryptocurrency

This makes a decentralized management of the participants possible and thus provides some degree of anonymity for them, because there is no central authority, where one might have to register. However, when using the same public key several times, all transactions made by this key or identity can be tracked and this might ultimately reveal the user behind it. Either way, it is easy to create a new identity and one can even use a new one for each transaction. Specifically a private-public keypair has to be used only twice, once to receive a transaction and once to spend it. Doing so is also encouraged from a security perspective, because it mitigates potential attacks, with which the private key can be recovered from one or multiple signatures. Furthermore, the public key might be transformed into a so called address with the help of a hash function as indicated in Figure 3.2. This allows the public key to stay hidden and secure, even when a weakness in the digital signature scheme is discovered.

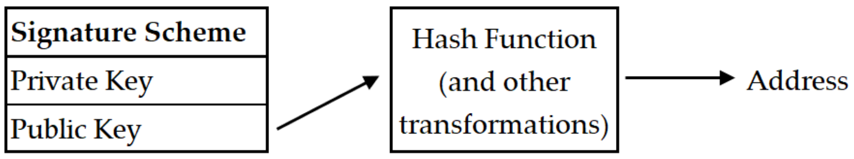
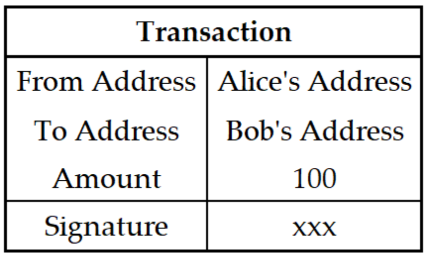


Figure 3.2: Derivation of an address from a public key

To transfer the coins of the cryptocurrency from one address to an other one, a so called transaction is created. The transaction basically states from and to which address the coins are transferred, as well as the amount of coins and some metadata. A simplified version of such a transaction is shown in Figure 3.3. In this example Alice transfers 100 of her coins from her address to Bob's. To ensure that the transaction cannot be altered and that Alice agrees to it, the whole transaction is signed with Alice's private key.



Transaction	
From Address	Alice's Address
To Address	Bob's Address
Amount	100
Signature	xxx

Figure 3.3: A simplified version of a transaction to transfer cryptographic coins from Alice to Bob

Coins, which are transferred from an address increase the balance within the transaction and are called transaction inputs. Coins, which are transferred to an address decrease the balance within the transaction and are called transaction outputs. A transaction might have one or multiple inputs and one or multiple outputs, as long as the balance remains zero. In Figure 3.4 such a transaction with multiple in- and outputs is shown. Because Alice does not have enough funds in one of here addresses, she combines three of them to be able to pay Bob and she transfers the surplus of coins back to one of her addresses. Each of her inputs has to be signed with the corresponding private key. The signature usually covers the transaction metadata and all outputs, but only the one input, the signature belongs to.

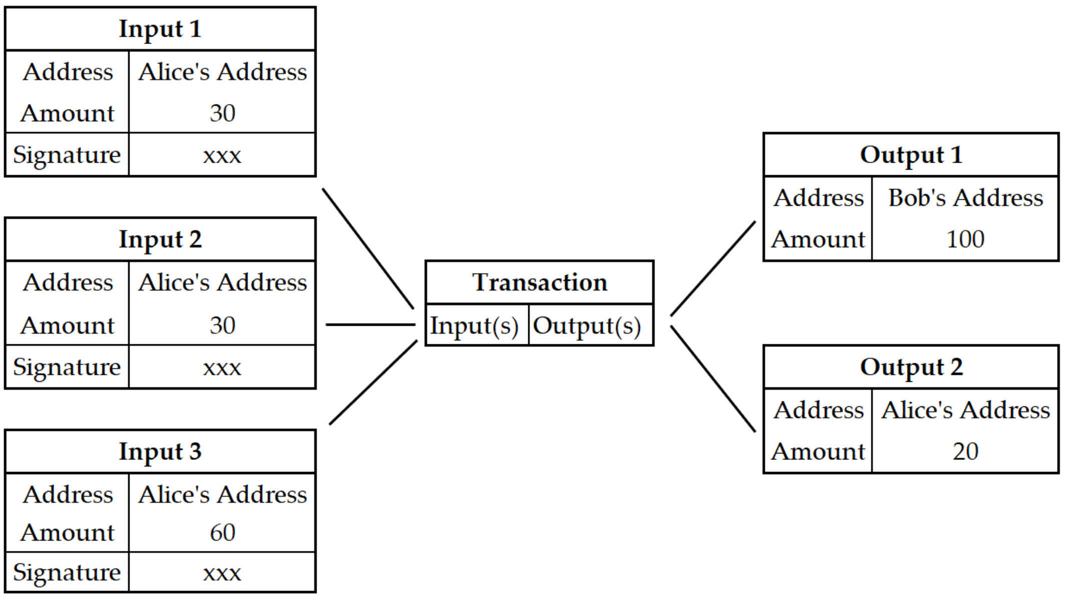


Figure 3.4: A transaction from Alice to Bob with an explicit display of the inputs and outputs.

To be able to verify the origin of the inputs, every input has to use a transaction output from a previous transaction. Thus, all coins, which one might own in a cryptocurrency are actually unspent transaction outputs. An unspent transaction output can only be used in an input, if a signature is provided, which verifies under the public key of the address specified in this output. Most cryptocurrencies additionally support other ways to make an output spendable and therefore each transaction output actually contains a locking script. To be able to spend these outputs, data has to be provided in the input, which makes this script evaluate to true. In the most basic use case the script in the output contains the address and an operand to verify the signature of the transaction with the public key from which the address is derived. Thus, the input has to provide the signature and a public key, which verifies the signature and which can be transformed into the address. A more detailed explanation of the scripting system in Bitcoin will be provided in Section 3.3. A transaction similar to the one in Figure 3.5 is usually used in current cryptocurrencies. Here the inputs only contain a reference to the output of a previous transaction as well as the data to fulfill the locking script. Data, like the amount of coins and the locking script are fetched from the reference.

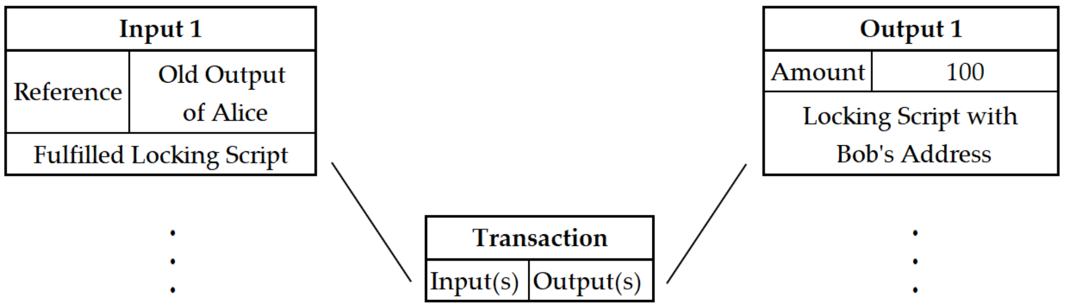


Figure 3.5: An example transaction, which references old outputs as inputs and creates new outputs

A transaction output can be referenced by the transaction id and the position within all outputs of this transaction. The transaction id is simply the hash of the whole transaction. This has the advantage, that each transaction has a unique id and that this id can be derived in a straightforward manner. Additionally, all related transactions are forming a linked list with hash pointers. A hash pointer makes it possible to detect changes in all previous elements in the linked list or transactions in the case of cryptocurrencies. Therefore, it is sufficient to add only the id of the last preceding transaction in the input of a new transaction to cover all related transactions.

So far users or identities can be identified by an address, where a user might control multiple addresses and coins can be transferred with transactions between these addresses. The flow of the coins can be traced back to the very first transaction, as each input in a transaction is an output of a previous transaction. Additionally, the signature in each input ensures that the input and the transaction as a whole cannot be altered afterwards and that the signer approves it. However, there is still a mechanism needed to store and manage transactions in a way, that no one can tamper with it. In a conventional payment system there is a trusted third party, like a bank, that takes care, that everything is save and sound. In decentralized cryptocurrencies, however, it is not possible to trust or even know every participant in the network. Therefore, a mechanism is needed to find consensus on which transactions are valid and which are not. To allow everyone to participate in the consensus finding, all transactions are stored publicly visible and everyone can verify their correctness. Additionally, everyone can maintain a copy of this chains of transactions. When a new transaction should be added to the chain, it has to be broadcasted to the network and the participants of the network have to find consensus on, if the transaction should be added or not. This consensus finding process has to take care of that no faulty transactions can be added, every valid transaction will be added, no one can spend the same transaction output multiple times (double spending) and the process has to terminate in finite time. In the overview of the next section the consensus finding algorithms used in Bitcoin, Ehereum and Ripple will be discussed.

To make the consensus finding simpler, multiple transactions are usually grouped in so called blocks. All blocks are connected in a linked list with hash pointers, called blockchain. The blockchain acts as a public ledger, which stores all blocks and therefore transactions in chronological order. A block will only be added to the blockchain, if all transactions within that block are valid and if the participants of the network consent, that it will be the next block. During the consensus finding a branching path may occur in the blockchain, due to, for instance, latency issues, but only the longest path will be considered as the valid one.

In Figure 3.6 an example for a part of a blockchain is shown. Each block contains the hash of the previous block, which serves simultaneously as the ID. Like for the transactions, when hash pointers are used, any changes in any of the previous blocks can be detected. A new block contains all transactions, which should be added to the blockchain and a hash of these transactions, to ensure their integrity. Furthermore the block contains a timestamp, data for consensus finding and other metadata, which enables an efficient management of the blockchain.

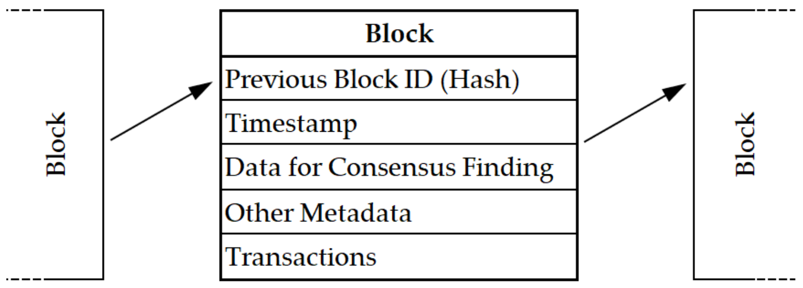


Figure 3.6: Example block within a blockchain

3.2 Overview

In this section a short overview of the cryptocurrencies Bitcoin [Nak09], Ethereum [But14] and Ripple [SYB14] is given and the differences between them are pointed out. These cryptocurrencies have been chosen, because they are among the most popular ones and additionally offer an option for a fast processing of payments.

The main differences between cryptocurrencies are usually the used cryptographic algorithms, the scripting language and the consensus finding process. In Bitcoin, Ethereum and Ripple the same digital signature scheme, ECDSA [JMV01], is used and Ripple has additionally an option to use EdDSA [Ber+12]. In contrast, the employed hash function differs between all of them. SHA-256 [01a] is mainly used in Bitcoin, Keccak-256 [Ber+11] in Ethereum and SHA-512Half, which is similar to SHA-512/256, in Ripple. Either way, the choice of primitives used in these three cryptocurrencies is deemed secure at the moment.

As for the scripting language, both, Ethereum and Bitcoin have a stack-based language. Ethereum offers a more extensive one, which is Turing complete [But14]. In essence, a language is Turing complete, if it can be used to solve any computational problem. The Turing completeness, however, comes with the downside, that scripts with an infinite run time can be build and the execution of such scripts would impact the availability of the network. To limit the amount of time and size of a script, Ethereum uses a unit called Gas, which has to be used for every executed operation in a script. The fee payed for the transaction then covers the Gas cost. The amount of Gas needed for a script is always the same, but the fee, which needs to be payed, varies according to the current value of the cryptocurrency. This system makes various applications on the Ethereum blockchain possible and even games have been developed for it. The scripting in Bitcoin on the other hand is not Turing complete as loop-instructions are not supported. This narrows down the number of possible scripts, but it keeps them simple and eliminates the occurrence of infinite loops. Nevertheless, it is still possible to use conditional statements for branching execution paths. Ripple does not directly support scripting, but there are several transaction types, with different predefined scripts.

The consensus finding process is quite similar for Bitcoin and Ethereum, but Ripple uses a different approach here. To recapitulate, when Alice wants to send coins to Bob, she creates a transaction, signs it and then broadcasts it to the network of the cryptocurrency. The transactions are gathered in blocks and the consensus process decides whether the block will be added to the blockchain or not. In Bitcoin and Ethereum a Proof of Work (PoW) system is employed. In this system, so called miners are the ones who collect transactions and put them into blocks. To be able to append a block to the blockchain, a miner has to solve a computationally difficult puzzle. Despite being difficult to solve, the solution of the puzzle is easily verifiable by everyone. Once the miner has solved the puzzle, she can broadcast the block to the network and the block will get added to the blockchain, if everything is sound. To reward the miners for the computational work, each new block generates new coins, which the miner can transfer to her address. Because everyone can participate in the mining process, the computational power and therefore the rate at which the blocks are created can fluctuate over time. To keep this rate constant, the difficulty of the puzzle is adjusted every time a specific amount of blocks has been mined and appended to the blockchain. Due to latency issues across the internet, different parts of the network may append a different block to the last common blockchain. In the long term however, the network will only consider the branch with the highest overall difficulty as valid. Consequently, one would need more than 50 % of the computational power of the whole network to gain control over the blockchain. This would make it possible to, for example deny specific transactions or launch a double spending attack. However, because a cryptocurrency inherently posses a value in itself, the participants are incentivized to act honestly. Otherwise, the value of the cryptocurrency would diminish with the trust in its security. Additionally, because of the high computational and therefore monetary cost to achieve a majority of the computational power in a network, acting fraudulent might not be profitable.

Ripple on the other hand does not rely on computationally hard puzzles for consensus finding, but instead utilizes the self-titled Ripple Protocol, which is a Byzantine consensus protocol. In this protocol every node in the network can vote on the transactions, which should be added to the blockchain next. The current state of the blockchain is called last-closed ledger and is the same for all nodes in the network. Each node individually has an open ledger to which all new and valid transactions are applied. Additionally, each node maintains a unique node list (UNL), which is different for every node. When executing the consensus algorithm, the candidates for the new transactions, which should be applied to the last closed ledger are exchanged with each node in the UNL. Then, the nodes vote on the veracity of all new transactions. Transactions with at least 80% of "yes" votes proceed, while the others are discarded or added to the candidate set of the next open ledger. For this algorithm to succeed, there have to be less than 20% of faulty nodes in the network and cliques (set of nodes, in which each node is connected to every other node in the set) have to have a connection to other nodes, which consist of at least 20 % of the whole network. This system enables Ripple to be one of the fastest cryptocurrencies in terms of transaction speed at the moment, but because of the necessary management of the UNLs, it is arguably less decentralized.

To improve the transaction speed for Bitcoin and Ethereum so called payment channels [PD16] can be used on top of the respective network. A payment channel is established between two participants and enables them to exchange coins between each other, without an interaction with the blockchain. During the lifetime of a channel only two transaction have to be added to the blockchain, one to open it and and one to close it and settle the balance. When opening a channel, both parties place funds in it and are then able to use these funds, once the initial transaction is added to the blockchain. Within the channel an arbitrary amount of transactions can be performed, but the participants cannot spend more coins as they have transferred to the channel in the initial transaction. Even so the transactions within a payment channel happen off-chain, the participants do not have to trust each other, because both participants can close the channel at any given time, without the involvement of the other one. Payment channels can also be expanded to a network of channels, so that the problem can be avoided, that every participant has to create a separate channel with everyone else on the blockchain. When each participant has at least one open payment channel a path can be routed through several intermediaries to make a payment to almost every other participant in the network. Again, this can be achieved in a way, that no trust has to be placed in any of the participants.

3.3 Bitcoin

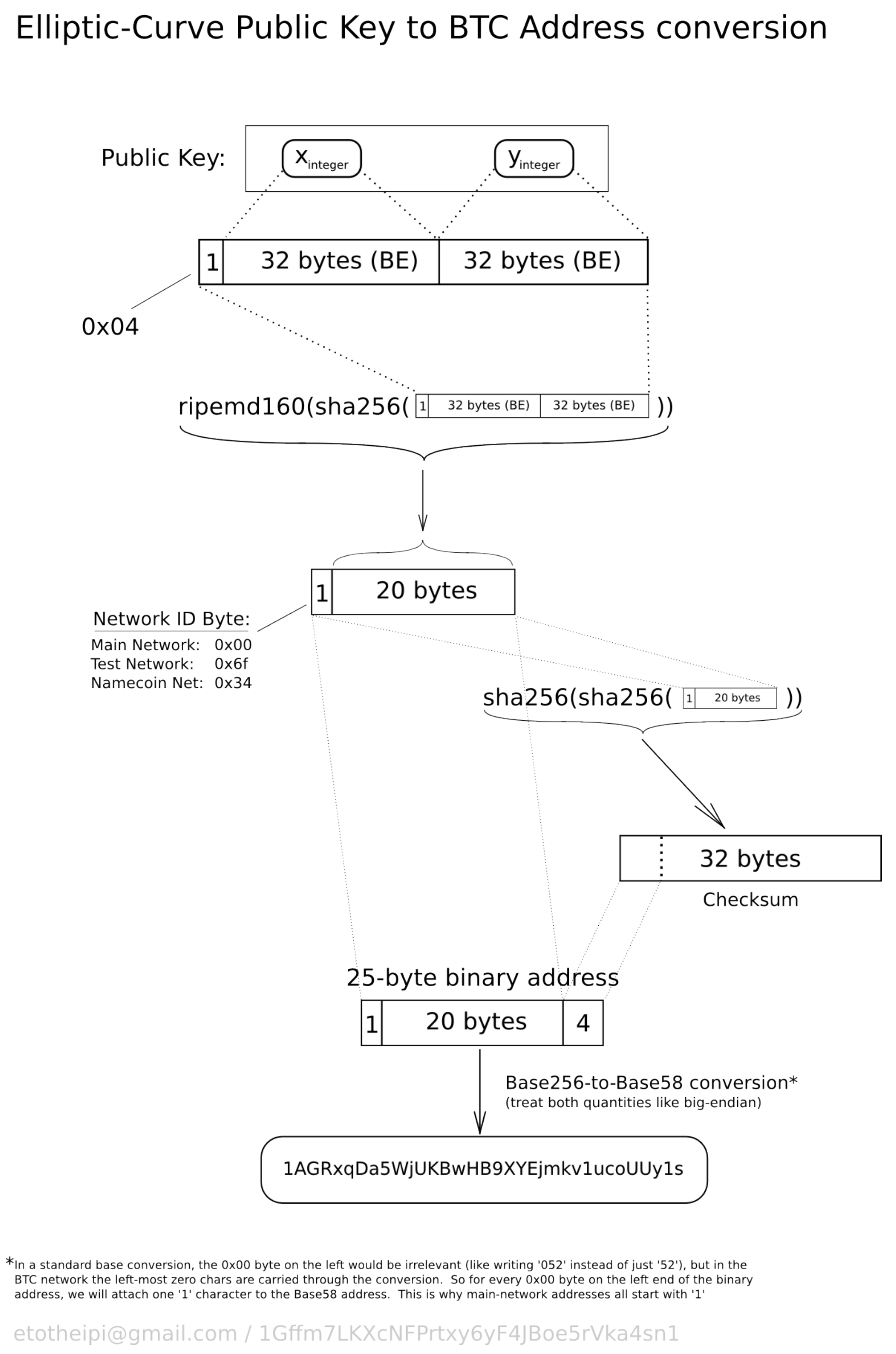
In this chapter a brief overview of Bitcoin will be presented, with some details, which are relevant for the application shown in Chapter 5, including the scripting language used in Bitcoin. Besides the work from [Nak09] and [Nar+16], additional information can be found in the Developer Guide[1] and the Bitcoin Wiki[2].

[1]https://bitcoin.org/en/developer-guide/
[2]https://en.bitcoin.it/

The digital signature scheme used in Bitcoin is ECDSA. Therefore, the address of a user or identity is derived from the public key of an ECDSA keypair. The process of deriving such an address is depicted in Figure 3.7. To obtain the address the uncompressed public key is first hashed with SHA-256 and the resulting output is hashed again with RIPEMD-160 [DBP96]. The RIPEMD-160 hash is then extended with a network ID, which specifies if the address belongs to the mainnet, the testnet and so on. This extended hash is then hashed twice with SHA-256 to produce a checksum and the first 4 bytes of this checksum are appended to the extended RIPEMD-160 hash to form the address. This address can then be used to receive and make payments. To increase the readability, the address is usually displayed as a base58 encoded string.

The structure of a transaction in Bitcoin is shown in Figure 3.8. Its version number specifies, which rules apply to the transaction or in other words, how to handle the transaction. The flag is an optional field, which signals if witness data is present or not. The In- and Out- counter are positive integer representing the number of inputs and outputs respectively. Witness is an optional field and if it is present, a witness has to be provided for each input. They are required for a protocol extension, called Segregated Witness (SegWit), which counters transaction malleability and is itself a requirement for the Lightning payment channel network. The locktime restricts a transaction to be added to the blockchain, if it is non-zero. It is either represented as an Unix time timestamp or a block height and a transaction might only be added to the blockchain once the respective block or point in time has passed. There are one or multiple outputs in a transaction, starting with Output0. An output contains a value and a locking script. The value represents the number of Satoshis hold by the output, where 10^8 Satoshi equate to 1 Bitcoin. The locking script makes the output only spendable, if a valid signature script is provided.

A transaction contains one or multiple inputs, starting with Input0. An input contains a reference to an unspent output, a signature script and a sequence number. The reference consists of the hash of a previous transaction (TX) and the index of the output in that transaction. The signature script has to make the locking script of the referenced output evaluate to true, to be able to redeem the output. The sequence number, in combination with the locktime, makes it possible to replace a transaction, as long as the locktime has not been expired. Therefore, once the point in time specified by locktime has passed, the transaction with the highest sequence number is added to the blockchain. When the sequence number is set to the highest possible value, the locktime is ignored, since no other transaction can supersede.


Elliptic-Curve Public Key to BTC Address conversion
Public Key:
$x_{integer}$
$y_{integer}$
1
32 bytes (BE)
32 bytes (BE)
0x04
ripemd160(sha256(1 32 bytes (BE) 32 bytes (BE)))
1
20 bytes
Network ID Byte:
Main Network: 0x00
Test Network: 0x6f
Namecoin Net: 0x34
sha256(sha256(1 20 bytes))
32 bytes
Checksum
25-byte binary address
1
20 bytes
4
Base256-to-Base58 conversion*
(treat both quantities like big-endian)
1AGRxqDa5WjUKBwHB9XYEjmkv1ucoUUy1s
etotheipi@gmail.com / 1Gffm7LKXcNFPrtxy6yF4JBoe5rVka4sn1

*In a standard base conversion, the 0x00 byte on the left would be irrelevant (like writing '052' instead of just '52'), but in the BTC network the left-most zero chars are carried through the conversion. So for every 0x00 byte on the left end of the binary address, we will attach one '1' character to the Base58 address. This is why main-network addresses all start with '1'

Figure 3.7: This figure shows the conversion steps from an ECDSA public key to a Bitcoin address. Source: https://en.bitcoin.it/w/images/en/9/9b/PubKeyToAddr.png

Transaction	
Version No.	
Flag	
In-counter	Out-counter
Inputlist:	Outputlist:
Input0	Output0
Previous TX Hash	Value
Previous TX out-index	Locking Script length
Sign. Script Length	Locking Script
Sign. Script	
Sequence No.	
…	…
Witness	
locktime	

Figure 3.8: Transaction in Bitcoin

The scripting system used for the transactions supports basically two different script types. The most common one is a Pay-To-Public-Key-Hash (P2PKH) script. Here a signature of the transaction, as well as a public key has to be provided to make the script evaluate to true. The other type of scripts are Pay-To-Script-Hash (P2SH) scripts, where an arbitrary script can be used to lock an output and the correct input as well as the script itself has to be provided to unlock it. To evaluate a script the locking script from the transaction output is combined with the signature script from the transaction input and then all commands are executed in order. An example for a P2PKH script evaluation is shown in Figure 3.9. First all constants (signature and public key) are pushed on the stack. Then the top element on the stack, here the public key, is duplicated and the address is created from the public key with the OP_HASH160 instruction. Thereafter the address (or public key hash) of the output is pushed on the stack and the two topmost elements are compared. The whole evaluation process will fail, if the two elements are not equal. Finally the signature is verified and true is pushed on the stack, if the verification is successful. The evaluation of an P2SH script is similar, but first the script provided in the input is hashed and compared with the script hash in the output. Afterwards only the data provided in the input is evaluated.

Locking Script:	OP_DUP	OP_HASH160	<pubKeyHash>	OP_EQUALVERIFY	OP_CHECKSIG
Signature Script:	<sig>	<pubKey>			

Stack	<sig>	<pubKey> <sig>	<pubKey> <pubKey> <sig>	<pubHashA> <pubKey> <sig>	<pubKeyHash> <pubHashA> <pubKey> <sig>	<pubKey> <sig>	TRUE
Executed Command	<sig>	<pubKey>	OP_DUP	OP_HASH160	<pubKeyHash>	OP_EQUALVERIFY	OP_CHECKSIG

Figure 3.9: Evaluation of an Pay-To-Public-Key-Hash script. The order of the executed commands equals to the signature script, followed by the locking script from left to right. Data is enclosed by angled brackets and is only pushed on the stack. Opcodes are preceded by OP.

A block in Bitcoin, as depicted in Figure 3.10, consists of a magic number, the blockheader and one or multiple transactions. The magic number has always the same value, which is 0xD9B4BEF9 for blocks in the mainnet and 0xDAB5BFFA for the blocks in the testnet. The blockheader itself contains a version number, a reference (hash) to the previous block, a Merkle root, a timestamp, Bits and a nonce. The version number specifies again, which set of rules apply to the block, but it is independent of the version number of the transactions. The Merkle root serves as a checksum and is calculated by creating a Merkle tree [Mer89], where all transactions in the block are used as the leaves of the tree. Therefore, it is sufficient to store only the Merkle root, to verify the integrity of all transactions. Additionally, due to the properties of a Merkle tree, it is possible to efficiently evaluate if some transaction is part of the block or not. The Bits field in the blockheader is a packed representation of the target, which defines the difficulty of mining the block. The nonce is the solution of the proof of work puzzle provided by a miner and has to comply with the target.

To add a transaction to the block chain, it has to be broadcasted to the Bitcoin network. Each peer in the network, who receives the transaction will check for its validity and valid transactions are then broadcasted to other peers. If the transaction was received by a miner, the miner can attempt to include the transaction into a block. Completed blocks are then broadcasted to the network and the block chain is updated accordingly, if the block is valid. A block is only valid, if all transactions within it are valid and if the miner is able to solve a computational puzzle. In the case of Bitcoin the miner has to find a value for the nonce of the block, so that two iterations of SHA-256, with the whole block as input, produce an output, which is smaller

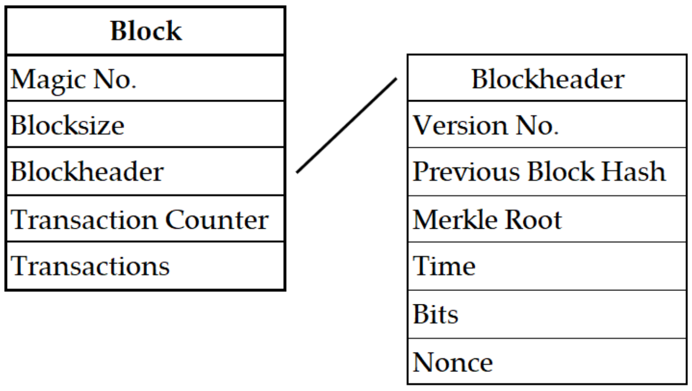


Figure 3.10: Block in Bitcoin

than the target value. The target value is included in the Bits field of the block in packed form. Once a miner has found the solution, she can broadcast the block to the network and peers will add the block to the blockchain, if no other block has been added yet. As a reward the miner can transfer the bitcoin, which are created with each new block, to an address under her control. The amount of bitcoin created is halved every 210,000 blocks and therefore, the number of bitcoin, which will ever exist is finite. Additionally, the miner receives a transaction fee, that is every bitcoin from the input, that is not consumed by an output. This mining process is part of the proof-of-work system and therefore the consensus finding in Bitcoin.

4 Double Authentication Preventing Signatures

In this chapter the functionality of Double Authentication Preventing Signature schemes or short DAPS is presented. Additionally, the differences to schemes similar to DAPS will be pointed out and a brief overview on the currently known schemes and instantiations of DAPS will be given. Afterwards the DAPS scheme used for the implementation of the presented payment protocol is detailed.

4.1 Definition

A DAPS scheme is similar to a conventional digital signature scheme in that, messages can be signed with a private key and verified with a public key. In a DAPS scheme however, a message consists of a payload and an address. Whenever two messages are signed with the same address, but different payloads, the private signing key can be extracted from the signatures. Therefore, a DAPS scheme disincentivizes signing under the same address twice, by exposing the signing key, when doing so. This makes it possible to bind data or statements to a specific context. So, when for example, Alice states 'Yes' in a survey with the address 'X', she can't state 'No' afterwards in the same survey, without revealing her private key.

When discussing DAPS a term frequently used throughout this work is colliding messages. A pair of messages $\{m_1, m_2\}$, with $m_1 = (a_1, p_1)$ and $m_2 = (a_2, p_2)$, is colliding, if $a_1 = a_2$ and $p_1 \neq p_2$, where a is the address and p is the payload of the message.

DAPS were first introduced by [PS14; PS17] and accordingly a DAPS scheme with a message space M can be described with the following four algorithms

$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(1^\kappa)$: The key generation algorithm takes the security parameter κ as input and outputs a private key sk to sign messages and a public key pk to verify signatures.

$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$: This method takes a message m and a private key sk as input and outputs a signature σ, if $m \in M$ and $\perp$ otherwise.

$\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{pk}, m, \sigma)$: This algorithm takes a public key pk, a message m and a signature σ as input and outputs 0, if the verification fails or if $m \notin M$ and 1 otherwise.

$\mathsf{sk} \leftarrow \mathsf{Ex}(\mathsf{pk}, m_1, m_2, \sigma_1, \sigma_2)$: The extraction algorithm takes the public key pk, a colliding message pair m_1 - m_2 and the signatures σ_1, σ_2 to the corresponding messages as inputs and outputs the private key sk.

The algorithm definitions of KGen, Sign and Verify are exactly the same as for digital signature schemes, but a DAPS scheme requires additionally an extraction algorithm, which makes it possible to extract the private key, if two messages are signed under the same address. A DAPS scheme is required to be correct and it is correct when

$$\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1, \tag{4.1}$$

for all possible keypairs generated from KGen, for all $m \in M$ and all signatures outputted from Sign. In a DAPS scheme the EUF-CMA security notion cannot be applied, as it would be, by design, trivial to forge a signature. Thus, a slightly restricted variant, due to [PS14] can be applied, as depicted in Figure 4.1. In this variant the adversary is only allowed to query signatures for messages with a distinct address. Otherwise the security game is exactly the same as for EUF-CMA and a PPT adversary should only have a negligible chance to win the game.

Another notion particularly important for DAPS is Double-Signature Extractability (DSE) defined by [PS14]. The DSE game is shown in Figure 4.2 and the chance, that an adversary can win the game should be negligible in the security parameter. DSE ensures, that if the DAPS keypair was generated honestly, the private key can always be extracted, when two messages are signed under the same address. In other words, an adversary should not be able to create valid signatures for two colliding messages, from which the private key cannot be extracted. There is also a stronger version of double-signature extractability denoted as DSE*, where the adversary is allowed to generate the DAPS keypair. Thus, DSE* guaranties, that even for maliciously generated keys, the signing key can be extracted. However, there is no efficient DAPS scheme known yet, which can achieve this notion.

```
$\mathbf{Exp}^{\mathsf{EUF-CMA}}_{\mathcal{A},\mathsf{DAPS}}(\kappa)$ :
    $(\mathsf{sk_D}, \mathsf{pk_D}) \leftarrow \mathsf{KGen_D}(1^\kappa)$
    $\mathcal{Q} \leftarrow \emptyset, \mathcal{R} \leftarrow \emptyset$
    $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign'_D}(\mathsf{sk_D}, \cdot)}(\mathsf{pk_D})$
        where oracle $\mathsf{Sign'_D}$ on input $m$ :
            $(a, p) \leftarrow m$
            if $a \in \mathcal{R}$, return $\perp$
            $\sigma \leftarrow \mathsf{Sign_D}(\mathsf{sk_D}, m)$
            $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{m\}, \mathcal{R} \leftarrow \mathcal{R} \cup \{a\}$
            return $\sigma$
    return 1, if $\mathsf{Verify_D}(\mathsf{pk_D}, m^*, \sigma^*) = 1 \wedge m^* \notin \mathcal{Q}$
    return 0
```

Figure 4.1: EUF-CMA security for Double Authentication Preventing Signature schemes

```
$\mathbf{Exp}^{\mathsf{DSE}}_{\mathcal{A},\mathsf{DAPS}}(\kappa)$ :
    $(\mathsf{sk_D}, \mathsf{pk_D}) \leftarrow \mathsf{KGen_D}(1^\kappa)$
    $(m_1, m_2, \sigma_1, \sigma_2) \leftarrow \mathcal{A}(\mathsf{sk_D}, \mathsf{pk_D})$
    return 0, if $m_1$ and $m_2$ are not colliding
    $v_i \leftarrow \mathsf{Verify}_D(\mathsf{pk_D}, m_i, \sigma_i)$ for $i \in [2]$
    return 0, if $v_1 = 0$ or $v_2 = 0$
    $\mathsf{sk'_D} \leftarrow \mathsf{Ex}_D(\mathsf{pk_D}, m_1, m_2, \sigma_1, \sigma_2)$
    return 1, if $\mathsf{sk'_D} \neq \mathsf{sk_D}$
    return 0
```

Figure 4.2: DSE security for Double Authentication Preventing Signature schemes

DAPS schemes cannot only be build as an ad-hoc construction, but conventional digital signature schemes might also be extended to a DAPS scheme. Making only black box-use of a conventional scheme has the advantage, that the integration in existing applications is easier and different schemes may be used as a basis. For such DAPS schemes an additional notion of extractability, namely weak Double-Signature Extraction (wDSE), can be defined, due to [DRS18b]. Again there is a variant for honestly and maliciously generated keys denoted as wDSE and wDSE* respectively. For wDSE(*) a DAPS scheme is not required to extract the full DAPS private key from two colliding, signed messages, but only the private key of the underlying conventional signature scheme. The security game for wDSE* is given in Figure 4.3, where the DAPS private key is denoted as pk_D and the private key of the signature scheme as pk_Σ. A PPT adversary should only be able to win this game with negligible probability.

```
$\mathbf{Exp}^{\mathsf{wDSE}^*}_{\mathcal{A},\mathsf{DAPS}}(\kappa)$ :
    $(\mathsf{pk}_D, m_1, m_2, \sigma_1, \sigma_2) \leftarrow \mathcal{A}(1^\kappa)$ where $\mathsf{pk}_D = (\mathsf{pk}_\Sigma, \dots)$
    return 0, if $m_1$ and $m_2$ are not colliding
    $v_i \leftarrow \mathsf{Verify}_D(\mathsf{pk}_\mathsf{D}, m_i, \sigma_i)$ for $i \in [2]$
    return 0, if $v_1 = 0$ or $v_2 = 0$
    $\mathsf{sk}'_\mathsf{D} \leftarrow \mathsf{Ex}_D(\mathsf{pk}_\mathsf{D}, m_1, m_2, \sigma_1, \sigma_2)$ where $\mathsf{sk}'_D = (\mathsf{sk}'_\Sigma, \dots)$
    return 1, if $\mathsf{sk}'_\Sigma$ is not the private key corresponding to $\mathsf{pk}_\Sigma$
    return 0
```

Figure 4.3: wDSE* security for Double Authentication Preventing Signature schemes

4.2 Self-enforcing Signatures

DAPS are a type of self-enforcing signatures and thus there are other schemes akin to DAPS, as enlisted by [PS14]. In the electronic cash system from [CFN90], double spending a token is penalized by revealing the identity of the misbehaving party. This is similar to DAPS, as in a DAPS scheme signing two colliding messages reveals the signing key.

In Lamport signatures [Lam79] signing more than once reduces the security of the scheme, because each additional signature enables an adversary to create new valid signatures. Several of these signatures can be combined using a Merkle tree [Mer89], which results in a more compact public key.
With Fail-stop signatures [WP89] and Forgery-resilient signatures [MO12] it is possible to prove that a forgery has occurred. When a forgery occurs, the security of Fail-stop signatures diminishes. Forgery-resilient signatures on the other hand remain secure. Fail-stop as well as forgery-resilient signatures enable an honest signer to prove, that an adversary was able to produce a forgery. In contrast, a DAPS scheme makes it possible to penalize a dishonest signer and therefore disincentivizes to act dishonest.
Chameleon hash functions [Tal00] are trapdoor one-way functions, with which a collision can be found efficiently, if a message-randomness pair of a specific hash and a trapdoor is known. This would make it possible to extract the private parameters, like in DAPS schemes and indeed [RKS15] were able to build a DAPS scheme from a chameleon hash function.
Accountable assertions, as introduced in [RKS15] have the same properties as DAPS, but they are not required to be unforgeable. In other words, they do not have to have DAPS EUF-CMA security as shown in Figure 4.1. The DAPS scheme presented in [RKS15] is in fact a modified version of an accountable assertion scheme.
Predicate authentication preventing signatures (PAPS) [BKN17] provide extractability for some secret information, if k signed messages fulfill a k-ary predicate. Therefore, DAPS are a special PAPS, where $k = 2$ and the secret information is the private signing key.

4.3 Overview of available DAPS Schemes

There are DAPS in several different settings and with differently large address spaces. Currently DAPS are known in the factoring-based [PS14; PS17; BPS17], discrete logarithm [RKS15; DRS18b; Poe18] and lattice-based [BKN17] setting, as well as DAPS from symmetric key primitives [DRS18a]. The DAPS constructions from [DRS18b; Poe18] have only a small address space, because the key size is growing linearly with the number of addresses. All other DAPS schemes offer an exponentially large address space.

The very first DAPS construction, which was introduced by [PS14; PS17], can be build from any extractable 2:1 trapdoor function. They presented an instantiation of this DAPS with a factoring-based 2:1 trapdoor function utilizing quadratic residues in Blum integer groups. In [BPS17] DAPS based on trapdoor identification schemes were introduced. For the instantiation the trapdoored version of the identification scheme from [GQ90] and the trapdoor identification scheme from [MR02] were used, which are both factoring-based. Instead of the Fiat-Shamir transform [FS87], which is usually used to create a signature scheme out of an identification scheme, they utilized the double-hash (H2) and double-id (ID2) dubbed transforms. This made a tighter security reduction possible in comparison to Fiat-Shamir and led to the three DAPS schemes H2[GQ], ID2[GQ] and H2[MR].

DAPS as a subset of predicate authentication preventing signatures (PAPS) were presented by [BKN17]. Their DAPS is based on a lattice trapdoor function and is therefore believed to be post-quantum secure. Additionally, it can be extended to either a PAPS or a multi-authority DAPS (MADAPS). With MADAPS the secret information can be extracted from the signatures of different signers, if their signed messages fulfill a predicate.

Ruffing et al. [RKS15] introduced a primitive dubbed accountable assertions (AS), which can be extended to a DAPS scheme. Their AS-based DAPS is the first DAPS in the discrete logarithm setting and uses chameleon hash functions in conjunction with a Merkle tree. This leads to a more efficient construction than the factoring-based DAPS from [PS14; PS17]. However, the DAPS from [BPS17] are faster and have significantly smaller signatures, even so their public key size is larger. In [DRS18b] a DAPS was introduced, which makes only a black box use of conventional digital signature schemes. Their DAPS utilizes secret sharing and a NIZK proof to ensure that a share of the private key is included in each signature. This enables the DAPS constructions to be instantiated with any signature scheme in the discrete logarithm setting and again improves in efficiency in comparison to previous schemes. However, the private and public key size is growing linearly with the size of the address space and therefore the construction is only practical for a small address space. Their construction additionally makes it possible to build n-times authentication preventing signatures (NAPS), which is denoted as k-way DAPS in [BKN17]. With NAPS the private key can be extracted for n colliding messages and thus DAPS can be seen as a NAPS for $n = 2$. Another DAPS for small address spaces was introduced by [Poe18], which is the most efficient DAPS known at the moment. Their DAPS have a probably optimum signature size for DAPS of 256-bit for a 128-bit security level and a constant sized private key. They showed how to construct DAPS from strictly one-time (SOT) signatures and presented an instantiation with SOT signatures from identification schemes using a modified version of the Fiat-Shamir transform. However, their construction does not make a black box reduction of the underlying signature scheme.

Recently a DAPS, which relies on symmetric primitives was presented in [DRS18a]. In their work, they build upon the approach from [DRS18b] and resolved the problem of the small address space. Because their DAPS utilizes solely symmetric key primitives, it is a candidate for a post-quantum secure DAPS. Additionally, they introduced a compiler to build a DAPS from any combination of conventional signature and DAPS schemes. This might be particularly useful for a wider adoption of DAPS, since the most efficient DAPS with an exponential address space known at the moment is ad-hoc and therefore does not make black box use of a conventional digital signature scheme.

4.4 DAPS in the Discrete Logarithm Setting

In this section the DAPS scheme used for the instantiation of the application, presented in Chapter 5, is detailed. The application requires a DAPS scheme, where the private key or a part of the private key can be simultaneously used in Bitcoin. Therefore, schemes which make a black box use of ECDSA are from advantage. Even so every DAPS scheme can extend a conventional signature scheme with the compiler presented in [DRS18a], only the DAPS from [DRS18b] are wDSE* secure and therefore allow the ECDSA secret key to be extracted, even under maliciously generated keys. The DAPS from [RKS15] have a similar property to wDSE*, where the private key of the chameleon hash function (*not* the whole DAPS private key) can be extracted from colliding messages, even when the adversary generates the DAPS keypair. However, when this private key is additionally used for a Bitcoin deposit and therefore in ECDSA, it might not be possible to prove EUF-CMA security. Therefore, in the application from [RKS15] the weaker accountable assertions were used, which increase the efficiency, but do not provide EUF-CMA security in the first place. To conclude, the DAPS from [DRS18b] seem to be the best choice for the application, because they make only a black box use of ECDSA, provide wDSE* security and are among the most efficient DAPS. Therefore, they will be detailed in the next paragraph.

Even so the DAPS construction only allows for a small address space, this is not a problem for this particular application, because the number of transactions a client is going to make during the period of time the deposit is valid, is only limited.

The DAPS uses verifiable secret sharing and zero-knowledge proofs to ensure that a share of the private key is included in each signature. This provides the extractability required for DAPS schemes. The four algorithms of the scheme are shown in Figure 4.4. In the key generation algorithm, besides the keypair of the digital signature scheme, a keypair for the ElGamal encryption scheme is created. The number of addresses is determined by the input n of KGen_D and for each address a ρ_i is chosen uniformly at random from $\mathbb{Z}_q$ and an encryption of each ρ_i is included in the public key. All ρ_i and the r_i used for the encryption are additionally included in the private key. The signing algorithm creates an ECDSA signature of the whole message $m = (i, p)$ and extends it with a share of the private key, as well as a non-interactive zero-knowledge proof (NIZK). The share is calculated with $f_i(X) = \mathsf{sk}_\Sigma + \rho_i \cdot X$, where the function is evaluated for $X = p$. The NIZK proof is a proof that $(g, \mathsf{pk}_E, C_{i,1}, C_2')$ form a DDH-tuple, where $C_{i,1} = g^r$ and $C_2' = \mathsf{pk}_E^r$. Thus, it is a proof for the verifiable secret sharing and ensures, that the share is really a share of the private key. The extraction algorithm allows to extract the private key of the underlying ECDSA signature scheme, but not the whole DAPS private key. Therefore, the DAPS scheme does not provide DSE security. However, it achieves the weaker wDSE* notion under malicious keys, because if a DAPS signature verifies correctly, it is ensured with high probability, that a share of the private key is included in the signature.

As apparent from the construction, the key size is growing linearly with the number of addresses and thus the practically usable address space is limited. To solve this issue, attempts have been made to reduce the dependency of the key size from linear to logarithmic or even keep the key size constant. The pursued approach was to use a verifiable random function (VRF) to replace the randomness in the secret key and the encryption of this randomness in the public key with respectively a constant parameter. As shown in [BG89] a tuple consisting of a document D, a PRF evaluation R of D for some secret key s and a NIZK proof are verifiable, if the NIZK proof is conducted for the $\mathcal{NP}$ statement

$$\exists s, \exists r : \mathsf{ct} = \mathsf{Enc}(r, s) \wedge R = \mathsf{PRF}_s(D), \tag{4.2}$$

where Enc is an encryption algorithm and ct is public knowledge. In context of this DAPS scheme, this means that given an address, a PRF evaluation of this address, an encryption of the secret PRF key and a NIZK proof, it can be verified if the PRF was evaluated for the given address. Accordingly, the key size of the DAPS would be constant and the secret key would only consist of the signature and PRF secret key and the public key of the signature, PRF and encryption public key. This construction was instantiated with the VRF from [DY05] in conjunction with the elliptic curve ElGamal encryption scheme. However, it was not possible to come up with a security proof for it.

These issues were resolved in [DRS18a], where a DAPS with a constant key size and therefore an exponentially large address space was presented. This DAPS scheme also builds upon the secret sharing approach, but all building blocks, including the PRF, solely consist of symmetric key primitives. To use their construction in combination with a signature scheme relying on structured hardness assumptions would require a PRF, which maps from $\mathbb{Z}_q$ to $\mathbb{Z}_q$. However, most PRFs in the discrete logarithm setting, like the VRF from [DY05] map from $\mathbb{Z}_q$ to $\mathbb{G}$. It is possible to transform the output of the PRF back to $\mathbb{Z}_q$, by for example using a random oracle, but this would make it hard to conduct a proof. Furthermore it is difficult to find an efficient proof system, which proofs simultaneously a statement in $\mathbb{Z}_q$ and $\mathbb{G}$. Development in this area has been made by [AGM18], but building such a DAPS in the discrete logarithm setting still remains an open topic.

```
$\underline{\{\mathsf{sk},\mathsf{pk}\} \leftarrow \mathsf{KGen}_D(1^\kappa, n)}$ :
    Let $\mathbb{G}$ be an elliptic curve group of prime order $q$ and with generator $g$ and
    $H : \{0,1\}^* \to \mathbb{Z}_q$ be a hash function mapping exactly to the order of the group :
1 :   $\mathsf{sk}_\Sigma \xleftarrow{R} \mathbb{Z}_q^*$ and $x_E \xleftarrow{R} \mathbb{Z}_q^*$
2 :   $\mathsf{pk}_\Sigma = g^{\mathsf{sk}_\Sigma}$ and $\mathsf{pk}_E = g^{x_E}$
3 :   $(\rho_i)_{i\in[n]} \xleftarrow{R} (\mathbb{Z}_q^*)^n$ and $(r_i)_{i\in[n]} \xleftarrow{R} (\mathbb{Z}_q^*)^n$
4 :   $(C_i)_{i\in[n]} \leftarrow (g^{r_i}, \mathsf{pk}_E^{r_i} \cdot g^{\rho_i})_{i\in[n]}$ and $\mathsf{crs} \leftarrow \mathsf{Setup}_{\mathsf{DDH}}(1^\kappa)$
5 :   $\mathsf{sk} \leftarrow (\mathsf{sk}_\Sigma, (r_i, \rho_i)_{i\in[n]})$
6 :   $\mathsf{pk} \leftarrow (\mathsf{pk}_\Sigma, \mathsf{pk}_E, (C_i)_{i\in[n]}, crs)$
$\underline{\sigma \leftarrow \mathsf{Sign}_D(\mathsf{sk}, m)}$ : parse sk as $(\mathsf{sk}_\Sigma, (r_i, \rho_i)_{i\in[n]})$. Parse $m$ as $(i, p)$ with $i \le n$ and $p \in \mathbb{Z}_q^*$
1 :   $k \xleftarrow{R} \mathbb{Z}_q^*$
2 :   $R \leftarrow g^k$
3 :   $r \leftarrow R_x \mod q$, if $r = 0$ goto step 1
4 :   $s \leftarrow k^{-1}(H(m) + r \cdot \mathsf{sk}_\Sigma) \mod q$ and if $s = 0$ goto step 1
5 :   $z \leftarrow \rho_i \cdot p + \mathsf{sk}_\Sigma$
6 :   $C_2' \leftarrow C_{i,2} \cdot (\mathsf{pk}_\Sigma \cdot g^{-z})^{\frac{1}{p}}$
7 :   $\pi \leftarrow \mathsf{Proof}_{\mathsf{DDH}}(crs, (g, \mathsf{pk}_E, C_{i,1}, C_2'), r_i)$
8 :   $\sigma \leftarrow (r, s, z, \pi)$
$\underline{\{0,1\} \leftarrow \mathsf{Verify}_D(\mathsf{pk}, m, \sigma)}$ : parse pk as $(\mathsf{pk}_\Sigma, \mathsf{pk}_E, (C_i)_{i\in[n]}, crs, \cdot)$. Parse $\sigma$ as $(r, s, z, \pi)$.
Parse $m$ as $(i, p)$ with $i \le n$ and $p \in \mathbb{Z}_q^*$
1 :   if $r = 0 \vee s = 0$ return 0
2 :   $z \leftarrow H(m) \mod q$ and $w \leftarrow s^{-1} \mod q$
3 :   $u_1 \leftarrow zw \mod q$ and $u_2 \leftarrow rw \mod q$
4 :   $R \leftarrow g^{u_1} \cdot \mathsf{pk}_\Sigma^{u_2}$
5 :   if $R_x \ne r \mod q$ return 0
6 :   $C_2' \leftarrow C_{i,2} \cdot (\mathsf{pk}_\Sigma \cdot g^{-z})^{\frac{1}{p}}$
7 :   return $\mathsf{Verify}_{\mathsf{DDH}}(crs, (g, \mathsf{pk}_E, C_{i,1}, C_2'), \pi)$
$\underline{\mathsf{sk}_\Sigma \leftarrow \mathsf{Ex}_D(\mathsf{pk}, m_1, m_2, \sigma_1, \sigma_2)}$ : parse $\sigma_i$ as $(\cdot, \cdot, z_i, \cdot)$. Parse $m_i$ as $(a_i, p_i)$
1 :   if $m_1$ and $m_2$ are not colliding, return $\bot$
2 :   if $\mathsf{Verify}_D(\mathsf{pk}, m_i, \sigma_i) = 0$ for any $i$, return $\bot$
3 :   $\mathsf{sk}_\Sigma \leftarrow z_1 \cdot \frac{p_2}{p_2 - p_1} + z_2 \cdot \frac{p_1}{p_1 - p_2}$
```

Figure 4.4: DAPS from ECDSA

5 Applications

At first an overview of possible applications for DAPS is presented, followed by the application focused on in this thesis, an analysis of this application and a comparison to similar ones. In all DAPS application a signer has to take additional care, that she does not accidentally sign two conflicting messages and thus the signer might want to maintain a list of signed addresses or employ other mechanisms, that prohibit doing so.

5.1 Overview

In [PS14; PS17] Poettering and Stebila showed two use-cases for DAPS, one in the public key infrastructure (PKI) of the web and one in time-stamping. In the context of PKI, a certificate authority (CA) is trusted to sign only unique certificates for a subject. It might, however occur, that a CA signs multiple certificates for the same subject, due to for example security breaches, poor management practices or coercion through third-parties. When using a DAPS scheme for signing certificates, a CA is discouraged to ever sign multiple certificates of the same subject, as this would leak the signing key and compromise their security.
Time-stamping authorities provide proof that certain data was available at a certain point in time, by signing the hash of the data together with a timestamp. Such authorities can additionally be disincentivized to act dishonestly by using a DAPS scheme and only publishing a signature at maximum once per increment of the timestamp.

Ruffing et al. [RKS15] proposed to use DAPS in combination with digital currencies so that the private key, which can be extracted from colliding messages, locks at the same time funds in the corresponding address of a cryptocurrency. Therefore, equivocation can be monetarily penalized for any non-equivocating contract. Additionally, they showed a payment protocol for cryptocurrencies, where a payee is able to receive payments from unsynchronized points of sale.

In [BKN17] it was suggested, that DAPS can be used to prohibit selling the rights to a patent to multiple parties. If the patent number is used for the address of a DAPS message and for example "owned by Party A" as the payload, party A can have some confidence, that the patent will not be sold to other parties.

Another application, proposed by [DRS18b], is to use DAPS for code signing. When the version number of an application is used as a DAPS address, the signing key can be extracted, whenever a normal and a backdoored variant for the same version number is released. With the signing key everyone can publish new versions or even updates of the application. This can also be applied for an app store, where every offered application is signed by the store owner with a DAPS. Here however, the address has to consist of an application ID and the version number, to allow for multiple applications. If the owner of the store signs a normal and a backdoored version of an application, the signing key can be extracted and everyone might publish applications on the store.

An additional use case for DAPS is to disincentivize publishing a censored version of digital goods. This can be achieved, by using an identifier, like the ISBN[1] for (e-)books, as the address of a DAPS message and the content as payload. Therefore, when a normal and a censored version of a digital good is published, the signing key can be extracted. If this does not suffice as a disincentive alone, the key might additionally be coupled with the deposit of a cryptocurrency, as proposed in [RKS15].

5.2 Off-chain Payments

The application presented in the following, is based on the payment protocol from [RKS15]. It enables a customer to make off-chain payments at independent points of sale with Bitcoin, where all points of sale have to trust a common provider. The points of sale are partners of the provider, but they may belong to different, independent companies. The provider may for example be a bank, which can generally be trusted.

[1]https://www.isbn-international.org/

Reasons why the provider has to be trusted and in what way will be shown in Section 5.3, but the amount of trust might be arguably minimal.

An example execution of the payment protocol is presented in Figure 5.1. If Alice wants to make payments at partners of Bob, she sets up a Bitcoin deposit and sends it to Bob. A simplified version of such a deposit is shown in Figure 5.2. It basically contains the maximum amount of bitcoin Alice is able to spend to Bob and a locktime, which specifies when the deposit expires. If the payment protocol executes successfully, Bob will receive (a part of) the funds from the deposit and Alice can reclaim the remainder. If Bob never closes the deposit, Alice can retrieve all funds after the locktime has expired. If Alice cheats by double spending, Bob can retrieve all funds. A more detailed description of the deposit transaction is given in Chapter 6.

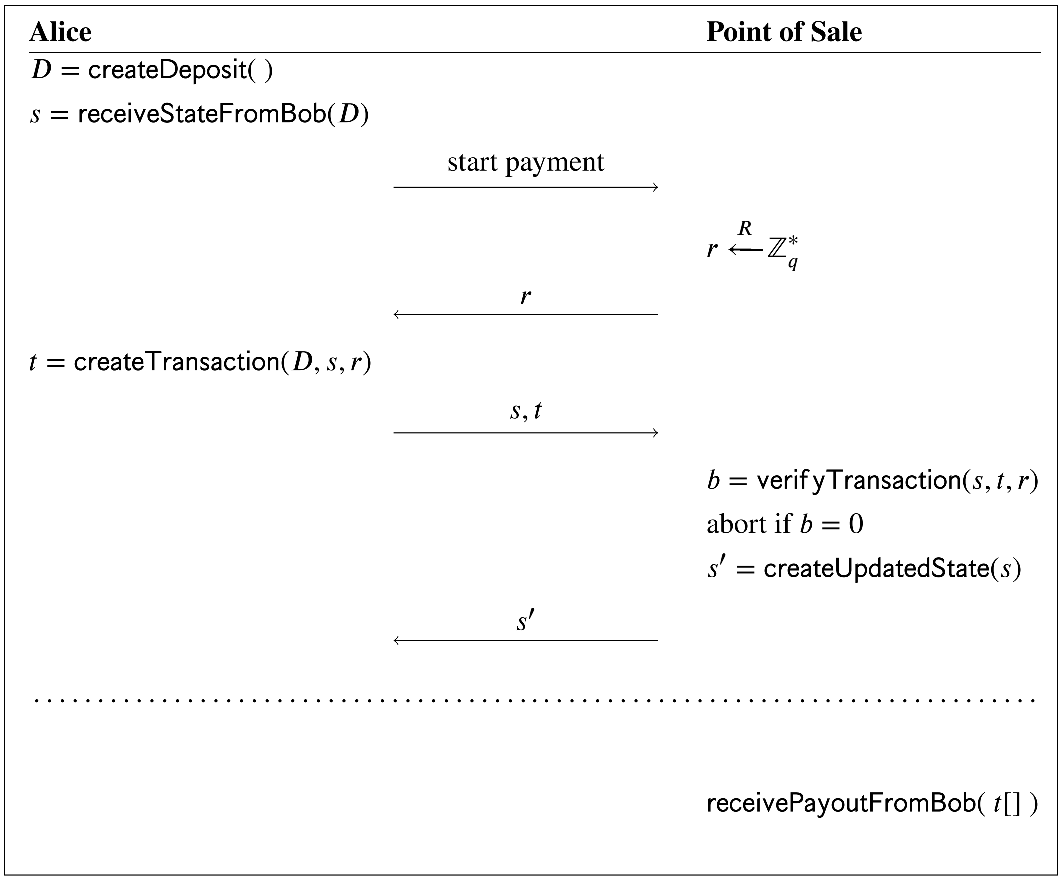

Alice
Point of Sale
$D = \mathsf{createDeposit}(\,)$
$s = \mathsf{receiveStateFromBob}(D)$
start payment
$r \xleftarrow{R} \mathbb{Z}_q^*$
r
$t = \mathsf{createTransaction}(D, s, r)$
s, t
$b = \mathsf{verifyTransaction}(s, t, r)$
abort if $b = 0$
$s' = \mathsf{createUpdatedState}(s)$
s'
$\mathsf{receivePayoutFromBob}(\, t[]\,)$

Figure 5.1: Example for an execution of the payment protocol, where Alice pays at a partner from Bob

Deposit	
From	Alice
Amount	X
Locktime	t_D

Figure 5.2: Simplified deposit of the client, to establish the payment channel

Once the deposit has enough confirmations on the blockchain, Bob sends Alice a state corresponding to the deposit. As shown in Figure 5.3 the state contains a reference to the deposit, the validity period of the transaction channel, the amount of bitcoins Alice is allowed to spend, the amount she has already spent, a revision number and a signature of Bob.

State	
Reference to Deposit	
Expire Time	t_{exp}
Limit	X_L
Spent	X_S
Rev. Nr.	R_S
ECDSA Signature	

Figure 5.3: Signed state, which keeps track of the spent bitcoin

With this state Alice can now make payments to any partner of Bob. When she wants to do so, she queries a random number from the point of sale and then creates a transaction, which contains this number. Such a transaction is depicted in Figure 5.4 and it contains a bitcoin transaction, which spends from the deposit Alice has created in the beginning, a revision number, the random number from the point of sale and a DAPS, with the revision number as address. The point of sale has to take care, that each time a new random number is handed over and that the number is really included in the received transaction. Otherwise the payload of the DAPS will be the same for each transaction with the same revision number and Bitcoin transaction and therefore the secret key will not be extractable upon double spending.

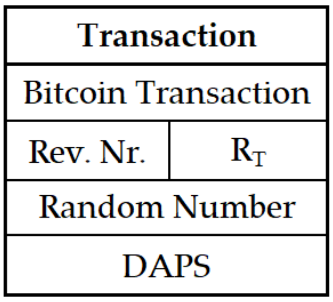


Figure 5.4: Signed transaction for a certain revision of the state

After Alice has created the transaction she sends it with the corresponding state to the point of sale. The point of sale verifies if:

- the ECDSA signature of the state is valid. It is valid, if it was created by Bob or if it verifies under a public key, which was signed by Bob.
- the random number handed out to Alice is really included in the transaction.
- the DAPS of Alice is valid.
- the revision number of the state R_S matches the revision number of the transaction R_T.
- the Bitcoin transaction is a valid spend from the deposit.
- the amount of bitcoin spend in the transaction do not exceed the limit X_L of the state
- the amount of bitcoin spend by the transaction subtracted by the amount of bitcoin already spent X_S in the state is (greater or) equal to the amount Alice has to pay.
- the expire time t_{exp} of the state has not been passed yet.
- a transaction with the revision number R_T has not been received yet in any of the previous transactions from Alice at this particular point of sale.

If everything is sound, the point of sale returns a new state, with an updated value for the revision number and the amount of bitcoin Alice has spent. The point of sale additionally includes its public key and Bob's signature of the key in the state, so that other points of sale can verify its validity. The point of sale stores all transactions received from Alice, but Alice only has to keep the latest state. Alice may now make payments at other points of sale with the new state. Before Alice's deposit expires, Bob collects all transactions from all points of sale, spends the most recent one and pays his partners accordingly. Bob may claim some fees for his participation as an intermediary.

5.3 Security Analysis

For this application either of the parties, namely the client, the point of sale or the provider or each possible pair of them might act dishonestly. Therefore, it will be analyzed, what each combination of misbehaving parties can achieve by altering the data they can control, retaining data or reusing old data.

A dishonest client can attempt to double spent the deposit transaction on the Bitcoin network, retrieve the funds of the deposit after the locktime has expired or reuse an old state and create a new transaction for that state. The client cannot alter a state or produce a faulty transaction, if the point of sale follows the protocol correctly. To prevent the client from double spending the deposit transaction, the provider has to wait until the transaction has an sufficient amount of confirmations on the blockchain[2] before the provider can hand out the state.
The client might attempt to transfer coins to an address under her control as soon as the locktime of the deposit transaction is expired. Thus, the provider has to set an appropriate value for the expire time of the state, which has to be earlier by some safety margin than the locktime of the deposit. The provider has to have enough time to spend the transaction, which pays him and closes the payment channel.
The client is able to reuse any old state at any point of sale, which has not received a more recent version of the state yet and it is not possible to hinder the client doing so. Therefore, a client can use the state s_1 and a corresponding transaction at the point of sale PS_1 and thereafter reuse s_1 at a distinct point of sale PS_2 and double spend coins. The client is able to continue doing so at any other point of sale, which did not receive this state or any state with a higher revision number. However, because the client is signing the transactions with a DAPS, the signing key can be extracted, whenever the client sings two transactions with the same revision number. Therefore, when the provider has collected all transactions and a double spend has occurred,

[2]https://en.bitcoin.it/wiki/Confirmation#How_Many_Confirmations_Is_Enough

the private key of the DAPS can be extracted and all funds from the deposit can be transferred to the provider. Additionally, the provider can set the limit in the state (how many coins the client is able to spend) to a value, which is lower than the actual funds in the deposit, to create a penalty for misbehaving.

A dishonest point of sale can hand out a faulty state to the client or refuse to transfer the collected transactions to the provider.
A point of sale can transmit a faulty state to the client and for example trick the client into double spending. Thus, the point of sale can return a state with an old revision number to the client. To counter this, the client should verify the state upon receiving it and should stop making any further transactions, if the state is incorrect. Once the locktime of the deposit has passed or the provider has closed the payment channel, the client can access his funds again. Additionally, the point of sale might falsify any other data in the state, but this can be countered in the same way.
The point of sale can also withhold the transactions received from the client and not transfer them to the provider. However, the point of sale is discouraged to do so, because it will consequently not receive a payout. Furthermore, this would not affect other points of sale, because their payout is covered in their own collected transactions.

A dishonest provider can, like a point of sale, hand over a faulty state to the client and the client can like before wait until the locktime has passed to retrieve the funds of the deposit.
In addition, a provider may refuse to pay the points of sale after the provider has received the collected transactions from them. If a provider cannot be trusted to pay its partners, a point of sale can require the client to include a hash of a secret, which only the point of sale knows, in the Bitcoin transaction. Furthermore, the reference of the Bitcoin transaction has to be included in the state and when making a new transaction the client has to send the signed old Bitcoin transaction along the normal transaction and the state. The point of sale can then validate if the old Bitcoin transaction matches with the reference in the state and therefore if all outputs with the hashes of the secret are still in the current transaction. When the provider collects all transactions, the points of sale may only reveal their secret if at first the provider hands them over a transaction, which spends from the output containing the hash of the secret. The provider may still include a fee in this transaction.

A dishonest client together with a point of sale can trick other points of sale in accepting false states. Therefore, the dishonest point of sale may hand out a state with a prolonged expire time, higher spending limit and reset spent value to the client. To counter this, the state can be split into two parts. One part, including the reference to the deposit, the expire time and the spending limit, which is signed by the provider and should not change and one part, including the amount of spent coins and the revision number, which can be changed and is signed by the points of sale. Now the client and the point of sale together can do not better than the client alone by reusing an old state.

A dishonest client together with a provider can trick the points of sale in accepting any transaction. This cannot be countered in a scenario, where the points of sale are completely offline. However, if they have a connection to the Bitcoin network, they can verify the reference to the deposit, which is included in each state. If the state does not match the deposit, they can simply refuse the transaction. Furthermore, the provider is discouraged to act dishonest to its points of sale, because on one hand this would be easily discovered as soon as the points of sale claim their payouts and on the other hand the provider would loose its trust and business.

A dishonest point of sale together with a provider has no additional way to cheat the client than each of them alone, because they have only control over the state.

To conclude, a point of sale cannot perform meaningful malicious actions, if the protocol is slightly enhanced and the provider is discouraged to misbehave and the misbehaving can be countered, if the points of sale have a connection to the Bitcoin network. A client however, can double spend at different points of sale, by reusing an old state. This can be detected by the provider, when all transactions are collected from the points of sale. The provider can then extract the DAPS signing key and transfer all funds of the deposit to an address of the provider. It remains still a difficult problem to set an according penalty for the client, because the client can double spend multiple times. A solution would be to apply this application in a constraint environment. Ruffing et al. [RKS15] suggested to use their application for public transport, where the penalty would be, for example the price of a day ticket and the deposit would have a validity period of one day. The application presented in this thesis can additionally be used at different transportation companies simultaneously and for example one could pay for a long distance train and thereafter use the local city transport. The penalty could then be a sum of day tickets for all participating companies. Another use case would be flights and especially flights with a connecting flight, where the client is able to pay on all airplains, even if they are owned by different companies. A different solution to the problem would be to synchronize a minimal amount of data between the points of sale at regular intervals. It would be sufficient to synchronize only a client ID or the reference to the deposit in combination with the latest revision number over a secure channel. This would for example amount per client to 256-bit for the reference (or 32-bit to 64-bit for the client ID) and 8-bit for the revision number.

5.4 Comparison with Alternatives

In this section a comparison with other fast (off-chain) payment methods for cryptocurrencies will be given. In cryptocurrencies like Bitcoin and Ethereum, which are proof of work based, it can take more than 30 minutes until a transaction gets a confirmation on the blockchain. This makes them impractical for real-life application, where for example one wants to pay for groceries at a supermarket and has to wait 1 hour until the transaction has enough confirmations. Cryptocurrencies like Ripple utilize a different consensus algorithm, which allows for fast payments and makes it possible to use them in a real-life scenario at the moment. However, if such a cryptocurrency would find a broad adaption the current transaction throughput might still not be enough. Furthermore, the heavy use of a cryptocurrency will cause storage problems, as every transaction increases the size of the blockchain. Payment protocols, like the one presented or payment networks, like Lightning and Raiden, build a payment channel on top of the cryptocurrency. This allows for a fast processing of transactions and a reduced amount of interactions with the blockchain. Therefore, with payment networks even cryptocurrencies with a low transaction throughput can be used for real-life applications and additionally less transactions have to be stored on the blockchain.

Payment channels only require two transactions broadcasted to the blockchain, one transaction to establish the channel and one to close it. No transaction within a channel requires an interaction with the blockchain. In a real-life scenario, like the supermarket example above, it would be inefficient, if each supermarket had to have a payment channel with each of their costumers. Thus, the channels have to be extended into a payment network. If enough participants of the cryptocurrency network have a payment channel in addition to the supermarket and the customer, a path can be routed from the costumer to the supermarket through a network of channels. Therefore, as long as there is a connection to the cryptocurrency network and a path can be found, the customer is able to pay. Any other participant in the route may charge a fee for acting as an intermediary. No intermediary has to trust the others, because an intermediary only has to make payments, if she received her pay first and it is possible to close out of the channel at any time. However, for payment channels it is important, that the participants regularly check on the blockchain, if their counterparty has closed the channel. Otherwise, they may end up making payments, even though the balance cannot be updated anymore.

The payment protocol presented also only requires two interactions with the blockchain per channel. However, in contrast to payment channels, it is only unidirectional. Therefore, only payments from a client to a provider can be made, but not the other way around. The presented protocol requires a certain amount of trust from the points of sale in the provider, but it can be executed completely offline in a constraint environment once the channel has been established. Furthermore, the provider has an incentive to not close the channel early, because the provider will potentially miss payments upon doing so and the client cannot close the channel before the locktime has passed. Therefore, there is no need to regularly check the blockchain for the status of the channel. Payment channels on the other hand cannot be used completely offline, because a connection to the network of the cryptocurrency is needed to find a path from payer to payee and to verify that the channel is still open. Additionally, it may occur that no path can be found at all.

A summary of this comparison is given in Table 5.1 and to conclude, even though a cryptocurrency might be fast enough to use in a real-life application, it is still beneficial to use a payment protocol on top of it. In payment protocols transactions are decoupled from the blockchain and are therefore faster and have no memory footprint on the blockchain in comparison to on-chain transactions. In payment channels, like Lightning and Raiden, transactions are bidirectional and off-chain, but require that a path of channels can be routed from payer to payee and that the participants regularly check if the channel is still open. In these payment channels none of the participants has to be trusted. The presented payment protocol only allows for unidirectional transactions, but this is sufficient for most of the every day use cases, where a customer wants to buy some kind of goods. While some trust is required in the provider, transactions can be performed completely offline in a constraint environment.

Payment Method	Advantages	Disadvantages	Constraints
Cryptocurrency	• funds are always accessible • transactions from everyone to everyone	• requires a network connection • scalability for a widespread adoption	
Payment Channel	• trustless • bidirectional transactions	• requires a network connection	• a path from payer to payee must exist
Presented Protocol	• can be used completely offline	• some trust has to be put into the provider • unidirectional transactions	• penalty has to be clearly definable

Table 5.1: Overview of payment methods with a cryptocurrency

6 Implementation

In this chapter details on the implementation[1] of the application presented in Chapter 5 will be given. The DAPS functionality was implemented using the OpenSSL[2] library in C and the Bitcoin related functions and interactions with the Bitcoin network was implemented using the Libbitcoin[3] library in C++.

6.1 DAPS

For the DAPS scheme all four algorithms described in Section 4.4 were implemented. In the default configuration the key generation algorithm uses the elliptic curve secp256k1 to be compatible with the ECDSA keys of Bitcoin. Optionally an external ECDSA secret key can be provided to generate a DAPS key from it. The keypair for ECDSA as well as the one for elliptic curve ElGamal are generated with the "EC_KEY" functions of OpenSSL. All other random numbers in the key generation process are obtained from the "BN_rand_range" function according to the order of the elliptic curve group.

In the signature generation algorithm SHA256 is used in conjunction with ECDSA to create the ECDSA signature. For the secret sharing, the payload of the message is hashed with SHA256 and then used in the calculation of the share. This makes it possible to use payloads of arbitrary length for the share. In the NIZK proof again SHA256 is used to create the challenge of the proof. In the verification and extraction algorithms equivalent choices have been made.

[1]The implementation is available at https://github.com/dosc919/bitcoin_payment_protocol
[2]https://www.openssl.org/
[3]https://libbitcoin.org/

6.2 Protocol

For the application the whole functionality of the protocol described in Chapter 5.2 was implemented. During the course of executing the payment protocol there are basically three interesting execution paths:

- The protocol ends normally: The provider collects all transactions from the client and spends the latest one.
- The client double spends: The provider collects all transactions, detects the double spend and retrieves all funds of the deposit.
- The client does not make a transaction or detects a fraud: The client waits until the locktime expires and retrieves the funds of the deposit.

At the beginning of all three paths the client and the provider are generating the ECDSA keys for Bitcoin from a hierarchical deterministic wallet[4] (HDW).

```
New Provider:
Mnemonic:
equip address calm seed priority garden fade thing axis used couch abuse
Address: miv8cV8pwU9CagsW7yMpqvHcUeh47YTM4W

New Client (id = 0):
Mnemonic:
shop convince absorb invite black myself harsh mother skin subject supply prefer
Address: mpZfrRvDxSct69T77AgkRAF4LeJqQ5nyQ6
```

In short, a HDW makes it easier to manage keys and for example use a different key for each deposit. Furthermore, the seed for the HDW is stored as a mnemonic, which is simply a string of words. It is sufficient to store the mnemonic alone, because all keys of the HDW can be derived from it. After the keys have been generated, the balance of the client is queried from the Bitcoin network for the address displayed (here: mpZfrRvDxSct69T77AgkRAF4LeJqQ5nyQ6).

```
client balance: 106860000 Satoshis
```

[4] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

The client can then choose one out of four options on how to proceed.

```
Please choose an option:
1) create a deposit only
2) create a deposit and make payments
3) make payments with an existing deposit
4) exit
```

In this case, the client creates a deposit and makes payments afterwards. The following data has to be supplied

```
please enter the hash of a transaction with an unspend transaction output (utxo) to create
the deposit:
c0426769a9327da01e0a9bdbb3dfb64b4f3e163cab4870178409452fd61dded2
please enter the output index of the utxo: 1
please enter the amount of Satoshi (1 Bitcoin = 100,000,000 Satoshi) to spent: 1000000
please enter the lock time for the deposit in days: 1
the deposit is locked until 1520420871 unix time. Please note down the time or otherwise
the deposit will become unspendable once the application has exited.
```

If everything is correct the deposit transaction is created and broadcasted to the Bitcoin network. The deposit transaction is a Pay-To-Script-Hash (P2SH) transaction. Therefore, only the hash of the script is stored in the transaction output and the script (including the lock time) has to be provided to be able to spent it. When the transaction was accepted by the Bitcoin network a success message will be displayed

```
transaction broadcast: success
```

followed by a summary of the deposit.

```
New Deposit (client id = 0):
transaction hash:
0c6e6654b94ff2a0d5ab7c59baaf567e052add3b2a69c934995b79114cde28a2
input transaction:
c0426769a9327da01e0a9bdbb3dfb64b4f3e163cab4870178409452fd61dded2
output adresses:
2NBzYA3xF1Kgw9ET9F1xRkKaQKb19fqaaif : 1000000 Satoshis
mpZfrRvDxSct69T77AgkRAF4LeJqQ5nyQ6 : 105850000 Satoshis
```

The first output address of the deposit (2NBz. . .) contains all locked funds. The second output (mpZf. . .) returns the remaining Satoshis back to the client and is not used in the protocol. Now the client sends the deposit to the provider and the provider verifies the deposit and hands out the initial state.

```
New initial state (provider):
expire time : 1520420871
satoshis limit: 500000
satoshis spent: 0
revision nr. : 0
client id : 0
deposit transaction hash:
0c6e6654b94ff2a0d5ab7c59baaf567e052add3b2a69c934995b79114cde28a2
```

Here the provider has set the spending limit to 500,000 Satoshis, which means that half of the funds in the deposit are used as a penalty for misbehaving. Thereafter the client can decide on what to do next and chooses to make a transaction.

```
Please choose an option:
1) make transactions
2) double-spent
3) close deposit
4) reclaim deposit
5) exit
```

The client chooses a point of sale and the amount of Satoshis he wants to pay and

```
please choose a point of sale (0 - 4): 1
please enter Satoshi (1 Bitcoin = 100,000,000 Satoshi) to spent: 100000
```

a new transaction is created accordingly and a summary printed.

```
New Transaction (client id = 0):
revision nr.: 0
random nr. :
3cffb4a986dc04fba006e922354e7a4a6adc6b7b4d0c3b91a335970092943706
input transaction:
0c6e6654b94ff2a0d5ab7c59baaf567e052add3b2a69c934995b79114cde28a2
output adresses:
miv8cV8pwU9CagsW7yMpqvHcUeh47YTM4W : 100000 Satoshis
mpZfrRvDxSct69T77AgkRAF4LeJqQ5nyQ6 : 890000 Satoshis
```

The first output address of the transaction (miv8...) is the address of the provider and the second one (mpZf...) of the client. The sum of these two outputs does not equate to the 1,000,000 Satoshis put into the deposit, because 10,000 Satoshis are paid to the miner as a transaction fee and are not included. The point of sale validates this transaction with the initial state of the provider and hands out a new state.

```
New State (point of sale id = 1):
expire time : 1520420871
satoshis limit: 500000
satoshis spent: 100000
revision nr. : 1
client id : 0
deposit transaction hash:
0c6e6654b94ff2a0d5ab7c59baaf567e052add3b2a69c934995b79114cde28a2
```

The client can now make transactions at other points of sale and at some point in time the provider will collect all transactions from the points of sale. If the client did not misbehave the provider takes the latest transaction, signs it and broadcasts it to the Bitcoin network. This transaction, like all transactions of this example, can be found on the blockchain of the Bitcoin testnet. The transaction of this particular example can be found here[5], where the client has spend additionally 100,000 Satoshis at other points of sale and the provider receives a total of 200,000 Satoshis. In the case the client double spends, the provider can extract the client's signing key and retrieve all funds of the deposit[6]. If the client does not make any transactions, all funds in the deposit can be transferred back to the client as shown here[7].

6.3 Transactions

In this section all Bitcoin transactions used in the application will be presented. There are three different types, which deviate from the standard Pay-To-Public-Key-Hash transaction, namely the transaction creating the deposit, the transaction paying the provider and the transaction, where the client retrieves the funds after the locktime has expired. The locking script of the output in the deposit transaction looks as follows

```
OP_IF
  OP_DUP  OP_HASH160
  <provider_address>
  OP_EQUALVERIFY  OP_CHECKSIGVERIFY
OP_ELSE
  <lock_time>
  OP_CHECKLOCKTIMEVERIFY
  OP_DROP
OP_ENDIF
OP_DUP  OP_HASH160
<client_address>
OP_EQUALVERIFY  OP_CHECKSIG
```

[5]https://live.blockcypher.com/btc-testnet/address/2NBzYA3xF1Kgw9ET9F1xRkKaQKb19fqaaif/
[6]https://live.blockcypher.com/btc-testnet/address/2NEbZRoe6GVFbqr3HRgLWWpC8J1QsCDAQms/
[7]https://live.blockcypher.com/btc-testnet/address/2Mu3GNYAd4JuyCieKhYTT6UubmkJQypj3jK/

This script requires either a signature of the provider and the client or the locktime to be expired and the signature of the client to evaluate successfully and thus make the output spendable. If the provider wants to receive funds, the provider takes a transaction signed be the client or extracts the client's secret key from a double spend and creates the following signature script

```
<client_signature>
<client_pubKey>
<provider_signature>
<provider_pubKey>
OP_1
```

Combined with the locking script, this is evaluated as shown in Figure 6.1. Like in the example from Chapter 3.3, data is put into angled brackets and is only pushed on the stack and operators are preceded by "OP". In the following only operators, which are not already covered in Chapter 3.3, will be described. The "OP_IF" operator checks if the top element of the stack is non-zero and if so, all operations of the if-branch are executed. "OP_0" and "OP_1" push zero respectively one on the stack. Therefore, it can be specified which branch shall be executed by adding either "OP_0" or "OP_1" to the signature script. Because the preceding "OP_IF"-branch was executed in this example, all operators from "OP_ELSE" to "OP_ENDIF" are skipped.

If the client wants to reclaim the funds of the deposit, the client waits until the locktime is expired and uses the signature script

```
<client_signature>
<client_pubKey>
OP_0
```

The processing of the signature in combination with the locking script is depicted in Figure 6.2. Here the operators in the if-branch are skipped and only the else-branch is executed. The "OP_CHECKLOCKTIMEVERIFY" operator compares the value on the top element of the stack with the actual locktime of the transaction. The whole validation process of the script will fail, if the locktime value of the transaction is smaller than the value of the top stack element. Because the top stack element is not removed from the stack by "OP_CHECKLOCKTIMEVERIFY", "OP_DROP" is used to do so.

Stack	<client_sig>	<client_pubKey> <c_sig>	<provider_sig> <c_pubKey> <c_sig>	<provider_pubKey> <p_sig> <c_pubKey> <c_sig>	1 <p_pubKey> <p_sig> <c_pubKey> <c_sig>	<p_pubKey> <p_sig> <c_pubKey> <c_sig>
Executed Command	<client_sig>	<client_pubKey>	<provider_sig>	<provider_pubKey>	OP_1	OP_IF

Stack	<p_pubKey> <p_pubKey> <p_sig> <c_pubKey> <c_sig>	<p_pubKeyHash> <p_pubKey> <p_sig> <c_pubKey> <c_sig>	<provider_address> <p_pubKeyHash> <p_pubKey> <p_sig> <c_pubKey> <c_sig>	<p_pubKey> <p_sig> <c_pubKey> <c_sig>	<c_pubKey> <c_sig>
Executed Command	OP_DUP	OP_HASH160	<provider_address>	OP_EQUALVERIFY	OP_CHECKSIGVERIFY

Stack	<c_pubKey> <c_sig>	<c_pubKey> <c_pubKey> <c_sig>	<c_pubKeyHash> <c_pubKey> <c_sig>	<client_address> <c_pubKeyHash> <c_pubKey> <c_sig>	<c_pubKey> <c_sig>	TRUE
Executed Command	OP_ELSE	OP_DUP	OP_HASH160	<client_address>	OP_EQUALVERIFY	OP_CHECKSIG

Figure 6.1: Evaluation of the script, where the signature of the provider and the client is used

Stack	<client_sig>	<client_pubKey> <client_sig>	0 <client_pubKey> <client_sig>	<client_pubKey> <client_sig>	<client_pubKey> <client_sig>
Executed Command	<client_sig>	<client_pubKey>	OP_0	OP_IF	OP_ELSE

Stack	<lock_time> <client_pubKey> <client_sig>	<client_pubKey> <client_sig>	<client_pubKey> <client_pubKey> <client_sig>	<lock_time> <client_pubKey> <client_sig>
Executed Command	OP_CHECKLOCKTIMEVERIFY	OP_DROP	OP_DUP	<lock_time>

Stack	<client_pubKeyHash> <client_pubKey> <client_sig>	<client_address> <client_pubKeyHash> <client_pubKey> <client_sig>	<client_pubKey> <client_sig>	TRUE
Executed Command	OP_HASH160	<client_address>	OP_EQUALVERIFY	OP_CHECKSIG

Figure 6.2: Evaluation of the script, where only the signature of the client is used, after the specified lock_time has expired

7 Conclusion

Cryptocurrencies see a gradual increase in public awareness and will most likely see a broader adoption in the future. One of the biggest problems a cryptocurrency has to solve is to prohibit the act of double spending. The most common solution to this problem is to use a public ledger, where everyone can verify the validity of all transactions made. A public ledger, however, requires some kind of consensus finding process, which largely impacts the time a transaction takes to be executed and confirmed. Therefore, payment channels have emerged, which reduce the amount of interaction with the public ledger to a minimum and allow for a fast payment processing. Additionally, they reduce the memory footprint on the public ledger as well as the load on the network of the cryptocurrency.

Very recently a new cryptographic primitive, called DAPS, has been developed, which makes it possible to penalize the act of making contradicting statements in a certain a context. Since the first publication [PS14; PS17] new DAPS constructions and designs have been made [RKS15; BPS17; BKN17; DRS18b; Poe18; DRS18a], which improve in signature and key sizes as well as computational cost. Additionally, several applications for DAPS have been proposed, which also include the use of cryptocurrencies. Double spending in a cryptocurrency can be seen as an act of making a contradicting statement and it seems only natural to use a DAPS to solve this issue. However, the key extracted from contradicting DAPS messages has to be compatible with the signing keys of the cryptocurrency. Thus, not all DAPS schemes can inherently be used to disincentivize double spending. Either way, it is possible to build an application with DAPS, to make offline transactions with a cryptocurrency possible. This can only be achieved in a constrained environment, because it is in fact not possible to prohibit someone to send the same digital data to an arbitrary amount of independent parties. Therefore, in an unconstrained environment, the parties would need some kind of synchronization to detect fraudulent behavior and to avoid using an absurdly high penalty.

7 Conclusion

The payment protocol presented in this thesis offers different trade-offs than the conventional payment channels of cryptocurrencies. The payment channels are bidirectional and completely trustless, but require a connection to the cryptocurrency network to function properly. The presented protocol only allows for unidirectional payments and some trust has to be put into the provider, but transactions can be performed completely offline. Interesting future work on this topic includes the development of a cryptocurrency with DAPS in mind, to for example create a hybrid consensus finding process with a combination of on-chain and off-chain/offline transactions or to further improve DAPS schemes, like for example finding a way to use signatures based on structured hardness assumptions in the DAPS construction from [DRS18a].

Bibliography

[01a] *FIPS PUB 180-2. Secure Hash Standard (SHA)*. https://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf. U.S. Department of Commerce/National Institute of Standards and Technology. 2001 (cit. on pp. 6, 28).

[01b] *FIPS PUB 197. Advanced Encryption Standard (AES)*. https://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. U.S. Department of Commerce/National Institute of Standards and Technology. 2001 (cit. on p. 11).

[15] *FIPS PUB 202. SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions*. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. U.S. Department of Commerce/National Institute of Standards and Technology. 2015 (cit. on p. 6).

[98] *FIPS PUB 186-1. Digital Signature Standard (DSS)*. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. U.S. Department of Commerce/National Institute of Standards and Technology. 1998 (cit. on p. 12).

[AGM18] Shashank Agrawal, Chaya Ganesh, and Payman Mohassel. “Non-Interactive Zero-Knowledge Proofs for Composite Statements.” In: *Advances in Cryptology – CRYPTO 2018*. Ed. by Hovav Shacham and Alexandra Boldyreva. Cham: Springer International Publishing, 2018, pp. 643–673. ISBN: 978-3-319-96878-0 (cit. on p. 45).

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. “Short Group Signatures.” In: *Advances in Cryptology – CRYPTO 2004*. Ed. by Matt Franklin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 41–55. ISBN: 978-3-540-28628-8 (cit. on p. 15).

[Ber+11] Guido Bertoni et al. *The Keccak reference*. https://keccak.team/files/Keccak-reference-3.0.pdf. 2011 (cit. on p. 28).

[Ber+12] Daniel J. Bernstein et al. “High-speed high-security signatures.” In: *Journal of Cryptographic Engineering* 2.2 (Sept. 2012), pp. 77–89. ISSN: 2190-8516. DOI: 10.1007/s13389-012-0027-1 (cit. on pp. 10, 12, 28).

[Ber08] Daniel J. Bernstein. *ChaCha, a variant of Salsa20*. https://cr.yp.to/chacha/chacha-20080128.pdf. 2008 (cit. on p. 11).

[BG89] Mihir Bellare and Shafi Goldwasser. “New Paradigms for Digital Signatures and Message Authentication Based on Non-interactive Zero Knowledge Proofs.” In: *Proceedings on Advances in Cryptology*. CRYPTO ’89. Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1989, pp. 194–211. ISBN: 0-387-97317-6 (cit. on p. 44).

[BKN17] Dan Boneh, Sam Kim, and Valeria Nikolaenko. “Lattice-Based DAPS and Generalizations: Self-enforcement in Signature Schemes.” In: *Applied Cryptography and Network Security*. Ed. by Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi. Cham: Springer International Publishing, 2017, pp. 457–477. ISBN: 978-3-319-61204-1 (cit. on pp. 41, 42, 48, 69).

[BPS17] Mihir Bellare, Bertram Poettering, and Douglas Stebila. “Deterring Certificate Subversion: Efficient Double-Authentication-Preventing Signatures.” In: *Public-Key Cryptography – PKC 2017*. Ed. by Serge Fehr. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 121–151. ISBN: 978-3-662-54388-7 (cit. on pp. 41, 42, 69).

[BR96] Mihir Bellare and Phillip Rogaway. “The Exact Security of Digital Signatures-How to Sign with RSA and Rabin.” In: *Advances in Cryptology — EUROCRYPT ’96*. Ed. by Ueli Maurer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 399–416. ISBN: 978-3-540-68339-1 (cit. on p. 10).

[Bro05] Daniel R. L. Brown. “Generic Groups, Collision Resistance, and ECDSA.” In: *Designs, Codes and Cryptography* 35.1 (Apr. 2005), pp. 119–152. ISSN: 1573-7586. DOI: 10.1007/s10623-003-6154-z. URL: https://doi.org/10.1007/s10623-003-6154-z (cit. on p. 14).

[But14] Vitalik Buterin. “A Next-Generation Smart Contract and Decentralized Application Platform.” In: 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper (cit. on p. 28).

[CFN90] David Chaum, Amos Fiat, and Moni Naor. “Untraceable Electronic Cash.” In: *Advances in Cryptology — CRYPTO’ 88*. Ed. by Shafi Goldwasser. New York, NY: Springer New York, 1990, pp. 319–327. ISBN: 978-0-387-34799-8 (cit. on p. 40).

[DBP96] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. “RIPEMD-160: A Strengthened Version of RIPEMD.” In: *FSE*. Ed. by Dieter Gollmann. Vol. 1039. LNCS. Springer, 1996, pp. 71–82. ISBN: 3-540-60865-6. URL: https://homes.esat.kuleuven.be/~bosselae/ripemd160/pdf/AB-9601/AB-9601.pdf (cit. on p. 31).

[DH76] W. Diffie and M. Hellman. “New directions in cryptography.” In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638 (cit. on p. 12).

[DRS18a] David Derler, Sebastian Ramacher, and Daniel Slamanig. “Generic Double-Authentication Preventing Signatures and a Post-quantum Instantiation.” In: *Provable Security - 12th International Conference, ProvSec 2018, Jeju, South Korea, October 25-28, 2018, Proceedings*. 2018, pp. 258–276. DOI: 10.1007/978-3-030-01446-9_15. URL: https://doi.org/10.1007/978-3-030-01446-9%5C_15 (cit. on pp. 41, 43, 45, 69, 70).

[DRS18b] David Derler, Sebastian Ramacher, and Daniel Slamanig. “Short Double- and N-Times-Authentication-Preventing Signatures from ECDSA and More.” In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. 2018, pp. 273–287. DOI: 10.1109/EuroSP.2018.00027. URL: https://doi.org/10.1109/EuroSP.2018.00027 (cit. on pp. 40–43, 48, 69).

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. “A Verifiable Random Function with Short Proofs and Keys.” In: *Public Key Cryptography - PKC 2005*. Ed. by Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 416–431. ISBN: 978-3-540-30580-4 (cit. on p. 45).

[Elg85] T. Elgamal. “A public key cryptosystem and a signature scheme based on discrete logarithms.” In: *IEEE Transactions on Information Theory* 31.4 (July 1985), pp. 469–472. ISSN: 0018-9448. DOI: 10.1109/TIT.1985.1057074 (cit. on pp. 12, 14).

[Fel87] Paul Feldman. “A Practical Scheme for Non-interactive Verifiable Secret Sharing.” In: *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. SFCS ’87. Washington, DC, USA: IEEE Computer Society, 1987, pp. 427–438. ISBN: 0-8186-0807-2. DOI: 10.1109/SFCS.1987.4. URL: https://doi.org/10.1109/SFCS.1987.4 (cit. on p. 21).

[FKP16] Manuel Fersch, Eike Kiltz, and Bertram Poettering. “On the Provable Security of (EC)DSA Signatures.” In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 1651–1662. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978413. URL: http://doi.acm.org/10.1145/2976749.2978413 (cit. on p. 14).

[FS87] Amos Fiat and Adi Shamir. “How To Prove Yourself: Practical Solutions to Identification and Signature Problems.” In: *Advances in Cryptology — CRYPTO’ 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194. ISBN: 978-3-540-47721-1 (cit. on pp. 18, 42).

[GMR85] 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. Providence, Rhode Island, USA: ACM, 1985, pp. 291–304. ISBN: 0-89791-151-2. DOI: 10.1145/22145.22178. URL: http://doi.acm.org/10.1145/22145.22178 (cit. on p. 17).

[GQ90] Louis Claude Guillou and Jean-Jacques Quisquater. “A “Paradoxical” Indentity-Based Signature Scheme Resulting from Zero-Knowledge.” In: *Advances in Cryptology — CRYPTO’ 88*. Ed. by Shafi Goldwasser. New York, NY: Springer New York, 1990, pp. 216–231. ISBN: 978-0-387-34799-8 (cit. on p. 42).

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. “The Elliptic Curve Digital Signature Algorithm (ECDSA).” In: *Int. J. Inf. Secur.* 1.1 (Aug. 2001), pp. 36–63. ISSN: 1615-5262. DOI: 10.1007/s102070100002. URL: http://dx.doi.org/10.1007/s102070100002 (cit. on pp. 10, 12, 28).

[Kat10] Jonathan Katz. *Digital Signatures*. Springer, 2010 2010, p. 192. ISBN: 9780387277110. DOI: 10.1007/978-0-387-27712-7 (cit. on p. 5).

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. ISBN: 9781466570269 (cit. on p. 5).

[Lam79] Leslie Lamport. “Constructing Digital Signatures from a One Way Function.” In: This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. Oct. 1979 (cit. on p. 41).

[Mer89] Ralph C. Merkle. “A Certified Digital Signature.” In: *Proceedings on Advances in Cryptology*. CRYPTO ’89. Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1989, pp. 218–238. ISBN: 0-387-97317-6 (cit. on pp. 34, 41).

[MJ16] Nadia El Mrabet and Marc Joye. *Guide to Pairing-Based Cryptography*. Chapman & Hall/CRC, 2016. ISBN: 1498729509, 9781498729505 (cit. on p. 15).

[MO12] Atefeh Mashatan and Khaled Ouafi. “Forgery-resilience for digital signature schemes.” In: *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS ’12, Seoul, Korea, May 2-4, 2012*. 2012, pp. 24–25. DOI: 10.1145/2414456.2414469 (cit. on p. 41).

[MR02] Silvio Micali and Leonid Reyzin. “Improving the exact security of digital signature schemes.” In: *Journal of Cryptology* 15.1 (Mar. 2002), pp. 1–18. ISSN: 1432-1378. DOI: 10.1007/s00145-001-0005-8 (cit. on p. 42).

[Nak09] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system.” In: 2009. URL: https://www.bitcoin.org/bitcoin.pdf (cit. on pp. 1, 28, 30).

[Nar+16] Arvind Narayanan et al. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton University Press, 2016. ISBN: 0691171696, 9780691171692. URL: http://bitcoinbook.cs.princeton.edu/ (cit. on pp. 23, 30).

[PD16] Joseph Poon and Thaddeus Dryja. “The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.” In: 2016. URL: https://lightning.network/lightning-network-paper.pdf (cit. on pp. 3, 30).

[Poe18] Bertram Poettering. “Shorter Double Authentication Preventing Signatures for Small Address Spaces.” In: *AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*. 2018, pp. 344–361. DOI: 10.1007/978-3-319-89339-6_19. URL: https://doi.org/10.1007/978-3-319-89339-6%5C_19 (cit. on pp. 41, 42, 69).

[PS14] Bertram Poettering and Douglas Stebila. “Double Authentication Preventing Signatures.” In: *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I*. 2014, pp. 436–453. DOI: 10.1007/978-3-319-11203-9_25 (cit. on pp. 37, 38, 40–42, 47, 69).

[PS17] Bertram Poettering and Douglas Stebila. “Double Authentication Preventing Signatures.” In: *International Journal of Information Security* 16.1 (Feb. 2017), pp. 1–22. ISSN: 1615-5270. DOI: 10.1007/s10207-015-0307-8 (cit. on pp. 37, 41, 42, 47, 69).

[Riv92] Ronald L. Rivest. “The MD5 Message-Digest Algorithm.” In: *RFC* 1321 (1992), pp. 1–21. DOI: 10.17487/RFC1321 (cit. on p. 7).

[RKS15] Tim Ruffing, Aniket Kate, and Dominique Schröder. “Liar, Liar, Coins on Fire!: Penalizing Equivocation By Loss of Bitcoins.” In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. (Denver, Colorado, USA). CCS ’15. New York, NY, USA: ACM, 2015, pp. 219–230. ISBN: 9781450338325. DOI: 10.1145/2810103.2813686. URL: https://crypsys.mmci.uni-saarland.de/projects/PenalizingEquivocation/penalizing.pdf (cit. on pp. 2, 41–43, 47, 48, 54, 69).

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems.” In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342 (cit. on p. 12).

[Sch90] Claus-Peter Schnorr. “Efficient Identification and Signatures for Smart Cards.” In: *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO ’89. Berlin, Heidelberg: Springer-Verlag, 1990, pp. 239–252. ISBN: 3-540-97317-6 (cit. on p. 17).

[Sha79] Adi Shamir. "How to Share a Secret." In: *Commun. ACM* 22.11 (Nov. 1979), pp. 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: http://doi.acm.org/10.1145/359168.359176 (cit. on p. 20).

[Sma15] Nigel P. Smart. *Cryptography Made Simple*. 1st. Springer Publishing Company, Incorporated, 2015. ISBN: 9783319219356. DOI: 10.1007/978-3-319-21936-3 (cit. on p. 5).

[SYB14] David Schwartz, Noah Youngs, and Arthur Britto. "The Ripple Protocol Consensus Algorithm." In: 2014. URL: https://ripple.com/files/ripple_consensus_whitepaper.pdf (cit. on pp. 3, 28).

[Tal00] Hugo Krawczyk and Tal Rabin. "Chameleon Signatures." In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA*. 2000 (cit. on p. 41).

[WP89] Michael Waidner and Birgit Pfitzmann. "The Dining Cryptographers in the Disco - Underconditional Sender and Recipient Untraceability with Computationally Secure Serviceability (Abstract)." In: *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*. Vol. 434. Lecture Notes in Computer Science. Springer, 1989, p. 690. DOI: 10.1007/3-540-46885-4_69 (cit. on p. 41).

[WY05] Xiaoyun Wang and Hongbo Yu. "How to Break MD5 and Other Hash Functions." In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 19–35. ISBN: 978-3-540-32055-5 (cit. on p. 7).