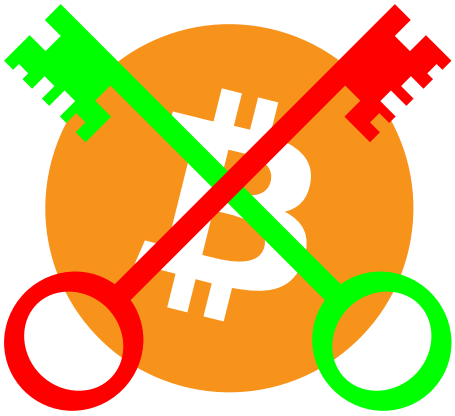
Masterthesis

Two-factor authentication for the Bitcoin protocol



7 May 2015

Author:
Christopher Mann
Student ID 2559142

Referees:
Prof. Dr. Joachim von zur Gathen
B-IT

Advisor:
Dr. Daniel Loebenberger
B-IT

Prof. Dr. Matthew Smith
Universität Bonn

Abstract

In this thesis, we present a two-party ECDSA signature protocol, which is based on the two-party DSA signature protocol by MacKenzie & Reiter (2004). Afterwards, we use the two-party ECDSA signature protocol to implement a Bitcoin wallet with two-factor authentication. We then compare our prototype to Bitcoin's built-in support for two-factor authentication and show that it is efficient enough for productive use. Furthermore, we show that our solution is transparent to the Bitcoin network.

Zusammenfassung

Im Rahmen dieser Masterarbeit wurde, basierend auf dem Zweiparteienprotokol zur Erstellung von DSA-Signaturen von MacKenzie & Reiter (2004), ein Zweiparteienprotokol zur Erstellung von ECDSA-Signaturen entwickelt. Mit Hilfe dieses Protokolls wurde anschließend eine Bitcoinwallet mit Zweifaktorauthentifizierung implementiert. Die hier entwickelte Wallet hat gegenüber anderen bereits existierenden Lösungen den Vorteil, dass sie für das Bitcoinnetzwerk völlig transparent ist. Außerdem erzeugt sie kleinere Transaktionen als andere Lösungen zur Zweifaktorauthentifizierung. Dadurch fallen für den Benutzer geringere Transaktionsgebühren an.

Selbstständigkeitserklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Preface

When I started with my research at the end of 2013, there was quite a hype around Bitcoin and a single Bitcoin was worth more than 1000$. At the same time, only rudimentary security solutions were available for Bitcoin users to protect their Bitcoins. I quickly developed the idea to implement two-factor authentication for the Bitcoin protocol similar to the existing solutions for online banking.

Other researchers and members of the Bitcoin community worked on this problem at the same time. Goldfeder *et al.* published their solution, which is based on a different threshold signature scheme, when I already had a first running proof of concept. Other solutions like *BitcoinAuthenticator* by Pacia & Muroch also became available.

Together with my advisor Dr. Daniel Loebenberger, we decided to publish an article (Mann & Loebenberger (2014)) with the results first. This thesis is a greatly extended version of the article. I would like to thank Daniel for the great support during the year-long process of work. I also thank Dr. Michael Nüsken for useful input provided in several discussions and I thank my referees Prof. Dr. Joachim von zur Gathen and Prof. Dr. Matthew Smith for enabling me to work on this very interesting topic. Finally, I thank Mike Hearn and Chris Pacia for the feedback on the article and for showing me that my work is of some interest for the Bitcoin community. Especially, Mike Hearn's hints on how to improve the performance of the prototype were very useful.

Christopher Mann
Kerpen, 7 May 2015

Contents

1. Introduction

Bitcoin (BTC) is a cryptographic currency proposed by Satoshi Nakamoto (2008) in the legendary email to the Cryptography Mailing list at `metzdowd.com`. One of the most important features of Bitcoin is that it is completely peer-to-peer, i.e. it does not rely on a trusted authority (the bank) which ensures that the two central requirements of any electronic cash system are met: Only the owner can spend money and it is impossible to spend money twice. In Bitcoin, these two features are realized with a common transaction history, the Bitcoin *block-chain*, known to all users. Each of the transactions in the chain contains the address to which some Bitcoins should be payed, the address from which the Bitcoins should be withdrawn and the amount of Bitcoins to be transferred. Both addresses are directly derived from the public key of the corresponding elliptic curve digital signature algorithm (ECDSA) key pairs of the recipient and the sender respectively. The whole transaction is then signed using the ECDSA private key of the sender. We describe the details in Section 2. Since any user might have multiple addresses, their *wallet* consists of several key-pairs and is typically stored on the owner's device or within some online service. Thus, from a thieves perspective, the only thing one has to do in order to steal some Bitcoins, is to get one's hands on the corresponding wallet, just like in real life. Indeed, Lipovsky (2013) describes an online banking trojan that also steals Bitcoin wallets.

Several solutions to protect the user's wallet already exist. Bitcoin-qt, the standard Bitcoin client, allows to encrypt the wallet file with a password. This makes stealing the wallet file useless, but a sophisticated malware could still break the encryption by key logging or extracting the encryption password or the private keys from the memory. There are also online wallets, which store the key pairs for their users. These websites offer two-factor authentication for the login. For example, one-time passwords sent via SMS are used. All these solutions have one problem in common: there is a single point of failure where the full private key is handled. If the attacker is able to compromise this single point of failure, he will have access to the private keys and therefore will be able to spend the victim's Bitcoins.

There are only two possibilities to remove the single point of failure:

1. The Bitcoin protocol itself is extended, so that it requires several ECDSA signatures to authorize a transaction. In this case, each party just holds one of the private keys.

2. The private key is shared between several parties in such a way that no party will learn anything about the secret shares of the other parties. The parties can then execute a multiparty ECDSA signature protocol to create a signature without revealing their secret shares to any of the other parties.

The first possibility has already been implemented in the Bitcoin protocol, but as the protocol has been extended, this solution is not transparent to the other Bitcoin users and indeed requires changes to their software. Furthermore, the transaction size increases because each transaction now contains several ECDSA signatures.

The second solution requires a multiparty ECDSA signature protocol. There was already considerable research effort in the area of threshold ECDSA signatures (see Section 3.2) and Goldfeder *et al.* (2014) tried to apply threshold signatures proposed by Ibrahim *et al.* (2003) to the Bitcoin protocol. Unfortunately, the used threshold signature protocol only works with three or more parties.

The optimal solution, which offers a high usability to the user, would be a Bitcoin wallet with two-factor authentication where the private key is shared between two different devices. This requires a two-party ECDSA signature protocol, which allows to share the ECDSA private key between two parties. As far as we know no multiparty ECDSA signature protocol exists, which allows the number of parties to be two. In MacKenzie & Reiter (2004), a two-party protocol for DSA has been presented. Some portions of this protocol have been used in Drake (2011) to implement a prototype for two-party ECDSA signatures for Bitcoin. Unfortunately, the authors provide no documentation besides the Javascript source code and have removed significant parts from the protocol in MacKenzie & Reiter (2004) without any justification.

The contributions made in this thesis are twofold. First, we adapted the protocol in MacKenzie & Reiter (2004) to ECDSA in such a way that the security guarantees made by MacKenzie & Reiter still hold. Our protocol seems to be the first two-party ECDSA signature protocol with a reasonable security analysis. Second, we use our protocol to implement a prototypic Bitcoin wallet which offers two-factor authentication. The prototype consists of a computer part and a smart phone part which interact over the network. This makes our solution easy to deploy as most users will already own both devices.

Our prototype has some unique advantages over the existing solutions. First of all, there is no single point of failure. The private ECDSA key is shared between the computer and the phone in a secure way. Second, our solution is completely transparent to the Bitcoin network. A transaction created by our wallet looks exactly as any other standard Bitcoin transaction and a third party cannot even tell whether the wallet is protected with our two-factor authentication or not. Furthermore, we show that our solution is efficient enough to be usable in real life. Indeed, signing a single transaction will take roughly 4 seconds in most cases when using a modern smart phone.

2. Bitcoin protocol

In this chapter, we will explain some of the technical details of the Bitcoin protocol as described by Nakamoto (2008). In this article, two major requirements are identified for an electronic cash scheme:

1. Only the owner can spend his coins.
2. It is not possible to spend coins twice.

In difference to other e-cash schemes such as the one proposed by Chaum *et al.* (1990) and many others, Bitcoin was designed to be completely decentral. The Bitcoin network consists of a large number of nodes which verify incoming transactions independently of each other. These nodes use a synchronization protocol which is based on a proof-of-work similar to the *hashcash* system described in Back (2002). With the help of this protocol, the nodes agree on a common transaction history, which is called the Bitcoin *block chain*.

The example in Figure 2.1 illustrates how Bitcoin works from a client's perspective when a participant Alice wants to pay 0.1 Bitcoin to Bob. A

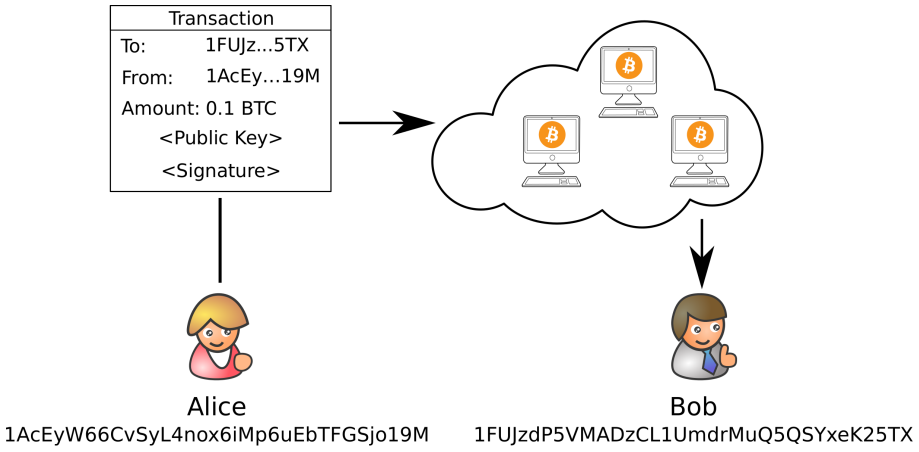


Figure 2.1: How a Bitcoin transaction works

Bitcoin transaction contains the address to which the Bitcoins should be payed, the address from which the Bitcoins should be withdrawn and the amount of Bitcoins to be transferred. Furthermore, the transaction contains a digital signature, which authorizes it, and the public key needed to verify the signature. Bitcoin uses the ECDSA signature scheme, specified by the Accredited Standards Committee X9 (2005) on the elliptic curve `secp256k1` as defined by Certicom Research (2000). All Bitcoin transactions must be correctly signed by the spender. In order to bind Bitcoin addresses and the public keys, the Bitcoin address of a user is directly derived from the user's public key by applying a cryptographic hash function to it (see Figure 2.2).

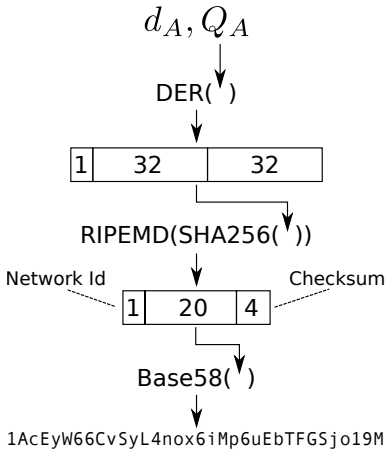


Figure 2.2: Derivation of a Bitcoin address from an ECDSA key pair (d_A, Q_A).

The hash function is only applied as to have a shorter string as address rather than the whole public key as an ECDSA public key is quite long. The whole address, including a byte identifying the Bitcoin network to which the address belongs and a checksum to prevent transmission and especially typing errors, is 25 bytes long. These bytes are then encoded using Base58 encoding, which works like Base64 encoding, except that the target character set does not contain any characters which could be easily misread as other characters. For example, lower case l is excluded as it looks very similar to the digit 1 in some fonts. In conclusion, a Bitcoin transaction is correctly signed if and only if the ECDSA signature of the transaction verifies under the provided public key and the provided public key yields the Bitcoin address from which is withdrawn (the from field in Figure 2.1) when applying the address derivation as in Figure 2.2.

The technical details are more complex than this. Figure 2.3 shows the actual structure of a Bitcoin transaction. (This illustration still omits some additional fields, but these are not relevant for the standard use case of just transferring Bitcoins from one address to another.) Any Bitcoin transaction actually consists of multiple inputs and outputs. Each output specifies a target address and an amount of Bitcoins to be transferred to the given address. Every input contains the hash of a preceding transaction and an index. Both values together unambiguously identify an output of a preceding transaction. All Bitcoins from this referenced output are spent by the current transaction. Consequently, every transaction output is only used a single time as an input and is completely spent at this time. This increases the efficiency of the network nodes as these only need to keep track of the unspent outputs instead of all transactions having an impact on the balance of the address. Furthermore, any input contains a signature and a public

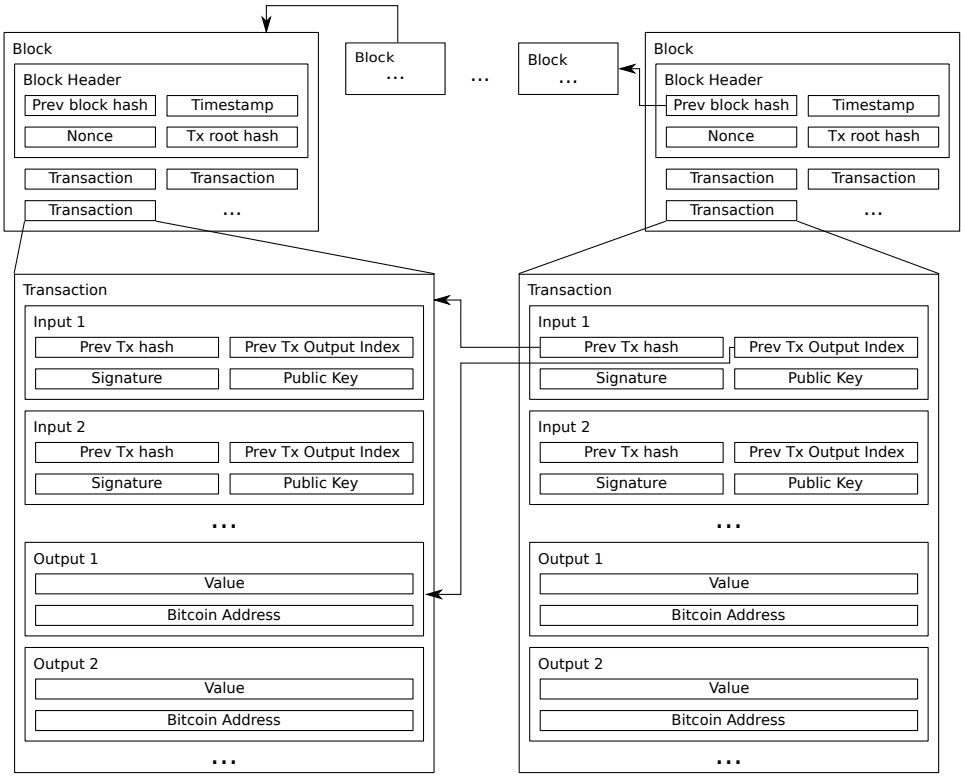


Figure 2.3: Detailed view of a Bitcoin transaction. (This illustration still omits some additional fields, but these are irrelevant for productive use.)

key, which must fit the address contained in the output referenced by this input. In consequence, if multiple inputs are used, multiple signatures of the transaction must be created, one for each input.

Clearly, the sum of the Bitcoins from all inputs must be greater than or equal to the sum of the Bitcoins spent by the outputs. If the sum of the inputs is greater, there is no problem. Any unused Bitcoins are transferred as a fee to the miner of the block containing this transaction and increase the miner's revenue. Therefore, this will increase the priority of the transaction as the miners will have an incentive to include it into a block.

For ease of exposition, we omitted the fact that Bitcoin uses a scripting language for transactions: In reality, a transaction does not really include a target address or a signature and a public key, but scripts which contain these as constants. Currently, only a very limited subset of the scripting functionality is actively used in the Bitcoin network and almost all transactions that currently occur use a standard set of scripts which behave exactly as described above and correspond to the standard use case of transferring Bitcoins from one address to another. Therefore, it is quite safe to ignore the scripting functionality and stay with the abstracted description provided

here. Nevertheless, we will describe the scripting in more detail as it was relevant for the implementation and knowledge of it is required to understand Bitcoin's built-in threshold signature support.

2.1. Scripting and transaction verification. As mentioned before, there are no fields for public key, signature and target address in the transaction outputs and inputs. Instead, each input or output just contains a script in Bitcoin's own scripting language and the aforementioned values are just stored as constants inside the scripts. The Bitcoin scripting language uses a syntax similar to Forth (Technical Committee X3J14 (1994)) and is a stack-based language. The feature set is highly restricted and the language is not Turing-complete. Indeed, it does not even contain any possibility to formulate loops. This is important as the scripts are executed by each node in the Bitcoin network when verifying transactions. If scripts with a significant execution time were allowed, denial-of-service attacks on the network would become possible.

In the following, we will not describe the full instruction set, but we will illustrate how the scripting works for standard transactions. A complete list of instructions can be found in Bitcoin Wiki (2014). In Figure 2.4, the example of Alice paying Bob is shown again, but this time with scripting in place. Each transaction input and each transaction output contains a script. The script in a transaction output specifies the conditions which must be met to spend this output. The script in the transaction input on the other hand is defined by the spender and usually just consists of constants which act as inputs to the script in the transaction output.

When a Bitcoin node verifies a transaction, it takes the script from each transaction input and concatenates it with the script from the transaction output that is spent by the transaction input. The result of the concatenation in our example is displayed at the bottom of Figure 2.4. The verification of a transaction input is successful if and only if the stack only contains a single true value after the execution of the script.

In Figure 2.5, the execution of the script from Figure 2.4 is shown in detail. The script consists of three major parts. The first part is the script of the transaction input and just contains the inputs provided by the spending user. The second part is responsible for checking whether the public key provided by the spending user fits the Bitcoin address from which the Bitcoins are to be spent. The last part of the script only consists of the `OP_CHECKSIG` instruction, which verifies that the signature provided by the spending user is valid for this transaction and the provided public key. Step-by-step, the script does the following:

a. In the beginning, the script is loaded into the instruction stack and the data stack is empty.

b. The signature and the public key provided by the spending user are

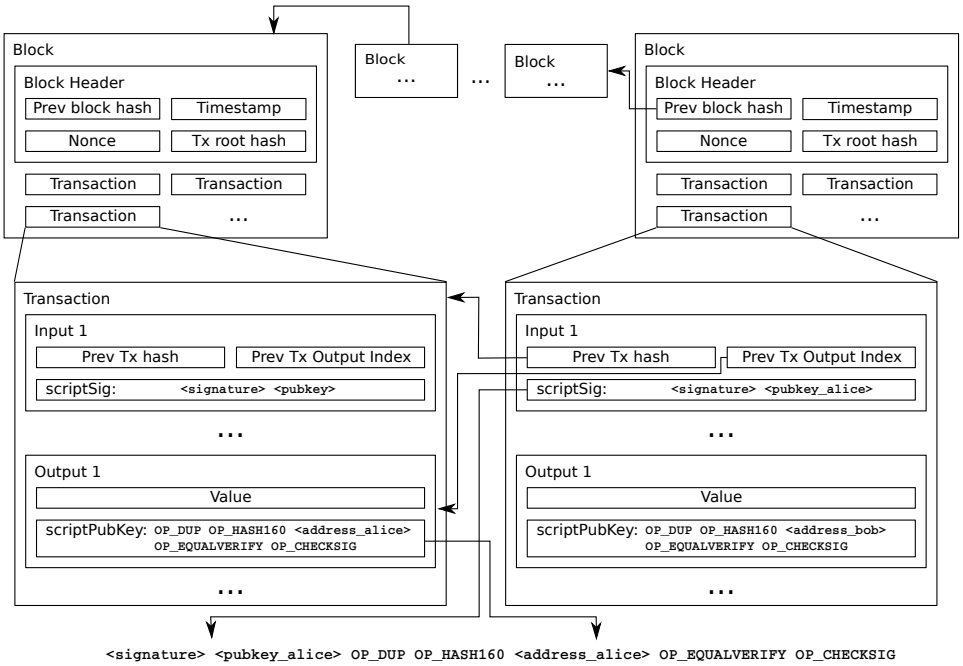


Figure 2.4: A Bitcoin transaction in detail with the scripts

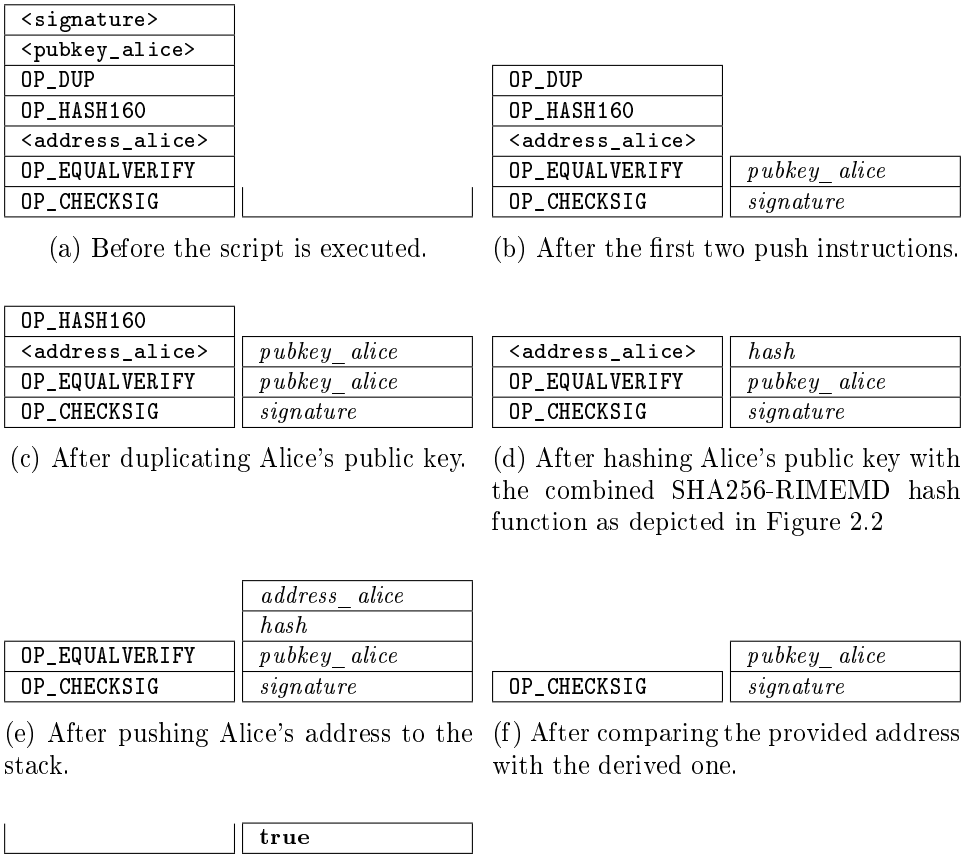
pushed onto the stack. The notation `<value>` is a short hand for the instruction `OP_PUSHDATA value`, where value is a number of constant bytes in the script which are pushed onto the stack.

c. `OP_DUP` duplicates the value on the top of the stack. As a result, the public key exists twice on the stack.

d. `OP_HASH160` applies the combined SHA256-RIPEMD hash function, depicted in Figure 2.2, to the top value of the stack. In our case, the public key is hashed with the combined hash function and the result address is pushed onto the stack.

e. The `<address_alice>` constant refers not to the Base58 encoded address but to the binary representation of the raw hash value outputted by the combined SHA256-RIPEMD hash function. The Base58 encoding and also the checksum and the network id are not used inside the transactions, but only to ensure the correct usage of the address by the user.

f. `OP_EQUALVERIFY` checks whether the two topmost values on the stack are equal. If they are not equal, the script execution is aborted with an error. In this case, the two top-most values on the stack are the Bitcoin address stored in the script of the spent output and the Bictoin address derived from the public key provided by the spending user in



(a) Before the script is executed.

(b) After the first two push instructions.

(c) After duplicating Alice's public key.

(d) After hashing Alice's public key with the combined SHA256-RIMEMD hash function as depicted in Figure 2.2

(e) After pushing Alice's address to the stack.

(f) After comparing the provided address with the derived one.

(g) After verifying the signature.

Figure 2.5: Step-by-step execution of a Bitcoin script.

the script of the transaction input. It is now checked, whether these two values are equal. Hence, this is the aforementioned check which aims to examine whether the public key provided by the spending user fits the Bitcoin address from which the Bitcoins are to be spent.

g. `OP_CHECKSIG` verifies a signature with a public key. Both the signature and the public key are popped from the stack. In our case, these are the signature and the public key provided by the spending user. `OP_CHECKSIG` now hashes the transaction and then checks whether the provided signature is valid under the provided public key. If this is the case, a `TRUE` is pushed on the stack and otherwise a `FALSE`.

The `OP_CHECKSIG` instruction The `OP_CHECKSIG` instruction should be discussed in more detail as it actually includes some quite complex operations. It verifies the provided signature of the current transaction under

the provided public key. To verify a signature, we need to know the exact message that has been signed, the used cryptographic hash function and the used signature scheme. Bitcoin uses SHA256 as a hash function and ECDSA as a signature scheme. The signed message is actually not the whole current transaction, but a modified copy of it. The copy is created as shown in Figure 2.6. The script fields of all inputs of the current transaction are emptied. Then, the input script field of the currently verified input is filled with the script of the connected output and in the end, all length fields are adjusted to the new contents. It makes sense to empty the input scripts as these contain the signatures which can only be computed after hashing the transaction, but as far as we know there is no rationale for copying the script of the connected output into the input. The final version of the transaction copy is hashed with SHA256 and then it is checked whether the given signature is a valid ECDSA signature for the computed hash value under the provided public key.

2.2. ECDSA. In this section, we describe the Elliptic Curve Digital Signature Algorithm (ECDSA), which is used by Bitcoin for digital signatures of transactions. ECDSA has been standardized by the Accredited Standards Committee X9 (2005) and is a variant of the ElGamal signature scheme on elliptic curves. The scheme works as follows:

Setup Fix the required parameters.

1. Choose a cryptographic hash function h.
2. Choose a prime power $q \in \mathbb{N}_{\geq 2}$ which denotes the size of the base field.
3. Choose the elliptic curve parameters $a, b \in \mathbb{F}_q$. The resulting elliptic curve is $E\colon y^2 = x^3 + ax + b$.
4. Choose a base point $G \in E$ of prime order $n \in \mathbb{N}$.

Additionally, we can define the cofactor $h = \#E/n \in \mathbb{N}$. This cofactor should be rather small. A cofactor of $h = 1$ is preferable and has the advantage, that every point on E is also a member of the group generated by G. Hence, to check group membership it is sufficient to confirm that the point satisfies the curve equation.

The described setup is not a sufficient guide to generating secure, easy-to-use (for implementers) and efficient elliptic curve parameters, but merely an outline of the basic requirements to make the scheme work. Bernstein (2006) and Bernstein *et al.* (2014) discuss in great detail their choices for a set of elliptic curve parameters and provide an extensive list of side requirements for elliptic curve parameters.

In the case of Bitcoin, the ECDSA parameters are fixed by the definition of the Bitcoin protocol. The used hash function h is SHA256 as

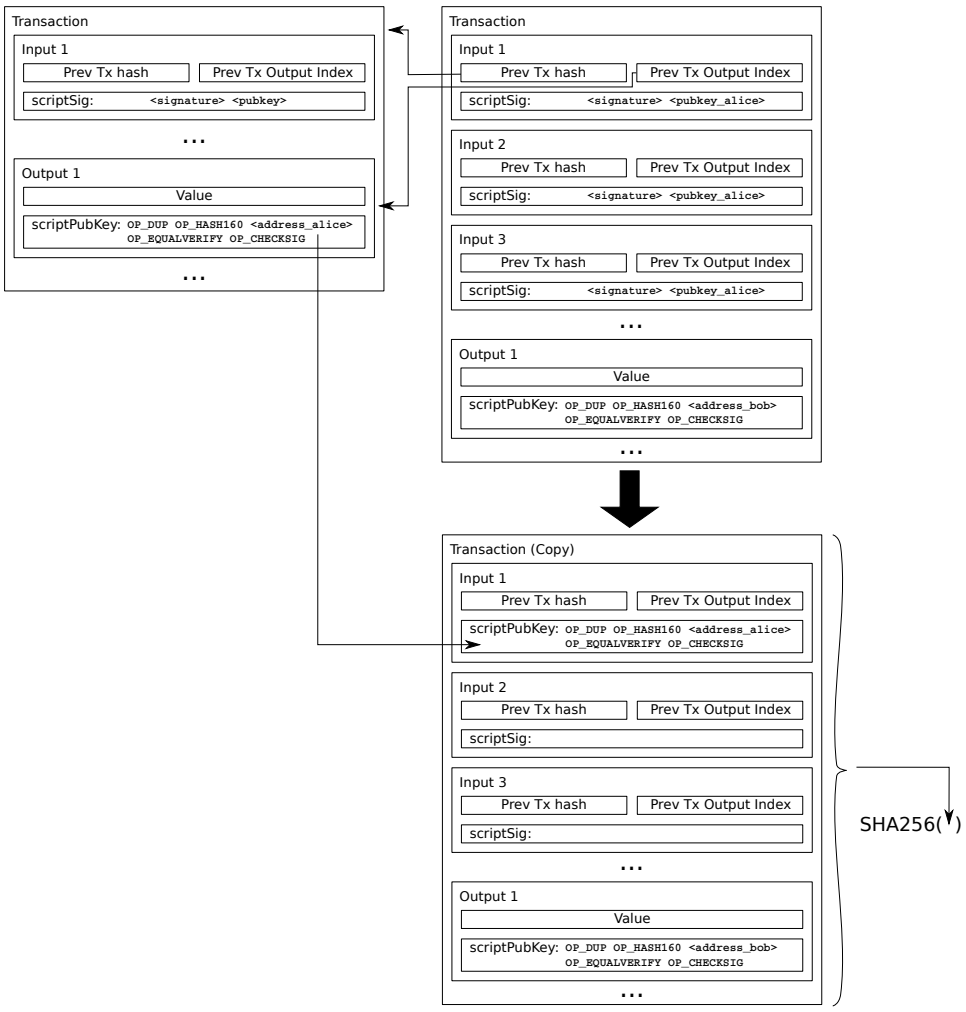


Figure 2.6: Computing the hash value of a transaction for the signature required for the first input.

defined in NIST (2012) and the elliptic curve is `secp256k1` as defined in Certicom Research (2000). In case of `secp256k1`: $a = 0$, $b = 7$ and q and n are large primes. The curve `secp256k1` has a cofactor of 1.

Key generation Generate a key pair consisting of the private key $d \in \mathbb{Z}_n^\times$ and the public key $Q \in E$.

1. Choose a $d \in \mathbb{Z}_n^\times$ uniformly at random.
2. Set $Q = dG$ on the elliptic curve E.

Signing Create a signature for the message $m \in \{0,1\}^*$.

1. Choose $k \in \mathbb{Z}_n^\times$ uniformly at random.
2. Set $R = kG$.
3. Set $r = \mathrm{coord_x}(R) \bmod n$. If $r = 0$, start again at 1.
4. Compute $s = k^{-1}(\mathrm{h}(m) + rd) \bmod n$. If $s = 0$, start again at 1.

The signature for the message m is the pair (r, s).

Verification Verify the signature (r, s) for the message m.

1. Check that $r, s \in \mathbb{Z}_n^\times$.
2. Compute $w = s^{-}1 \bmod n$.
3. Compute $u_1 = \mathrm{h}(m)\, w \bmod n$ and $u_2 = rw \bmod n$.
4. Compute $R' = u_1 G + u_2 Q$.
5. Check that $r \equiv_n \mathrm{coord_x}(R')$.

It is important to note, that ECDSA requires much shorter key sizes than DSA for the same level of security. Furthermore, any points on the elliptic curve can be stored in a compressed way. Any elliptic curve $E\colon y^2 = x^2 + ax + b$ is symmetric with respect to the x-axis and consequently for any value of x, there are at most two points on the curve with this value of x. Therefore, it is sufficient to store just the x coordinate and a single bit, telling whether the y coordinate is "positive" or "negative". The y coordinate itself can then be retrieved by solving the equation defining E for y.

Like any ElGamal signature scheme, ECDSA uses an ephemeral secret k during the signing process. It is crucial, that different values of k are used when creating signatures for different messages. If an attacker learns about two different signatures $(r, s), (r, s')$ for two messages m, m' with $m \neq m'$ which use the same ephemeral secret k, the attacker can easily retrieve the private key d by computing

$$k = \frac{\mathrm{h}(m) - \mathrm{h}(m')}{s - s'}$$

and then

$$d = \frac{sk - \mathrm{h}(m)}{r}.$$

Two different approaches can be taken to ensure that different values of k are used for different messages. The older and more natural approach is to choose $k \in \mathbb{Z}_n^\times$ uniformly at random. The chance to pick the same value of k twice is negligible if we really choose uniformly at random. Unfortunately, in the real world only a pseudo random number generator (PRNG) is used to generate k. If the PRNG fails to generate good random values of k, this is fatal for the security of the signature scheme. Problems with the PRNG have already occcured in the Bitcoin world and Bitcoin users experienced financial losses because of this. The error happened in several Bitcoin wallets for Android, which created multiple transaction signatures with the same value of k. These wallets used the underlying PRNG of Android, which was not correctly seeded. The flaw of the random number generator is explained in Kim *et al.* (2013) and its use to exploit Bitcoin in Klyubin (2013).

The second approach to generate the ephemeral secret is to derive it from the message m. As the ephemeral secret must be secret, it must also depend on some private knowledge of the signer. Consequently, the ephemeral secret is derived from the message m and the private key d with the help of a cryptographic hash function. One possible derivation is described in Pornin (2013). Using this method will generate the same value of k for the same message, but this is not a problem for ECDSA.

Elliptic curves and the NSA In 2013, it became public that the NSA developed a pseudo-random number generator called Dual_EC_DBRG and standardized it with the help of the NIST. This random number generator uses elliptic curve arithmetic on an elliptic curve that has been specially designed to contain a backdoor, which allows the NSA to reliably predict the PRNG's outputs. Some cryptographers at Microsoft found out that the random number generator might contain a backdoor as early as in 2007 and this has been confirmed by the secret NSA documents that leaked in 2013. It is also known that Certicom already knew about the potential backdoor in 2005 as they filed a patent application describing the general idea of the backdoor in Dual_EC_DBRG. A summary of all this can be found in Green (2013).

This story has created some general concerns that the NSA might have also weakened other standardized crypto algorithms and especially standardized elliptic curves. As mentioned before, the creation of secure elliptic curves is not trivial and it might be possible that the NSA knows about additional weaknesses that are still not known to the public. This lead to general suspicion against standardized elliptic curves especially in cases where it is not clear how the parameters of the curve have exactly been created. It is important to note, that the curve `secp256k1` has been standardized in Certicom

Research (2000) which has been published by the aforementioned Certicom. So far, no weaknesses of `secp256k1` are known.

Even if `secp256k1` had weaknesses that allow forging signatures, they would be insufficient to maliciously spend Bitcoins in most cases. It is recommended (and the standard client behaves this way) not to reuse Bitcoin addresses. This means that the user generates a new Bitcoin address every time he wants to receive Bitcoins and when spending Bitcoins all Bitcoins in an address are spent and the change is sent to a new address. Consequently, any address only appears in a single spent transaction. When spending from an address, the user must provide the public key that belongs to this address. Before spending the first time, the public key is only known to the address owner. An attacker who can break ECDSA can still not spend Bitcoins as he also needs the public key corresponding to the address. To find this public key, the attacker would have to mount a pre-image attack on the combined SHA256-RIPEMD hash function used to derive addresses from public keys (see Figure 2.2).

2.3. Privacy in Bitcoin. In the Bitcoin network, each node verifies transactions on its own and then agrees with the other nodes in the network on a common transaction history, the *block chain*. Consequently, by design, each node in the network stores a copy of the full Bitcoin transaction history. This is a great threat to the users' privacy as the full transaction history is publically available to anyone. To protect the users' privacy, transactions only occur between the pseudonymous Bitcoin addresses and each user can have an arbitrary number of Bitcoin addresses and generate new ones without any interaction with the Bitcoin network. It is highly recommended to generate a new Bitcoin address for each payment a user wants to receive and also for any change. This is implemented in the standard Bitcoin-qt client.

Unfortunately, recent work shows, that one-time addresses only offer a very minor protection. In Ron & Shamir (2013b), a quite successful deanonymization strategy is presented. The authors convert the transaction history into a transaction graph where Bitcoin addresses are represented as nodes and transactions form edges between these nodes. The authors then apply a heuristic to identify addresses which belong to the same entity. The authors use this information to contract their address graph into an entity graph, which for example reveals the wealth of a certain Bitcoin entity to them even if the Bitcoins are stored in a lot of different addresses. In Ron & Shamir (2013a), the same authors use their strategy to track the actions of the Silk Road founder Dread Pirate Roberts in the transaction history and make substantiated speculations about a potential link between the Bitcoin inventor Satoshi Nakamoto and the Silk Road founder Dread Pirate Roberts. Dread Pirate Roberts indirectly received some Bitcoins from an account which was created in the very early days of Bitcoin, when Satoshi Nakamoto mined most Bitcoins himself and never spent them.

One way to protect one's anonymity when using Bitcoin are mixing services. Mixing services collect the payments of several Bitcoin users in one of their accounts and then later pay the Bitcoins to the real recipients. When this is done correctly, anyone from the outside cannot tell which sender paid which recipient. The disadvantage of this approach is that the mixing service can deanonymize any user and additionally owns the Bitcoins while the mixing is in progress and therefore can easily fraud its users.

In Miers *et al.* (2013), a sophisticated solution, called Zerocoin, is presented which requires an extension to the Bitcoin protocol. The solution allows users to swap their Bitcoins into Zerocoins with a fixed denomination. When the user later spends a Zerocoin, he does so by providing a zero knowledge proof which shows that he earlier bought one of the existing Zerocoins and that he knows the secret required to spend it. The network then checks whether the Zerocoin the user wants to spend has not been spent before. As the proof is a zero knowledge proof, the user does not reveal the transaction with which he bought his Zerocoin. Therefore, there exists no visible link between the Bitcoins used to buy the Zerocoin and the Bitcoins gained by redeeming the Zerocoin. Consequently, the Bitcoins can belong to any Zerocoin user and as long as the number of Zerocoin users is large enough, this provides sufficient anonymity to all users.

3. Threshold signatures

For a polynomial p, a *$p(t)$-out-of-u threshold signature scheme* allows $p(t)$ members out of a group of u to cooperate in creating a signature for a certain message. At the same time, the scheme is secure against an eavesdropping attacker who compromises less than t parties, which means that the attacker learns the secrets of less than t parties. In general, three different attacker models exist, Gennaro *et al.* (1996):

Eavesdropping attacker This attacker only passively spies on less than t parties and tries to forge a signature with the acquired knowledge.

Halting attacker This attacker can actively stop less than t parties from participating in the threshold signature scheme. He succeeds as soon as the remaining benign parties are no longer able to create a valid signature any more. The only way to protect the parties against a halting attacker is to increase the number of parties, such that there is a replacement for each party that has been disabled by the halting attacker.

Fully malicious attacker The fully malicious attacker replaces less than t parties in the threshold signature scheme and actively inserts invalid messages into the conversation. The fully malicious attacker will succeed if the remaining benign parties are not able to create a valid signature any longer. To defend against a fully malicious attacker, the number of parties must be increased, such that it is possible to compensate for any invalid values when the messages of the different parties are combined into the result signature. Consequently, some kind of error correction mechanism must be applied when combining the messages.

In the following, we will concentrate on the eavesdropping attacker, simply because we restrict ourselves to using only two devices and there is no way to defend against a halting or a fully malicious attacker with only two parties respectively devices. A direct consequence from the above definition is that none of the parties are allowed to learn anything about the secrets of any of the other parties. Therefore, the notion of zero-knowledge naturally comes into play when talking about these schemes.

A special case of a threshold signature scheme, is a 2-out-of-2 threshold signature scheme which is also called a *two-party signature scheme*. In a two-party signature scheme, two parties must work together to create a signature and the scheme is secure against an eavesdropping attacker who is able compromise one of the parties. This also means, that if one party becomes malicious, this party will not be able to forge any signatures.

3.1. A naive approach to threshold signatures. The naive approach to creating a threshold signature scheme, which is compatible with any existing signature scheme, would be to share the private key of the original scheme between several parties. Using Shamir (1979) secret sharing is the logical next step. Shamir's secret sharing is based on the observation that t points are required to uniquely specify a polynomial of degree $t-1$ over a finite field. The general approach to sharing a secret s with u parties, where t parties can reconstruct the secret, is the following:

1. The dealer creates a polynomial $p(x) = a_{t-1}x^{t-1} + \ldots + a_1 x + a_0 \in \mathbb{F}_q$ of degree $t-1$ where $a_{t-1}, \ldots, a_1$ are chosen at random from $\mathbb{F}_q$, $a_{t-1} \neq 0$ and $a_0 = s$.

2. The dealer deals each of the u different participants a secret share $(x, p(x))$, where the x are pairwise disjunct and not zero.

3. t participants can now decide to reconstruct the secret with the help of a combiner. The combiner retrieves t different points of the polynomial p and uses them to reconstruct p with the help of Lagrange interpolation. The secret can then be reconstructed by computing $p(0)$.

In this basic scheme both the dealer and the combiner know the shared secret. This may be acceptable in case of the dealer, because the dealer is only required during the setup, but it is unacceptable regarding the combiner. To create a signature, t parties send their secret shares to the combiner, the combiner combines them to retrieve the shared secret, which is the private key, and then creates the signature. But now, the combiner knows the private key and can create additional signatures without the other parties.

We would like to have a threshold signature scheme where the combiner does not learn anything about the secret shares of the other parties. This means that the combiner does not receive shares of the private key but shares of the signature and when these shares are combined, the result is a valid signature. Consequently, the participating parties would need to perform some mathematical operations on their secret shares and would then only send the result to the combiner. From the mathematical point of view, this means that each party modifies its secret share, which is a point on a polynomial, such that the polynomial interpolated from the modified shares contains the correct result as the constant term. To sum up, we would like to translate mathematical operations on the shared secret to operations on the secret sharing polynomial.

Ben-Or *et al.* (1988) gives completeness theorems for the addition and multiplication of a shared secret with a constant or another shared secret. Consequently, if the signing algorithm of the original signature scheme only requires addition and multiplication, we can conclude from the completeness theorems, that the signature can be computed securely in a threshold signature scheme. Furthermore, the authors explain how to do the translation.

Unfortunately, the multiplication of two secrets requires $2t-1$ participants, but is still only secure against less than t conspiring attackers. This is caused by the fact that multiplying two secrets translates to multiplying the two secret sharing polynomials. The resulting polynomial will then have a degree of $2\,(t-1)$ and will require $2t-1$ points for interpolation. Additionally, actively used signature schemes like RSA, DSA or ECDSA require further, more complex operations besides addition and multiplication. For example, DSA and ECDSA require an ephemeral secret which has to be chosen uniformly at random during the signing process. In the literature overview, several protocols are mentioned which extend the approach from above.

3.2. Existing threshold signature schemes for DSA and ECDSA. For our two-factor Bitcoin wallet, we are interested in a two-party signature scheme which creates signatures that are compatible with ECDSA. The signature algorithm of ECDSA is quite similar to the one of DSA, standardized in NIST (2013). Thus, a DSA-compatible threshold signature scheme can be ported to ECDSA by replacing the modular operations in the DSA group with the corresponding operations on elliptic curves. Of course, while doing so, the operations in the exponent groups have to be modified accordingly. This is easy to achieve as the exponent group is in both cases $\mathbb{Z}_n^\times$ where n is a large prime. Furthermore, the elements in the different exponent groups have similar bit lengths when considering the same security level.

We have searched for threshold signature schemes for both DSA and ECDSA. Several secure and efficient threshold signature schemes exist for modified versions of the El-Gamal signature scheme, see for example Harn (1994). Compatibility with DSA or ECDSA on the other hand is harder to achieve as the signature algorithm requires the inversion of a secret value and the multiplication of two secret values. This has also been mentioned by MacKenzie & Reiter when explaining the design choices in their protocol.

Most threshold signature schemes follow the ideas outlined in the naive approach in Section 3.1 and use polynomial shares similar to Shamir (1979) secret sharing. As mentioned before, this does not work well as the multiplication of two polynomials increases the degree of the resulting polynomial. It took several years to actually reach the bound given by the completeness theorems in Ben-Or *et al.* (1988) for DSA. In Langford (1995), a (t^2-t+1)-out-of-u threshold scheme for DSA is presented. The scheme does not require a trusted party or a separate combiner. The scheme is secure against an attacker compromising less than t signers. An improved $(2t-1)$-out-of-u scheme can be found in Gennaro *et al.* (1996), which also does not require a trusted party or a separate combiner. In Wang & Hwang (1997), the authors present a $(t+1)$-out-of-u scheme which is compatible with DSA. The scheme also does not need a trusted party or separate combiner, but to keep the $(t+1)$-out-of-u property the t dealers in the scheme must not act as signers. If the dealers also act as signers, the scheme is only $2t$-out-of-u.

Theoretic (Multiplication)	Ben-Or *et al.* (1988)	$2t-1$
DSA	Langford (1995)	t^2-t+1
	Gennaro *et al.* (1996)	$2t-1$
	Wang & Hwang (1997)	$t+1$ / t
ECDSA	Ibrahim *et al.* (2003)	$2t-1$
	Goldfeder *et al.* (2014)	$2t+1$

Figure 3.1: Minimal number of parties required to be secure against an eavesdropping attacker compromising less than t parties. Wang & Hwang require additional t independent combiners. Goldfeder *et al.* corrected their assumption of $t+1$ to $2t+1$.

For ECDSA, Ibrahim *et al.* (2003) present a $(2t-1)$-out-of-u threshold signature scheme. Their scheme is based on the DSA threshold scheme by Gennaro *et al.* (1996). Goldfeder *et al.* (2014) apply the Ibrahim *et al.* scheme to secure Bitcoin wallets. However, as the authors point out, it is difficult to respect the restrictions on the threshold value in the scheme, rendering it somewhat unsuitable for two-factor authentication. Namely, it was erroneously assumed that one could further improve the protocol to $(t+1)$-out-of-u by applying the degree reduction protocol from Ben-Or *et al.* (1988) to circumvent the degree doubling caused by the multiplication of two secret sharing polynomials. Unfortunately, the protocol requires $2t-1 \geq 3$ cooperating parties with secret shares to reduce the polynomial. Figure 3.1 offers an overview over the discussed threshold signature schemes.

In MacKenzie & Reiter (2004), a two-party signature scheme for DSA with a different approach is presented. Instead of working with polynomial shares, the authors use a homomorphic cipher such as the Paillier (1999) crypto system. This allows one party to operate with cipher texts of another party's secrets without ever learning about the secrets. In difference to the other threshold signature schemes, this one works for only two parties. As we need a two-party signature scheme for ECDSA to implement our two-factor wallet, we decided to port their scheme to ECDSA. As it turned out, Goldfeder *et al.* came to the same conclusion: In the blog post related to their article, they note that the scheme by MacKenzie & Reiter seems to be "close to ideal".

3.3. Threshold signature support in Bitcoin. As part of the scripting functionality, Bitcoin supports t-out-of-u threshold signatures. Instead of only a single signature, a user must provide t signatures to spend a transaction output. Each of the t signatures must verify under one of the u public keys. Bitcoin's threshold signature support has been used by Bitpay Inc. (2014) to implement a web application that offers shared control over Bitcoin addresses. Pacia & Muroch (2014) use Bitcoin's threshold signature

support to implement a two-factor authenticated wallet.

In the standard single signature case, Bitcoins are sent to a Bitcoin address which is directly derived from a public key, see Section 2. The payee can spend the received Bitcoins by providing a transaction with a signature that verifies under the public key. In the threshold signature case, the payer must specify a list of u public keys instead of a single one. The payee can spend the received Bitcoins by providing a transaction with t signatures where each of the signatures verifies under one of the u public keys.

As a list of public keys is now used to identify the payee instead of a single one, no Bitcoin address can be derived any more. Consequently, the payer must not only know a short Bitcoin address, but the whole list of u public keys to send Bitcoins to the payee. This is very inconvenient for the payer.

Another Bitcoin feature called Pay-to-script-hash (P2SH) solves this problem by adding another level of indirection. Instead of specifying the whole list of public keys, the payer only specifies the hash value of a Bitcoin script, which contains the list of the public keys. The script is hashed with the same combined SHA256-RIPEMD hash function that is used to derive a Bitcoin address from a public key (see Figure 3.3). The resulting hash value is used as a Bitcoin address. Another network id is used for these addresses to allow the client to differ between standard public key hash addresses and script hash addresses. To spend Bitcoins from a P2SH address, the payee must not only provide the t signatures, but also a Bitcoin script that fits the script hash address specified by the payer. The signatures in the spending transaction are then verified against the public keys in the script.

The combination of both features yields a threshold signature support that is as convenient for the payer as the single signature version of Bitcoin. In the following, we illustrate this threshold signature support with the same example of Alice paying some Bitcoins to Bob. This time, Alice's wallet is protected with two key pairs: `pubkey_alice_1` and `pubkey_alice_2` with corresponding private keys. One key pair is stored on her desktop pc and the other one is stored on her phone. Alice now creates `signature_1` on her desktop with the private key belonging to `pubkey_alice_1` and `signature_2` on her phone with the private key belonging to `pubkey_alice_2`. With both signatures, Alice can create a spending transaction as depicted in Figure 3.2.

Transaction inputs which spend Bitcoins from P2SH addresses are verified using a special verification procedure. These inputs are identified with help of the special layout of the script in the connected output. This detection is implemented with a byte mask and consequently the script of the connected output must look exactly as specified. Otherwise, P2SH does not work. The verification itself is a two phase process. In the first phase, it is checked whether the script provided by the spender fits the address to spend from by applying the combined SHA256-RIPEMD hash function to it (see Figure 3.3). In the second phase, the script provided by the spender is

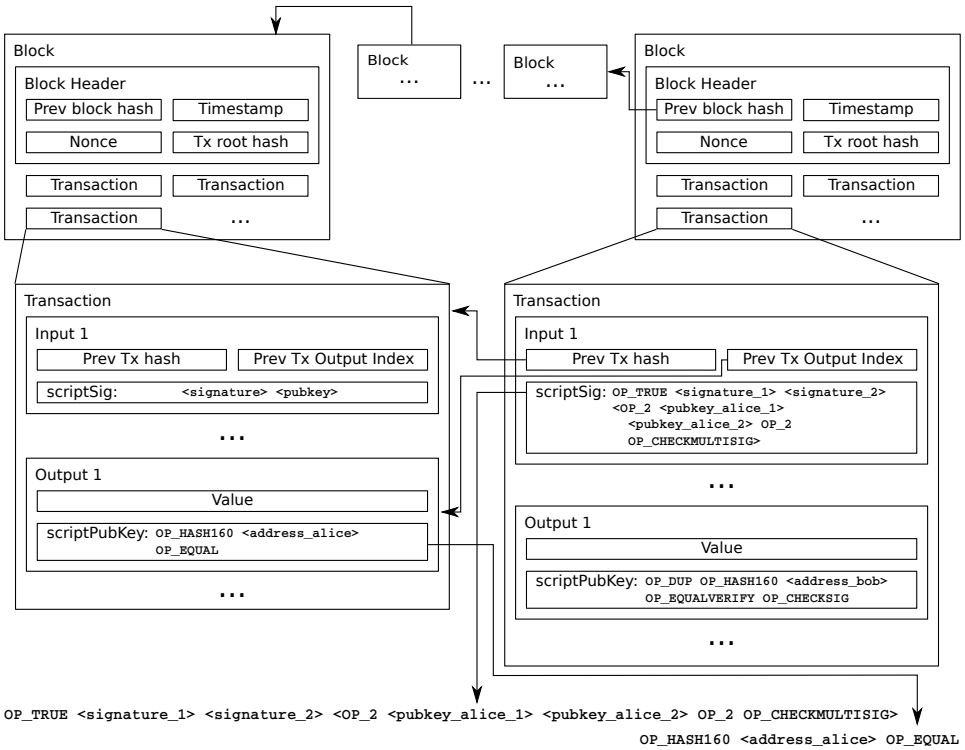


Figure 3.2: Detailed view of a Bitcoin transaction using Bitcoin's threshold signature support.

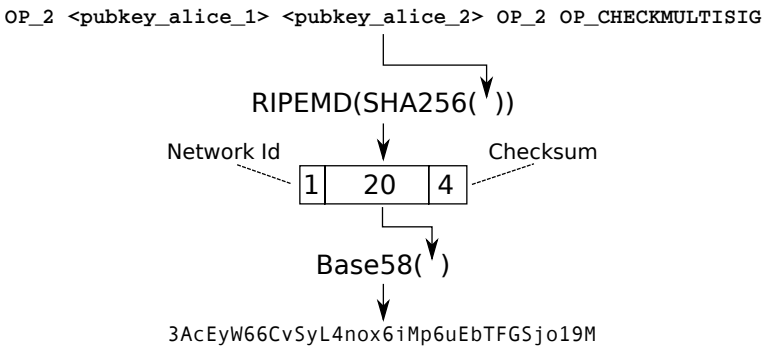


Figure 3.3: Derivation of a Bitcoin address from a Bitcoin script as defined for Pay-to-script-hash(P2SH) addresses.

executed. This script performs the signature verification. In Figure 3.4, the verification of Alice's spending transaction in Figure 3.2 is shown step-by-step:

a. In the beginning, the script, which was constructed combining the input script and the output script, is loaded into the instruction stack and the data stack is empty. In our example, the script from Figure 3.2 is loaded.

b. As explained before, the notation `<value>` is a short hand for the instruction `OP_PUSHDATA value`, where value is a number of constant bytes in the script which are pushed onto the stack. In our example, the constant value **true**, two signatures created by Alice with her private keys and the serialized threshold signature verification script are pushed onto the stack.

c. `OP_HASH160` applies the combined SHA256-RIPEMD hash function (see Figure 3.3) to the top value of the stack, which is the serialized script.

d. Now, the address stored as a constant in the script of the spent output is pushed onto the stack. In our example, this is Alice's address. As mentioned before, the `<address_alice>` constant refers not to the Base58 encoded address but to the binary representation of the raw hash value outputted by the combined SHA256-RIPEMD hash function. The Base58 encoding and also the checksum and the network id are not used inside the transactions, but only to ensure correct usage of the address by the user.

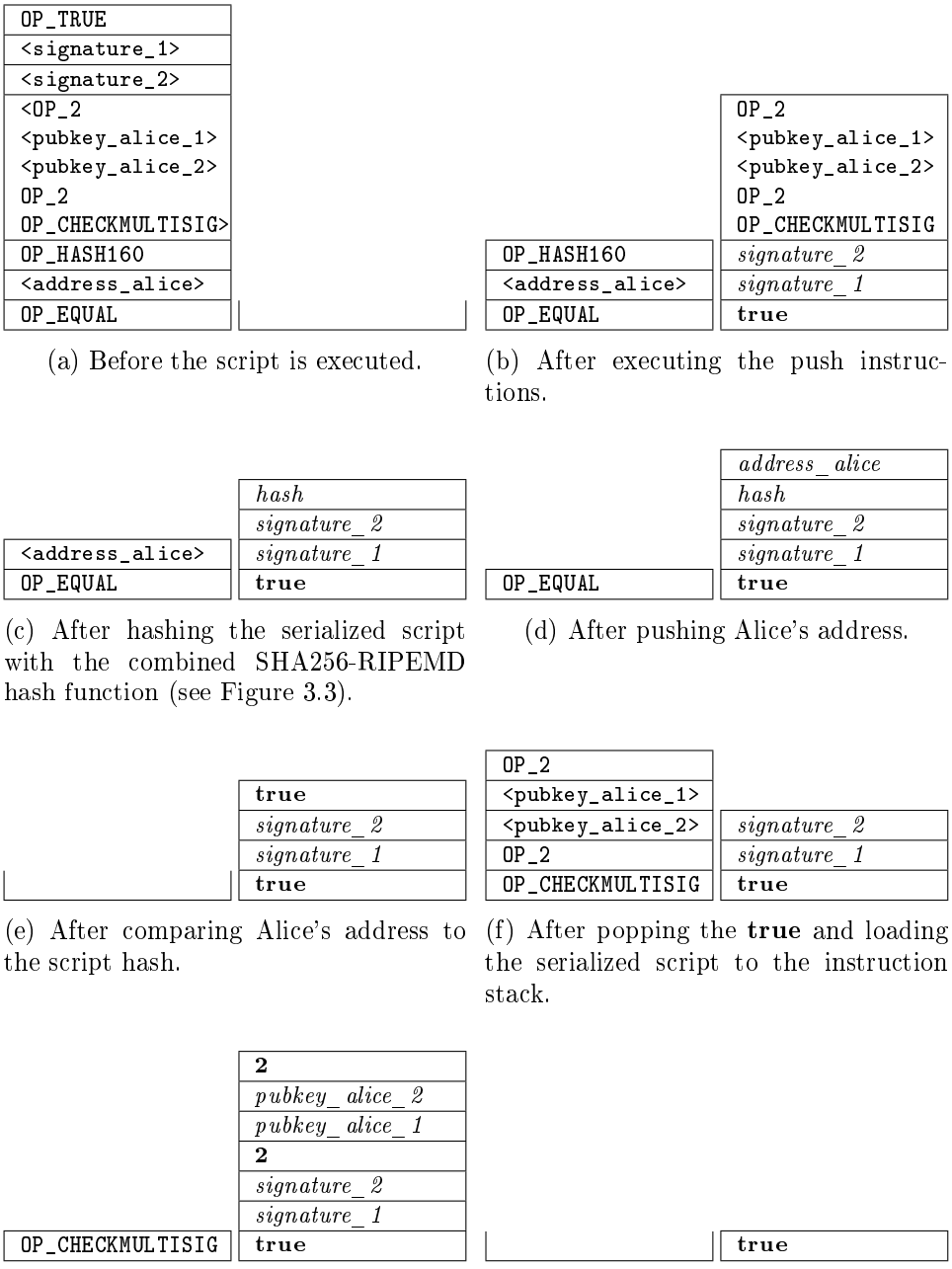
e. `OP_EQUAL` pops two values from the stack and checks whether they are equal. The two values are the Bitcoin address provided in the output script and the hash of the serialized script. Consequently, this instruction ensures that the serialized script fits the script hash used as a Bitcoin address. In our example, Alice's address and the serialized script she provided are compared.

f. The first phase of the verification has been completed. If there is a **true** on the top of stack, the verifier can be sure that the serialized script fits the P2SH Bitcoin address. The verifier now continues with the second phase of the verification. Therefore, he removes the **true** from the data stack and loads the serialized script into the instruction stack. The rest of the data on the data stack remains untouched.

g. All the push instructions from the serialized script are executed. The stack then contains Alice's two signatures and her two public keys.

h. `OP_CHECKMULTISIG` now performs the whole threshold signature verification in a single step. It starts by popping a number from the stack which denotes the number of public keys to pop and then pops the public keys. Afterwards, it pops another number which denotes the required number of signatures. Finally, it pops the signatures which are part of the input script. Now, `OP_CHECKMULTISIG` performs the threshold signature verification by checking whether each provided signature is valid under one of the specified public keys. Each public key may be used at most once. The verification of each signature is performed as described in Section 2.1. Note, that the serialized script containing the `OP_CHECKMULTISIG` instruction and not the script from the referenced output is copied into the transaction input when the transaction is hashed. Because of an implementation bug in Bitcoin, another value is afterwards popped from the stack. This value is not used at all and is most times simply set to **true** or **false**. If this additional value is missing, the script execution aborts and the transaction verification fails. In the end, a **true** or **false** is pushed onto the stack depending on whether the signature verification was successful or not. In our example, the two signatures provided by Alice are verified under Alice's two public keys which are contained in the serialized script.

This variant of threshold signatures for Bitcoin has several disadvantages that are also mentioned in Goldfeder *et al.* (2014): First of all, the spending transaction becomes much larger as it contains both the t signatures and the script where the u public keys are listed. Signatures and public keys are responsible for most of the data in a transaction. Consequently, multiple signatures significantly increase the size of the transaction and can increase the transaction fees as these depend on the size of the transaction. Second, it is visible in the public block chain that threshold signatures are used.



(a) Before the script is executed.

(b) After executing the push instructions.

(c) After hashing the serialized script with the combined SHA256-RIPEMD hash function (see Figure 3.3).

(d) After pushing Alice's address.

(e) After comparing Alice's address to the script hash.

(f) After popping the **true** and loading the serialized script to the instruction stack.

(g) After executing the push instructions.

(h) After executing `OP_CHECKMULTISIG` which performs the 2-out-of-2 threshold signature verification in one step.

Figure 3.4: Verification of a sample P2SH spent transaction.

4. Two-party ECDSA signatures

In this chapter, we present a two-party ECDSA signature protocol which is an adaption of the MacKenzie & Reiter (2004) two-party DSA signature protocol. ECDSA itself has been described in Section 2.2.

4.1. Building blocks. For the protocol to work, it is necessary to prove several facts to the other party using non-interactive zero-knowledge proofs, which we will denote by zkp, see Blum *et al.* (1988). Also, the protocol is based on the use of a (additively) homomorphic, asymmetric cipher. For any key pair $(\mathrm{sk}, \mathrm{pk})$, we define $\mathcal{M}_{\mathrm{pk}} \subset \mathbb{Z}$ to be the message space and $\mathcal{C}_{\mathrm{pk}}$ to be the cipher text space. The homomorphic property of a cipher gives rise to an operation

$$+_{pk}\colon \begin{array}{rcl} \mathcal{C}_{\mathrm{pk}} \times \mathcal{C}_{\mathrm{pk}} & \longrightarrow & \mathcal{C}_{\mathrm{pk}}, \\ (\mathrm{Enc}_{\mathrm{pk}}(m_1), \mathrm{Enc}_{\mathrm{pk}}(m_2)) & \longmapsto & \mathrm{Enc}_{\mathrm{pk}}(m_1 + m_2) \end{array}.$$

We stress that the encryption function $\mathrm{Enc}_{\mathrm{pk}}$ can be randomized such that $\mathrm{Enc}_{\mathrm{pk}}(m_1 + m_2)$ only denotes *one valid* encryption of the addition of the messages m_1 and m_2.

Applying the function $+_{\mathrm{pk}}$ repeatedly defines the function

$$\times_{pk}\colon \begin{array}{rcl} \mathcal{C}_{\mathrm{pk}} \times \mathbb{N} & \longrightarrow & \mathcal{C}_{\mathrm{pk}}, \\ (\mathrm{Enc}_{\mathrm{pk}}(m_1), m_2) & \longmapsto & \mathrm{Enc}_{\mathrm{pk}}(m_1 \cdot m_2) \end{array}.$$

Another building block are range bounded integer commitments, which are used inside of the zero-knowledge proofs. These allow a prover to commit to a secret $s \in \mathbb{Z}$ and to prove at the same time that s is inside a certain range.

Paillier cryptosystem Our protocol uses the asymmetric crypto system described in Paillier (1999). This crypto system has the aforementioned homomorphy properties. The scheme is proven to be semantically secure if and only if the Decisional Composite Residuosity Assumption holds, see also Paillier (1999). The encryption is randomized, which is required to achieve semantic security. The scheme works as follows:

Setup Generation of the public key pk and the private key sk:

1. Choose two large primes $\mathbf{p}$ and $\mathbf{q}$ uniformly at random. Set $\mathbf{N} = \mathbf{pq}$ and $\lambda = \mathrm{lcm}\,(\mathbf{p}-1, \mathbf{q}-1)$.
2. Select a $\mathbf{g} \in \mathbb{Z}_{\mathbf{N}^2}$, s.t. $\gcd\left(\mathrm{L}\left(\mathbf{g}^{\lambda} \bmod \mathbf{N}^2\right), \mathbf{N}\right) = 1$ and $\mathrm{L}(u) = u-1/\mathbf{N}$. The $\mathbf{g}$ does not need to be selected at random for security and Damgård & Jurik (2001) suggest to always use $\mathbf{g} = \mathbf{N} + 1$. This allows for some additional performance optimizations.

The public key is $pk = (\mathbf{N}, \mathbf{g})$ and the private key is $sk = (\lambda)$ or equivalently $(\mathbf{p}, \mathbf{q})$.

Encryption Encryption of a plain text $m \in \mathcal{M}_{\text{pk}}$ with $\mathcal{M}_{\text{pk}} = \mathbb{Z}_{\mathbf{N}}$:

1. Select $r \in \mathbb{Z}^*_{\mathbf{N}}$ uniformly at random.
2. Compute the cipher text $c = \mathbf{g}^m \cdot r^{\mathbf{N}} \bmod \mathbf{N}^2$.

Decryption Decryption of a cipher text $c \in \mathcal{C}_{\text{pk}}$ with $\mathcal{C}_{\text{pk}} \subset \mathbb{Z}^*_{\mathbf{N}^2}$:

1. Compute the decrypted plain text $m = \frac{\mathrm{L}\left(c^\lambda \bmod \mathbf{N}^2\right)}{\mathrm{L}\left(\mathbf{g}^\lambda \bmod \mathbf{N}^2\right)} \bmod \mathbf{N}$.

Similar to RSA, the knowledge of the prime factors $\mathbf{p}, \mathbf{q}$ enables the reconstruction of the private key. Therefore, $\mathbf{N}$ should be hard to factor. In our implementation, we will use a $\mathbf{N}$ with 2560 bits. Now, we will show how the homomorphic operations are implemented:

$$+_{pk}\colon \begin{array}{rcl} \mathcal{C}_{\text{pk}} \times \mathcal{C}_{\text{pk}} & \longrightarrow & \mathcal{C}_{\text{pk}}, \\ (c_1, c_2) & \longmapsto & c_1 \cdot c_2 \bmod \mathbf{N}^2 \end{array}$$

$$\times_{pk}\colon \begin{array}{rcl} \mathcal{C}_{\text{pk}} \times \mathbb{N} & \longrightarrow & \mathcal{C}_{\text{pk}}, \\ (c_1, k) & \longmapsto & (c_1)^k \bmod \mathbf{N}^2 \end{array}$$

Integer commitments Another repeatedly used construction are integer commitments as described in Fujisaki & Okamoto (1997). In an integer commitment scheme, a prover $\mathcal{P}$ commits a certain secret value x to a verifier $\mathcal{V}$, such that it cannot be changed later, but at the same time does not reveal the value itself to $\mathcal{V}$. There are two basic requirements for any commitment scheme:

Hiding The commitment must not leak any information about the committed secret.

Binding It must be infeasible to find a $x' \neq x$ which produces the same commitment.

The integer commitment scheme in Fujisaki & Okamoto (1997) is more sophisticated as it allows for proving certain properties of the secret as well, for example that the secret falls into a certain range. The two-party ECDSA signature protocol uses a modified version which is non-interactive. This scheme works as follows:

Setup The verifier $\mathcal{V}$ generates certain parameters for the integer commitment scheme.

1. $\mathcal{V}$ chooses two large Sophie Germain primes $\mathfrak{p}, \mathfrak{q}$ uniformly at random and $\mathfrak{p} \neq \mathfrak{q}$. There are primes $\mathfrak{p}', \mathfrak{q}'$, s.t. $\mathfrak{p} = 2\mathfrak{p}'+1, \mathfrak{q} = 2\mathfrak{q}'+1$.

2. $\mathcal{V}$ computes the modulus $\mathfrak{N} = \mathfrak{p}\mathfrak{q}$.
3. $\mathcal{V}$ chooses a random $\mathfrak{g} \in \mathbb{Z}_{\mathfrak{N}}^{\times}$ with multiplicative order $\mathfrak{p}'\mathfrak{q}'$. The unique cyclic sub group with order $\mathfrak{p}'\mathfrak{q}'$ of $\mathbb{Z}_{\mathfrak{N}}^{\times}$ is the set of quadratic residues modulo $\mathfrak{N}$ denoted $\mathbb{QR}_{\mathfrak{N}}$. A generator $\mathfrak{g}$ of $\mathbb{QR}_{\mathfrak{N}}$ can be easily computed as $\mathfrak{g} \equiv_{\mathfrak{N}} a^2$ with $a \in \mathbb{Z}_{\mathfrak{N}}^{\times}$ chosen uniformly at random and $\gcd(a-1, \mathfrak{N}) = 1$ and $\gcd(a+1, \mathfrak{N}) = 1$. This holds for most a, see Schmidt (2012).
4. $\mathcal{V}$ chooses $\chi \in \mathbb{Z}_{\mathfrak{p}'\mathfrak{q}'}^{*}$ uniformly at random and computes $\mathfrak{h} = \mathfrak{g}^{\chi} \bmod \mathfrak{N}$.
5. $\mathcal{V}$ proves to $\mathcal{P}$ that $\mathfrak{h} \in \langle \mathfrak{g} \rangle$. This can be achieved by proving that $\mathfrak{h}$ is a quadratic residue modulo $\mathfrak{N}$ (see Section 4.5).

The public parameters are $(\mathfrak{g}, \mathfrak{h}, \mathfrak{N})$. All other values must stay secret.

Commitment The prover $\mathcal{P}$ commits a secret $x \in [0, m]$ with $m << \mathfrak{N}$.

1. $\mathcal{P}$ chooses $r \in \mathbb{Z}_{m\mathfrak{N}}$ uniformly at random.
2. $\mathcal{P}$ computes the commitment $z = \mathrm{IC}(x, r) = \mathfrak{h}^x \mathfrak{g}^r \bmod \mathfrak{N}$.

The prover $\mathcal{P}$ can now send z to the verifier $\mathcal{V}$ to commit to the secret value x.

The randomizer r can intentionally exceed $\mathfrak{p}'\mathfrak{q}'$, which is the order of $\mathfrak{g}$. This ensures that the commitment z is statistically close to uniform in $\langle \mathfrak{g} \rangle$ and consequently that this commitment scheme has the hiding property, see Damgård & Fujisaki (2002).

The commitment alone is not really useful. To make use of it, it must either be opened later in a protocol or the commitment is accompanied by proofs of certain properties of the committed secret x. One interesting extension are range bounded commitments which are also used in the two-party ECDSA protocol. With this extension, the prover $\mathcal{P}$ does not only commit to a secret x, but at the same time also proves that x is in a certain range. The range bounded commitments used by the two party ECDSA protocol are described in Fujisaki & Okamoto (1997), Chan *et al.* (1998), Damgård & Fujisaki (2002), Boudot (2000) and work as follows:

Setup The same setup as for the plain commitment scheme is used.

Commitment The prover $\mathcal{P}$ commits to a secret $x \in [0, m]$ and at the same time proves that $x \in \left[-m^3, m^3\right]$.

1. $\mathcal{P}$ chooses $r \in \mathbb{Z}_{m\mathfrak{N}}$ uniformly at random.
2. $\mathcal{P}$ computes the commitment $z_1 = \mathrm{IC}(x, r) = \mathfrak{h}^x \mathfrak{g}^r \bmod \mathfrak{N}$.
3. $\mathcal{P}$ chooses $\alpha \in \mathbb{Z}_{m^3}, \gamma \in \mathbb{Z}_{m^3\mathfrak{N}}$ uniformly at random.
4. $\mathcal{P}$ computes $z_2 = \mathrm{IC}(\alpha, \gamma) = \mathfrak{h}^{\alpha} \mathfrak{g}^{\gamma} \bmod \mathfrak{N}$.

5. $\mathcal{P}$ computes $e = \mathrm{h}(z_2) \bmod m$.
6. $\mathcal{P}$ computes $s_1 = \alpha + xe$ and $s_2 = \gamma + re$. These computations are performed in $\mathbb{Z}$. If $s_1 \notin [em, m^3 - 1]$, $\mathcal{P}$ starts over again.

The prover $\mathcal{P}$ can now send (z_1, e, s_1, s_2) to the verifier $\mathcal{V}$ as a range bounded commitment to the secret value x.

Verification The verifier $\mathcal{V}$ checks a range bounded commitment of the form (z_1, c, s_1, s_2) from the prover $\mathcal{P}$.

1. $\mathcal{V}$ checks, that $s_1 \in [em, m^3 - 1]$.
2. $\mathcal{V}$ computes $e' = \mathrm{h}\left(\mathfrak{h}^{s_1}\mathfrak{g}^{s_2}z_1^{-e}\right) \bmod m$.
3. $\mathcal{V}$ checks, that $e' = e$.

The verifier $\mathcal{V}$ is now convinced that $x \in [-m^3, m^3]$.

4.2. The protocol. In Figure 4.1, the full two-party ECDSA signature protocol is shown. The protocol is a modified version of the protocol in MacKenzie & Reiter (2004) for DSA. In this article, the authors provide an extensive analysis of their protocol including a security reduction.

The protocol consists of three different phases for jointly signing a message $m \in \{0,1\}^*$:

Initialization. In this phase, Alice and Bob run the pairing protocol in Figure 4.2 to agree on the required parameters. Especially, they agree on a common ECDSA public key Q which is used to verify the cooperatively created signatures. Therefore, Alice and Bob choose private key shares $d_A, d_B \in \mathbb{Z}_n^\times$ pseudorandomly. Afterwards, they exchange the corresponding public keys $Q_A = d_A G$ and $Q_B = d_B G$. Both sides now compute the common public key as $Q = d_A Q_B = d_A d_B G$ and $Q = d_B Q_A = d_B d_A G$ respectively. As the scalar multiplication on elliptic curves is commutative, both sides now hold the same common public key Q. We can define the fictive private key $d = d_A d_B$ which is the private key corresponding to the public key Q. None of the two parties ever hold the full private key d nor are they able to compute it.

Furthermore, Alice and Bob generate key pairs $(\mathrm{sk}_A, \mathrm{pk}_A)$ and $(\mathrm{sk}_B, \mathrm{pk}_B)$ respectively for the Paillier (1999) cryptosystem, and public parameters $(\mathfrak{g}_A, \mathfrak{h}_A, \mathfrak{N}_A)$ and $(\mathfrak{g}_B, \mathfrak{h}_B, \mathfrak{N}_B)$ respectively for the Fujisaki & Okamoto (1997) integer commitment scheme. Afterwards, they exchange the public parts with each other.

The initialization only needs to be executed the first time. Afterwards, Alice and Bob can create additional signatures without repeating the initialization.

Constructing an ephemeral key. In the second phase, a shared ephemeral secret $k = k_A k_B \in \mathbb{Z}_n^\times$ is generated together with the corresponding public point $R = kG \in E$. Alice and Bob also compute the public points corresponding to their shares of the ephemeral secret as $R_A = k_A G$ and $R_B = k_B G \in E$. Similar to the shared private key d, none of the parties ever know or compute the full ephemeral secret k. Furthermore, Alice commits to the two values k_A^{-1} and $k_A^{-1} d_A$ in $\mathbb{Z}_n^\times$ by sending the corresponding encryptions under pk_A to Bob.

Form the signature. In the final phase, Bob uses the two commitments together with the homomorphic property of the encryption scheme to finally compute s, the second part of the ECDSA signature.

The protocol uses two zero-knowledge proofs π_A and π_B to ensure the correct execution of the protocol. The first proof, constructed by Alice, proves the existence of values $x, y \in \left[-n^3, n^3\right]$ to Bob, such that

$$xR = R_B, \quad (y/x)\, G = Q_A, \quad \mathrm{Dec}_{\mathrm{sk}_A}(\alpha_A) \equiv_n x, \quad \mathrm{Dec}_{\mathrm{sk}_A}(\beta) \equiv_n y.$$

All these conditions are satisfied by setting $x = z_A$ and $y = z_A d_A$. In other words, Alice proves to Bob that she has properly executed the previous steps in the protocol. The second zero-knowledge proof is used by Bob to prove to Alice that he has also executed the necessary steps in the protocol and that the operations he performed fit the operations Alice performed. Specifically, he proves that there are values $x, y \in \left[-n^3, n^3\right], z \in \left[-n^7, n^7\right]$, such that $xR_B = G$, $(y/x)\, G = Q_B$ and

$$\begin{aligned} &\mathrm{Dec}_{\mathrm{sk}_B}(\alpha_B) \equiv_n x, \\ &\mathrm{Dec}_{\mathrm{sk}_A}(\sigma) = \mathrm{Dec}_{\mathrm{sk}_A}\Big(\big((\alpha_A \times_{\mathrm{pk}_A} \mathrm{h}(m)) \times_{\mathrm{pk}_A} x\big) \\ &\qquad +_{\mathrm{pk}_A} \big((\beta \times_{\mathrm{pk}_A} r) \times_{\mathrm{pk}_A} y\big)\Big) + zn. \end{aligned}$$

All these conditions are satisfied by setting $x = z_B$, $y = z_B d_B$ and $z = c$. It seems counterintuitive, that Bob can argue about decryptions of cipher texts that were encrypted with Alice's public key pk_A. One would expect that this requires knowledge of Alice's secret key sk_A. But Bob is arguing about homomorphic operations with the cipher texts, which are deterministic for him, as he also knows the randomization term zn. In the zero knowledge proof, he can encode the equality of the two decryptions as the equality of two related cipher texts, which Bob can prove without any problems.

Before continuing, we illustrate the correctness of the two-party signature scheme by showing that Alice and Bob indeed create a valid ECDSA

Alice $(\mathbf{d_A}, \mathbf{Q_A}, \mathbf{sk_A})$		**Bob** $(\mathbf{d_B}, \mathbf{Q_B}, \mathbf{sk_B})$
$k_A \xleftarrow{R} \mathbb{Z}_n^\times$		
$z_A \leftarrow k_A^{-1}$		
$\alpha_A \leftarrow \mathrm{Enc}_{\mathrm{pk}_A}(z_A)$		
$\beta \leftarrow \mathrm{Enc}_{\mathrm{pk}_A}(d_A z_A)$	$\xrightarrow{m,\alpha_A,\beta}$	check $\alpha_A, \beta \in \mathcal{C}_{\mathrm{pk}_A}$
		$k_B \xleftarrow{R} \mathbb{Z}_n^\times$
check $R_B \in \langle G \rangle$	$\xleftarrow{R_B}$	$R_B \leftarrow k_B G$
$R \leftarrow k_A R_B$		
$\pi_A \leftarrow \mathrm{zkp}_A(R_B, R, \alpha_A, \beta)$	$\xrightarrow{R,\pi_A}$	check $R \in \langle G \rangle, \pi_A$
		$r \leftarrow \mathrm{coord_x}(R) \bmod n$
		$z_B \leftarrow k_B^{-1}$
		$c \xleftarrow{R} \mathbb{Z}_{n^5}$
		$\sigma \leftarrow \big((\alpha_A \times_{\mathrm{pk}_A} \mathrm{h}(m)) \times_{\mathrm{pk}_A} z_B\big)$ $+_{\mathrm{pk}_A} \big((\beta \times_{\mathrm{pk}_A} r) \times_{\mathrm{pk}_A} d_B z_B\big)$ $+_{\mathrm{pk}_A} \mathrm{Enc}_{\mathrm{pk}_A}(c \cdot n)$
		$\alpha_B \leftarrow \mathrm{Enc}_{\mathrm{pk}_B}(z_B)$
		$\pi_B \leftarrow \mathrm{zkp}_B\big(R_B, \alpha_A \times_{\mathrm{pk}_A} \mathrm{h}(m), \beta \times_{\mathrm{pk}_A} r, \sigma, \alpha_B\big)$
check $\sigma \in \mathcal{C}_{\mathrm{pk}_A}$	$\xleftarrow{\sigma,\alpha_B,\pi_B}$	
check $\alpha_B \in \mathcal{C}_{\mathrm{pk}_B}, \pi_B$		
$s \leftarrow \mathrm{Dec}_{\mathrm{sk}_A}(\sigma) \bmod n$		
$r \leftarrow \mathrm{coord_x}(R) \bmod n$		
publish (r, s)		

Figure 4.1: Generating a two-party ECDSA signature with a modified version of the protocol by MacKenzie & Reiter (2004).

Alice		**Bob**
$d_A \xleftarrow{R} \mathbb{Z}_n^\times$		
$Q_A \leftarrow d_A G$		
$(\mathrm{sk}_A, \mathrm{pk}_A) \leftarrow \mathrm{Gen}_{\mathrm{Pai}}()$		
$(\mathfrak{g}_B, \mathfrak{h}_B, \mathfrak{N}_B) \leftarrow \mathrm{Gen}_{\mathrm{IC}}()$		
$\tau_A \leftarrow (\mathrm{p}_{\mathrm{init}}(\mathfrak{g}_B), \mathrm{p}_{\mathrm{init}}(\mathfrak{h}_B))$	$\xrightarrow[(\mathfrak{g}_B, \mathfrak{h}_B, \mathfrak{N}_B)]{Q_A, \mathrm{pk}_A, \tau_A}$	check $\tau_A, Q_A \in \langle G \rangle$
		check $\neg\,\mathrm{prime}(\mathfrak{N}_B)$
		$d_B \xleftarrow{R} \mathbb{Z}_n^\times$
		$Q_B \leftarrow d_B G$
		$(\mathrm{sk}_B, \mathrm{pk}_B) \leftarrow \mathrm{Gen}_{\mathrm{Pai}}()$
		$(\mathfrak{g}_A, \mathfrak{h}_A, \mathfrak{N}_A) \leftarrow \mathrm{Gen}_{\mathrm{IC}}()$
check $\tau_B, Q_B \in \langle G \rangle$	$\xleftarrow[(\mathfrak{g}_A, \mathfrak{h}_A, \mathfrak{N}_A)]{Q_B, \mathrm{pk}_B, \tau_B}$	$\tau_B \leftarrow (\mathrm{p}_{\mathrm{init}}(\mathfrak{g}_A), \mathrm{p}_{\mathrm{init}}(\mathfrak{h}_A))$
check $\neg\,\mathrm{prime}(\mathfrak{N}_A)$		
$Q \leftarrow d_A Q_B$		$Q \leftarrow d_B Q_A$

Figure 4.2: Pairing protocol for performing the setup for the two-party ECDSA signature protocol in Figure 4.1. $\mathrm{Gen}_{\mathrm{Pai}}()$ executes the setup for the Paillier crypto system and $\mathrm{Gen}_{\mathrm{IC}}()$ executes the setup for the Fujisaki & Okamoto integer commitment scheme as described in Section 4.1.

signature by executing the protocol:

$$\begin{aligned}
s =& \operatorname{Dec}_{\mathrm{sk}_A}(\sigma) \\
=& \operatorname{Dec}_{\mathrm{sk}_A}\Big(\big((\alpha_A \times_{\mathrm{pk}_A} \mathrm{h}(m)\big) \times_{\mathrm{pk}_A} z_B\big) \\
&+_{\mathrm{pk}_A} \big((\beta \times_{\mathrm{pk}_A} r) \times_{\mathrm{pk}_A} d_B z_B\big) \\
&+_{\mathrm{pk}_A} \operatorname{Enc}_{\mathrm{pk}_A}(c \cdot n)\Big) \\
=& \operatorname{Dec}_{\mathrm{sk}_A}\Big(\big((\operatorname{Enc}_{\mathrm{pk}_A}(z_A) \times_{\mathrm{pk}_A} \mathrm{h}(m)\big) \times_{\mathrm{pk}_A} z_B\big) \\
&+_{\mathrm{pk}_A} \big((\operatorname{Enc}_{\mathrm{pk}_A}(d_A z_A) \times_{\mathrm{pk}_A} r) \times_{\mathrm{pk}_A} d_B z_B\big) \\
&+_{\mathrm{pk}_A} \operatorname{Enc}_{\mathrm{pk}_A}(c \cdot n)\Big) \\
=& z_A \,\mathrm{h}(m)\, z_B + d_A z_A r d_B z_B + c \cdot n \\
\equiv_n& \, k_A^{-1} k_B^{-1} (\mathrm{h}(m) + rd) \\
\equiv_n& \, k^{-1} (\mathrm{h}(m) + rd)
\end{aligned}$$

Thus, the resulting two-party signature is indeed a valid ECDSA signature under the shared private key $d = d_A d_B \in \mathbb{Z}_n^\times$ and the shared ephemeral secret $k = k_A k_B \in \mathbb{Z}_n^\times$.

4.3. Removing π_B. When studying the protocol in detail, one might notice that Bob sends the zero-knowledge proof π_B together with the encrypted signature σ to Alice. Alice then verifies π_B and afterwards decrypts σ and publishes the signature (r, s) without performing any further verification on it.

This behaviour is quite counterintuitive. One would expect that Alice verifies the decrypted signature (r, s) with the standard ECDSA signature verification before publishing it. This leads to a simplified protocol where Bob only sends σ in the second to last step, but no zero-knowledge proof. In the last step, Alice decrypts σ and then verifies the resulting signature (r, s) with the standard ECDSA signature verification. If the verification succeeds, Alice publishes the signature and otherwise aborts the protocol.

This approach has two problems: First, the zero-knowledge proof assured to Alice that Bob constructed σ exactly as required by the protocol. Without the zero-knowledge proof Bob can forge arbitrary σ and send them to Alice who will then decrypt them. Second, in the modified protocol, Alice must abort the protocol if the verification of the decrypted signature (r, s) fails. In combination, this could lead to an information leak as a malicious Bob could potentially forge specific σ which will allow him to retrieve a single bit of information by observing whether the verification of the decrypted signature fails or not. In the unmodified protocol, Alice will abort if the verification of π_B fails, but this does not leak any information besides the fact that Alice noticed that Bob has not followed the protocol. The proof π_B restricts Bob

so much, that he does not have any leeway to create values which could cause an information leak on Alice's side.

4.4. Attack scenarios and counter measures in the protocol. In MacKenzie & Reiter (2004), the authors provide a proof of soundness for their protocol, but no rationale for their design choices. In this section, we present three attack scenarios and the corresponding counter measures in the protocol to clarify why certain elements exist in the protocol. When looking at the protocol, it is quite obvious that Alice is in a much stronger position than Bob as she controls the decryption of σ, which contains Bob's secrets, and also the publication of the resulting ECDSA signature. Bob on the other hand is in a rather vulnerable position as he must multiply his secret values with some cipher texts over which he has no control. Therefore, Bob needs strong guarantees that the values encrypted in the cipher texts have been constructed as required by the protocol. These guarantees are given to Bob with the zero knowledge proof π_A.

Zero knowledge proofs In the beginning, we illustrate what happens when the zero knowledge proof π_A is missing completely. In this case, Alice is free to choose any $\alpha_A, \beta \in \mathcal{M}_{\mathrm{pk}_A}$. Alice can now sign the malicious message m' with the help of Bob who thinks that he is signing the benign message m. Alice proceeds as follows:

1. Alice follows the first step of protocol, but computes α_A differently as $\alpha_A = z_A \,\mathrm{h}\,(m)^{-1} \,\mathrm{h}\,(m') \bmod n$.
2. Alice and Bob follow the protocol, but Bob does not verify the missing proof π_A in the second to last step.
3. In the last step, Alice computes the ECDSA signature (r, s) as described in the protocol.

The signature Alice just computed is a valid signature for the message m' instead of m as:

$$
\begin{aligned}
s =& \,\mathrm{Dec}_{\mathrm{sk}_A}(\sigma) \\
=& \,\mathrm{Dec}_{\mathrm{sk}_A}\left(\left(\alpha_A \times_{\mathrm{pk}_A} \mathrm{h}\,(m)\, z_B\right) +_{\mathrm{pk}_A} \left(\beta \times_{\mathrm{pk}_A} r d_B z_B\right)\right. \\
&\left. +_{\mathrm{pk}_A} \mathrm{Enc}_{\mathrm{pk}_A}(c \cdot n)\right) \\
=& \,\mathrm{Dec}_{\mathrm{sk}_A}\left(\left(\mathrm{Enc}_{\mathrm{pk}_A}\left(z_A \,\mathrm{h}\,(m)^{-1} \,\mathrm{h}\left(m'\right)\right) \times_{\mathrm{pk}_A} \mathrm{h}\,(m)\, z_B\right)\right. \\
&+_{\mathrm{pk}_A} \left(\mathrm{Enc}_{\mathrm{pk}_A}(d_A z_A) \times_{\mathrm{pk}_A} r d_B z_B\right) \\
&\left. +_{\mathrm{pk}_A} \mathrm{Enc}_{\mathrm{pk}_A}(c \cdot n)\right) \\
=& z_A \,\mathrm{h}\,(m) \,\mathrm{h}\,(m)^{-1} \,\mathrm{h}\left(m'\right) z_B + d_A z_A r d_B z_B + c \cdot n \\
\equiv_n& \, k_A^{-1} k_B^{-1} \left(\mathrm{h}\left(m'\right) + rd\right) \equiv_n k^{-1} \left(\mathrm{h}\left(m'\right) + rd\right)
\end{aligned}
$$

Alice has just succeeded in creating a signature for the malicious message m' while Bob is thinking that he signed the benign message m. This attack has been verified in an experiment with the prototype's implementation of the two-party ECDSA protocol.

Randomization of σ Another interesting element of the protocol is the randomization of σ which is performed by Bob in the second to last step of the protocol. Bob computes σ as:

$$\sigma = \left(\alpha_A \times_{\mathrm{pk}_A} \mathrm{h}\left(m\right) z_B\right) +_{\mathrm{pk}_A} \left(\beta \times_{\mathrm{pk}_A} r d_B z_B\right) +_{\mathrm{pk}_A} \mathrm{Enc}_{\mathrm{pk}_A}\left(c \cdot n\right)$$

with a randomizing term $\mathrm{Enc}_{\mathrm{pk}_A}\left(c \cdot n\right)$ where $c \in \mathbb{Z}_{n^5}$ is chosen uniformly at random. Alice later computes $s = \mathrm{Dec}_{\mathrm{sk}_A}\left(\sigma\right) \bmod n$, but she can also see the unreduced decryption of σ. Without the randomization, the unreduced decryption leaks information about Bob's secret values. This has been verified in an experiment with the prototype's implementation of the two-party ECDSA protocol. When the randomization is removed, even a small number of protocol runs (less than 10) leak the magnitude of Bob's secret key share, such that it can be computed quite precisely by a malicious Alice.

To work properly, the randomization requires the range proofs for the clear texts x, y of α_A, β in the zero knowledge proof π_A. These range proofs assure that $x, y \in \left[-n^3, n^3\right]$. When these range proofs are missing and the message space of the Paillier crypto scheme is large enough, a malicious Alice can easily recover Bob's secret key as follows:

1. Alice executes the first step of the protocol, but computes α_A, β as $\alpha_A = z_A + z_A n^6$ and $\beta = d_A z_A + d_A z_A n^{10}$.
2. Alice executes the protocol together with Bob until the last step.
3. In the last step, Alice receives σ and computes

$$\begin{aligned} s' =& \mathrm{Dec}_{\mathrm{sk}_A}\left(\sigma\right) \\ =& \left(z_A + z_A n^6\right) \mathrm{h}\left(m\right) z_B + \left(d_A z_A + d_A z_A n^{10}\right) r d_B z_B + c \cdot n \\ =& z_A \,\mathrm{h}\left(m\right) z_B + z_A n^6 \,\mathrm{h}\left(m\right) z_B + d_A z_A r d_B z_B \\ &+ d_A z_A n^{10} r d_B z_B + c \cdot n \end{aligned}$$

4. Alice computes

$$s'' = s' \operatorname{div} n^6 = z_A \,\mathrm{h}\left(m\right) z_B + d_A z_A n^4 r d_B z_B$$

5. Alice retrieves Bob's ephemeral secret z_B by computing

$$\begin{aligned} & \left(s'' \bmod n^4\right) z_A^{-1} \,\mathrm{h}\left(m\right)^{-1} \\ =& \left(\left(z_A \,\mathrm{h}\left(m\right) z_B + d_A z_A n^4 r d_B\right) \bmod n^4\right) z_A^{-1} \,\mathrm{h}\left(m\right)^{-1} \\ =& z_A \,\mathrm{h}\left(m\right) z_B z_A^{-1} \,\mathrm{h}\left(m\right)^{-1} \equiv_n z_B \end{aligned}$$

6. Finally, Alice retrieves Bob's secret key share d_B by computing

$$\begin{aligned}&\left(s'' \operatorname{div} n^4\right)\left(z_A z_B r d_A\right)^{-1}\\ =&\left(\left(z_A \operatorname{h}(m) z_B + d_A z_A n^4 r d_B\right) \operatorname{div} n^4\right)\left(z_A z_B r d_A\right)^{-1}\\ =&d_A z_A r d_B \left(z_A z_B r d_A\right)^{-1} \equiv_n d_B\end{aligned}$$

   Alice now knows all the inverted values, as she retrieved z_B in the step before, r is public and z_A, d_A are her own secrets.

4.5. The protocol proofs. In this section, we present the zero-knowledge proofs used by the two-party ECDSA signature protocol in Figure 4.1. The zero-knowledge proofs π_A and π_B were taken from MacKenzie & Reiter (2004) and modified to be used with the two-party ECDSA signature protocol. We will refrain from presenting the full soundness and zero-knowledge proofs which can be found MacKenzie & Reiter (2004), but we will shortly illustrate how the proofs work and that switching from DSA to ECDSA does not have any influence on the security. We will also present the proof π_{init}, which is used in the pairing protocol in Figure 4.2. The idea for this proof was taken from Kunz-Jacques *et al.* (2006).

The zero-knowledge proof π_A The following non-interactive zero-knowledge proof is used by Alice to prove that she has executed her part of the two-party ECDSA signature protocol as specified:

$$\begin{aligned}\pi_A =& \operatorname{zkp}_A\left(R_B, R, c_1, c_2\right)\\ =&\begin{bmatrix}\exists x, y & x, y \in \left[-n^3, n^3\right]\\ \wedge & xR = R_B\\ \wedge & (y/x)\,G = Q_A\\ \wedge & \operatorname{Dec}_{\mathrm{sk}_A}(c_1) \equiv_n x\\ \wedge & \operatorname{Dec}_{\mathrm{sk}_A}(c_2) \equiv_n y\end{bmatrix}\end{aligned}$$

The proof works as follows:

Setup The prover and the verifier have executed the pairing protocol in Figure 4.2, where the prover performed Alice's part and the verifier performed Bob's part. Especially, the verifier has generated a set of parameters for the Fujisaki & Okamoto integer commitment scheme and has sent the public parameters $(\mathfrak{N}_A, \mathfrak{g}_A, \mathfrak{h}_A)$ to the prover. The prover has generated a key-pair for the Paillier crypto system and has sent the public key $\mathrm{pk}_A = (\mathbf{N}_A, \mathbf{g}_A)$ to the verifier.

Construction We assume, that the prover knows $x, y \in \mathbb{Z}_n^\times$ and randomiz-

ers $r_1, r_2 \in \mathbb{Z}^*_{\mathbf{N}_A}$, such that

$$c_1 \equiv_{\mathbf{N}_A^2} \mathbf{g}_A^x \cdot r_1^{\mathbf{N}_A} \equiv_{\mathbf{N}_A^2} \mathrm{Enc}_{\mathrm{pk}_A}(x)$$
$$c_2 \equiv_{\mathbf{N}_A^2} \mathbf{g}_A^y \cdot r_2^{\mathbf{N}_A} \equiv_{\mathbf{N}_A^2} \mathrm{Enc}_{\mathrm{pk}_A}(y)\,.$$

The construction of the proof then works as shown in Figure 4.3.

Verification The verification of the proof works as shown in Figure 4.4.

We will now discuss the security properties of the proof. We will only state the results by MacKenzie & Reiter and point out any differences.

Completeness Follows directly from inspection.

Soundness An in-depth soundness proof for this zero-knowledge proof can be found in MacKenzie & Reiter (2004). The authors use a proof technique by Pointcheval & Stern (2000) which is called *oracle replay attack*. They proceed as follows: in the beginning, they assume that an efficient attacker exists who can forge a malicious proof. They also show that the initialization of the Fujisaki & Okamoto integer commitment scheme can be simulated by selecting $\mathfrak{h}'_A \in \mathbb{Z}^*_{\mathfrak{N}_A}$ uniformly at random. With a probability of 1/4, $\mathfrak{h}'_A$ will be of order $\mathfrak{p}'_A\mathfrak{q}'_A$ and consequently a quadratic residue modulo $\mathfrak{N}_A$ and the simulated initialization will be indistinguishable from the real initialization. Then, the authors apply the forking lemma from Pointcheval & Stern (2000) which allows them to rerun the attacker polynomially many times. This provides them with two different instances of the proof which are suitably related. The two instances were created with the same input values but the challenge e has a different value as the random oracle, which generates the challenge, has been reset after each run. The authors then compute the differences between several values in the two instances of the proof and with the help of these differences they solve the instance $(\mathfrak{N}_A, \mathfrak{h}'_A)$ of the strong RSA problem using the extended Euclidean algorithm. Finally, they show that the one case, where the extended Euclidean algorithm does not return a valid solution, leads to a contradiction. Consequently, the soundness of the proof π_A can be reduced to the strong RSA assumption.

This proof can be straight-forwardly applied to the ECDSA case. All important computations in the soundness proof are performed in the exponent group which in the case of both DSA and ECDSA is the group $\mathbb{Z}_q^\times$ for a large prime q.

Zero-knowledge The proof is statistical zero-knowledge. A simulator can be found in MacKenzie & Reiter (2004). The presented simulator can easily be adapted for the ECDSA case by replacing the operations in the DSA group with the corresponding operations in the elliptic curve group.

$$\begin{aligned}
&\alpha \xleftarrow{R} \mathbb{Z}_{n^3} && \delta \xleftarrow{R} \mathbb{Z}_{n^3}\\
&\beta \xleftarrow{R} \mathbb{Z}^*_{\mathbf{N}_A} && \mu \xleftarrow{R} \mathbb{Z}^*_{N_A}\\
&\gamma \xleftarrow{R} \mathbb{Z}_{n^3\mathfrak{N}_A} && \nu \xleftarrow{R} \mathbb{Z}_{n^3\mathfrak{N}_A}\\
&\varrho_1 \xleftarrow{R} \mathbb{Z}_{n\mathfrak{N}_A} && \varrho_2 \xleftarrow{R} \mathbb{Z}_{n\mathfrak{N}_A}\\
& && \varrho_3 \xleftarrow{R} \mathbb{Z}_n\\
& && \varepsilon \xleftarrow{R} \mathbb{Z}_n
\end{aligned}$$

$$\begin{aligned}
&z_1 \leftarrow (\mathfrak{h}_A)^x (\mathfrak{g}_A)^{\varrho_1} \bmod \mathfrak{N}_A && z_2 \leftarrow (\mathfrak{h}_A)^y (\mathfrak{g}_A)^{\varrho_2} \bmod \mathfrak{N}_A\\
&U_1 \leftarrow \alpha R && Y \leftarrow (y+\varrho_3)\, G\\
&u_2 \leftarrow \mathfrak{g}_A^{\alpha} \beta^{\mathbf{N}_A} \bmod \mathbf{N}_A^2 && V_1 \leftarrow (\delta+\varepsilon)\, G\\
&u_3 \leftarrow (\mathfrak{h}_A)^{\alpha} (\mathfrak{g}_A)^{\gamma} \bmod \mathfrak{N}_A && V_2 \leftarrow \alpha Q_A + \varepsilon G\\
& && v_3 \leftarrow \mathfrak{g}_A^{\delta} \mu^{\mathbf{N}_A} \bmod \mathbf{N}_A^2\\
& && v_4 \leftarrow (\mathfrak{h}_A)^{\delta} (\mathfrak{g}_A)^{\nu} \bmod \mathfrak{N}_A
\end{aligned}$$

$$e \leftarrow \mathrm{hash}\,(R, R_B, G, Q_A, c_1, c_2, z_1, U_1, u_2, u_3, z_2, Y, V_1, V_2, v_3, v_4)$$

$$\begin{aligned}
&s_1 \leftarrow ex + \alpha && t_1 \leftarrow ey + \delta\\
&s_2 \leftarrow (r_1)^e \beta \bmod \mathbf{N}_A^2 && t_2 \leftarrow e\varrho_3 + \varepsilon \bmod n\\
&s_3 \leftarrow e\varrho_1 + \gamma && t_3 \leftarrow (r_2)^e \mu \bmod \mathbf{N}_A^2\\
& && t_4 \leftarrow e\varrho_2 + \nu
\end{aligned}$$

$$\Pi_A \leftarrow \langle z_1, z_2, Y, e, s_1, s_2, s_3, t_1, t_2, t_3, t_4 \rangle$$

Figure 4.3: Construction of π_A

$$\langle z_1, z_2, Y, e, s_1, s_2, s_3, t_1, t_2, t_3, t_4 \rangle \leftarrow \pi_A$$

$$\begin{aligned}
&\textbf{check } s_1, t_1 \in \left[0, n^3 - 1\right] && V_1 \leftarrow (t_1 + t_2)\, G + (-e)\, Y\\
&U_1 \leftarrow s_1 R + (-e)\, R_B && V_2 \leftarrow s_1 Q_A + t_2 G + (-e)\, Y\\
&u_2 \leftarrow \mathfrak{g}_A^{s_1} (s_2)^{\mathbf{N}} (c_1)^{-e} \bmod \mathbf{N}_A^2 && v_3 \leftarrow \mathfrak{g}_A^{t_1} (t_3)^{\mathbf{N}} (c_2)^{-e} \bmod \mathbf{N}_A^2\\
&u_3 \leftarrow (\mathfrak{h}_A)^{s_1} (\mathfrak{g}_A)^{s_3} (z_1)^{-e} \bmod \mathfrak{N}_A && v_4 \leftarrow (\mathfrak{h}_A)^{t_1} (\mathfrak{g}_A)^{t_4} (z_2)^{-e} \bmod \mathfrak{N}_A
\end{aligned}$$

$$\textbf{check } e = \mathrm{hash}\,(R, R_B, G, Q_A, c_1, c_2, z_1, U_1, u_2, u_3, z_2, Y, V_1, V_2, v_3, v_4)$$

Figure 4.4: Verification of π_A

The zero-knowledge proof π_B The following non-interactive zero-knowledge proof is used by Bob to prove that he has executed his part of the two-party ECDSA signature protocol as specified:

$$\pi_B = \mathrm{zkp}_B\left(R_B, c_1, c_2, c_3, c_4\right)$$
$$= \begin{bmatrix} \exists x, y, z & x, y \in \left[-n^3, n^3\right] \wedge z \in \left[-n^7, n^7\right] \\ \wedge & xR_B = G \\ \wedge & (y/x)\, G = Q_B \\ \wedge & \mathrm{Dec}_{\mathrm{sk}_B}(c_4) = x \\ \wedge & \mathrm{Dec}_{\mathrm{sk}_A}(c_3) = \left(\mathrm{Dec}_{\mathrm{sk}_A}(c_1)\right) x + \left(\mathrm{Dec}_{\mathrm{sk}_A}(c_2)\right) y + nz \end{bmatrix}$$

The proof works as follows:

Setup The prover and the verifier have executed the pairing protocol in Figure 4.2, where the prover performed Bob's part and the verifier performed Alice's part. Especially, the prover has generated a key-pair for Paillier crypto system and has sent the public key $\mathrm{pk}_B = (\mathbf{N}_B, \mathbf{g}_B)$ to the verifier. The verifier on the other hand has generated a set of parameters for the Fujisaki & Okamoto integer commitment scheme and has sent the public parameters $(\mathfrak{N}_B, \mathfrak{g}_B, \mathfrak{h}_B)$ to the prover. Furthermore, he has generated a key-pair for Paillier crypto system and has also sent the public key $\mathrm{pk}_A = (\mathbf{N}_A, \mathbf{g}_A)$ to the prover.

Construction We assume, that the prover knows $x, y \in \mathbb{Z}_n^\times$, $z \in \mathbb{Z}_{n^5}^\times$ and randomizers $r_4 \in \mathbb{Z}_{\mathbf{N}_B}^*$, $r_3 \in \mathbb{Z}_{\mathbf{N}_A}^*$, such that

$$\begin{aligned} c_4 &\equiv_{\mathbf{N}_B^2} \mathbf{g}_B^x \cdot r_4^{\mathbf{N}_B} \equiv_{\mathbf{N}_B^2} \mathrm{Enc}_{\mathrm{pk}_B}(x) \\ c_3 &\equiv_{\mathbf{N}_A^2} c_1^x \cdot c_2^y \cdot \mathbf{g}_A^{nz} \cdot r_3^{\mathbf{N}_A} \\ &\equiv_{\mathbf{N}_A^2} \left(c_1 \times_{\mathrm{pk}_A} x\right) +_{\mathrm{pk}_A} \left(c_2 \times_{\mathrm{pk}_A} y\right) +_{\mathrm{pk}_A} \mathrm{Enc}_{\mathrm{pk}_A}(nz)\,. \end{aligned}$$

The construction of the proof then works as shown in Figure 4.5.

Verification We assume that the verifier knows $\mathrm{Dec}_{\mathrm{sk}_A}(c_1)$ and $\mathrm{Dec}_{\mathrm{sk}_A}(c_2)$. The verification of the proof then works as shown in Figure 4.6.

We will now also quickly discuss the security properties of this zero-knowledge proof. We will only state the results by MacKenzie & Reiter and point out any differences.

Completeness Follows directly from inspection.

Soundness The in-depth soundness proof can be found in MacKenzie & Reiter (2004). This proof is very similar to the soundness proof for π_A, which has been described above. The soundness of this proof can also be reduced to the strong RSA assumption. Also, the proof can be applied to the ECDSA case with the same argument as for π_A.

$$\alpha \xleftarrow{R} \mathbb{Z}_{n^3} \qquad \delta \xleftarrow{R} \mathbb{Z}_{n^3}$$
$$\beta \xleftarrow{R} \mathbb{Z}^*_{\mathbf{N}_B} \qquad \mu \xleftarrow{R} \mathbb{Z}^*_{\mathbf{N}_A}$$
$$\gamma \xleftarrow{R} \mathbb{Z}_{n^3\mathfrak{N}_B} \qquad \nu \xleftarrow{R} \mathbb{Z}_{n^3\mathfrak{N}_B}$$
$$\varrho_1 \xleftarrow{R} \mathbb{Z}_{n\mathfrak{N}_B} \qquad \varrho_2 \xleftarrow{R} \mathbb{Z}_{n\mathfrak{N}_B}$$
$$\varrho_3 \xleftarrow{R} \mathbb{Z}_n$$
$$\varrho_4 \xleftarrow{R} \mathbb{Z}_{n^5\mathfrak{N}_B}$$
$$\varepsilon \xleftarrow{R} \mathbb{Z}_n$$
$$\sigma \xleftarrow{R} \mathbb{Z}_{n^7}$$
$$\tau \xleftarrow{R} \mathbb{Z}_{n^7\mathfrak{N}_B}$$

$$z_1 \leftarrow (\mathfrak{h}_B)^x (\mathfrak{g}_B)^{\varrho_1} \bmod \mathfrak{N}_B \qquad z_2 \leftarrow (\mathfrak{h}_B)^y (\mathfrak{g}_B)^{\varrho_2} \bmod \mathfrak{N}_B$$
$$U_1 \leftarrow \alpha R_B \qquad Y \leftarrow (y + \varrho_3) G$$
$$u_2 \leftarrow (\mathbf{g}_B)^\alpha \beta^{\mathbf{N}_B} \bmod (\mathbf{N}_B)^2 \qquad V_1 \leftarrow (\delta + \varepsilon) G$$
$$u_3 \leftarrow (\mathfrak{h}_B)^\alpha (\mathfrak{g}_B)^\gamma \bmod \mathfrak{N}_B \qquad V_2 \leftarrow \alpha Q_B + \varepsilon G$$
$$v_3 \leftarrow (c_1)^\alpha (c_2)^\delta \mathbf{g}^{n\sigma} \mu^{\mathbf{N}_A} \bmod (\mathbf{N}_A)^2$$
$$v_4 \leftarrow (\mathfrak{h}_B)^\delta (\mathfrak{g}_B)^\nu \bmod \mathfrak{N}_B$$
$$z_3 \leftarrow (\mathfrak{h}_B)^z (\mathfrak{g}_B)^{\varrho_4} \bmod \mathfrak{N}_B$$
$$v_5 \leftarrow (\mathfrak{h}_B)^\sigma (\mathfrak{g}_B)^\tau \bmod \mathfrak{N}_B$$

$$e \leftarrow \text{hash}\,(R_B, G, Q_B, c_4, c_3, z_1, U_1, u_2, u_3, z_2, z_3, Y, V_1, V_2, v_3, v_4, v_5)$$

$$s_1 \leftarrow ex + \alpha \qquad t_1 \leftarrow ey + \delta$$
$$s_2 \leftarrow (r_4)^e \beta \bmod (\mathbf{N}_B)^2 \qquad t_2 \leftarrow e\varrho_3 + \varepsilon \bmod n$$
$$s_3 \leftarrow e\varrho_1 + \gamma \qquad t_3 \leftarrow (r_3)^e \mu \bmod (\mathbf{N}_A)^2$$
$$t_4 \leftarrow e\varrho_2 + \nu$$
$$t_5 \leftarrow ez + \sigma$$
$$t_6 \leftarrow e\varrho_4 + \tau$$

$$\pi_B \leftarrow \langle z_1, z_2, z_3, Y, e, s_1, s_2, s_3, t_1, t_2, t_3, t_4, t_5, t_6 \rangle$$

Figure 4.5: Construction of π_B

$$\langle z_1, z_2, z_3, Y, e, s_1, s_2, s_3, t_1, t_2, t_3, t_4, t_5, t_6 \rangle \leftarrow \pi_B$$

check $s_1, t_1 \in [0, n^3 - 1]$	$V_1 \leftarrow (t_1 + t_2)\, G + (-e)\, Y$
check $t_5 \in [0, n^7 - 1]$	$V_2 \leftarrow s_1 Q_B + t_2 G + (-e)\, Y$
$U_1 \leftarrow s_1 R_B + (-e)\, G$	$v_3 \leftarrow \Big((c_1)^{s_1} (c_2)^{t_1} \mathbf{g}_A^{nt_5} (t_3)^{\mathbf{N}_A} (c_3)^{-e} \Big) \bmod (\mathbf{N}_A)^2$
$u_2 \leftarrow (\mathbf{g}_B)^{s_1} (s_2)^{\mathbf{N}_B} (c_4)^{-e} \bmod (\mathbf{N}_B)^2$	$v_4 \leftarrow (\mathfrak{h}_B)^{t_1} (\mathfrak{g}_B)^{t_4} (z_2)^{-e} \bmod \mathfrak{N}_B$
$u_3 \leftarrow (\mathfrak{h}_B)^{s_1} (\mathfrak{g}_B)^{s_3} (z_1)^{-e} \bmod \mathfrak{N}_B$	$v_5 \leftarrow (\mathfrak{h}_B)^{t_5} (\mathfrak{g}_B)^{t_6} (z_3)^{-e} \bmod \mathfrak{N}_B$

check $e = \text{hash}\,(R_B, G, Q_B, c_4, c_3, z_1, U_1, u_2, u_3, z_2, z_3, Y, V_1, V_2, v_3, v_4, v_5)$

Figure 4.6: Verification of Π_B

Zero-knowledge This proof is statistical zero-knowledge. A simulator can be found in MacKenzie & Reiter (2004). The presented simulator can also easily be adapted for the ECDSA case by replacing the operations in the DSA group with the corresponding operations in the elliptic curve group.

The proof π_{init} The following non-interactive proof is used by Alice and Bob to prove that they chose $\mathfrak{g}$ and $\mathfrak{h}$ for the Fujisaki & Okamoto integer commitment scheme as required in the setup described in Section 4.1:

$$\pi_{\text{init}} = \text{p}_{\text{init}}\,(x) = \text{p}\,[x \in \mathbb{QR}_{\mathfrak{N}}]$$

where $\mathbb{QR}_{\mathfrak{N}}$ denotes the set of quadratic residues modulo $\mathfrak{N}$. The proof works as follows:

Setup No special setup is required, but we assume that both the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ know the safe RSA modulus $\mathfrak{N}$. The verifier does not need to know the order of $\mathbb{QR}_{\mathfrak{N}}$.

Construction The prover $\mathcal{P}$ wants to show that x is a quadratic residue modulo $\mathfrak{N}$. We assume that the prover knows the prim factors $\mathfrak{p}$ and $\mathfrak{q}$ of $\mathfrak{N}$.

1. $\mathcal{P}$ computes the positive square roots of x modulo $\mathfrak{p}$ and modulo $\mathfrak{q}$ as follows: $r_{\mathfrak{p}} \equiv_{\mathfrak{p}} \mathfrak{h}^{(\mathfrak{p}+1)/4}$ and $r_{\mathfrak{q}} \equiv_{\mathfrak{q}} \mathfrak{h}^{(\mathfrak{q}+1)/4}$. This is possible as $\mathfrak{p}$ and $\mathfrak{q}$ are safe primes greater than 5 and therefore $\mathfrak{p} \equiv_4 3$ and $\mathfrak{q} \equiv_4 3$ and the square roots can be computed with the formula we used.

2. $\mathcal{P}$ computes the square root $r \in \mathbb{Z}_{\mathfrak{N}}$ with $x \equiv_{\mathfrak{N}} r^2$ with the help of $r_{\mathfrak{p}}$ and $r_{\mathfrak{q}}$ by using the Chinese Remainder Theorem and the Extended Euclidean Algorithm.

The prover $\mathcal{P}$ can know send r as a proof to the verifier.

Verification The verifier wants to verify that $x \in \mathbb{QR}_{\mathfrak{N}}$.

1. The verifier $\mathcal{V}$ checks if $x \equiv_{\mathfrak{N}} r^2$.

If the check succeeds, the verifier $\mathcal{V}$ is convinced that r is a square root of x and consequently that $x \in \mathbb{QR}_{\mathfrak{N}}$.

Completeness Follows directly from inspection.

Soundness Assume that we know $r \in \mathbb{Z}^*_{\mathfrak{N}}$ such that $r^2 \equiv_{\mathfrak{N}} x$ but $x \notin \mathbb{QR}_{\mathfrak{N}}$. The contradiction follows directly from the definition of quadratic residuosity as $\mathbb{QR}_{\mathfrak{N}} = \left\{y \in \mathbb{Z}^*_{\mathfrak{N}} : \exists r \in \mathbb{Z}^*_{\mathfrak{N}}.\ y \equiv_{\mathfrak{N}} r^2\right\}$.

In difference to π_A and π_B, this proof is not zero-knowledge as it leaks the root r of x.

4.6. Security analysis. In MacKenzie & Reiter (2004), the authors give a detailed security analysis for their protocol. We will only summarize their results and provide a list of the assumptions under which the protocol is secure. MacKenzie & Reiter have proven that their two-party DSA protocol is EUF-CMA (existential unforgeability under chosen message attack) secure against an Alice-compromising or a Bob-compromising attacker under certain assumptions. In the ECDSA case, these are the following assumptions:

ECDSA is EUF-CMA secure. This assumption is obvious. If ECDSA is not EUF-CMA secure, an attacker can simply use the existential forgery attack for ECDSA directly on the common public key Q. There is no need to attack the two-party ECDSA protocol itself.

The Paillier crypto system is semantically secure. Semantic security under chosen plaintext attack is equivalent to indistinguishability under chosen plain text attack (IND-CPA), see Goldwasser & Micali (1984). This automatically means that any crypto system that should be used here must be randomized as otherwise IND-CPA security is not possible. In Paillier (1999), it is proven that the presented crypto system is semantically secure if and only if the decisional composite residuosity assumption (DCRA) holds. In short, DCRA states that no polynomial time distinguisher exists for $\mathbf{N}$-th residues modulo $\mathbf{N}^2$ with $\mathbf{N} = \mathbf{pq}$ and $\mathbf{p}, \mathbf{q}$ prime.

In Paillier (1999), it is additionally proven that deciding composite residuosity is polynomially reducible to solving the RSA problem for

the instance $(\mathbf{N}, \mathbf{N})$. The RSA problem $(\mathbf{N}, e, c)$ is to find m, s.t. $c = m^e \bmod \mathbf{N}$, which means to extract an e-th root modulo $\mathbf{N}$. Consequently, the DCRA does not hold if the RSA assumption does not hold. This does not imply that the Paillier crypto system is semantically secure if the RSA assumption holds. We must be believe in the DCRA for this.

The proofs π_A and π_B are sound. MacKenzie & Reiter have proven that the soundness of the proofs π_A and π_B can be reduced to the strong RSA assumption in the DSA case. This means, that an attacker who is able to produce a malicious proof that is accepted by the verifier can also solve a certain instance of the strong RSA problem and consequently break that strong RSA assumption. In Section 4.5, we have justified the application of their results to the ECDSA case. A further soundness proof for the Fujisaki & Okamoto integer commitment scheme is not required. MacKenzie & Reiter reuse the commitment scheme but the soundness proof is part of the soundness proof of π_A and π_B.

The proofs π_A and π_B are statistically zero-knowledge. MacKenzie & Reiter have proven this for DSA and the applicability of their results in the ECDSA case has been justified in Section 4.5. The statistical zero-knowledge of π_A and π_B does not require any further assumptions.

The initialization has been correctly performed. All the parameters in the protocol must be correctly generated as required. This can be done by executing the pairing protocol in Figure 4.2. We discuss the security of the pairing protocol separately.

In conclusion, the two-party ECDSA signature protocol is EUF-CMA secure under the following assumptions: ECDSA is EUF-CMA secure, the strong RSA assumption holds and the decisional composite residuosity assumption (DCRA) holds. Furthermore, the random oracle model is used when hash functions appear and consequently the security proofs only hold in the random oracle model.

Security of the pairing protocol During the pairing protocol, each party should prove to the other party that their parameters have been created according to the setup instructions. Three different types of parameters occur during the pairing:

- The public points Q_A and Q_B which correspond to the private key shares d_A and d_B respectively.
- The Paillier public keys pk_A and pk_B which correspond to the secret keys sk_A and sk_B respectively.

- The public parameter sets $(\mathfrak{g}_A, \mathfrak{h}_A, \mathfrak{N}_A)$ and $(\mathfrak{g}_B, \mathfrak{h}_B, \mathfrak{N}_B)$ for the Fujisaki & Okamoto integer commitment scheme.

MacKenzie & Reiter also provide a draft for a pairing protocol in an appendix of their article, but they focus on a different aspect in it. Their pairing protocol includes proofs of knowledge for the private key shares for DSA and also for the private keys for the Paillier crypto system. The parameters for the Fujisaki & Okamoto integer commitment scheme are not part of their pairing protocol.

Proving the knowledge of a private key is usually done in public key infrastructures when applying for a certificate for a certain public key. Otherwise, a user could bind an arbitrary public key to his identity and signatures which verify under this public key would be attributed to this user even though he has not created them. This problem does not apply to Bitcoin as the key pairs are exclusively used to identify the owner of a certain Bitcoin address. If Alice or Bob send a point Q_A or Q_B for which they do not know the private key share, the Bitcoin address will be inaccessible, but no other consequences will occur. We cannot handle the availability problem in this context. Alice or Bob always have the ability to render the Bitcoin address under their shared control inaccessible by just forgetting their private key share. The availability problem can only be solved by introducing additional parties (see Section 3 and Section 6.4). On the other hand, we want to ensure during the pairing that $Q_A, Q_B \in \langle G \rangle$. As the used elliptic curve `secp256k1` has a cofactor of 1, this can easily be done by just verifying that Q_A and Q_B are on the elliptic curve.

Proofs of knowledge of the private key or proofs of the correct construction of the parameters of the Paillier crypto system are also unnecessary. The Paillier key pair of Bob is only used for the commitment which is required for the zero-knowledge proof π_B. By damaging the security of the crypto system, Bob only endangers his own secret. The Paillier key pair of Alice on the other hand is also used by Bob as he multiplies his own secrets into the cipher texts encrypted with Alice's public key. Nevertheless, if Alice damages the security of the Paillier crypto system, this has no consequences for Bob. Bob protects his secrets by randomizing σ and this randomization must be sufficient to protect Bob's secrets as Alice can see the clear texts by design. Even if Alice breaks the crypto system for her key pair in a way that even the homomorphic operations do not work correctly any longer, it is easy to see that the randomization will still work. Adding the randomization term $\mathrm{Enc}_{\mathrm{pk}_A}(cn)$ with the homomorphic addition $+_{\mathrm{pk}_A}$ maps to multiplying the rest of σ with $\mathbf{g}^{(cn)} \cdot r^{\mathbf{N}_A}$ in $\mathbb{Z}^*_{\mathbf{N}_A^2}$ and $r \in \mathbb{Z}^*_{\mathbf{N}_A}$ is chosen uniformly at random. In the end, the rest of σ is always randomized by multiplication with an exponentiation of a random element from $\mathbb{Z}^*_{\mathbf{N}_A}$.

The most problematic part are the parameters for the Fujisaki & Okamoto integer commitment scheme. These parameters must indeed be

constructed as required in the setup procedure (see Section 4.1). The RSA moduli $\mathfrak{N}_A$ and $\mathfrak{N}_B$ must especially be formed from two safe primes, $\mathfrak{g}_A$ and $\mathfrak{g}_B$ must be quadratic residues modulo $\mathfrak{N}_A$ and $\mathfrak{N}_B$ respectively, and $\mathfrak{h}_A \in \langle \mathfrak{g}_A \rangle$ and $\mathfrak{h}_B \in \langle \mathfrak{g}_B \rangle$. Otherwise, the integer commitment scheme does not hide the committed secrets and a malicious verifier can mount a quite efficient attack as described by Kunz-Jacques *et al.* (2006). The aforementioned attack is meant for the Σ^+-protocol, but can directly be applied to the Fujisaki & Okamoto integer commitment scheme. Unfortunately and also mentioned by Kunz-Jacques *et al.*, achieving provable security is really hard as the construction of $\mathfrak{N}_A$ and $\mathfrak{N}_B$ from safe primes must be proven and no zero-knowledge proof with an acceptable execution time exists for this problem. Consequently, we decided to only apply two attack mitigations which are suggested by Kunz-Jacques *et al.*:

- Check that $\mathfrak{g}_A, \mathfrak{h}_A$ and $\mathfrak{g}_B, \mathfrak{h}_B$ are quadratic residues modulo $\mathfrak{N}_A$ and $\mathfrak{N}_B$ respectively. This implies that $\mathfrak{h}_A \in \langle \mathfrak{g}_A \rangle$ as the RSA modulus $\mathfrak{N}_A$ is formed by the two safe primes $\mathfrak{p}_A$ and $\mathfrak{q}_A$ with $\mathfrak{p}_A = 2\mathfrak{p}'_A + 1$ and $\mathfrak{q}_A = 2\mathfrak{q}'_A + 1$ and $\mathfrak{p}'_A$ and $\mathfrak{q}'_A$ prime. Hence, $\mathfrak{g}_A$ is a generator of order $\mathfrak{p}'_A\mathfrak{q}'_A$ of $\mathbb{QR}_{\mathfrak{N}_A}$, which is the sub group of quadratic residues modulo $\mathfrak{N}_A$. We can prove $\mathfrak{h}_A \in \langle \mathfrak{g}_A \rangle$ by showing that $\mathfrak{h}_A \in \mathbb{QR}_{\mathfrak{N}_A}$. The same applies to $\mathfrak{g}_B, \mathfrak{h}_B$. The proof π_{init} can be used to prove the required quadratic residuosities.
- Check that $\mathfrak{N}_A$ and $\mathfrak{N}_B$ are not prime.

These two counter measures significantly increase the complexity of the attack described in Kunz-Jacques *et al.* (2006). The malicious verifier then needs multiple protocol runs with different sets of parameters to recover a single bit of the prover's secret. This is sufficiently secure for our two-factor Bitcoin wallet as the parameters for the Fujisaki & Okamoto integer commitment scheme are fixed during the pairing and are not changed afterwards. Consequently, rerunning the protocol with different sets of parameters is not possible. Furthermore, commitments are only created during the two-party ECDSA signature protocol. Hence, several hundred runs of the signature protocol are required to apply the attack by Kunz-Jacques *et al.*. In our prototype, each run of the two-party ECDSA signature protocol must be triggered manually by the user on both devices. All in all, the attack described in Kunz-Jacques *et al.* (2006) becomes infeasible at least in the setting used by our prototype.

Parameter choices for Bitcoin In Figure 4.7, the required parameters sizes for the two-factor Bitcoin wallet are given. The parameter sizes were chosen based on the established recommendations for key sizes. ECDSA with the curve `secp256k1`, as used in the Bitcoin protocol, uses 256 bit keys. This corresponds to 128 bits of security. To achieve 128 bits of security with

RSA, a 2048 bit modulus is required according to ANSSI (2014). Note that others are more pessimistic: NIST (Barker *et al.* 2012) recommends at least 3072 bit moduli. On the other hand, there is also an implicit lower bound for the moduli sizes by the protocol itself, since some of the above mentioned arguments only work when the used parameter sizes are large enough. We decided to use 2560 bit RSA moduli for the Paillier crypto system and 2048 bit moduli for the Fujisaki & Okamoto integer commitment scheme, which is a good compromise between the different recommendations and also offers acceptable performance on the smart phone.

It should be stressed, that we are only talking about short term security. The Paillier crypto system is only used to encrypt private keys and ephemeral secrets for the ECDSA signature scheme, which uses 256 bit keys. The security can later be easily increased to the level provided by 256 bit ECDSA by increasing the RSA modulo size beyond 3072 bit and transferring all Bitcoins to new addresses with new ECDSA key pairs, which were not yet used in the two-party ECDSA signature protocol. Boosting the level of security any further is not possible as the used elliptic curve `secp256k1` is fixed in the Bitcoin protocol.

n	◦ Order of the group generated by G over the elliptic curve. ◦ Fixed by Bitcoin, see `secp256k1`. ◦ $\text{len}(n) = 256$.
$\mathbf{N}_A$	◦ Alice's RSA modulus for the Paillier cryptosystem. ◦ Protocol requires message space $[-n^8, n^8] \Rightarrow \mathbf{N}_A$ determines the message space $\Rightarrow \text{len}(\mathbf{N}_A) = \text{len}(n) \cdot 8 = 2048$. ◦ The soundness proof for the zero-knowledge proof π_B requires $\mathbf{N}_A > n^9 \Rightarrow \text{len}(\mathbf{N}_A) > \text{len}(n) \cdot 9 = 2304$.
$\mathbf{N}_B$	◦ Bob's RSA modulus for the Paillier cryptosystem. ◦ Protocol requires message space $[-n^6, n^6] \Rightarrow \mathbf{N}_B$ determines the message space $\Rightarrow \text{len}(\mathbf{N}_B) = \text{len}(n) \cdot 6 = 1536$. ◦ The soundness proof for the zero-knowledge proof π_B requires $\mathbf{N}_B > n^6 \Rightarrow \text{len}(\mathbf{N}_B) > \text{len}(n) \cdot 6 = 1536$.
$\mathfrak{N}_A$	◦ Alice's RSA modulus for the integer commitment scheme. ◦ Fujisaki & Okamoto commitment scheme requires a RSA modulus consisting of two safe primes. ◦ Largest committed value is in $[-n^3, n^3] \Rightarrow \text{len}(\mathfrak{N}_A) > \text{len}(n) \cdot 3 = 768$.
$\mathfrak{N}_B$	◦ Bob's RSA modulus for the integer commitment scheme. ◦ Fujisaki & Okamoto commitment scheme requires a RSA modulus consisting of two safe primes. ◦ Largest committed value in $[-n^7, n^7] \Rightarrow \text{len}(\mathfrak{N}_B) > \text{len}(n) \cdot 7 = 1792$.

Figure 4.7: Required parameter sizes for ECDSA as used in Bitcoin. Paramter sizes chosen for the prototype: 2560 bit for $\mathbf{N}_A$ and $\mathbf{N}_B$ and 2048 bit for $\mathfrak{N}_A$ and $\mathfrak{N}_B$.

5. Two-factor Bitcoin wallets

As mentioned in Lipovsky (2013), a first Bitcoin stealing online banking trojan has already been discovered in the wild. We can assume, that when Bitcoin becomes used by a wider public, attackers will come up with more sophisticated attacks inspired by the attacks on online banking systems. Therefore, it makes sense to analyze existing attacks on online banking systems and to consider the existing counter measures when designing a Bitcoin wallet.

In Sancho *et al.* (2014), a common attack on online banking is described. First, the user's computer is compromised with a trojan, which modifies the victim's DNS resolver and installs an additional attacker controlled certification authority on the system. Consequently, the trojan can now become a man-in-the-middle between the user and the bank. After the user successfully logs in, the attacker displays a warning to trick the user into installing a malicious app on his phone, which finally allows the attacker to intercept incoming session tokens and transaction numbers. It is important to note, that the phone is compromised by tricking the user into installing the spyware app and not by vulnerabilities in the phone's software.

To complicate such attacks as much as possible, state-of-the-art online banking systems offer both two-factor authentication and verification over a separate channel. In the (at least in Germany) commonly used SMS TAN system, the user creates a bank transaction on his computer and then needs to enter a transaction number (TAN) to confirm the transaction. The user receives this TAN via SMS from his bank. The SMS does not only contain the TAN but also the transaction data and the user can verify it once again. A compromised computer cannot modify this information and the user can detect any modifications done to the transaction by an online banking trojan on his computer.

With our Bitcoin wallet, we provide both two-factor authentication and verification via a separate channel to Bitcoin users. We thus offer users a similar level of security for Bitcoin as they currently have in online banking.

As mentioned before, a Bitcoin address is directly derived from an ECDSA public key and anyone having access to the corresponding private key can spend all Bitcoins stored in this address. Therefore, the only secure way to implement two-factor authentication is to share the private key and to create transaction signatures with a two-party signature protocol. Any other solution would require the handling of the full private key in a single place, which consequently becomes a single point of failure. Several Bitcoin service providers offer SMS TAN or one-time-password two-factor authentication, but in these cases the service providers store the private key and become a single point of failure. Bitcoin service providers are hardly regulated at the moment and considering the bankruptcy of Mt. Gox, it is clear that leaving the security to the service provider is too risky.

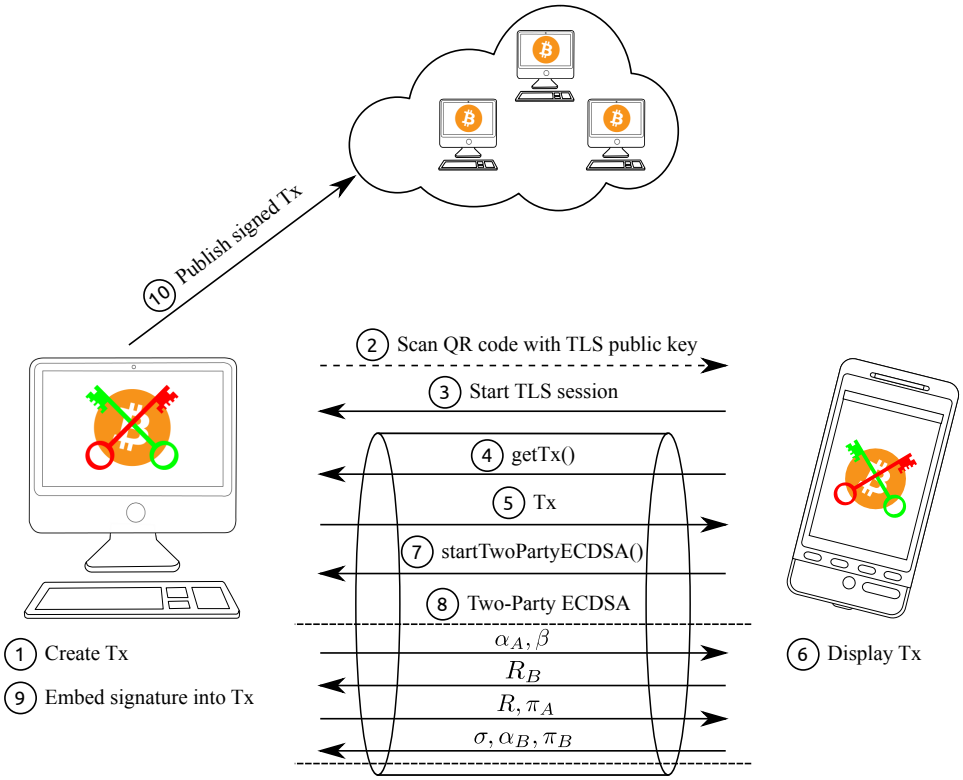


Figure 5.1: Signing a Bitcoin transaction with our prototype of a two-factor Bitcoin wallet.

For our Bitcoin wallet, the two-party ECDSA signature protocol, described in Section 4, is used. This allows us to share the private key belonging to a Bitcoin address between two different devices. Both devices are then required to sign a transaction and the full private key is never recombined.

5.1. Description of the prototype. Our two-factor wallet consists of a desktop wallet in form of a Java graphical user interface, and a phone counterpart that is realized as an Android application. Only the desktop application is a full Bitcoin wallet, which stores and processes all incoming transactions relevant to the user. Consequently, only the desktop wallet can display the transaction history and the current balance. The phone wallet is only required when signing a new transaction and does not need to connect to the Bitcoin network at all, which makes the implementation much more lightweight.

In Figure 5.1, the dataflow when signing a transaction is displayed. When a user wants to send Bitcoins to another person, he starts by creating a Bitcoin transaction using the desktop wallet ①. When the transaction is ready for signing, the desktop wallet displays a QR-Code which contains the IP

address of the desktop wallet and the public key for a TLS connection. The desktop ad-hoc generates the key pair and a corresponding server certificate for the TLS connection.

The user now opens the smart phone wallet and scans the QR Code with the phone's camera ②. The smart phone wallet connects to the desktop wallet via the IP address specified in the QR code. The phone wallet establishes a TLS connection with the desktop wallet ③. During the connection setup, the phone wallet verifies that the public key from the desktop's certificate matches the public key in the QR code. This prevents any man-in-the-middle attacks.

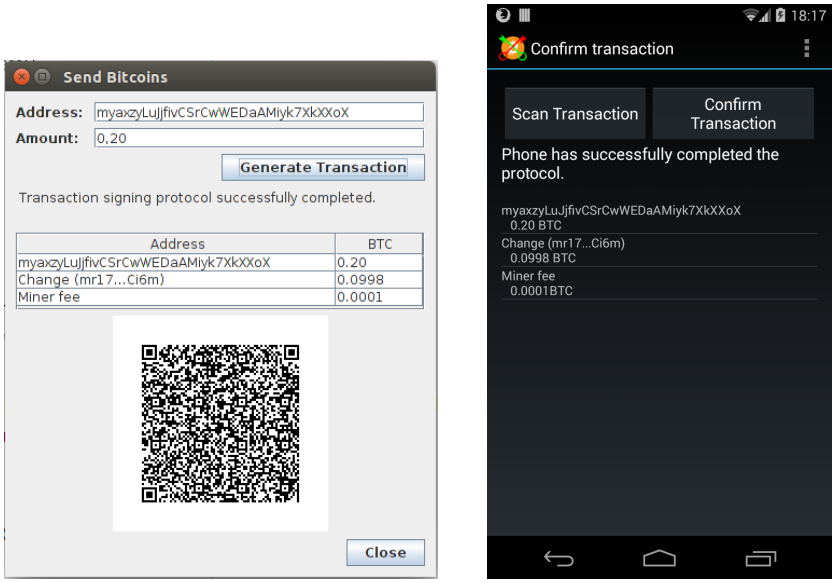
Over the secured connection, the phone wallet requests the transaction to be signed from the desktop wallet ④ and after receiving it from the desktop ⑤ displays it on the phone's screen ⑥. The user now has the possibility to review the transaction once again to make sure that is has not been modified by a compromised desktop wallet.

When the user confirms the transaction on the phone, the phone wallet asks the desktop wallet to start the two-party ECDSA signature protocol ⑦. The two wallets then exchange the messages required for the two-party ECDSA signature protocol over the TLS connection ⑧.

In the end, the desktop wallet holds the correct ECDSA signature for the transaction. It can now embed the signature into the transaction ⑨. Afterwards, the desktop wallet publishes the now correctly signed transaction to the Bitcoin network ⑩. Figure 5.2 shows the desktop and the phone wallet after successfully completing the two-party ECDSA protocol in ⑧.

We currently assume that the desktop and the phone wallet are located in the same, most likely wireless, local area network. Over the IP connection, the two wallets then establish a TLS channel as described above. Afterwards, the wallets exchange messages with the help of the Apache Avro serialization protocol over the TLS channel. To further reduce the attack surface, the two wallets could be connected via Bluetooth by using the Bluetooth network encapsulation protocol (BNEP) which allows to establish IP connections over Bluetooth. This only allows connections between previously paired devices. Therefore, attacks would become much harder as an attacker could not directly connect to the desktop wallet any more.

5.2. Runtime analysis. In general, solutions that use zero-knowledge proofs can be quite slow. We have benchmarked our prototype of a two-factor Bitcoin wallet to ensure that it achieves an acceptable execution time. Furthermore, we benchmarked a modified version of our prototype which uses Bitcoin's threshold signature support (see Section 3.3) instead of the two-party ECDSA signature protocol (see Section 4.2). The benchmarks were performed on a core-i5-2520M notebook running Ubuntu 14.04 with OpenJDK, and a Nexus 4 smart phone running Android 4.4.4. During the benchmark, the execution time of each prototype has been measured for



(a) Desktop GUI (b) Smartphone GUI

Figure 5.2: The desktop and the smart phone GUI after completing a transaction.

	1 input	2 inputs	3 inputs
Two-party ECDSA protocol	22.9s	44.6s	66.1s
Bitcoin's threshold signatures	2.3s	3.0s	4.0s

Figure 5.3: Timings for the prototypes using bitcoinj 0.11.3 and bouncycastle 1.50.

transactions which have one, two or three inputs. The execution time measured is the time it takes to run the complete protocol on the computer and the phone. The results are shown in Figure 5.3.

The first version of the prototype was quite slow and needed more than 20 seconds to sign a transaction with a single input. Most of the runtime was used by operations on the elliptic curve `secp256k1`. The prototype uses the bitcoinj library, which implements the Bitcoin protocol in Java, and the bouncycastle crypto library for the cryptographic operations on the elliptic curve. With the help of Mike Hearn, we found out that the version of bouncycastle we used at that time (bouncycastle 1.50 in combination with bitcoinj 0.11.3) contained an unoptimized implementation of the elliptic curve `secp256k1`. Therefore, the prototype using Bitcoin's threshold signature support also was quite slow. When signing a transaction with a single input, this prototype only needed to create two ECDSA signatures and it still took more than two seconds to sign the transaction.

	1 input	2 inputs	3 inputs
Two-party ECDSA protocol (2048 bit RSA)	3.8s	7.4s	11.1s
Two-party ECDSA protocol (3072 bit RSA)	7.3s	12.8s	18.6s
Bitcoin's threshold signatures	0.22s	0.18s	0.25s

Figure 5.4: Timings for the improved prototypes using bitcoinj 0.12 and bouncycastle 1.51.

We made some efforts to improve the performance. First of all, we upgraded to bitcoinj 0.12 and bouncycastle 1.51, which contains significantly improved code for the operations on the elliptic curve `secp256k1`. Second, most of the code has been parallelized to make use of the multithreading capabilities of modern processors. This is especially important as the single threading performance of smartphone CPUs is quite bad. The benchmark for the improved versions of the prototypes can be found in Figure 5.4. All measured execution times have improved significantly. Note, that the prototype using Bitcoin's threshold signature support now only needs 0.22 seconds to sign a transaction instead of 2.3 seconds. This shows how significant the performance improvements in the bouncycastle library are. We have also benchmarked a version of the two-party ECDSA signature protocol where the size of all RSA modulus type parameters has been increased to 3072 bits. RSA moduli of 3072 bit offer roughly the same level of security as 256 bit ECDSA keys (see Section 4.6). When comparing Bitcoin's threshold signatures with the 2048 bit version of the two-party ECDSA signature protocol, the two-party ECDSA signature protocol has a significantly longer execution time. On the other hand, when using online banking with SMS TAN, the user has to wait at least several seconds for the SMS. The execution time of our prototype is therefore well within the user's expectations.

As mentioned in Section 3.3, Bitcoin's built-in threshold signature support has the disadvantage of increasing the transaction size significantly. We have verified this by recording the sizes of the resulting transactions during a benchmark. The result in Figure 5.5 shows, that the transaction size increases by at least 40% when using Bitcoin's threshold signature support. It should be noted, that a transaction with only three inputs is already

	1 input	2 inputs	3 inputs
Two-party signature protocol	257 bytes	438 bytes	619 bytes
Bitcoin's threshold signatures	370 bytes	696 bytes	1022 bytes

Figure 5.5: Final size of signed transaction.

larger than 1000 bytes. The Bitcoin network expects a transaction fee for

transactions exceeding 1000 bytes while smaller transactions usually do not require a transaction fee. In general, each transaction is assigned a certain priority for it to be included into a new block. This priority is the ratio between the transaction fee and the transaction size. Consequently, a larger transaction needs a larger transaction fee to keep the same priority. The lower the priority, the longer the user has to wait for the confirmation of his transaction by the Bitcoin network. The transaction priority is not strict as each miner decides on his own which transactions to include into the block he mines, but usually miners try to fit as many transactions with as much fee as possible into a block to maximize their profit. Therefore, the above definition of priority is natural and is the one used by the standard Bitcoin implementation. As Bitcoin's threshold signature support produces larger transactions, the user must either increase the transaction fee or wait longer for the confirmation of the transaction. Commonly, the users will increase the transaction fee as transactions with a low priority can stay unconfirmed for a longer time. Consequently, using Bitcoin's threshold signature support has a financial cost for the user. Our approach with the two-party ECDSA signature protocol on the other hand is transparent to the Bitcoin network and as a result cost-neutral for the user.

5.3. Implementation aspects. During the implementation, some challenges came up which were not obvious in the beginning. In this section, we present these challenges and our solutions for them.

Displaying the transaction fee in the phone wallet As explained in Section 2, the transaction fee, which is payed to the miner, is the difference between the sum of Bitcoins in the transaction inputs and the sum of Bitcoins in the transaction outputs. The inputs actually only reference the outputs of preceding transactions. Consequently, to correctly compute the fee, one needs access to the preceding transactions. In our case, the phone itself must compute the overpay, which makes up the fee. Otherwise, the desktop can create a transaction which only contains benign outputs, but spends far too large inputs. The result would be a large fee for the miner and a financial damage to the user. Implementing full Bitcoin network access is possible as wallet software exists for Android, but would make the phone wallet much more complex. Instead, in our solution, the phone does not only request the transaction to be signed from the desktop, but also all transactions that are referenced in the inputs of the transaction to be signed. The phone verifies that the hash values of the provided transactions fit the hash values in the transaction inputs. In Figure 2.3, the reference from a transaction input to a preceding transaction and its output is depicted. Now, the phone can be sure that it has obtained the correct transactions and can use the information to compute the fee independently. The phone wallet can then display the fee to the user as shown in Figure 5.2.

Transactions with multiple inputs As mentioned in Section 2, a Bitcoin transaction can have multiple inputs. In this case, an ECDSA signature must be provided for each input. The signed hash value differs for each input, as some of the transaction's fields are emptied before the hash value is computed (see Section 2.1). Consequently, multiple ECDSA signatures are required to correctly sign a transaction with multiple inputs. These signatures are all created with the two-party ECDSA signature protocol and we rely on the fact that the protocol is secure under parallel execution as MacKenzie & Reiter have shown for the DSA version. This allows us to sign a transaction with multiple inputs with the same number of messages as required for a transaction with a single input. This is possible as the two-party ECDSA signature protocol can be executed in parallel for all inputs and the messages of the protocol runs can be combined into larger messages. The handling of a transaction with multiple signatures is also shown in the code sample in Figure A.1.

6. Future work

As our implementation is only a prototype, there is still some work to do. Most certainly, before using our software in production, a thorough code review is required to make sure that no implementation mistakes have been made both in the protocol itself and in the supporting code.

6.1. Improving the performance. Our prototype already achieves an acceptable execution time when signing a Bitcoin transaction, but there is still some space for improvements. Analyzing the prototype carefully, we found that most of the execution time is used by modular arithmetic on large integers. Especially operations of the Paillier crypto system are quite expensive as they require modular arithmetic with 5120 bit integers. Currently, the prototype uses the BigInteger class of the Java platform, which seemingly only contains a straight-forward implementation for integer multiplication. More efficient methods have been known for a long time now, see for example Karatsuba & Ofman (1963) or Schönhage & Strassen (1971). We point out that we are dealing with different implementations on the desktop and on the Android smart phone, as Android brings its own BigInteger implementation which utilizes native code.

6.2. Random number generation on Android. Several versions of Android were shipped with a broken default pseudo random number generator (PRNG), that has not been correctly seeded on start up. This allowed an attacker to recover the state of the PRNG. The details are described in Kim *et al.* (2013). This was fatal for several Android Bitcoin wallets which generated predictable private keys as described in Klyubin (2013). As Android devices often lack security updates, we must expect to deal with users with Android versions that are still vulnerable. Consequently, we must use a PRNG provided by our application and we must seed it correctly from a reliable entropy source. Furthermore, the protocol, especially the zero-knowledge proofs π_A and π_B, requires a large number of random values. Consequently, a good PRNG with a truly random seed is even more crucial than it would be for ECDSA alone.

6.3. Generation of parameters for the integer commitment scheme. The zero-knowledge proofs make use of the integer commitment scheme by Fujisaki & Okamoto (1997), which requires the verifier to generate certain parameters. These parameters include a RSA modulus consisting of two safe primes. The primes must indeed be safe for the scheme to work. The proof π_A which is verified by the phone is essential for the protocol's security. Therefore, a set of parameters for the Fujisaki & Okamoto integer commitment scheme including the safe primes, which are very expensive to generate, must be generated on the phone. It is possible to generate the parameters only once during the pairing phase in the beginning and reuse them after-

wards, but the generation is still very time consuming on the phone. We implemented the prime sieve idea from Wiener (2003) and have achieved a great speedup compared to our first trivial implementation, but on the phone the generation of a safe prime with 2048 bits still takes several minutes.

In Damgård & Fujisaki (2002), a generalization of the commitment scheme mentioned in Fujisaki & Okamoto (1997) is presented, where the requirement of safe primes has been relaxed to the requirement of strong primes. Generating RSA moduli consisting of strong primes is much cheaper. An efficient method for their generation is presented in von zur Gathen & Shparlinski (2013). Consequently, it would be favorable to adapt the protocol to use the generalized scheme from Damgård & Fujisaki (2002). As part of this adaption, the soundness proofs of π_A and π_B must be modified as the probability for a successful simulation of the initialization changes. A probability of 1/4 only holds when safe primes are used. Damgård & Fujisaki already reason about the probability of a successful simulation of the initialization.

6.4. Key derivation, backup and halting attackers. A Bitcoin address is directly derived from the user's ECDSA public key as depicted in Figure 2.2. Consequently, an attacker only needs control over the corresponding private key to spend all Bitcoins in a certain address. At the same time, when the user looses the private key all Bitcoins in the corresponding address are lost forever. This is a somewhat special case as in standard public key infrastructures lost signature keys can easily be replaced by creating a new key pair and then issuing a new certificate. The design of Bitcoin poses the special challenge on users to store their private keys securely and at the same time ensuring the availability with the help of backups. Furthermore, it is desirable to use a new address and consequently a new key pair for each transaction to provide a higher level of privacy to the user (see Section 2.3). The standard Bitcoin client just generates a new key pair for each transaction and stores it in the user's wallet. Therefore, the wallet file can easily contain hundreds of key pairs. The whole wallet file must now be backed up while being kept secret at the same time.

A solution to this problem is key derivation as described in Wuille (2014). When using such a scheme, all key pairs and consequently all Bitcoin addresses are derived from a single random seed with the help of a secure key derivation function using HMAC-SHA512. Hence, the user only needs to backup the seed securely, which is short enough to be, for example, written down and put into a safe.

A consequence of using our two-factor authentication, which requires two devices to sign a transaction, is that data loss or a halting attack on a single device makes the Bitcoins in the address under shared control inaccessible. Hence, it is highly reasonable to offer support for key derivation by implementing a modified version of the scheme described in Wuille (2014). This

would allow a user to easily backup his two-factor authenticated wallet by just securely storing the two seeds used by the two devices, for example with the piece of paper in a safe method.

7. Conclusion

We have presented a secure and efficient two-party ECDSA signature protocol and the corresponding pairing protocol which allows to setup all required parameters without relying on a trusted party.

Then, we have used this protocol to realize two-factor authentication for a Bitcoin wallet. As far as we know, we were able to implement the first fully functional prototype which is compatible with and completely transparent to the Bitcoin production network. Specifically, transactions created by our two-factor wallet are indistinguishable from standard Bitcoin transactions. This transparency is a unique feature that has not been available before and allows users to combine two-factor authentication with CoinJoin (Maxwell (2013)), which is a very promising solution for Bitcoin's privacy problem, without experiencing a degraded privacy.

A. Code samples

```
public Transaction addEncryptedSignaturesToTransaction(
      EncryptedSignatureWithProof[] encryptedSignatureMap){
    for(Map.Entry<Integer, DesktopSigner> entry : desktopSignerMap.entrySet()){
        int i = entry.getKey();
        DesktopSigner desktopSigner = entry.getValue();
        EncryptedSignatureWithProof encSignature = encryptedSignatureMap[i];
        if(encSignature == null)
            throw new ProtocolException("An encrypted signature is missing!");
        TransactionInput transactionInput = transaction.getInput(i);
        Script scriptPubKey = transactionInput.getConnectedOutput().
            getScriptPubKey();
        byte[] hash = transaction.hashForSignature(i,
        scriptPubKey, Transaction.SigHash.ALL, false).getBytes();
        ECKey.ECDSASignature ecdsaSignature = desktopSigner
            .decryptEncryptedSignature(encSignature, hash);
        TransactionSignature txSignature = new TransactionSignature(ecdsaSignature,
            Transaction.SigHash.ALL, false);
        transaction.getInput(i).setScriptSig(replaceSignatureInScriptSig(
            transactionInput.getScriptSig(), txSignature));
    }
    return transaction;
}
```

Figure A.1: Method from **DesktopTransactionSigner** that iterates over the signatures in a transaction and replaces the dummy signatures inserted by bitcoinj with the signatures created with the two-party ECDSA signature protocol.

```
public ECKey.ECDSASignature decryptEncryptedSignature(
     EncryptedSignatureWithProof encryptedSignature, byte[] hash){
    if(! state.equals(States.DecryptEncryptedSignature))
        throw new ProtocolException("Operation not allowed in this protocol state.
            ");
    if(encryptedSignature.getSigma().compareTo(BigInteger.ONE) < 0 ||
        encryptedSignature.getSigma().compareTo(pkpDesktop.getN().pow(2)) >= 0){
        throw new ProtocolException("Sigma is out of bounds.");
    }
    if(encryptedSignature.getAlphaPhone().compareTo(BigInteger.ONE) < 0 ||
        encryptedSignature.getAlphaPhone().compareTo(pkpPhone.getN().pow(2))
        >=0 ){
        throw new ProtocolException("alpha_B is out of bounds.");
    }

    BigInteger hm = new BigInteger(1, hash);
    BigInteger r = R.normalize().getAffineXCoord().toBigInteger().mod(nEC);
    BigInteger nsquared = pkpDesktop.getN().pow(2);
    ForkJoinTask<BigInteger> c1 = zkProofHelper.PowMult(alphaDesktop, hm,
        nsquared);
    ForkJoinTask<BigInteger> c2 = zkProofHelper.PowMult(beta, r, nsquared);
    encryptedSignature.getProof().verify(c1.join(), c2.join(),
    encryptedSignature.getSigma(), encryptedSignature.getAlphaPhone(), ECKey.
        CURVE.getG(), otherPublicKey, RPhone, pkpDesktop, pkpPhone,
        phoneBCParameters, zkProofHelper);
    BigInteger s = pkpDesktop.decrypt(encryptedSignature.getSigma()).mod(nEC);
    state = States.Finished;
    return new ECKey.ECDSASignature(r, s);
}
```

Figure A.2: Method from **DesktopSigner** that implements the last step of the two-party ECDSA signature protocol where the resulting ECDSA signature is created.

B. Source code

The source code of the prototype can be found on the attached CD or at `https://github.com/ChristopherMann/2FactorWallet`.

References

Accredited Standards Committee X9 (2005). ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Standard (ECDSA). Technical report, American National Standards Institute, American Bankers Association.

ANSSI (2014). *Mécanismes cryptographiques – Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques, Rev. 2.03*. Agence nationale de la sécurité des systèmes dinformation. URL `http://www.ssi.gouv.fr/uploads/2015/01/RGS_v-2-0_B1.pdf`.

Adam Back (2002). Hashcash - A Denial of Service Counter-Measure. URL `http://www.hashcash.org/papers/hashcash.pdf`.

Elaine Barker, William Barker, William Burr, William Polk & Miles Smid (2012). *NIST Special Publication 800-57 — Recommendation for Key Management - Part 1: General (Revision 3)*. National Institute of Standards and Technology. URL `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`.

Michael Ben-Or, Shafi Goldwasser & Avi Widgerson (1988). Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, 1–10. ACM, New York, NY, USA. ISBN 0-89791-264-0. URL `http://dx.doi.org/10.1145/62212.62213`.

Daniel J. Bernstein (2006). Curve25519: New Diffie-Hellman Speed Records. In Yung *et al.* (2006), 207–228. URL `http://dx.doi.org/10.1007/11745853_14`.

Daniel J. Bernstein, Chitchanok Chuengsatiansup & Tanja Lange (2014). Curve41417: Karatsuba Revisited. In *Cryptographic Hardware and Embedded Systems - CHES 2014*, Lejla Batina & Matthew Robshaw, editors, volume 8731 of *Lecture Notes in Computer Science*, 316–334. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-662-44708-6 (Print) 978-3-662-44709-3 (Online). URL `http://dx.doi.org/10.1007/978-3-662-44709-3_18`.

Bitcoin Wiki (2014). Script. URL `https://en.bitcoin.it/wiki/Script`.

Bitpay Inc. (2014). Copay: A secure Bitcoin wallet for friends and companies. URL `www.copay.io`.

Manuel Blum, Paul Feldman & Silvio Micali (1988). Proving Security Against Chosen Cyphertext Attacks. In *Advances in Cryptology: Proceedings of CRYPTO 1988*, Santa Barbara, CA, number 403 in Lecture Notes in Computer Science, 256–268. Springer-Verlag. ISSN 0302-9743.

Fabrice Boudot (2000). Efficient Proofs that a Committed Number Lies in an Interval. In *Advances in Cryptology - EUROCRYPT 2000*, Bart Preneel, editor, volume 1807 of *Lecture Notes in Computer Science*, 431–444. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-67517-4 (Print) 978-3-540-45539-4 (Online). URL `http://dx.doi.org/10.1007/3-540-45539-6_31`.

Certicom Research (2000). SEC 2: Recommended Elliptic Curve Domain Parameters. Technical report, Certicom Corporation.

Agnes Chan, Yair Frankel & Yiannis Tsiounis (1998). Easy come - Easy go divisible cash. In *Advances in Cryptology - EUROCRYPT 98*, Kaisa Nyberg, editor, volume 1403 of *Lecture Notes in Computer Science*, 561–575. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-64518-4 (Print) 978-3-540-69795-4 (Online). URL http://dx.doi.org/10.1007/BFb0054154.

David Chaum, Amos Fiat & Moni Naor (1990). Untraceable Electronic Cash. In *Advances in Cryptology - CRYPTO 88*, Shafi Goldwasser, editor, volume 403 of *Lecture Notes in Computer Science*, 319–327. Springer-Verlag, Berlin, Heidelberg. ISBN 978-0-387-97196-4 (Print) 978-0-387-34799-8 (Online). URL http://dx.doi.org/10.1007/0-387-34799-2_25.

Ivan Damgård & Eiichiro Fujisaki (2002). A Statistically-Hiding Integer Commitment Scheme Based on Groups with Hidden Order. In *Advances in Cryptology - ASIACRYPT 2002*, Yuliang Zheng, editor, volume 2501 of *Lecture Notes in Computer Science*, 125–142. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-00171-3 (Print) 978-3-540-36178-7 (Online). URL http://dx.doi.org/10.1007/3-540-36178-2_8.

Ivan Damgård & Mads Jurik (2001). A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography - PKC 2001*, Kwangjo Kim, editor, volume 1992 of *Lecture Notes in Computer Science*, 119–136. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-41658-6 (Print) 978-3-540-44586-9 (Online). URL http://dx.doi.org/10.1007/3-540-44586-2_9.

Kyle Drake (2011). Two-party ECDSA signature generation. URL http://www2.bitcoinjs.org/bitcoinjs-lib/demo/split-key.html.

Eiichiro Fujisaki & Tatsuaki Okamoto (1997). Statistical zero knowledge protocols to prove modular polynomial relations. In *Advances in Cryptology: Proceedings of CRYPTO 1997*, Santa Barbara, CA, B. S. Kaliski Jr., editor, volume 1294 of *Lecture Notes in Computer Science*, 16–30. Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-63384-7. ISSN 0302-9743. URL http://dx.doi.org/10.1007/BFb0052225.

Joachim von zur Gathen & Igor Shparlinski (2013). Generating safe primes. *Journal of Mathematical Cryptology* **7**(4), 333–365. ISSN 1862-2984 (Online) 1862-2976 (Print)). URL http://dx.doi.org/10.1515/jmc-2013-5011.

Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk & Tal Rabin (1996). Robust Threshold DSS Signatures. In *Advances in Cryptology - EUROCRYPT 96*, Ueli Maurer, editor, volume 1070 of *Lecture Notes in Computer Science*, 354–371. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-61186-8 (Print) 978-3-540-68339-1 (Online). URL http://dx.doi.org/10.1007/3-540-68339-9_31.

Steven Goldfeder, Joseph Bonneau, Edward W. Felten, Joshua A. Kroll & Arvind Narayanan (2014). Securing Bitcoin wallets via threshold signatures. URL http://www.cs.princeton.edu/~stevenag/bitcoin_threshold_signatures.pdf. Preprint.

Shafi Goldwasser & Silvio Micali (1984). Probabilistic Encrypion. *Journal of Computer and System Sciences* **28**, 270–299. URL http://groups.csail.mit.edu/cis/pubs/shafi/1984-jcss.pdf.

Matthew Green (2013). A few more notes on NSA random number generators. URL http://blog.cryptographyengineering.com/2013/12/a-few-more-notes-on-nsa-random-number.html.

Lein Harn (1994). Group-oriented (t, n) threshold digital signature scheme and digital multisignature. *Computers and Digital Techniques, IEE Proceedings* **141**(5), 307–313. URL http://dx.doi.org/10.1049/ip-cdt:19941293.

M.H. Ibrahim, I.A. Ali, I.I. Ibrahim & A.H. El-sawi (2003). A robust threshold elliptic curve digital signature providing a new verifiable secret sharing scheme. In *MWCAS03*, 276 – 280 Vol. 1. IEEE Computer Society, Cairo, Egypt. ISBN 0-7803-8294-3. ISSN 1548-3746. URL http://dx.doi.org/10.1109/MWSCAS.2003.1562272.

A. Karatsuba & Yu. Ofman (1963). Multiplication of multidigit numbers on automata. *Soviet Physics–Doklady* **7**(7), 595–596. Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July, 1962.

Soo Hyeon Kim, Daewan Han & Dong Hoon Lee (2013). Predictability of Android OpenSSL's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 659–668. ACM, New York, NY, USA. ISBN 978-1-4503-2477-9. URL http://dx.doi.org/10.1145/2508859.2516706.

Alex Klyubin (2013). Some SecureRandom Thoughts. URL http://android-developers.blogspot.de/2013/08/some-securerandom-thoughts.html.

Sébastien Kunz-Jacques, Gwenaëlle Martinet, Guillaume Poupard & Jacques Stern (2006). Cryptanalysis of an Efficient Proof of Knowledge of Discrete Logarithm. In Yung *et al.* (2006), 27–43. URL http://dx.doi.org/10.1007/11745853_3.

Susan K. Langford (1995). Threshold DSS Signatures without a Trusted Party. In *Advances in Cryptology - Crypto 95*, D. Coppersmith, editor, volume 963 of *Lecture Notes in Computer Science*, 397–409. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-60221-7 (Print) 978-3-540-44750-4 (Online). URL http://dx.doi.org/10.1007/3-540-44750-4_32.

Robert Lipovsky (2013). New Hesperbot targets: Germany and Australia. URL http://www.welivesecurity.com/2013/12/10/new-hesperbot-targets-germany-and-australia/.

Philip MacKenzie & Michael K. Reiter (2004). Two-party generation of DSA signatures. *International Journal of Information Security* **2**(3-4), 218–239. URL http://dx.doi.org/10.1007/s10207-004-0041-0.

Christopher Mann & Daniel Loebenberger (2014). Two-factor authentication for the Bitcoin protocol. *Cryptology ePrint Archive* **2014/629**. URL http://eprint.iacr.org/2014/629.

Gregory Maxwell (2013). CoinJoin: Bitcoin privacy for the real world. URL https://bitcointalk.org/index.php?topic=279249.0.

Ian Miers, Christina Garman, Matthew Green & Aviel D. Rubin (2013). Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, 397–411. IEEE Computer Society. ISBN 978-0-7695-4977-4. URL http://dx.doi.org/10.1109/SP.2013.34.

Satoshi Nakamoto (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. Cryptography Mailing list at metzdowd.com. URL https://bitcoin.org/bitcoin.pdf. 9 pages.

NIST (2012). *Federal Information Processing Standards Publication 180-4 - Secure Hash Standard*. National Institute of Standards and Technology. URL http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf. Federal Information Processings Standards Publication 180-4.

NIST (2013). Federal Information Processing Standards Publication 186-4 - Digital Signature Standard (DSS). Technical report, Information Technology Laboratory, National Institute of Standards and Technology. URL http://dx.doi.org/10.6028/NIST.FIPS.186-4.

Chris Pacia & Alon Muroch (2014). bitcoin authenticator. URL https://www.bitcoinauthenticator.org.

Pascal Paillier (1999). Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology: Proceedings of EUROCRYPT 1999,* Prague, Czech Republic, J. Stern, editor, volume 1592 of *Lecture Notes in Computer Science*, 233–238. Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-65889-0. ISSN 0302-9743. URL http://dx.doi.org/10.1007/3-540-48910-X_16.

David Pointcheval & Jaques Stern (2000). Security arguments for digital signatures and blind signatures. *Journal of Cryptology* **13**(3), 361–396. ISSN 0933-2790 (Print) 1432-1378 (Online).

T. Pornin (2013). Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). URL http://tools.ietf.org/html/rfc6979. RFC 6979.

Dorit Ron & Adi Shamir (2013a). How Did Dread Pirate Roberts Acquire and Protect His Bitcoin Wealth? Cryptology ePrint Archive, Report 2013/782. URL eprint.iacr.org/2013/782.pdf.

Dorit Ron & Adi Shamir (2013b). Quantitative Analysis of the Full Bitcoin Transaction Graph. In *Financial Cryptography and Data Security*, Ahmad-Reza Sadeghi, editor, volume 7859 of *Lecture Notes in Computer Science*, 6–24. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-39883-4 (Print) 978-3-642-39884-1 (Online). ISSN 0302-9743. URL http://dx.doi.org/10.1007/978-3-642-39884-1_2.

David Sancho, Feike Hacquebord & Rainer Link (2014). Finding Holes Operation Emmental. Technical report, Trend Micro Incorporated. URL http://housecall.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-finding-holes-operation-emmental.pdf.

Jack Schmidt (2012). Answer to "How to compute a generator of this cyclic quadratic residue group?". URL http://math.stackexchange.com/questions/167478.

Arnold Schönhage & Volker Strassen (1971). Schnelle Multiplikation großer Zahlen. *Computing* **7**, 281–292.

Adi Shamir (1979). How to Share a Secret. *Communications of the ACM* **22**(11), 612–613.

Technical Committee X3J14 (1994). ANSI X3.215-1994, Forth (Working Document). Technical report, American National Standards Institute. URL http://www.forth.org/svfig/Win32Forth/DPANS94.txt.

Chih-Hung Wang & Tzonelih Hwang (1997). (t+1,n) threshold and generalized DSS signatures without a trusted party. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC 97)*, 221–226. IEEE. ISBN 0-8186-8274-4. URL http://dx.doi.org/10.1109/CSAC.1997.646193.

Michael J. Wiener (2003). Safe Prime Generation with a Combined Sieve. *Cryptology ePrint Archive* **2003/186**. URL http://eprint.iacr.org/2003/186.

Pieter Wuille (2014). BIP32 Hierarchical Deterministic Wallets. URL https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

Moti Yung, Yevgeniy Dodis, Aggelos Kiayias & Tal Malkin (editors) (2006). *Public Key Cryptography - PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-33851-2 (Print) 978-3-540-33852-9 (Online). URL http://dx.doi.org/10.1007/11745853.