Adaptor Signature Based Atomic Swaps Between Bitcoin and a Mimblewimble Based Cryptocurrency

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Jakob Abfalter, BSc
Registration Number 01126889

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Matteo Maffei
Assistance: Dr. Pedro Moreno-Sanchez

Vienna, 6th April, 2020

Jakob Abfalter	Matteo Maffei

Erklärung zur Verfassung der Arbeit

Jakob Abfalter, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. April 2020

Jakob Abfalter

Acknowledgements

Enter your text here.

Abstract

Enter your text here.

Contents

CHAPTER 1

Introduction

Pedro: We need to discuss a structure for the introduction. Proposal:
- Introduce why coin exchanges are interesting
- Explain why atomic swaps protocols (e.g., one could use a trusted server for this and problem solved, right?)
- Why coin exchanges between Bitcoin and Mimblewimble?
- Why what you are proposing in this thesis is challenging?
- What are the main contributions of these thesis?
- What do you think is an interesting future research direction?

Mimblewimble The Mimblewimble protocol was introduced in 2016 by an anonymous entity named Jedusor, Tom Elvis [10]. The author's name, as well as the protocols name, are references to the Harry Potter franchise. [1] In Harry Potter, Mimblewimble is a tongue-typing curse which reflects the goal of the protocol's design, which is improving the user's privacy. Later, Andrew Poelstra took up the ideas from the original writing and published his understanding of the protocol in his paper [19]. The protocol gained increasing interest in the community and was implemented in the Grin [2] and Beam [3] Cryptocurrencies, which both launched in early 2019. In the same year, two papers were published, which successfully defined and proved security properties for Mimblewimble [7, 5].

Pedro: I would not add a line break at the end of each paragraph. The template should do that

Pedro: If you are going to compare to Bitcoin, you need to introduce Bitcoin before

Compared to Bitcoin, there are some differences in Mimblewimble:

[1] `https://harrypotter.fandom.com/wiki/Tongue-Tying_Curse`
[2] `https://grin.mw/`
[3] `https://beam.mw/`

- Use of Pedersen commitments instead of plaintext transaction values

  Pedro: The reader does not know what Pedersen commitments are at this point. Perhaps say transaction values are hidden from a blockchain observer while this is not the case in Bitcoin

- No addresses. Coin ownership is given by the knowledge of the opening of the coins Pedersen commitment.

  Pedro: This is also unclear. Could one see the commitment as the "address" in Mimblewimble? Perhaps you want to say that there is no scripting language supported?

- Spend outputs are purged from the ledger such that only unspent transaction outputs remain.

- No scripting features.

Pedro: Use "we" for contributions that you do in the thesis and "they" for parts that are borrowed from other works

Pedro: An intuition of these two terms is required here

Pedro: another sentence that shows that you need to explain before how Bitcoin works (the basics)

By utilizing Pedersen commitments in the transactions, we hide the amounts transferred in a transaction, improving the systems user privacy, but also requiring additional range proofs, attesting to the fact that actual amounts transferred are in between a valid range. Not having any addresses enables transaction merging and transaction cut through, which we will explain in section **??**. However, this comes with the consequence that building transactions require active interaction between the sender and receiver, which is different than in constructions more similar to Bitcoin, where a sender can transfer funds to any address without requiring active participation by the receiver. Through transaction merging and cut-through and some further protocol features, which we will see later in this section, we gain the third mentioned property of being able to delete transaction outputs from the Blockchain, which have already been spent before. This ongoing purging in the Blockchain makes it particularly space-efficient as the space required by the ledger only grows in the number of UTXOs, in contrast to Bitcoin, in which space requirement increases with the number of overall mined transactions. Saving space is especially relevant for Cryptocurrencies employing confidential transactions because the size of the range proofs attached to outputs can be significant.

Pedro: What comes next is hard to read. It requires better organization: Advantages of Mimblewimble are: (i) .., (ii)...; Disadvantages are: (i)..., (ii),...).

Another advantage of this property is that new nodes joining the system do not have to verify the whole history of the Blockchain to validate the current state, making it much easier to join the network. Another limitation of Mimblewimble- based Cryptocurrencies is that at least the current construction does not allow scripts, such as they are available in Bitcoin or similar systems. Transaction validity is given solely by a single valid signature plus the balancedness of inputs and outputs. This shortcoming makes it challenging to realize concepts such as multi signatures or conditional transactions which are required for Atomic Swap protocols. However, as we will see in **??** there are ways we can still construct the necessary transactions by merely relying on cryptographic primitives [7].

CHAPTER 2

Motivation & Objectives

TODO

CHAPTER 3

Preliminaries

In this chapter we will lay down the general notations and definitions required for the later parts of the thesis. In section 3.1 we will define several cryptographic primitives which are required for our constructions. Section 3.2 will describe several definitions around Bitcoin, particularly its transaction structure. After that in secion 3.3 we will discuss the notion of privacy enhancing cryptocurrencies, and then range proofs in section 3.3.2 of which both are needed to understand the Mimblewimble protocol discussed in section 3.4. Finally we explain the concept of scriptless scripts in section 3.5 and adaptor signatures 3.6 which are both relevant building blocks for the constructions found in this thesis.

3.1 General Notation and Definitions

Notation We first define the general notation used in the following chapters to formalize procedures and protocols. Let $\mathbb{G}$ denote a cyclic group of prime order p and $\mathbb{Z}_p$ the ring of integers modulo p with identity element 1_p. $\mathbb{Z}_p^*$ is $\mathbb{Z}_p \setminus \{0\}$. $g,\ h$ are adjacent generators in $\mathbb{G}$, where adjacent means the discrete logarithm of h in regards to g is not known. Exponentiation stands for repeated application of the group operation. We define the group operation between two curve points as $g^{a} \ \cdot \ g^{g^{b}} \ \stackrel{?}{=} \ g^{a \ + \ b}$.

Definition 3.1 (Hard Relation)**.** Given a language $L_R \ := \ \{A \mid \exists a \text{ s.t. } (A, a) \in R\}$ then the relation R is considered hard if the following three properties hold: [3]

1. $\mathsf{genRel}((\mathit{1}^{n}))$ is a PPT sampling algorithm which outputs a statement/witness of the form $(A, a) \in R$.

2. Relation R is poly-time decidable.

3. For all PPT adversaries $\mathcal{A}$ the probability of finding a given A is negligible.

Definition 3.2 (Discrete Logarithm)**.** We define the discrete logarithm in a group $\mathbb{G}$ of a number n as the number m such that for the groups generator g the following holds:

$$g^m \;=\; n$$

The discrete logarithm is a hard relation as defined in 3.1.

Definition 3.3 (Signature Scheme)**.** A signature scheme Φ is a tuple of algorithms $(\mathsf{keyGen},\ \mathsf{sign},\ \mathsf{verf})$ defined as follows: [8]

$$\Phi = (\mathsf{keyGen},\ \mathsf{sign},\ \mathsf{verf})$$

- $(sk, pk)\ \leftarrow\ \mathsf{keyGen}(1^n)$: The keygen function creates a keypair (sk, pk), the public key can be distributed to the verifier(s) and the secret key has to be kept private.

- $\sigma\ \leftarrow\ \mathsf{sign}(m, sk)$: The signing function creates a signature consisting of a variable s and R which is a commitment to the secret nonce n used during the signing process. As an input it takes a message m and the secret key sk of the signer.

- $\{1, 0\}\ \leftarrow\ \mathsf{verf}(m, \sigma, pk)$: The verification function allows a verifier knowing the signature σ, message m and the provers public key pk to verify the signatures validity.

A valid signature scheme has to fulfill two security properties:

- Correctness: For all messages m and valid keypairs (sk, pk) the following must hold with overwhelming probability: $\mathsf{verf}(pk, \mathsf{sign}(sk, m), m)\ \stackrel{?}{=}\ 1$

- Unforgeability $(\mathsf{EUF-CMA}\ )$: Informally the existential unforgeability under chosen message attacks holds if an attacker $\mathcal{A}$ is unable to forge a valid signature for a chosen message. A formalization of the property can be found in section 4.4.2

Definition 3.4 (Cryptographic Hash Function)**.** A cryptographic hash function H is defined as $\mathsf{H}(I) \rightarrow \{0, 1\}^n$ for some fixed number n and some input I [1]. A secure hashing function has to fulfill the following security properties:

- Collision-Resistance (CR): Collision-Resistance means that it is computationally infeasible to find two inputs I_1 and I_2 such that $\mathsf{H}(I_1)\ :=\ \mathsf{H}(I_2)$ with $I_1\ \neq\ I_2$.

- Pre-image Resistance (Pre): In a hash function H that fulfills Pre-image Resistance it is infeasible to recover the original input I from its hash output $\mathsf{H}(I)$. If this security property is achieved, the hash function is said to be non-invertible.

- 2nd Pre-image Resistance (Sec): This property is similar to Collision-Resistance and is sometimes referred to as *Weak Collision-Resistance*. Given such a hash function H and an input I, it should be infeasible to find a different input I^* such that $I \neq I^*$ and $\mathsf{H}(I) \stackrel{?}{=} \mathsf{H}(I^*)$.

The relation between the input I and the output $\mathsf{H}(I)$ is a hard relation as defined in 3.1.

Definition 3.5 (Commitment Scheme)**.** A cryptographic Commitment Scheme *COM* is defined by a pair of functions $(\mathsf{keyGen}, \mathsf{commit})$ [4].

- $rs \leftarrow \mathsf{setupCom}(1^n)$: The setup procedure is a DPT function, it takes as input a security parameter 1^n and outputs public parameters PP. Depending on PP we define a input space $\mathbb{I}_{PP}$, a randomness space $\mathbb{K}_{PP}$ and a commitment space $\mathbb{C}_{PP}$.
- $C \leftarrow \mathsf{commit}(I, n)$ The commit routine is DPT function that takes an arbitrary input $I \in \mathbb{I}_{PP}$, a random value $n \in \mathbb{K}_{PP}$ and generates an output $C \in \mathbb{C}_{PP}$.

Secure commitments must fulfill the *Binding* and *Hiding* security properties:

- *Binding:* If a Commitment Scheme is binding it must hold that for all *PPT* adversaries $\mathcal{A}$ given a valid input $I \in \mathbb{I}_{PP}$ and randomness $n \in \mathbb{K}_{PP}$ the probabilty of finding a $I^* \neq I$ and a n^* with $\mathsf{commit}(I, n) = \mathsf{commit}(I^*, n^*)$ is negligible.
- *Hiding:* For a *PPT* adversary $\mathcal{A}$, commitment inputs $I_0, I_1 \in \mathbb{I}_{PP}$ randomness $n \in \mathbb{K}_{PP}$ and a commitment output $C := \mathsf{commit}(I_b, n,)$ the probabilty of the adversary choosing the correct b out of $\{0, 1\}$ must not be higher then $\frac{1}{2} + \mathsf{negl}(P)$.

Definition 3.6 (Additive Homomorphic Commitment)**.** A Commitment Scheme as defined in 3.5 is said to be addtive homomorphic if the following holds [4]

$$\mathsf{commit}(I_1, n_1) \cdot \mathsf{commit}(I_2, n_2) = \mathsf{commit}(I_1 + I_2, n_1 + n_2)$$

Definition 3.7 (Pedersen Commitment Scheme)**.** A *Pedersen Commitment Scheme* is an instance of a Commitment Scheme as defined in definition 3.5 that has the additive homomorphic property as defined in 3.6.

This can be achieved as follows: $\mathbb{C}_{PP} := \mathbb{G}$ of order p, $\mathbb{I}_{PP}$, $\mathbb{K}_{PP} := \mathbb{Z}_p$. the procedures $(\mathsf{setupCom}, \mathsf{commit})$ are then instantiated as:

$$\begin{aligned} rs \leftarrow \mathsf{setupCom}(g, h) &:= g, h \leftarrow (g, h) \\ C \leftarrow \mathsf{commit}(I, n) &:= g^n h^I \end{aligned}$$

An instantiation of the pedersen commitment scheme must pick two adjacent generators g, h for the setup to be secure in terms of hiding and binding. Formally adjacent means that there exists a hard relation between g and h in terms of the discrete logarithm 3.2. That means no x is known such that $h = g^x$. In practice this is often achieved by hashing g and using the hash output as h.

To prove the security of our protocols we define the notion of security in the presence of malicious adversaries, which may deviate from the protocol arbitrarily. To construct the definition we must first define two terms, IDEAL the execution in the ideal model and REAL, the execution in the real model. The following definitions are based on a tutorial paper on simulation proofs by Yehuda Lindell. [13]

Execution in the Ideal Model We have two parties P_1 with input x and P_2 with input y that cooperate to compute a two-party functionality $\mathsf{f} : \{0,1\}^* \times \{0,1\}^* \rightarrowtail \{0,1\}^* \times \{0,1\}^*$. The adversary $\mathcal{A}$ either controls P_1 or P_2. The ideal execution IDEAL relies on the assumption that we have access to a trusted third party and proceeds in the following steps:

1. **Inputs:** The input of P_1 is x and the input of P_2 is y. Both parties get an additional auxiliary input z. We note that we can generalize the concept to functions which require multiple inputs or even functions which do not require any input. In the case of multiple inputs the inputs of P_1 would then b a list $[x_i]$ and the inputs of P_2 a list $[y_i]$. For the case of simplicity we here describe the case with one single parameter provided by each party.

2. **Send Inputs:** The honest party (the one which is not controlled by $\mathcal{A}$) sends its input x (resp. y) to the trusted third party. The malicious party can either abort the execution by sending the symbol abort to the trusted third party, send its input x (resp. y), or send an arbitrarily chosen string k with the same length to x to proceed with the protocol execution. The decision is made by $\mathcal{A}$ and may depend on the input or auxilary input z. We denote (x^*, y^*) as the inputs received by the trusted third party. If P_1 is malicious then $(x^*, y^*) = (k, y)$, if P_2 is malicious then $(x^*, y^*) = (x, k)$.

3. **Abort:** If the trusted third party has received abort from one of the parties, then it sends abort to both parties.

4. **Answer to Adversary:** After having received both inputs the trusted third party computes $\mathsf{f}(x^*, y^*) = (\mathsf{f}_1(x^*, y^*), \mathsf{f}_2(x^*, y^*))$ and proceeds by sending $\mathsf{f}_1(x^*, y^*)$ (respective $\mathsf{f}_2(x^*, y^*)$) to the adversary.

5. **Adversary Instructs Trusted Party:** $\mathcal{A}$ now again has the option of sending abort to the trusted third party to stop the execution. Otherwise it may send $\mathsf{continue}$ which means the output $\mathsf{f}_1(x^*, y^*)$ (respective $\mathsf{f}_2(x^*, y^*)$) will be delivered to the honest party.

6. **Outputs:** The honest party outputs the answer of the trusted third party. The malicious party may output an arbitrary function of its input, the auxiliary string z, or the answer for the trusted party.

Let $\mathcal{A}$ be a non-uniform PPT algorithm and $i \in \{1, 2\}$ be the index of the corrupted party. We then denote $\mathsf{IDEAL}_{\mathsf{f},\mathsf{P}(z),i}(x, z)$ as the ideal execution of f on inputs (x, y) with auxiliary input z to $\mathcal{A}$ and security param 1^n defined as the output pair of the honest party and $\mathcal{A}$ from the ideal execution.

Execution in the Real Model Again let $\mathcal{A}$ be a non-uniform PPT adversary and $i \in \{1, 2\}$ be the index of the corrupted party. In this model a real two-party protocol γ is executed but the adversary $\mathcal{A}$ sens alls messages in place of the corrupted party, and may follow an arbitrary polynomial-time strategy. Then the real execution of the two-party protocol γ between P_1 and P_2 on inputs (x, y) and auxiliary input z to $\mathcal{A}$ and security parameter 1^n is denoted by $\mathsf{REAL}_{\mathsf{f},\mathsf{P}(z),i}(x, z)$ and is defined as the output pair of the honest party and the adversary $\mathcal{A}$ from the real execution of γ.

Definition 3.8 (Security in the Malicious Setting)**.** We say a two-party protocol γ securely computes a function f with aborts and inputs (x, y) in the malicious setting if for every non-uniform PPT adversary $\mathcal{A}$ in the real model, there exists a non-uniform PPT algorithm $\mathcal{S}$, refered to as simulator, such that

$$\{\mathsf{IDEAL}_{\mathsf{f},\mathcal{S}(z),i}(x, z) \ \equiv_c \ \mathsf{REAL}_{\mathsf{f},\mathcal{A}(z),i}(x, z)\}$$

where $|x| = |y|$ and $z = \mathsf{poly}(|\mathsf{x}|)$. [13]

3.2 Bitcoin

In this section we will discuss the basics of the Bitcoin transaction protocol. We will find definitions which we will use later in section 5.5 to construct an atomic swap protocol. The main reference of this section is the book Mastering Bitcoin by Andreas Antonopoulos [2].

3.2.1 Bitcoin Transaction Protocol

A *Bitcoin Transaction* is a data structure which allows transferring value between participants of the network. In Bitcoin there are no user balances or user accounts, instead the UTXO model (unspent transaction outputs) is empoloyed. An UTXO is a output constructed in a previous transaction which holds value in the form of an amount expressed in Bitcoin (more precisely in Satoshis, which is the smallest unit of Bitcoin) and a locking condition (referred to as scriptPubKey). Unspent means that this output has not been spent yet in a transaction and its funds are therefore available to be redeemed by a participant capable of unlocking the output. To unlock this value one has to provide a script fulfilling the locking condition, referred to as scriptSig. In the most common case the lock condition will be to provide a valid signature under a public key. This is referred to as a P2PK or P2PKH output which we will see in more detail in section 3.2.1. However, more complex conditions, such we shall see in section 3.2.1 are possible.

Definition 3.9 (Unspent Transaction Output - UTXO)**.** An unspent transaction output is a data structure consisting of a locking condition spk, a value expressed in Bitcoin v and an unlocking script σ which is initially empty and has to be provided by the owner when spending the UTXO in a transaction. In this paper we generally use ψ to refer to a singular UTXO and Ψ to refer to a set of UTXOs.

$$\psi \ := \ \{v, \ spk, \ \sigma\}$$

We define three auxiliary functions for the creation, spending and verification of an UTXO. Note that we use verf as a generalization of a verification function. In practice verifcation of a UTXO will most of the time correspond to the verification of a digital signature. However, as we shall see in 3.2.1 this is not necessarily always the case.

$\mathsf{createUTXO}(v, spk)$

1 : **return** $\psi \ := \ \{v \ := \ v, spk \ := \ spk, \sigma \ := \ \emptyset\}$

$\mathsf{spendUTXO}(\psi, \sigma)$

1 : $\{v, spk\} \ \leftarrow \ \psi$
2 : **return** $\psi \ := \ \{v \ := \ v, spk \ := \ spk, \sigma \ := \ \sigma\}$

$\mathsf{verfUTXO}(\psi)$

1 : $\{v, spk, \sigma\} \ \leftarrow \ \psi$
2 : **return** $\mathsf{verf}(spk, \sigma, v)$

Now a full transaction consists of one, or many UTXOs as inputs and one or many UTXOs as output. For the transaction to be considered valid the σ fields in the inputs need the be correctly filled, and the value in the newly created output UTXOs must not exceed the value stored in the spending UTXOs. A value lower than what is provided in the inputs is allowed, this means the miner of the transaction gets to collect the difference as a fee. The higher this fee, the more incentive the miners will have to include your transaction in the blockchain. Additionally a transaction consists of a version number, and a locktime field which semantically means that a transaction will only be seen as valid after a certain block number in the Bitcoin blockchain was mined. Figure 3.1 shows a decoded Bitcoin transaction. [1]

Definition 3.10 (Bitcoin Transaction)**.** A Bitcoin transaction consists of a series of input UTXOs Ψ_{inp}, a series of output UTXOs Ψ_{out}, a transaction version vs, and an optional locktime t:

$$tx_{btc} \ := \ \{vs, t, \Psi_{inp}, \Psi_{out}\}$$

[1] https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298ca
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG",
    }
  ]
}
```


Figure 3.1: A decoded Bitcoin transaction

A transaction is valid if the follwing conditions are fullfilled:

- The total value of inputs is greater or equal the total value of outputs.
- For all $\psi \in \psi$ $\mathsf{verfUTXO}(\psi) = 1$ must hold.
- All input UTXOs have not been spent before.
- If a locktime t is given, the current block on the Bitcoin blockchain needs to be higher or equal t.

Definition 3.11 (Bitcoin Transaction Scheme)**.** We define a Bitcoin Transaction scheme as a tupel of three DPT functions $(\mathsf{buildTransaction}, \mathsf{signTransaction}, \mathsf{verfTransaction})$.

- $tx_{btc} \leftarrow \mathsf{buildTransaction}(\Psi_{inp}, \Psi_{out}, vs, t)$: The transaction building algorithm is a DPT function which takes as input a set of unspent transaction outputs Ψ_{inp}, a set of newly created transaction outputs Ψ_{out} a version number vs and a optional locking time t. The algoritm will output an unsigned transaction tx_{btc}.
- ${tx_{btc}}^* \leftarrow \mathsf{signTransaction}(tx_{btc}, [\sigma])$: The transaction signing algoritm is DPT function which takes as input a unsigned Bitcoin transaction tx_{btc} and an array of unlocking scripts $[\sigma]$ for all inputs of the transaction. The algoritm outputs a signed Bitcoin transaction which can now be broadcast to the network.
- $\{1, 0\} \leftarrow \mathsf{verfTransaction}(tx_{btc})$: The verification algorithm is a DPT function taking as input a transaction tx_{btc} outputing 1 on a successfull verification or 0 otherwise. The function will check the well-balancedness of the transaction,

verify the unlocking scripts, locktime as well as scanning through the blockchain if all inputs are indeed unspent. Note that any public verifier with access to the blockchain ledger and tx_{btc} will be able to perform the verification.

Following we will outline two common structures of Bitcoin outputs the P2PK/P2PKH and the P2SH outputs.

P2PK, P2PKH

P2PK stands for Pay-to-Public-Key and P2PKH for Pay-to-Public-Key-Hash. In this type of output *spk* will be constructed such that its value unlocks if a correct signature is provided in σ for a corresponding public key *pk*. P2PKH is an update to this script in which the *spk* contains a hashed version of the public key *pk*, instead of the public key itself. To spend a P2PKH output one has to provide the unhashed public key in addition to a valid signature. This type of output, is the most commonly used output in the Bitcoin blockchain to transfer value from one participant to another. Delgado et at. found in their paper Analysis of the Bitcoin UTXO set from 2017 that more then 80% of the UTXO set at that time consisted of P2PKH transactions, whereas about 17% were P2SH and 0.12% P2PK outputs. [6] P2PKH outputs can be encoded into a Bitcoin address using base58 encoding. This addresses can be handed out to request a payment from somebody.

P2SH

If more advanced spending conditions, such as multi signature are required, P2SH (Pay-to-script-hash), introduced in 2012, is a way to implement those in a space efficient and simple matter. Here the locking condition *spk* does not contain a script, but instead the hash of a script. Upon spending the spender has to provide the original script as well as the unlocking requirements for the script itself. Upon verification the hash of the provided script will be computed and compared with the value given in the locking condition, if those match the actual script will be executed. The advantages of using this approach over just handcrafting a custom locking script is that the locking scripts are rather short making the transactions smaller and therefore reducing fees, or rather shifting the fees from the sender to the owner of the output. Additionally this type of output can be encoded again into a Bitcoin address similar to a P2PKH output, making it easy to request a payment.

3.3 Privacy-enhancing Cryptocurrencies

3.3.1 Zero Knowledge Proofs

3.3.2 Range Proofs

Definition 3.12 (Range proof System)**.** A range proof system $\Pi_{RP}[COM]$ with regards to a homomorphic commitment scheme COM consists of a tupel of functions

$(\mathsf{ranPrfSetup}, \mathsf{ranPrf}, \mathsf{vrfRanPrf})$.

- $ps \leftarrow \mathsf{ranPrfSetup}(1^n, i, j)$: The rangeproof setup algorithm takes as input a security paramter 1^n as well as two numbers i and j which are treated as exponents of 2 to define the lower and upper bound of the rangeproof protocol.

- $\pi \leftarrow \mathsf{ranPrf}(C, v, r)$: The proof algorithm is a DPT function which takes as input a commitment C a value v and a blinding factor r. It will output a proof π attesting to the statement that the value v of commitment C is in between the range $\langle lb, ub\rangle$ as defined during the $\mathsf{ranPrfSetup}$ function.

- $\{1, 0\} \leftarrow \mathsf{vrfRanPrf}(\pi, C)$: The proof verification algorithm is a DPT function which verifies the validity of the proof π with regards to the commitment C. It will output 1 upon a successfull verification or 0 otherwise.

Definition 3.13 (Multiparty Rangeproof System)**.** A Multiparty Rangeproof Sytem $\Pi_{RP-MP}[COM]$ with regards to a homomorphic commitment scheme COM is an extension of the regular Rangeproof Sytem with the following distributed protocol **dRanPrf**.

- $\pi \leftarrow \mathbf{dRanPrf}\langle(C, v, r_A), (C, v, r_B)\rangle$: The distributed proof protocol allows two parties Alice and Bob, each owning a share of the commitment C to cooperate in order to produce a valid range proof π without a party learning the blinding factor share from the other party.

For MP proofs [11]

3.4 Mimblewimble

In this section we will outline the fundamental properties of the protocols employed in Mimblewimble which are relevant for the thesis and particularly the construction of the Atomic Swap protocol constructed in chapter 5.

Transaction Structure

First we will define the notion of a coin in Mimblewimble which has similarity to an unspent transaction output (UTXO) in Bitcoin.

Definition 3.14 (Mimblewimble Coin)**.** For two adjacent elliptic curve generators g and h a coin in Mimblewimble is a tuple of the form $(\mathcal{C}, \pi)$, where $\mathcal{C} := g^v \cdot h^n$ is a Pedersen Commitment [18] to the value v with blinding factor n. π is a range proof attesting to the statement that v is in a valid range in zero-knowledge. The valid range is defined by the specific implementation, in pratice $\langle 0, 2^{64} - 1\rangle$ is used in the most prominant implementations.

A Mimblewimble transaction consists of $\mathcal{C}_{inp} := (\mathcal{C}_1, \ldots, \mathcal{C}_n)$ input coins, $\mathcal{C}_{out} := (\mathcal{C}'_1, \ldots, \mathcal{C}'_n)$ output coins and kernel K, which we will define throughout this section.

Definition 3.15 (Transaction well-balancedness)**.** A transaction is considered *well-balanced* if for a list of input coins with values $[v]$, a list of output coins with values $[v^*]$, and a fee f $\sum[v] - \sum[v^*] - f = 0$ so the sum of all output values and the fee subtracted from the sum of input values has to be 0.

Definition 3.16 (Transaction validity)**.** A transaction is valid if:

- The transaction is well-balanced as defined in definition 3.15
- $\forall\ (\mathcal{C}_i \pi_i) \in \mathcal{C}_{out}\ \mathsf{vrfRanPrf}(\pi_i, \mathcal{C}_i) = 1$

From the definition of *Transaction validity* we can derive the following equation:

$$\sum \mathcal{C}_{out} - \sum \mathcal{C}_{inp} = \sum (h^{v'_i} \cdot g^{n'_i}) - \sum (h^{v_i} \cdot g^{n_i}) - h^f$$

So if we assume that a transaction is valid then we are left with the following so called excess value:

$$\mathcal{E} = g^e = g^{(\sum n'_i - \sum n_i)}$$

Knowledge of the opening of all coins, and the well-balancedness of the transaction implies knowledge of the discrete logarithm e of $\mathcal{E}$. Directly revealing e would leak too much information, an adversary knowing the openings for input coins and all but one output coin, could easily calculate the unknown opening given e. Therefore instead knowledge of the discrete logarithm to $\mathcal{E}$ is proven by providing a valid signature for $\mathcal{E}$ as public key. Finally we would like to add that coinbase transactions (transactions creating new money as part of mining reward) additionally include the newly minted money as supply s in the excess equation as follows:

$$\mathcal{E} := g^{(\sum n'_i - \sum n_i)} - h^s$$

For non coinbase transactions, s will simply be set to 0. Finally, a Mimblewimble transaction is of form:

$$tx := (s,\ \mathcal{C}_{inp},\ \mathcal{C}_{out},\ K) \text{ with } K := (\{\pi\},\ \{\mathcal{E}\},\ \{\sigma\})$$

where s is the transaction supply amount, $\mathcal{C}_{inp}$ is the list of input coins, $\mathcal{C}_{out}$ is the list of output coins and K is the transaction Kernel. The Kernel consists of $\{\pi\}$ which is a set of all output coin range proofs, $\{\mathcal{E}\}$ a set of excess values and finally $\{\sigma\}$ a set of signatures [7]. Even though normally a transaction would only require a single excess value and signature, for reasons we will see in the next section these fields always have to be lists instead of just a single value.

Transaction Merging

An intriging property of the Mimblewimble protocol is that two transactions can easily be merged into a single one, which is essentially a non-interactive version of the CoinJoin protocol on Bitcoin [15]. Assume we have the following two transactions:

$$\begin{aligned} tx_0 &:= (s_0,\ \mathcal{C}^0_{inp},\ \mathcal{C}^0_{out},\ (\{\pi_0\},\ \{\mathcal{E}_0\},\ \{\sigma_0\})) \\ tx_1 &:= (s_1,\ \mathcal{C}^1_{inp},\ \mathcal{C}^1_{out},\ (\{\pi_1\},\ \{\mathcal{E}_1\},\ \{\sigma_1\})) \end{aligned}$$

Then we can build a single merged transaction:

$$tx_m := (s_0 + s_1,\ \mathcal{C}^0_{inp} \,||\, \mathcal{C}^1_{inp},\ \mathcal{C}^0_{out} \,||\, \mathcal{C}^1_{out},\ (\{\pi_0\} \,||\, \{\pi_1\},\ \{\mathcal{E}_0\} \,||\, \{\mathcal{E}_1\},\ \{\sigma_0\} \,||\, \{\sigma_1\})$$

We can easily deduce that if tx_0 and tx_1 are valid, it must follow that tx_m is valid:
If tx_0 and tx_1 are valid as of definition 3.16 that means $\mathcal{C}^0_{inp} - \mathcal{C}^0_{out} - h^{s_0} = \mathcal{E}_0$, $\{\pi_0\}$ contains valid range proofs for the outputs $\mathcal{C}^0_{out}$ and $\{\sigma_0\}$ contains a valid signature to $\mathcal{E}_0 - h^{s_0}$ as public key, the same must hold for tx_1.

By the rules of arithmetic it then must also hold that

$$\mathcal{C}^0_{inp} \,||\, \mathcal{C}^1_{inp} - \mathcal{C}^0_{out} \,||\, \mathcal{C}^1_{out} - h^{s_0 + s_1} = \mathcal{E}_0 \cdot \mathcal{E}_1$$

$\{\pi_0\} \,||\, \{\pi_1\}$ must contain valid range proofs for the output coins and $\{\sigma_0\} \,||\, \{\sigma_1\}$ must contain valid signatures to the respective Excess points, which makes tx_m a valid transaction.

Subset Problem A subtle problem arises with the way transactions are merged in Mimblewimble. From the construction shown earlier, it is possible to reconstruct the original separate transactions from a merged one, which can be a privacy issue. Given a set of inputs, outputs, and kernels, a subset of these will recombine to reconstruct one of the valid transaction which were aggregated since kernel excess values are not combined. Recall the merged transaction from earlier:

$$tx_m := (s_0 + s_1,\ \mathcal{C}^0_{inp} \,||\, \mathcal{C}^1_{inp},\ \mathcal{C}^0_{out} \,||\, \mathcal{C}^1_{out},\ (\{\pi_0\} \,||\, \{\pi_1\}),\ \{\mathcal{E}_0\} \,||\, \{\mathcal{E}_1\},\ \{\sigma_0\} \,||\, \{\sigma_1\})$$

Since the attacker has access to both $\mathcal{E}_0$ and $\mathcal{E}_1$ as well as σ_0 and σ_1, he can simply try different combinations of input values $\{\mathcal{C}_{inp}\}^*$ and output values $\{\mathcal{C}_{out}\}^*$ until he finds a combination under which the transaction is valid with $\mathcal{E}_0, \sigma_0$ or $\mathcal{E}_1, \sigma_1$. Thereby the attacker was able to reconstruct one of the original transactions from which tx_m was constructed. Following this method he might be able to uncover all original transactions from the merged one.

This problem has been mitigated in cryptocurrencies implementing the protocol by including an additional variable o in the Kernel, called offset value. Briefly recall the construction of the excess value $\mathcal{E}$:

$$\mathcal{E} := g^e$$

In order to solve the problem we redefine $\mathcal{E}$ as:

$$\mathcal{E} \;:=\; g^{e\;-\;o}$$

Since o is now also included in the transaction kernel and therefore known to the verifier, the public verification is still possible. Now every time two transactions are merged with the method layed out previously, the two individual offset values o_0, o_1 are combined into a single value o_m. If offsets are picked truly randomly, and the possible range of values is broad enough, the probability of recovering the uncombined offsets from a merged one becomes negligible, making it infeasible to recover original transactions from a merged one [19].

Cut Through From the way transactions are merged together, we can now learn how to purge spent outputs securely. Let's assume $\mathcal{C}_i$ appears as an output in tx_0 and as an input in tx_1:

$$\begin{aligned} tx_0 \;&:=\; (s_0,\ \mathcal{C}^0_{inp},\ \mathcal{C}^i_{out},\ (\{\pi_0\},\ \{\mathcal{E}_0\},\ \{\sigma_0\})) \\ tx_1 \;&:=\; (s_1,\ \mathcal{C}^i_{inp},\ \mathcal{C}^1_{out},\ (\{\pi_1\},\ \{\mathcal{E}_1\},\ \{\sigma_1\})) \end{aligned}$$

Essentially this means tx_1 spends a coin created in tx_0. Now lets recall the equation given for transaction well-balancedness in 3.15:

$$\sum \mathcal{C}_{out} \;-\; \sum \mathcal{C}_{inp} \;=\; \sum (g^{n'_i}) \;-\; \sum (g^{n_i})$$

If we merge tx_0 with tx_1 as done previously the coin $\mathcal{C}_i$ will appear both in $\sum \mathcal{C}_{inp}$ and $\sum \mathcal{C}_{out}$. Therefore we can erase $\mathcal{C}_i$ from both lists, while maintaining transaction balancedness. Informally this means that every time a coin gets spend, it can be erased from the ledger, without breaking the rules of the system. This property is employed in the Mimblewimble protocol to reduce the space requirements of the protocol as well as provide a notion of unlinkability, as transaction histories can be erased.

Transaction Building

As already pointed out, building transactions in Mimblewimble is an interactive process between the sender and receiver of funds. Jedusor, Tom Elvis originally envisioned the following two-step process to build a transaction: [10]

Throughout the thesis whenever we are concerned with Mimblewimble transactions we generally refer to the sending party (owning the input coins) as Alice and the receiving party (owning the newly created output coin) as Bob. Assume for the following that Alice wants to transfer coins of value p to Bob:

1. Alice first selects an input coin $\mathcal{C}_{inp}$ (or potentially multiple) in her control with total stored value v with $v \geq p$. She then creates change coin outputs $\mathcal{C}^A_{out}$ (could

Alice		**Bob**
Select $\mathcal{C}_{inp}$ of value $v \geq p$		
Create $\mathcal{C}^A_{out}$ of value $v \ - \ p$		
$\mathcal{E}_A \ := \ \sum \mathcal{C}^A_{out} \ - \ \sum \mathcal{C}_{inp}$		
$(-p, \ x)$ opening to $\mathcal{E}_A$		
Create range-proof π for $\mathcal{C}^A_{out}$		
	$\xrightarrow{(-p, \ x), \ \pi}$	
		Create $\mathcal{C}^B_{out}$ with value p and keys (x^*_i)
		$x_{shared} \ := \ x \ + \ \sum x^*_i$
		Create σ with x_{shared}
		Create range-proof π for $\mathcal{C}^B_{out}$
		Finalize transaction tx

Figure 3.2: Original transaction building process

again be multiple) with the remainder of her input value substracted by the value send to Bob. For her newly created output coins and her input coins she calculates her part of discrete logarithm x (her part of the key) to the final $\mathcal{E}$ and sends all this information to Bob as a pre-transaction.

2. Bob creates himself additional output coins $\mathcal{C}^B_{out}$ of total value p and similar to Alice creates his share x^* of the discrete logarithm of $\mathcal{E}$. Together with the share received by Alice he can now create a signature to $\mathcal{E}$ and finalize the transaction

Figure 3.2 depicts the original transaction flow.

This protocol however turned out to be insecure as it is vulnerable to the following attack: The receiver could spend Alice's change coins $\mathcal{C}^A_{out}$ by reverting the transaction. Doing this would give the sender his coins back, however as the sender might not have the keys for his spent outputs anymore, the coins could then be lost.

In detail this reverting transaction would look like:

$$tx_{rv} \ := \ (0, \ \mathcal{C}^A_{out} \ || \ \mathcal{C}^B_{out}, \ \mathcal{C}_{inp}, \ (\pi_{rv}, \ \mathcal{E}_{rv}, \ \sigma_{rv}))$$

So in essence it is exactly the reverse of the previous transaction. Again remembering the construction of the excess value of this construction would look like this:

$$\mathcal{E}_{rv} \ := \ \sum \mathcal{C}^A_{out} \ || \ \mathcal{C}^B_{out} \ - \ \mathcal{C}_{inp}$$

The key x originally sent by Alice to Bob is a valid opening to $\sum \mathcal{C}_{inp} \ - \ \sum \mathcal{C}^A_{out}$. With the inverse of this key x_{inv} we get the opening to $\sum \mathcal{C}^A_{out} \ - \ \mathcal{C}_{inp}$. Now all Bob has to do is add his key x^* to get:

$$x_{rv} := -x + x^*$$

which is the opening to $\mathcal{E}_{rv}$. Therefore Bob is able to construct a valid signature under $\mathcal{E}_{rv}$. Range proofs can just be reused, because this transaction spends to a coin which has already existed on the ledger with the same blinding factor and value, meaning the proof will still be valid.

In essence this means Bob spends the newly created outputs and sends them back to the original input coins, chosen by Alice. It might at first seem unclear why Bob would do that. An example situation could be if Alice pays Bob for some good which Bob is selling. Alice decides to pay in advance, but then Bob discovers that he is already out of stock of the good that Alice ordered. To return the funds to Alice, he reverses the transaction instead of participating in another interactive process to build a new transaction with new outputs. If Alice already deleted the keys to her initial coins, the funds are now lost. The problem was solved in the Grin and Beam Mimblewimble implementations by making the signing process itself a two-party process which will be explained in more detail in chapter 4.

Alternatively Fuchsbauer et al. [7] proposed another way to build transactions which would not be vulnerable to this problem:

1. Alice constructs a full-fledged transaction tx_A spending her input coins $\mathcal{C}_{inp}$ and creates her change coins $\mathcal{C}^A_{out}$, plus a special output coin $\mathcal{C}^{sp}_{out} := h^p \cdot g^{x_{sp}}$, where p is the desired value which should be transferred to Bob and x_{sp} is a randomly choosen key. She proceeds by sending tx_A as well as (p, x_{sp}) and the necessary range proofs to Bob.

2. Bob now creates a second transaction tx_B spending the special coin $\mathcal{C}^{sp}_{out}$ to create an output only he controls $\mathcal{C}^B_{out}$ and merges tx_A with tx_B into tx_m. He then broadcasts tx_m to the network. Note that when the two transactions are merged the intermediate special coin $\mathcal{C}^{sp}_{out}$ will be both in the coin output and input list of the transaction and therfore will be discarded.

One drawback of this approach is that we have two transaction kernels instead of just one because of the merging step, making the transaction slightly bigger, however there is still only one interaction required between Alice and Bob. In the solution employed by the Grin and Beam implementations which we will discuss in chapter 5, at least one additional round of interaction will be required. A figure showing the protocol flow is depicted in Figure 3.3.

Mimblewimble Ledger

In Mimblewimble the ledger itself is a transaction of the form defined in section 3.4 with a set of input and outputs which initially start out empty [7]. The list of outputs as

Alice		**Bob**
Select $\mathcal{C}_{inp}$ of value $v \geq p$		
Create $\mathcal{C}_{out}^{A}$ of value $v - p$		
$x_{sp} \leftarrow\$ \mathbb{Z}_p^*$		
Create $\mathcal{C}_{out}^{sp} := h^p + g^{x_{sp}}$		
Construct and sign tx_A with $\mathcal{C}_{inp}, \mathcal{C}_{out}^{A}, \mathcal{C}_{out}^{sp}$		
	$\xrightarrow{tx_A,\ p,\ x_{sp}}$	
		Create $\mathcal{C}_{out}^{B}$ with value p
		Create tx_B spending $\mathcal{C}_{out}^{sp}$ to $\mathcal{C}_{out}^{B}$
		Merge tx_A and tx_B to tx_m
		Publish tx_m

Figure 3.3: Salvaged transaction protocol by Fuchsbauer et al. [7]

given in the ledger is the list of spendable coins, similar to the list of UTXOs (unspent transaction outputs) in Bitcoin. Upon publishing a new transaction to the ledger it will be merged with the ledger itself as seen in section 3.4, after which a cut through as seen in section 3.4 is executed. By running the cut through all coins that now appear in both the output and input list are discarded. It is easy to see that the input list of the ledger must therefore always be empty as whenever an output coin is spent it will be discarded immediately after. We can further see that with this setup the ledger only every grows in size of the unspent output list, which is very helpful given that each output coin must also attach a range proof that usually has high space requirements. In Grin and Beam updates to the ledger are made in the form of blocks requiring proof of work which is the same as it is in Bitcoin [2]. A miner that found a new block by having solved the proof of work is allowed to include one coinbase transaction creating a fixed amount of new supply which he can send to himself as a reward.

3.5 Scriptless Scripts

3.6 Adaptor Signatures

CHAPTER 4

Two Party Fixed Witness Adaptor Signatures

In this chapter, we will define a variant of the adaptor signature scheme as explained in section 3.6. This new variant is tailored specifically to meet the requirements of being applicable in the scenario of two-party signature protocols, constructable for aggregateable signature schemes such as Schnorr [16]. In a two-party signature protocol each party holds only a share of a private key (to a composite public key) for which they want to cooperatively create a signature. The advantage of our adapted signature scheme in comparison to the original definition is that in the two-party scenario mentioned above we do not need to introduce an additional pre-signature step. Instead one of the partial signatures created and exchanged by the two parties will serve as what is defined as the pre-signature by Aumayr et al., allowing for a simpler protocol. In particular our protocol will allow one of the two parties, to hide a witness value x of $(x, X) \in R$ (where R is a hard relation as of definition 3.1) inside his partial signature. The second party (knowing X, but not x) can verify that x is indeed contained in the partial signature received by the peer. To complete the final signature the party knowing x has to first replace his original partial signature (adapted with x) with the unadapted version. The second party, having previously received the adapted partial signature is now able to extract x from the final signature. This feature can then be leveraged to build an Atomic Swap protocol as we will show in chapter 5.

The rest of the chapter is organized as follows: First we will define the general two-party Schnorr signature protocol, as it is currently implemented in Mimblewimble-based cryptocurrencies. We will then show that the final signatures output by the protocol fulfill the same properties as regular Schnorr signatures seen in [21] and prove correctness of the protocol. From this two-party protocol, we then derive the adapted variant already mentioned before. We start by defining our extended signature scheme in section 4.1, proceed by providing a Schnorr-based instantiation of the protocol in section 4.2 and finally prove its correctness and security in section 4.4.

4.1 Definitions

A two-party signature scheme is an extension of a signature scheme as defined in definition 3.3, which allows us to distribute signature generation for a composite public key shared between two parties Alice and Bob. Alice and Bob want to collaborate to generate a signature valid under the composite public key $pk := pk_A \cdot pk_B$ without having to reveal their secret keys to each other. The definition below was constructed with the goal in mind of formalizing exactly what is currently implemented and used in Mimblewimble-based cryptocurrencies.

Definition 4.1 (Two Party Signature Scheme)**.** A *two party signature scheme* Φ_{MP} extends a signature scheme Φ with a tuple of protocols and algorithms $(\mathsf{dKeyGen}, \mathsf{signPt}, \mathsf{vrfPt}, \mathsf{finSig})$ defined as follows:

- $((sk_A, pk_A, n_A, \Lambda), (sk_B, pk_B, n_B, \Lambda)) \leftarrow \mathsf{dKeyGen}\langle 1^n, 1^n\rangle$: The distributed key generation protocol takes as input the security parameter from both Alice and Bob and returns the tuple $(sk_A, pk_A, n_A, \Lambda)$ to Alice (similar to Bob) where (sk_A, pk_A) is a pair of private and corresponding public keys, n_A a secret nonce and Λ is the signature context containing parameters shared between Alice and Bob. We introduce Λ for the participants to share as well as update parameters with each other during the protocol execution. Note that this context always has to be consistent between the two parties. That means if Alice were to update Λ, she has to sent the updated version to Bob to continue protocol.

- $(\tilde{\sigma_A}) \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$: The partial signing algorithm is a DPT function that takes as input the message m, the share of the secret key sk_A and nonce n_A (similiar for Bob) as well as the shared signature context Λ. The procedure outputs $(\tilde{\sigma_A})$, that is, a share of the signature to a participant.

- $\{1, 0\} \leftarrow \mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A)$: The share verification algorithm is a DPT function that takes as input a signature share $\tilde{\sigma_A}$, a message m, and the other participants public key pk_A (similiar pk_B for Bobs partial signature). The algorithm returns 1 if the verification was successful or 0 otherwise.

- $\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$: The finalize signature algorithm is a DPT function that takes as input two shares of the signatures and combines them into a final signature valid under the composite public key $pk = pk_A \cdot pk_B$.

We require the two party signature scheme to be correct as well as secure as of definition 3.8. For the security of the distributed key-generation protocol $\mathsf{dKeyGen}$, special care needs to be taken to protect the scheme against rogue-key attacks. In such an attack one of the public keys is computed as a function of the other parties public key, allowing the corrupted signer to produce forged signatures under the honest users public key without knowing its secret key [16].

From the definition 4.1, we now derive an adapted signature scheme Φ_{Apt}, which allows one of the participants to hide a secret witness value inside his partial signature.

Definition 4.2 (Two Party Fixed Witness Adaptor Signature Scheme)**.** Given a pair $(x, X) \in R$ (where R is a hard relation as of definition 3.1) a Two Party Fixed Witness Adaptor Signature Scheme Φ_{Apt} is an extension to Φ_{MP} with the following algorithms.

$$\Phi_{Apt} \ := \ (\Phi_{MP} \ || \ \mathsf{adaptSig} \ || \ \mathsf{vrfAptSig} \ || \ \mathsf{extWit})$$

- $\hat{\sigma_A} \ \leftarrow \ \mathsf{adaptSig}(\tilde{\sigma_A}, x)$: The adapt signature algorithm is a DPT function that takes as input a partial signature $\tilde{\sigma_A}$ and a secret witness value x. The procedure will output an adapted partial signature $\hat{\sigma_A}$ which can be verified to contain x using the vrfAptSig function, without having to reveal x.

- $\{1, 0\} \ \leftarrow \ \mathsf{vrfAptSig}(\hat{\sigma_A}, m, pk_A, X)$: The verification algorithm is a DPT function that takes as input an adapted partial signature $\hat{\sigma_A}$, the other participants public key pk_A and a statement X. The function will verify the partial signature's validity as well that it contains the secret witness x.

- $x \ \leftarrow \ \mathsf{extWit}(\sigma_{fin}, \tilde{\sigma_A}, \hat{\sigma_B})$: The witness extraction algorithm is a DPT function that lets Alice extract the secret witness x after having learned the final composite signature σ_{fin}. As input it expects the partial signatures $\tilde{\sigma_A}$ and $\hat{\sigma_B}$ shared between the participants during protocol execution, as well as the final composite signature σ_{fin}. Consequently, only protocol participants knowing the partial signatures will be able to run this algorithm.

Similar to how it is defined in [3] additionally to regular Correctness, as defined in definition 3.3, we require our signature scheme to satisfy Adaptor Signature Correctness. This property is given when every adapted partial signature generated by adaptSig can be completed into a final signature for all pairs $(x, \ X) \in R$, from which it will then be possible to extract the witness computing extWit with the required parameters.

Definition 4.3 (Adaptor Signature Correctness)**.** More formally *Adaptor Signature Correctness* is given if for every security parameter $n \in \mathbb{N}$, message $m \in \{0, 1\}^*$, keypairs $\langle (sk_A, pk_A, n_A, \Lambda), (sk_B, pk_B, n_B, \Lambda) \rangle \ \leftarrow \ \mathsf{dKeyGen}\langle 1^n, 1^n \rangle$ with their composite public key $\Lambda.pk \ = \ pk_A \ \cdot \ pk_B$ and every statement/witness pair $(X, \ x) \ \leftarrow \ \mathsf{genRel}(1^n)$

$\mathsf{keyGen}(1^n)$	$\mathsf{sign}(m, sk)$	$\mathsf{verf}(m, \sigma, pk)$
1 : $x \leftarrow\!\$\ \mathbb{Z}_p^*$	1 : $n \leftarrow\!\$\ \mathbb{Z}_p^*$	1 : $(s,\ R) \leftarrow \sigma$
2 : **return** $(sk := x,\ pk := g^x)$	2 : $R := g^n$	2 : $e := \mathsf{H}(m \mid\mid R \mid\mid pk)$
	3 : $e := \mathsf{H}(m \mid\mid R \mid\mid pk)$	3 : **return** $g^s = R \cdot pk^e$
	4 : $s := n + e \cdot sk$	
	5 : **return** $\sigma := (s, R)$	

Figure 4.1: Schnorr Signature Scheme as first defined in [21]

it must hold that:

$$\Pr\left[\begin{array}{c} \mathsf{verf}(m, \sigma_{fin}, \Lambda.pk) \ = \ 1 \\ \wedge \\ \mathsf{vrfAptSig}(\hat{\sigma_B}, m, pk_B, X) \ = \ 1 \\ \wedge \\ (x^*, X) \in R \end{array}\ \middle|\ \begin{array}{l} (x, X) \ \leftarrow \ \mathsf{genRel}(1^n) \\ \tilde{\sigma_A} \ \leftarrow \ \mathsf{signPt}(m, sk_A, n_A, \Lambda) \\ \tilde{\sigma_B} \ \leftarrow \ \mathsf{signPt}(m, sk_B, n_B, \Lambda) \\ \hat{\sigma_B} \ \leftarrow \ \mathsf{adaptSig}(\tilde{\sigma_B}, x) \\ \sigma_{fin} \ \leftarrow \ \mathsf{finSig}(\tilde{\sigma_A}, \hat{\sigma_B}) \\ x^* \ \leftarrow \ \mathsf{extWit}(\sigma_{fin}, \tilde{\sigma_A}, \hat{\sigma_B}) \end{array}\right] = 1.$$

4.2 Schnorr-based instantiation

We start by providing a general instantiation of a signature scheme (see definition 3.3): We assume we have a group $\mathbb{G}$ with prime p and generator point g, H is a secure hash function in the random oracle model as defined in definition 3.4 and $m \in \{0,1\}^*$ is a message.

A concrete implementation can be seen in fig. 4.1. The signature scheme is called Schnorr signature scheme, first defined in [21] and valued for its simplicity and extensively analyzed security. Due to being patented its practical use originally was limited, however since the patent expired in 2008 the signature scheme sees increasing use in practical applications. Cryptocurrencies such as Grin and Beam now use Schnorr as its primary signature scheme, also Bitcoin is planning to add Schnorr signatures as an alternative to the currently used ECDSA signatures. [1]

Correctness of the scheme can be derived as following: As shown in fig. 4.1, verf, line 3 we need to show that $g^s = R \cdot pk^e$ returns 1 for correct signatures. As s is calculated as $n + e \cdot sk$ (sign, line 4), when generator g is raised to s, we get $g^{n + e \cdot sk}$ which we can transform into $g^n \cdot g^{sk \cdot e}$, and finally into $R \cdot pk^e$ which is the same as the right side of the equation.

From the regular Schnorr signature scheme we now provide an instantiation for the two-party case defined in definition 4.1. Note that this two-party variant of the scheme

[1] `https://en.bitcoin.it/wiki/BIP_0341`

is what is currently implemented in the Mimblewimble-based Cryptocurrencies and will provide a basis from which we can then build an instantiation for the Two Party Fixed Witness Adaptor Signature Scheme.

First we define a auxiliary function setupCtx to use for the instantion:

$$
\begin{array}{l}
\underline{\mathsf{setupCtx}(\Lambda, pk_A, R_A)} \\
1: \quad \Lambda.pk := \Lambda.pk \cdot pk_A \\
2: \quad \Lambda.R := \Lambda.R \cdot R_A \\
3: \quad \textbf{return } \Lambda
\end{array}
$$

This function helps the participants to setup and update the signature context shared between them. In fig. 4.2 we show a concrete instantiation of the protocol and functions.

In dKeyGen Alice and Bob will each randomly chose their secret key and nonce. They further require to create a zero-knowledge proof attesting to the fact that they have generated their key before any message was exchanged. This is essential to avoid the rogue key attacks mentioned earlier. The idea of achieving this using zero-knowledge proofs of knowledge was introduced by Thomas Ristenpart and Scott Yilek in [20]. Another secure key generation setup for a Schnorr-based multi-signature protocol was found by Micali et al. in [17]. However, the protocol requires additional impractical steps such as splitting signers into individual subgroups $\mathbb{S}_1, \mathbb{S}_2, \cdots$ of a group $\mathbb{G}$. In our instantiation of dKeyGen Alice will initially setup the signature context and send it to Bob, together with her public key and zk-proof of knowledge. Bob verifies the proof and will then proceed by adding his parameters to the shared signature context and send it back to Alice, together with his parameters, which then again Alice will verify.

We note here that this is only one possible way of securely computing the parties keypairs and nonce values, as well as setting up the shared context. Alternative ways of generating these values could be employed, depending on the use case. For instance one might envision a scenario in which Alice and Bob would like to reuse their keypairs multiple times and only regenerate the random nonces before each signing process, in this case we could split up dKeyGen into two separate functions. Another scenario, one which we will encounter throughout this thesis, is that both the keypairs and nonce values have been generated by a protocol similar to dKeyGen beforehards, but the shared context Λ is not yet setup. In this case the setupCtx can be incorporated into the signing protocol as we shall see in section 4.3. Whatever method of key generation is used, it must not be vulnerable to rogue key attacks.

signPt and vrfPt are generally similar to the instantiation of the normal Schnorr signature scheme. Note however that for computing the Schnorr challenge e the input into the hash function will be the already combined public key pk and combined nonce commitment R, which the participants can read from the context object Λ. This has the effect that the signature shares itself are not yet a valid signature (neither under pk nor under pk_A or pk_B) and furhter means that signing can only start after the context Λ has been fully setup. This is because to be valid under pk the signature shares are missing the s values from the other participants. They are also not valid under the partial public keys pk_A or pk_B because the Schnorr challenge is computed already with the combined values. Therefore we have to introduce the slightly adjusted vrfPt to be able to verify specifically the partial signatures.

For a correctness proof and a generally more extensive explanation of this two-party Schnorr signature scheme we refer the reader to a paper by Maxwell et al. [16].

In fig. 4.3 we further provide a Schnorr-based instantiation for the fixed witness adapted signature scheme as defined in definition 4.2.

adaptSig will add the secret witness x to the s value of the signature, changing the partial signature it this way means that it can't be verified using verf any longer. Therefore we introduce vrfAptSig which takes as additional parameter the statement X which will be included in the verifiers equation. Now the function verifies not only validity of the partial signature, but also that it indeed has been adapted with the witness value x, being the discrete logarithm of X. After obtaining σ_{fin}, we can then cleverly unpack the secret x, which is shown in the extWit function.

4.3 Protocols

We now formalize two protocols **dSign** and **dAptSign** which will later be used when constructing Mimblewimble transactions. **dSign** is a two-party protocol creating a signature under a composite public key $pk = pk_A \cdot pk_B$ using the algorithms outlined in fig. 4.2. **dAptSign** additionally, uses the functionality of fig. 4.3 allowing one party to adapt his partial signature with a secret witness value x, which is then revealed to the other party by the final signature.

Note that for these protocols we assume that the secret keys as well as nonce values used in the signatures have already been generated beforehand, for example by running a secure setup protocol similar to dKeyGen. However, in this case we furthermore assume that the signature context Λ has not yet been setup between the parties, the reason for this is that we are faced with exactly this scenario in the Mimblewimble transaction protocols, which we shall see later in section 5.2. Both parties input the shared message m as well as their secret keys and secret nonces. The instantiation of the protocol can be seen in fig. 4.4. The protocol outputs a signature σ_{fin} to the message m, valid under the composite public key $pk = pk_A \cdot pk_B$. Additionally, to the final signature the protocol also outputs the composite public key pk.

The final signature is a valid signature to the message m under the composite public key $pk := pk_A \cdot pk_B$. A verifier knowing the signed message m, the final signature σ_{fin} and the composite public key pk can now verify the signature using the regular verf procedure as shown in fig. 4.1.

$\mathsf{dKeyGen}\langle 1^n, 1^n \rangle$

Alice		**Bob**
$sk_A \leftarrow\$ \mathbb{Z}_p^*$		$sk_B \leftarrow\$ \mathbb{Z}_p^*$
$n_A \leftarrow\$ \mathbb{Z}_p^*$		$n_B \leftarrow\$ \mathbb{Z}_p^*$
$pk_A := g^{sk_A}$		$pk_B := g^{sk_B}$
$R_A := g^{n_A}$		$R_B := g^{n_B}$
$st_A := \exists sk_A \ s.t. \ g^{sk_A} = pk_A$		$st_B := \exists sk_B \ s.t. \ g^{sk_B} = pk_B$
$\pi_A \leftarrow \mathsf{P}_{NIZK}(sk_A, st_A)$		$\pi_B \leftarrow \mathsf{P}_{NIZK}(sk_B, st_B)$
$\Lambda := \langle pk := 1_p, \ R := 1_p \rangle$		
$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, pk_A, R_A)$		
	$\xrightarrow{\Lambda, pk_A, \pi_A}$	
		if $\mathsf{V}_{NIZK}(\pi_A) = 0$
		return $\perp$
		$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, pk_B, R_B)$
	$\xleftarrow{\Lambda, pk_B, \pi_B}$	
if $\mathsf{V}_{NIZK}(\pi_B) = 0$		
return $\perp$		
return $(sk_A, pk_A, n_A, \Lambda)$		**return** $(sk_B, pk_B, n_B, \Lambda)$

$\mathsf{signPt}(m, sk_A, n_A, \Lambda)$

$(R, \ pk) \leftarrow \Lambda$
$R_A := g^{n_A}$
$e := \mathsf{H}(m \ || \ R \ || \ pk)$
$s_A := n_A + sk_A \cdot e$
return $\tilde{\sigma_A} := (s_A, R_A, \Lambda)$

$\mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A)$

$(s_A, \ R_A, \ \Lambda) \leftarrow \tilde{\sigma_A}$
$(pk, \ R) \leftarrow \Lambda$
$e := \mathsf{H}(m \ || \ R \ || \ pk)$
return $g^{s_A} = R_A \cdot pk_A^e$

$\mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$

$(s_A, \ R_A, \ \Lambda) \leftarrow \tilde{\sigma_A}$
$(s_B, \ R_B, \ \Lambda) \leftarrow \tilde{\sigma_B}$
$(pk, \ R) \leftarrow \Lambda$
$s := s_A + s_B$
$\sigma_{fin} := (s, R)$
return σ_{fin}

Figure 4.2: Two Party Schnorr Signature Scheme

$\mathsf{adaptSig}(\tilde{\sigma}, x)$

$(s,\ R_A,\ \Lambda) \leftarrow \tilde{\sigma}$
$s^* := s + x$
return $\hat{\sigma} := (s^*,\ R_A,\ \Lambda)$

$\mathsf{vrfAptSig}(\hat{\sigma_A}, m, pk_A, X)$

$(s_A,\ R_A,\ \Lambda) \leftarrow \hat{\sigma_A}$
$(pk,\ R) \leftarrow \Lambda$
$e := \mathsf{H}(m \mid\mid R \mid\mid pk)$
return $g^{s_A} = R_A \cdot pk_A^e \cdot X$

$\mathsf{extWit}(\sigma_{fin}, \tilde{\sigma_A}, \hat{\sigma_B})$

$(s,\ R) \leftarrow \sigma_{fin}$
$(s_A,\ R_A,\ \Lambda) \leftarrow \tilde{\sigma_A}$
$(\hat{s_B},\ R_B,\ \Lambda) \leftarrow \hat{\sigma_B}$
$s_B := s - s_A$
$x := \hat{s_B} - s_B$
return (x)

Figure 4.3: Fixed Witness Adaptor Schnorr Signature Scheme

We now define the **dAptSign** protocol between Alice and Bob again creating a signature σ_{fin} under the composite public key $pk := pk_A \cdot pk_B$. Now Bob will hide his secret x which Alice can extract after the signing process has completed. The concrete instantiation can be seen in fig. 4.5. One thing to note is that in this protocol only Bob is able to call the signature finalization algorithm finSig for computing the final signature, which is different from the previous protocol, in which both had the ability to do so. The reason for this is that the function requires Bob's unadapted partial signature $\tilde{\sigma_B}$ as input, which Alice does not know (She only knows Bobs adapted partial signature). Therefore, one further interaction is needed to send the final signature to Alice. The protocol outputs $(x, (\sigma_{fin}, pk))$ for Alice as she manages to learn x and (σ_{fin}, pk) for Bob.

4.4 Correctness & Security

We now prove that the outlined Schnorr-based instantiation is correct, i.e. Adaptor Signature Correctness holds, and is secure with regards to the definition 3.8.

4.4.1 Adaptor Signature Correctness

To prove that Adaptor Signature Correctness holds we have three statements to prove as given by definition 4.3, first we prove that $\mathsf{verf}(m, \sigma_{fin}, \Lambda.pk) = 1$ holds in our Schnorr-based instantiation of the signature scheme, where Λ is setup such that $pk = pk_A \cdot pk_B$.

Proof. For this proof we assume the setup already specified in definition 4.3. The proof is by showing equality of the equation checked by the verifier of the final signature by

dSign$\langle(m, sk_A, n_A), (m, sk_B, n_B)\rangle$

Alice		*Bob*
$\Lambda := \{pk := 1_p, R := 1_p\}$		
$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
	$\xrightarrow{\Lambda, pk_A := g^{sk_A}}$	
		$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
		$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
	$\xleftarrow{\tilde{\sigma_B}, \Lambda, pk_B := g^{sk_B}}$	
if $\mathsf{vrfPt}(\tilde{\sigma_B}, m, pk_B) = 0$		
return $\perp$		
$\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$		
	$\xrightarrow{\tilde{\sigma_A}}$	
		if $\mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A) = 0$
		return $\perp$
$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$		$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$
$pk \leftarrow \Lambda.pk$		$pk \leftarrow \Lambda.pk$
return (σ_{fin}, pk)		**return** (σ_{fin}, pk)

Figure 4.4: Instantiation of the **dSign** protocol.

continuous substitutions in the left side of equation:

$$
\begin{aligned}
g^s &= R \cdot pk^e && (4.1)\\
g^{s_A} \cdot g^{s_B} & && (4.2)\\
g^{n_A + e \cdot sk_A} \cdot g^{n_B + e \cdot sk_B} & && (4.3)\\
g^{n_A} \cdot pk_A^e \cdot g^{n_B} \cdot pk_B^e & && (4.4)\\
R_A \cdot pk_A^e \cdot R_B \cdot pk_B^e & && (4.5)\\
R \cdot pk^e &= R \cdot pk^e && (4.6)\\
1 &= 1 && (4.7)
\end{aligned}
$$

It remains to prove that with the same setup $\mathsf{vrfAptSig}(\hat{\sigma_B}, m, pk_B, X) = 1$ and $(X,\ x) \in R$ for $x \leftarrow \mathsf{extWit}(\sigma_{fin}, \tilde{\sigma_A}, \hat{\sigma_B})$:

$$\mathsf{vrfAptSig}(\hat{\sigma_B}, m, pk_B, X) = 1$$

dAptSign$\langle (m, sk_A, n_A), (m, sk_B, n_B, x) \rangle$

Alice		*Bob*
$\Lambda := \{pk := 1_p, R := 1_p\}$		
$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
	$\xrightarrow{\Lambda, pk_A := g^{sk_A}}$	
		$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
		$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
		$\hat{\sigma_B} \leftarrow \mathsf{adaptSig}(\tilde{\sigma_B}, x)$
		$pk_B := g^{sk_B}$
		$X := g^x$
	$\xleftarrow{\hat{\sigma_B}, \Lambda, pk_B, X}$	
if $\mathsf{vrfAptSig}(\tilde{\sigma_B}, m, pk_B, X) = 0$		
return $\perp$		
$\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$		
	$\xrightarrow{\tilde{\sigma_A}}$	
		if $\mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A) = 0$
		return $\perp$
		$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$
	$\xleftarrow{\sigma_{fin}}$	
$pk \leftarrow \Lambda.pk$		$pk \leftarrow \Lambda.pk$
if $\mathsf{verf}(m, \sigma_{fin}, pk) = 0$		
return $\perp$		
$x \leftarrow \mathsf{extWit}(\sigma_{fin}, \tilde{\sigma_A}, \hat{\sigma_B})$		
return $(x, (\sigma_{fin}, pk))$		**return** (σ_{fin}, pk)

Figure 4.5: Instantiation of the **dAptSign** protocol.

The proof is by continuous substitutions in the equation checked by the verifier:

$$g^{\tilde{\sigma_B}} \;=\; R_B \;\cdot\; pk_B^e \;\cdot\; X \tag{4.8}$$
$$g^{\tilde{\sigma_B} \;+\; x} \tag{4.9}$$
$$g^{n_B \;+\; sk_B \;\cdot\; e \;+\; x} \tag{4.10}$$
$$g^{n_B} \;\cdot\; g^{sk_B \;\cdot\; e} \;+\; g^x \tag{4.11}$$
$$R_B \;\cdot\; pk_B^e \;\cdot\; X \;=\; R_B \;\cdot\; pk_B^e \;\cdot\; X \tag{4.12}$$
$$1 \;=\; 1 \tag{4.13}$$

We now continue to prove the last equation required:

$$(X,\; x) \in R$$

We do this by showing that x is calculated correctly in extWit: $\hat{s_B}$ is the s value in Bob's adapted partial signature

$$x \;= \hat{s_B} \;-\; (s \;-\; s_A) \tag{4.14}$$
$$\hat{s_B} \;-\; ((s_A \;+\; s_B) \;-\; s_A) \tag{4.15}$$
$$s_B \;+\; x \;-\; (s_B) \tag{4.16}$$
$$x \;= x \tag{4.17}$$
$$1 \;= 1 \tag{4.18}$$

□

4.4.2 Security

We have shown that the outlined signature scheme is correct, next we have to prove its security. Our goal is to proof security in the malicious setting (as defined in definition 3.8) that means the adversary might or might not behave as specified by the protocol. For achieving this we will prove security for both the **dSign** and **dAptSign** protocols in the hybrid model which was layed out by Yehuda Lindell in [13]. In particular, we will use the f_{zk}^R-model in which we assume that we have access to a constant-round protocol f_{zk}^R that computes the zero-knowledge proof of knowledge functionality for any NP relation R. The function is parameterized with a relation R between a witness value x (or potentially multiple) and a statement X. One party provides the witness statment pair (x, X), the second the statement X^*. If $X \;=\; X^*$ and $(x, X) \in R$ the functionality returns 1, otherwise 0. More formally:

$$\mathsf{f}_{zk}^R(((x, X), X^*)) \;=\; \begin{cases} (\lambda, R(X, x)) & \text{if } X \;=\; X^* \\ (\lambda, 0) & \text{otherwise} \end{cases}$$

That a constant-round zero-knowledge proof of knowledge exists was proven in [12]. A secure zero-knowledge proof must fulfill Completeness, Soundness and Zero-Knowledge properties which are defined for instance in [9].

Hybrid functionalities: The parties have access to a trusted third party that computes the zero-knowledge proof of knowledge functionality f_{zk}^{R}. R is the relation between a secret key sk and its public key $pk = g^{sk}$, for the elliptic curve generator point g. The participants have to call the functionality in the same order. That means if the prover first sends the pair (x_1, X_1) and then (x_2, X_2) the verifier needs to first send X_1 and then X_2.

Proof idea: In order to construct our simulation proof in the hybrid-model we make some adjustments to the **dSign** protocol utilizing the capabilities of the f_{zk}^{R} functionality. The adjusted protocol can be seen in figure fig. 4.6 with the newly added lines marked in blue. That means both Alice and Bob will verify the validity of the public key and nonce commitments of the other party and will stop protocol execution in case an invalid value has been sent. We assume parties have access to a trusted third party computing f_{zk}^{R} which will return 1 if $pk_A = pk_A^*$ (where pk_A^* is the public key that Bob received from Alice) and $pk_A = g^{sk_A}$. (The same holds for the reversed case)

Theorem 1. Assume we have two key pairs (sk_A, pk_A) and (sk_B, pk_B) which were setup securely as for instance with the distributed keygen protocol $\mathsf{dKeyGen}$ and a hash function $\mathsf{H}(\cdot)$ modeled in the random oracle model. Then **dSign** securely computes a signature σ_{fin} under the composite public key $pk := pk_A \cdot pk_B$ in the f_{zk}^{R}-model.

Proof. We proof security of the protocol by constructing a simulator $\mathcal{S}$ who is given output (σ_{fin}, pk) from a TTP (trusted third party) that securely computes the protocol in the ideal world upon receiving the inputs from Alice and Bob. The task of the simulator will be to extract the inputs used by $\mathcal{A}$ such that he is able to call the TTP and receive the outputs. From this output the simulator $\mathcal{S}$ will have to construct a transcript which is indistinguishable from the protocol transcript in the real world in which the corrupted party is controlled by a deterministic polynomial adversary $\mathcal{A}$. The simulator uses the calls to f_{zk}^{R} in order to do this. Furthermore, we assume that the message m is known to both Alice and Bob. All other inputs (including public keys) are only known to the respective party at the start of the protocol. We have to prove two cases, one in which Alice is the corrupted party and one in which Bob is the corrupted party.

Alice is corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ receives and saves (sk_A, pk_A), as well as (n_A, R_A) that $\mathcal{A}$ sends to f_{zk}^{R}.

2. Next $\mathcal{S}$ receives the message (Λ, pk_A^*, R_A^*) as sent to Bob by $\mathcal{A}$. If $pk_A^* \neq pk_A$ or $R_A^* \neq R_A$ $\mathcal{S}$ externally sends abort to the TTP computing **dSign** and outputs whatever $\mathcal{A}$ outputs, otherwise he will send the inputs (m, sk_A, n_A) and receive back (σ_{fin}, pk).

$\mathbf{dSign}\langle(m, sk_A, n_A), (m, sk_B, n_B)\rangle$

Alice		*Bob*
$\Lambda := \{pk := 1_p, R := 1_p\}$		
$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
$\mathsf{f}^{R}_{zk}((sk_A, pk_A))$		
$\mathsf{f}^{R}_{zk}((n_A, R_A))$		
	$\xrightarrow{\Lambda, pk_A, R_A}$	
		$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
		$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
		if $\mathsf{f}^{R}_{zk}(pk_A) = 0 \lor$
		$\quad \mathsf{f}^{R}_{zk}(\Lambda.R) = 0$
		$\quad$ **return** $\perp$
		$\mathsf{f}^{R}_{zk}((sk_B, pk_B))$
		$\mathsf{f}^{R}_{zk}((n_B, R_B))$
	$\xleftarrow{\tilde{\sigma_B}, \Lambda, pk_B}$	
if $\mathsf{f}^{R}_{zk}(pk_B) = 0 \lor$		
$\quad \mathsf{f}^{R}_{zk}(\Lambda.R \cdot R_A^{-1}) = 0$		
$\quad$ **return** $\perp$		
if $\mathsf{vrfPt}(\tilde{\sigma_B}, m, pk_B) = 0$		
$\quad$ **return** $\perp$		
$\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$		
	$\xrightarrow{\tilde{\sigma_A}}$	
		if $\mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A) = 0$
		$\quad$ **return** $\perp$
$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$		$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$
$pk \leftarrow \Lambda.pk$		$pk \leftarrow \Lambda.pk$
return (σ_{fin}, pk)		**return** (σ_{fin}, pk)

Figure 4.6: Adjustment to the **dSign** protocol seen in fig. 4.4

3. $\mathcal{S}$ now calculates pk_B, R_B and $\tilde{\sigma_B}$ as follows:

$$\begin{aligned} (s, R) &\leftarrow \sigma_{fin} \\ pk_B &:= pk \cdot pk_A^{-1} \\ R_B &:= R \cdot R_A^{-1} \\ \Lambda &\leftarrow \mathsf{setupCtx}(\Lambda, pk_B, R_B) \\ \tilde{\sigma_A} &\leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda) \\ (s_A, R_A, \Lambda) &\leftarrow \tilde{\sigma_A} \\ s_B &:= s - s_A \\ \tilde{\sigma_B} &:= (s_B, R_B, \Lambda) \end{aligned}$$

4. After having done the calculations $\mathcal{S}$ is able to send $\Lambda, \tilde{\sigma_B}, pk_B$ to $\mathcal{A}$ as if coming from Bob.

5. When $\mathcal{A}$ calls f_{zk}^R and f_{zk}^R (as the verifier) $\mathcal{S}$ checks equality with pk_B (respective R_B) and thereafter sends back either 0 or 1.

6. Eventually $\mathcal{S}$ will receive $\tilde{\sigma_A}^*$ from $\mathcal{A}$ and finally output whatever $\mathcal{A}$ outputs.

We now show that the joint output distribution in the ideal model with $\mathcal{S}$ is identically distributed to the joint distribution in a real execution in the f_{zk}^R-hybrid model with $\mathcal{A}$. We consider three phases : **(1)** Alice sends (sk_A, pk_A) as well as (n_A, R_A) to f_{zk}^R and (Λ, pk_A, R_A) to Bob. **(2)** Bob sends pk_A and $\Lambda.R$ to f_{zk}^R as the verifier, and (sk_B, pk_B), (n_B, R_B) to f_{zk}^R as the prover. Afterward he sends $(\tilde{\sigma_B}, \Lambda, pk_B)$ to Alice. **(3)** Alice sends pk_B and R_B to f_{zk}^R as the verifier and $\tilde{\sigma_A}$ to Bob. Finally, we will have to show that the simulators output is indistinguishable from that of $\mathcal{A}$.

- *Phase 1* Since $\mathcal{A}$ is required to be deterministic, the distribution in this phase is identical to what is expected in a real execution.
- *Phase 2* As $\mathcal{S}$ managed to calculate Bobs $\tilde{\sigma_B}, pk_B, R_B$, as if they would be expected in a real execution, from the final (σ_{fin}, pk), we can conclude that the transcript of this phase must be computationally indistinguishable from a real transcript.
- *Phase 3* The messages sent by the deterministic $\mathcal{A}$ again have to be identically distributed to a real execution, therefore the transcript produced by this phase again has to be indistinguishable.
- Regarding the protocol output we note that if the adversary deviates from the protocol specification at any time the simulator will note this, halt and output whatever $\mathcal{A}$ outputs. In the case that $\mathcal{A}$ behaves correctly $\mathcal{S}$ will play the protocol until the end and finally again output whatever $\mathcal{A}$ outputs. So both in the case that $\mathcal{A}$ acts honestly and in the case that he does not the outputs of $\mathcal{A}$ and $\mathcal{S}$ will be indistinguishable.

We have shown that the distributions of transcript messages are indistinguishable in every phase of the protocol in the case that Alice is corrupted.

Bob is corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ starts by sampling $sk_A, n_A \leftarrow\$ \mathbb{Z}_*^*$ and proceeds by setting up the initial signature context as defined by the protocol:

$$\Lambda := \{pk := 1, R := 1\}$$
$$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$$

2. $\mathcal{S}$ now invokes $\mathcal{A}$ and sends (Λ, pk_A, R_A) as if coming from Alice.
3. When $\mathcal{A}$ calls f_{zk}^R (as verifier) $\mathcal{S}$ checks equality to the parameters he sent in step 1 and returns either 1 or 0. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^R((sk_B, pk_B))$ and $\mathsf{f}_{zk}^R((n_B, R_B))$ the simulator saves those values to its memory.
4. Now $\mathcal{S}$ externally sends the inputs (m, sk_B, n_B) to the TTP and receives back (σ_{fin}, pk)
5. When $\mathcal{A}$ queries $\mathsf{H}(m \mid\mid R_A \cdot R_B \mid\mid pk_A \cdot pk_B)$ during the signPt call, $\mathcal{S}$ sends back e^* such that:

$$\sigma_{fin} = n_A + sk_A \cdot e^* + n_B + sk_B \cdot e^*$$
$$e^* = \frac{\sigma_{fin} - n_A - n_B}{sk_A + sk_B}$$

6. $\mathcal{S}$ receives $(\tilde{\sigma_B}, \Lambda, pk_B)$ from $\mathcal{A}$. He verifies the values sent to him by comparing them with pk_B and R_B from its memory. If the simulator finds the values to be invalid, or if he doesn't receive any values at all, he will send abort to the TTP and output whatever $\mathcal{A}$ outputs.
7. $\mathcal{S}$ calculates as defined in the protocol as $\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$ and then sends it to $\mathcal{A}$ as if coming from Alice and finally outputs whatever $\mathcal{A}$ outputs.

Again we argue why the transcript is indistinguishable from the real one for each of the three phases layed out before:

- *Phase 1:* The values (pk_A, R_A) sent by $\mathcal{S}$ to $\mathcal{A}$ only depend on Alice's input parameters (and to some extend on the public elliptic curve parameters). As $\mathcal{A}$ does not know pk_A or R_A yet, he has no way of determining for two public keys pk_A, pk_A^* which of the two is the correct one (other than guessing).

- *Phase 2:* When $\mathcal{A}$ calls f_{zk}^{R} with the correct parameters sent to him he will still receive 1 back, or 0 otherwise, which is the same as would be expected in a real execution. The hash function $\mathsf{H}(\cdot)$ is expected to output a random value for the Schnorr challenge as it is defined in the random oracle model. In the simulated case $\mathcal{S}$ calculates the output value from the final signature that depends on the input values of Alice and Bob of which at least Alice input is chosen randomly by $\mathcal{S}$. As dependent on random tape the calculation output will as well be distributed uniformly across the possible values and is therefore indistinguishable from a real hash function output. Furthermore, $\mathcal{A}$ can not recover the original input from the hash output. Imagine that he would be able to do so, he would then be able to guess the correct input from any hash output and thereby break the Pre-image Resistance property of the hash function. The remaining messages sent by $\mathcal{A}$ are identical to what would be expected in a real execution due to the deterministic nature of $\mathcal{A}$.
- *Phase 3:* The simulator will now verify the values sent to him by $\mathcal{A}$ and will halt and output $\perp$ in the case that he sends something invalid which is again identical to what is expected in a real execution. In this case $\mathcal{A}$ must not receive (σ_{fin}, pk) in the ideal setting which is modelled by $\mathcal{S}$ sending abort to the TTP. Otherwise $\mathcal{S}$ will calculate his part of the partial signature as defined by the protocol. It will therefore found to be valid by $\mathcal{A}$ and will complete to σ_{fin} with finSig, because of the fixed, calculated Schnorr challenge $\mathcal{S}$ calculated in Phase 2.
- If $\mathcal{A}$ behaves dishonestly at any point of the protocol then the simulator will notice, sent abort to the TTP and output whatever $\mathcal{A}$ outputs. If the adversary instead behaves as defined in the protocol specification, the protocol will be played until the end after which $\mathcal{S}$ again outputs whatever $\mathcal{A}$ outputs. Therefore in any case the outputs must be indistinguishable from the adversaries output in a real execution.

We have managed to show that in the case that Bob is corrupted the transcript is indistinguishable from a real transcript. We can therefore conclude that the transcript output will be indistinguishable from a real one in all cases and have thereby proven that the protocol **dSign** is secure in the f_{zk}^{R}-model and theorem theorem 1 must hold. □

We now do the same for **dAptSign**: Again we adjust the protocol with calls to f_{zk}^{R}, note that we now have one additional call f_{zk}^{R}, for the pair (x, X). The relation R is equally defined as in the previous proof. The adjusted protocol can be seen in fig. 4.7.

Theorem 2. Assume we have two key pairs (sk_A, pk_A) and (sk_B, pk_B) which were setup securely as for instance with the distributed keygen protocol $\mathsf{dKeyGen}$ and a hash function $\mathsf{H}(\cdot)$ modeled in the random oracle model. Additionally, we have a pair (x, X) in the relation $X = g^x$ for which x was chosen randomly. Then **dAptSign** securely computes a signature σ_{fin} under the composite public key $pk := pk_A \cdot pk_B$ after which x is revealed to Alice, in the f_{zk}^{R}-model.

dAptSign$\langle(m, sk_A, n_A), (m, sk_B, n_B, x)\rangle$

Alice		*Bob*
$\mathsf{f}_{zk}^{R}((sk_A, pk_A))$		
$\mathsf{f}_{zk}^{R}((n_A, R_A))$		
$\Lambda := \{pk := 1_p, R := 1_p\}$		
$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
	$\xrightarrow{\Lambda, pk_A, R_A}$	
		if $\mathsf{f}_{zk}^{R}(pk_A) = 0 \lor$ $\mathsf{f}_{zk}^{R}(\Lambda.R) = 0$
		return $\perp$
		$\mathsf{f}_{zk}^{R}((sk_B, pk_B))$
		$\mathsf{f}_{zk}^{R}((n_B, R_B))$
		$\mathsf{f}_{zk}^{R}((x, X))$
		$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
		$\tilde{\sigma}_B \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
		$\hat{\sigma}_B \leftarrow \mathsf{adaptSig}(\tilde{\sigma}_B, x)$
		$pk_B := g^{sk_B}$
		$X := g^x$
	$\xleftarrow{\tilde{\sigma}_B, \Lambda, pk_B, X}$	
if $\mathsf{vrfAptSig}(\tilde{\sigma}_B, m, pk_B, X) = 0$		
return $\perp$		
if $\mathsf{f}_{zk}^{R}(pk_B) = 0 \lor$ $\mathsf{f}_{zk}^{R}(\Lambda.R \cdot R_A^{-1}) = 0 \lor$ $\mathsf{f}_{zk}^{R}(X) = 0 \lor$		
return $\perp$		
$\tilde{\sigma}_A \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$		
	$\xrightarrow{\tilde{\sigma}_A}$	
		if $\mathsf{vrfPt}(\tilde{\sigma}_A, m, pk_A) = 0$
		return $\perp$
		$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma}_A, \tilde{\sigma}_B)$
	$\xleftarrow{\sigma_{fin}}$	
$pk \leftarrow \Lambda.pk$		$pk \leftarrow \Lambda.pk$
if $\mathsf{verf}(m, \sigma_{fin}, pk) = 0$		
return $\perp$		
$x \leftarrow \mathsf{extWit}(\sigma_{fin}, \tilde{\sigma}_A, \hat{\sigma}_B)$		
return $(x, (\sigma_{fin}, pk))$		**return** (σ_{fin}, pk)

Figure 4.7: Adjustments to the **dAptSign** protocol seen in fig. 4.5

Proof. We proof the security of **dAptSign** by constructing a simulator $\mathcal{S}$ who is given the output (σ_{fin}, pk) (resp. $(x, (\sigma_{fin}, pk))$) from a TTP that securly computes the protocol in the ideal world after receiving the inputs from Alice and Bob. The simulators task again is to extract the adversaries inputs and send them to the trusted third party to receive the protocol outputs. From this output the simulator $\mathcal{S}$ will construct a transcript that is indistinguishable from the protocol transcript in the real world. The simulator uses the calls to f_{zk}^{R} in order to do this. As in the proof before we assume the message m is known to both participants. All other inputs (including public keys) are only known to the respective party at the start of the protocol. We proof that the transcript is indistinguishable in case Alice is corrupted as well as in the case that Bob is corrupted.

Alice is corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$. When $\mathcal{A}$ internally calls f_{zk}^{R} and f_{zk}^{R} $\mathcal{S}$ saves (sk_A, pk_A) and (n_A, R_B) to its memory.

2. $\mathcal{S}$ receives $(\Lambda, pk_A^*, pk_B^*)$ from $\mathcal{A}$. $\mathcal{S}$ checks the equalities $pk_A^* = pk_A$ and $R_A^* = R_A$ as well as checking $pk_A = g^{sk_A}$ and $R_A = g^{n_A}$. If any of those checks fail, or he doesn't receive some of the values at all $\mathcal{S}$ sends abort to the TTP and outputs whatever $\mathcal{A}$ outputs. Otherwise he sends (m, sk_A, n_A) to the TTP and receives $(x, (\sigma_{fin}, pk))$

3. Again $\mathcal{S}$ calculates $\tilde{\sigma_B}, pk_B, R_B$ and finalizes the context Λ as follows:

$$
\begin{aligned}
(s, R) &\leftarrow \sigma_{fin} \\
pk_B &:= pk \cdot pk_A^{-1} \\
R_B &:= R \cdot R_A^{-1} \\
\Lambda &\leftarrow \mathsf{setupCtx}(\Lambda, pk_B, R_B) \\
\tilde{\sigma_A} &\leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda) \\
(s_A, R_A, \Lambda) &\leftarrow \tilde{\sigma_A} \\
s_B &:= s - s_A \\
\tilde{\sigma_B} &:= (s_B, R_B, \Lambda)
\end{aligned}
$$

4. $\mathcal{S}$ calculates $s_B^* := s_B + x$ (extracted from the TTP output) from which he sets $\hat{\sigma_B} := (s_B^*, R_B, \Lambda)$.

5. $\mathcal{S}$ sends $(\hat{\sigma_B}, \Lambda, pk_B, X := g^x)$ as if coming from Bob.

6. When $\mathcal{A}$ calls f_{zk}^{R} we compare the parameters send by $\mathcal{A}$ to the real one, in case he sent a invalid value $\mathcal{S}$ returns 0, otherwise 1.

7. $\mathcal{S}$ receives $\tilde{\sigma_A}^*$ from $\mathcal{A}$ and in any case outputs whatever $\mathcal{A}$ outputs.

The phases are similar to the ones defined in section 4.4.2 with the only two adjustments being that a) in Phase 2 Bob additionally sends X to Alice and b) we introduce a new Phase 4 in which Bob sends σ_{fin} to Alice. Yet for the sake of completeness we write the full proof in the following: **(1)** Alice sends (sk_A, pk_A) as well as (n_A, R_A) to f_{zk}^R and (Λ, pk_A, R_A) to Bob. **(2)** Bob sends pk_A and $\Lambda.R$ to f_{zk}^R as the verifier, and (sk_B, pk_B), (n_B, R_B) to f_{zk}^R as the prover. Afterward he sends $(\tilde{\sigma_B}, \Lambda, pk_B, X)$ to Alice. **(3)** Alice sends pk_B and R_B to f_{zk}^R as the verifier and $\tilde{\sigma_A}$ to Bob. **(4)** Bob sends the final signature σ_{fin} to Alice. They both output (σ_{fin}, pk) and Alice additionally outputs x. Finally, again we have to show that the simulators protocol output is equivalent to what is expected of $\mathcal{A}$ in a real execution.

We now again argue why the transcript of each phase has to be indistinguishable from a real transcript:

- *Phase 1:* As $\mathcal{A}$ is required to be deterministic, we can conclude that the transcript in this phase must be indistinguishable from a real transcript.
- *Phase 2:* In this phase $\mathcal{S}$ sends $X := g^x$ to $\mathcal{A}$ for which x was received from the TTP, therefore it will resamble the value that would have been expected in a real execution.
- *Phase 3:* The transcript in this phase must be indistinguishable for the same reasons already layed out in Phase 1.
- *Phase 4:* Now the $\mathcal{A}$ expects to receive σ_{fin}, from which he is able to extract the witness x. Indeed he will receive σ_{fin} as $\mathcal{S}$ has received from the TTP, which is exactly what would have been expected in a real execution. It must furthermore hold that $\mathcal{A}$ will be able to extract the correct x using the extWit procedure, as the simulator calculated $X = g^x$ in step 5.
- In that case that $\mathcal{A}$ behaves dishonestly and at any time of the protocol by sending invalid (or no) values to the simulator, he will detect this, abort the further protocol execution and output whatever $\mathcal{A}$ outputs. Similarily in the case that $\mathcal{A}$ behaves honestly the protocol is played until the end after which $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. So in both cases the outputs will be equivalent to what is expected in a real execution.

We have shown that in the case that Alice is corrupt the simulated transcript produced by $\mathcal{S}$ is indeed distributed equally to a real execution and is thereby computationally indistinguishable.

Bob is corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ starts by sampling $sk_A, n_A \leftarrow\$ \mathbb{Z}_*^*$ and proceeds by setting up the initial signature context as defined in the protocol:

$$\Lambda := \{pk := 1, R := 1\}$$
$$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$$

2. $\mathcal{S}$ now invokes $\mathcal{A}$ and sends (Λ, pk_A, R_A) as if coming from Alice.
3. When $\mathcal{A}$ calls f_{zk}^R (as the verifier) $\mathcal{S}$ checks for equality with the values sent by him and returns either 0 or 1. Once $\mathcal{A}$ sends (sk_B, pk_B), (n_B, R_B), (x, X) internally to f_{zk}^R as the prover, $\mathcal{S}$ saves them to his memory.
4. $\mathcal{S}$ sends (m, sk_A, n_A, x) to the TTP and receives (σ_{fin}, pk).
5. When $\mathcal{A}$ queries $\mathsf{H}(\cdot)$ the simulator again sets the output to e^* calculated with the following steps already seen in the previous proof:

$$\sigma_{fin} = n_A + sk_A \cdot e^* + n_B + sk_B \cdot e^*$$
$$e^* = \frac{\sigma_{fin} - n_A - n_B}{sk_A + sk_B}$$

6. $\mathcal{S}$ receives $(\hat{\sigma_B}^*, pk_B^*, \Lambda, X^*)$ from $\mathcal{A}$ and verifies those values checking equality with the ones stored in its memory. If the equality checks succeed $\mathcal{S}$ sends $\mathsf{continue}$ to the TTP, otherwise sends abort and outputs whatever $\mathcal{A}$ outputs.
7. The simulator now calculates $\tilde{\sigma_A}$ as defined by the protocol using the signPt procedure and sends the result to $\mathcal{A}$ as if coming from Alice.
8. Finally $\mathcal{S}$ will receive σ_{fin}^* from $\mathcal{A}$ and in any case output whatever $\mathcal{A}$ outputs.

Again we argue why the transcript is indistinguishable in phases 1–4:

- *Phase 1:* As argued in section 4.4.2 in this phase the adversary will just receive some public, nonce commitment and signature context. As he does not know Alice real inputs he has no way of knowing if the values received are the correct ones fitting with Alice inputs, other than by guessing.
- *Phase 2:* As argued before due to the hash function being modeled in the random oracle its output is expected to be randomly distributed. As the calculation done by $\mathcal{S}$ to create the hash output relies itself on randomly chosen values (sk_A, n_A), we can conclude that the output is distributed indistinguishably from a real hash output. Further $\mathcal{A}$ must not know the original input value by seeing the hash output he receives as he then would also be able to break the Pre-image Resistance property of the hash function.

- *Phase 3:* In this section $\mathcal{S}$ will verify equality of the values sent by $\mathcal{A}$ with the variables saved prior to its memory and if any of the values are unequal or invalid. In this case $\mathcal{A}$ should not receive the final outputs (σ_{fin}, pk) which is modelled by sending abort to the TTP. The same behaviour is expected in a real execution when Alice calls f_{zk}^{R} and receives a 0 bit. $\tilde{\sigma_A}$ must be indistinguishable from a real execution because it was calculated by $\mathcal{S}$ exactly as of protocol definition.

- *Phase 4:* In this phase $\mathcal{S}$ is expected to receive σ_{fin}^{*} from $\mathcal{A}$ after which $\mathcal{S}$ will simply output whatever $\mathcal{A}$, which must be indistinguishable from a real execution because of the deterministic adversary.

- Again in both the case that $\mathcal{A}$ deviates from protocol specification and in the case that he follows it $\mathcal{S}$ will output whatever $\mathcal{A}$ outputs, therefore being equal to what would be the expected output from $\mathcal{A}$ in a real execution.

We have shown that the transcript produced by $\mathcal{S}$ in an ideal world with access to a TTP computing **dAptSign** is indistinguishable from a transcript produced during a real execution both in the case that Alice and that Bob is corrupted. By managing to show this we have proven that the protocol is secure in f_{zk}^{R}-model and theorem theorem 2 therefore holds. □

CHAPTER 5

Adaptor Signature Based Atomic Swaps Between Bitcoin and a Mimblewimble Based Cryptocurrency

In this section, we will first define procedures and protocols to construct Mimblewimble transactions and prove their security. The formalizations will be similar to those found by Fuchsbauer et al. in their cryptographic investigation of the Mimblewimble protocol [7]. In particular the final transaction output from our protocols should be a valid transaction as by the definitions of Fuchsbauer et al. As we will only focus on the transaction building protocol (section 3.4), the notions of cut through (section 3.4), transaction merging (section 3.4), coin minting (see coinbase transactions section 3.4), and publishing transactions to the ledger (section 3.4), all formalized by Fuchsbauer et al. in [7], will not be topic of this formalization.

As an extension to the regular transaction protocol *Mimblewimble Transaction Scheme*, which we will define first, we will additionally define two further schemes: The first of them titled *Extended Mimblewimble Transaction Scheme* will provide additional functions to create and spend coins owned by two parties instead of just one, thereby enabling coins owned by mutliple parties, which is similar to a mutlisig address in Bitcoin [2]. The second extended definition is called *Contract Mimblewimble Transaction Scheme* in which we further add algorithms that allow embedding primitive smart contracts to the transaction building protocol. Both the *Extended Mimblewimble Transaction Scheme* and *Contract Mimblewimble Transaction Scheme* are constructed with the goal in mind of providing the functionality that is later needed to build the final Atomic Swap protocol which we will introduce in section 5.5.

We will proceed by providing an instantiation of the three transactions schemes in section 5.2 which can be implemented and deployed on a Mimblewimble-based Cryptocurrency such as Beam or Grin. In section 5.3 we define two-party protocols from the outlined schemes to construct mimbewimble transactions. Section 5.4 shows the proofs that the schemes are correct and the protocols secure in the malicious setting as defined in definition 3.8. Finally, in section 5.5, we define a Atomic Swap protocol from these building blocks, allowing two parties to securely and trustlessly swap funds from a Mimblewimble based blockchain with those on another blockchain, such as Bitcoin.

5.1 Definitions

As we have already discussed in section 3.4 for the creation of a transaction in Mimblewimble, it is immanent that both the sender and receiver collaborate and exchange messages via a secure channel. To construct the transaction protocol we assume that we have access to a two-party signature scheme Φ_{MP} as defined in definition 4.1, a range proof system as defined in definition 3.12 such as Bulletproofs, as described in section 3.3.2 and a homomorphic commitment scheme COM as defined in definition 3.6 such as Pedersen Commitments seen in definition 3.7.

Fuchsbauer et al. have defined three procedures Send, Rcv and Ldgr with regards to the creation of a transaction. Send called by the sender will create a pre-transaction, Rcv takes the pre-transaction and adds the receivers output and Ldgr (again called by the sender) verifies and publishes the final transaction to the blockchain ledger. As we have already pointed out in this thesis we won't discuss the transaction publishing phase therefore we will not cover the publishing functionality of the Ldgr procedure, however we will use the verification capabilities of the algorithm. That means the transactions created by our protocol must be compatible with the $\mathsf{MW.Ver}(1^n, tx)$ functionality formalized by Fuchsbauer et at. and internally used by Ldgr. We can however assume that a transaction tx for which $\mathsf{MW.Ver}(1^n, tx) = 1$ holds, could be published to the ledger using the Ldgr algorithm. (Given the inputs used in the transaction are in fact present and unspent on the ledger)

Originally Fuchsbauer et al. have defined the creation of a Mimblewimble transaction as a two step two-party protocol in which a sender owning a set of input coins calls Send to create an initial pre-transaction which is signed already by the sender and then forwarded to the fund receiver. The receiver then calls Rcv to add his own output coins with the correct value, and his signature which is then aggregated with the senders signature and thereby finalizing the transaction tx. Any party (knowing the final tx) can now call Ldgr to verify and publish the transaction to the ledger.

We now want to motivate why in the following we found it necessary to redefine some of the algoritm's already layed out by Fuchsbauer et al. The main reason is that in our formalization we are using the notion of two-party signatures as of definition 4.1 instead of aggregateable signatures, which are employed in their paper. While aggregateable signatures are quite similar to the two-party signatures we can find some important differences, ultimately making the two-party signatures, as we shall see, the more appropriate and secure choice for the formalization. First of all we need to define the notion of a aggregateable signature scheme:

Definition 5.1 (Aggregateable Signature Scheme)**.** A signature scheme Φ can be called aggregateable if for two signatures σ_1 and σ_2, valid for a message m under the public keys pk_1 and pk_2 we can construct a aggregated signature σ_a valid for the same message m under the composite public key $pk_a = pk_1 \cdot pk_2$

In the case of the Schnorr signature scheme we can only aggregate signatures by concatenating the individual signatures like $\sigma_1 \mid\mid \sigma_2$. The verifier would then check the validity of σ_1 and σ_2 independently under the public keys pk_1, pk_2 and finally check if $pk_a \stackrel{?}{=} pk_1 \cdot pk_2$ [7].
The reason why we can not simply add up the signatures is the following: Recall the structure of a Schnorr signature (s, R), imagine we would try to create an aggregated signature like $\sigma_a = (s_1 + s_2, R_1 \cdot R_2)$, then this would not be a valid signature anymore. This is because, again recalling the structure of Schnorr, s is calculated as $s = n + e \cdot sk$ where $e = \mathsf{H}(m \mid\mid R \mid\mid pk)$. Now as we have changed the nonce commitment R as well as the public key pk_a in our aggregated signature the Schnorr challenge e will be different from the one used by the individual signers and thereby making the verification algorithm return 0. We can fix this issue by having the individual signers use the final composite R and pk_a for their Schnorr challenge calculation, which is indeed exactly what we are doing in the Schnorr-based instantiation of two two-party signature scheme in fig. 4.2. This however introduces the necessity for an initial setup phase in which the parties exchange messages to compute R and pk_a from their individual shares. By using the two-party Schnorr model instead of the aggregated Schnorr we save space, as we only need to store one single signature instead of multiple. Further we also only need to store the final public key pk_a and can disregard the individual public keys shares. We also note that the two-party version is what is implemented currently in Grin and Beam in practice.[1] Finally, there is another critical advantage that comes with the two-party Schnorr approach. To start the signing process the final composite pk_a and nonce commitment R need to be known. That also entails that the flow pointed out in [7], in which the transaction sender starts the signing process and the receiver completes it is no longer possible. Instead, the signing process can only start with the receivers turn and we need to introduce a third round in which the sender receives the partially signed pre-transaction from the receiver, adds his partial signature and only now is able to finalize the signature and therby the transaction. While having to add an additional round would seem like an inconvenience at first, we discover that by doing so we avoid being vulnerable to a *Transaction Sniff Attack*.

For the following attack to be possible we need to assume that the channel between the sender (Alice) and receiver (Bob) has been compromised, and therefore can no longer

[1] `https://tinyurl.com/y63hc4ua`

be considered secure. We show that under this assumption the formalization layed out by Fuchsbauer et al. would be vulnerable to the *Transaction Sniff Attack*, while our formalization, using two-party signatures instead of aggregateable signatures would still be secure.

Transaction Sniff Attack Imagine a sender Alice and receiver Bob. Alice owns three Mimblewimble coins and wants to send one of them to Bob to pay for service offered by Bob. They start the transaction building process and communicate via a channel that they assume to be secure. However, in reality the channel they are using is insecure and an attacker $\mathcal{A}$ has managed to compromise it and is secretly listening to every message exchanged between the two. With the notions defined by Fuchsbauer et al. Alice starts the protocol by running $ptx \leftarrow \mathsf{Send}(\cdot)$ and sending ptx to Bob via the channel. Bob has received ptx from Alice but decides to wait with the protocol continuation because of some urgent task that came up. In the meantime the malicious attacker managed to sniff ptx sent by Alice. Already containing Alice signature all the attacker has to do is guess the value that Alice might want to send, create an output coin with that value, add his own signature, aggregate it with Alice and broadcast the final transaction to the network. Since the range of possible amounts that Alice might want to transfer is limited, it is trivial for the attacker to guess it in polynomial time. When now Bob comes back to finalize the transaction, he will discover that he is unable to continue with the protocol, as the transactions input coins are already been spent and are now in possession of the attacker.

Starting the signing process only at the receivers turn and introducing a third round solves this issue, because Alices signature for her input coins will be added only at the last step. Using the notion of the two-party signature scheme instead of a aggregateable signature scheme forces us to make this change, because of the additional setup phase required. Even if the attacker would be able to sniff one of the pre-transactions sent between the parties now, because Alice will only ever add the signature for her input coins at end of the protocol, the attacker would not be able to compute a valid transaction.

We now define the standard *Mimblewimble Transaction Scheme* that intuitively allows a sender to transfer value stored in a Mimblewimble coin to a receiver. To improve the readability of our following formalizations we introduce a wrapper $sp\mathcal{C}$ which represents a spendable coin and contains a reference to the coin commitment C, rangeproof π, as well as its (secret) spending information which consist of the coins value v and blinding factor r.

$$sp\mathcal{C} \ := \ \{C, v, r, \pi\}$$

If we want to indicate that a spendable coin is used as an output coin in a transaction we write $sp\mathcal{C}^*$.

Definition 5.2 (Mimblewimble Transaction Scheme)**.** A Mimblewimble Transaction Scheme $MW[COM, \Phi_{MP}, \Pi_{RP}]$ with commitment scheme COM, two-party signature scheme Φ_{MP}, and range proof system Π_{RP} consists of the following tuple of procedures:

$$MW[COM, \Phi_{MP}, \Pi_{RP}] := (\mathsf{spendCoins}, \mathsf{recvCoins}, \mathsf{finTx}, \mathsf{verfTx})$$

- $(ptx, sp\mathcal{C}_A^*, (sk_A, n_A)) \leftarrow \mathsf{spendCoins}([sp\mathcal{C}], p, t)$: The spendCoins algorithm is a DPT function called by the sending party to initiate the spending of some input coins. As input, it takes a list of spendable coins $[sp\mathcal{C}]$ and a value p which should be transferred to the receiver. Optionally a sender can pass a block height t to make this transaction only valid after a specific time. It outputs a pre-transaction ptx which can be sent to a receiver, Alice's spendable change output coin $sp\mathcal{C}_A^*$ as well as the senders signing key and secret nonce (sk_A, n_A) later used in the transaction signing process.

- $(ptx^*, sp\mathcal{C}_B^*) \leftarrow \mathsf{recvCoins}(ptx, p)$: The receiveCoins algorithm is a DPT routine called by the receiver and takes as input a pre-transaction ptx and a fund value p. It will output a modified pre-transaction ptx^* together with Bob's new spendable output coin $sp\mathcal{C}_B^*$ which has been added to the transaction. At this stage the transaction already has to be partially signed by the receiver.

- $tx \leftarrow \mathsf{finTx}(ptx, sk_A, n_A)$: The finalize algorithm is a DPT routine again called by the transaction sender that takes as input a pre-transaction ptx and the senders signing key sk_A and nonce n_A. The function will output a finalized signed transaction tx.

- $\{1, 0\} \leftarrow \mathsf{verfTx}(tx)$: The verification algorithm is exactly the same as defined in the paper by Fuchsbauer et at. [7], we still add it here for completeness. Note that in the paper it can be found under the name MW.Ver, we renamed it here to verf to fit with our naming scheme. If an invalid transaction is passed to the routine, it will output 0, 1 otherwise. Informally the algoritm verifies four conditions:

  1. Condition 1: Every input and output coin only appears once in the transaction.
  2. Condition 2: The union of input and output coins is the empty set.
  3. Condition 3: For every output coin the range proof verifies.
  4. Condition 4: The transaction signature verifies with the excess value of the transaction as public key, which is calculated by summing up the output coins and subtracting the input coins. (See section 3.4)

We say a Mimblewimble Transaction Scheme is correct if the verification algorithm verfTx returns 1 if and only if the added transaction is well balanced and its signature is valid. More formally:

Definition 5.3 (Transaction Scheme Correctness)**.** For any transaction fund value p and list of spendable input coins $[sp\mathcal{C}]$ with combined value $v \geq p$ the following must hold:

$$\Pr\left[\; \mathsf{verfTx}(tx) = 1 \;\middle|\; \begin{array}{l} p \leq \sum_{i := 0}^{i \leq n}(sp\mathcal{C}_i.v) \\ (ptx, \cdot, (sk_A, n_A)) \leftarrow \mathsf{spendCoins}([sp\mathcal{C}], v, \perp) \\ (ptx^*, \cdot) \leftarrow \mathsf{recvCoins}(ptx, p) \\ tx \leftarrow \mathsf{finTx}(ptx^*, sk_A, n_A) \end{array} \right] = 1.$$

In the following we define the *Extended Mimblewimble Transaction Scheme*, which intuitively extends the previous scheme with shared coin ownership functionalities, similiar to multisignature addresses available in Bitcoin.

Definition 5.4 (Extended Mimblewimble Transaction Scheme)**.** An extended Mimblewimble transaction scheme $MW_{ext}[COM, \Phi_{MP}, \Pi_{RP-MP}]$ is an extension to MW with the following three procedures:

$$MW_{ext}[COM, \Phi_{MP}, \Pi_{RP-MP}] := MW[COM, \Phi_{MP}, \Pi_{RP-MP}] \mid\mid (\mathbf{dSpendCoins}, \mathbf{dRecvCoins}, \mathbf{dFinT}$$

Note that for this scheme we require a multiparty range proof system Π_{RP-MP} as defined in definition 3.13. Specifically we require the proof system to provide a distributed proof computation protocol **dRanPrf**.

- $\langle (ptx, sp\mathcal{C}_A^*, (sk_A, n_A)), (ptx, sp\mathcal{C}_C^*, (sk_C, n_C)) \rangle$ $\leftarrow$ $\mathbf{dSpendCoins}\langle ([sp\mathcal{C}_A], p, t), ([sp\mathcal{C}_C], p) \rangle$: The distributed coin spending algorithm takes as input a list of spendable input coins for which ownership is shared between Alice and Carol. Assume that a coin $\mathcal{C}$ is owned by both Alice and Carol, then we have two blinding factors r_A, r_C, where r_A is known only to Alice and r_C only to Carol. Both blinding factors are required needed to spend the coin. Again optionally a block height t can be given to time lock the transaction. Similar to the single party version of the function its outputs are a pre-transaction ptx and change coin for each party $sp\mathcal{C}_A^*$ (resp. $sp\mathcal{C}_C^*$), as well as their signing information.

- $\langle (ptx^*, psp\mathcal{C}_B^*), (ptx^*, psp\mathcal{C}_C^*) \rangle \leftarrow \mathbf{dRecvCoins}\langle (ptx, p), () \rangle$: The distributed coin receive procedure takes as input a pre-transaction ptx and a value p which should be transferred with the transaction. The distributed algorithm will generate a output coin with value v, owned by both Bob and Carol (each knowing only a share of the coin commitment's blinding factor). The output will be an updated pre-transaction ptx^*, and the spendable shared output coins for each party $psp\mathcal{C}_B^*$ (resp. $psp\mathcal{C}_C^*$). Note that the newly generated output coin can only be spent by both parties cooperating, as each share of the blinding factor is strictly required. We note here that creating more complex schemes in which a coin is spendable by knowing N out M keys would be possible by implementing Shamir's Secret Sharing algorithm which can be found in [22].

- $\langle tx, tx\rangle \leftarrow$ **dFinTx**$\langle(ptx, sk_A, n_A), (ptx, sk_C, n_C)\rangle$: The distributed finalized transaction protocol has to be used if we are creating a transaction spending a shared coin (i.e. the transaction was created with the **dSpendCoins** algorithm). In this case we require signing information from both Alice and Carol.

Correctness is given very similar to the standard scheme:

Definition 5.5 (Extended Transaction Scheme Correctness)**.** For any list of spendable coins $[sp\mathcal{C}]$ with total value v greater than the transaction fund value p and split blinding factors $([r_A], [r_C])$ the following must hold:

$$\Pr\left[\ \mathsf{verfTx}(tx) \ = \ 1 \ \middle|\ \begin{array}{l} p \ \leq \ \sum_{i\ :=\ 0}^{i\ <\ n}(sp\mathcal{C}_i.v) \\ \langle(ptx, \cdot, (sk_A, n_A)), (ptx, (sk_C, n_C))\rangle \ \leftarrow \\ \textbf{dSpendCoins}\langle([sp\mathcal{C}_A], p, \perp), ([sp\mathcal{C}_C], p)\rangle \\ \langle(ptx^*, \cdot)(ptx^*, \cdot)\rangle \ \leftarrow \ \textbf{dRecvCoins}\langle(ptx, p), ()\rangle \\ tx \ \leftarrow \ \textbf{dFinTx}\langle(ptx^*, sk_A, n_A), (ptx^*, sk_C, n_C)\rangle \end{array} \right] = 1.$$

Now we define the *Contract Mimblewimble Transaction Scheme*, which will extend the scheme with additional algorithms allowing to create primitive contracts between the sending and receiving party.

Definition 5.6 (Contract Mimblewimble Transaction Scheme)**.** The contract version of the Mimblewimble Transaction Scheme updates the Extended Mimblewimble Transaction Schme by providing a modified version of the single party receive routine and the distributed finalize transaction protocol.

$$MW_{apt}[COM, \Phi_{MP}, \Pi_{RP-MP}] \ := \ MW_{ext}[COM, \Phi_{MP}, \Pi_{RP-MP}] \ || \ (\mathsf{aptRecvCoins}, \textbf{dAptFinTx})$$

- $(ptx^*, sp\mathcal{C}_B^*, \tilde{\sigma_B}) \ \leftarrow \ \mathsf{aptRecvCoins}(ptx, p, x)$: The contract variant of the receive function takes an additional input a secret witness value x which will be hidden in the transaction signature and extractable by the other party after the protocols' completion. Note that the routine also returns Bob's unadapted partial signature. The reason for this is that we later still need the unadapted version to complete the signature und thereby finalize the transaction. By not sharing this unadapted signature with Alice, Bob is the one who gets to finalize the transaction which is different from the simpler protocol and is an important feature necessary for our atomic swap protocol. We want to stress here that aptRecvCoins is only a single party algorithm, as such it can only be used in the case that we want to create an output coin owned by a single receiver. It would of course be conceivable to also define a distributed version similar to **dRecvCoins** of this functionality, allowing two receivers (or one of the two) to hide secret witness values, extractable later by the sender(s). However, as for the following protocols such functionality is not needed we omit it here for clarity.

- $\langle \tilde{\sigma_{AB}}, tx \rangle \leftarrow$ **dAptFinTx**$\langle (ptx^*, sk_A, n_A, X), (ptx^*, sk_B, n_B, \tilde{\sigma_B}) \rangle$: The contract variant of the finalize transaction algorithm is a distributed protocol between the sender(s) and receiver. Additionally to the pre-transaction ptx^* the senders need to input their signing information, Bob needs to input the unadapted version of his partial signature as it is needed for transaction completion. This protocol could also be implemented as a three party protocol, two senders controlling a shared coin and a third receiver. However, as in our case, which we will describe later in section 5.3, one of the two senders is also the receiver, we allowed ourselves to model this protocol as being between only two parties to simplify the formalization. In this version of the protocol only Bob will be able to finalize the transaction, which is different to finTx and **dFinTx**. This has the practical reason that for the atomic swap execution Bob needs to be the one in control of building the final transaction. If Alice were to build the final transaction before Bob, she will be able to extract the witness value before the transaction has been published, which in the atomic swap scenario would mean she could steal the funds stored on the other chain. This is why the protocol does not return the final transaction tx to Alice, instead the protocol will output the senders partial signature, which Alice can later use to extract the witness value from the final transaction.

Similar as before we define correctness for the adapted scheme:

Definition 5.7 (Contract Transaction Scheme Correctness)**.** For any transaction fund value p and list of input coins $[sp\mathcal{C}]$ with combined value $v \geq p$ and any witness value $x \in \mathbb{Z}_p^*$ the following must hold:

$$\Pr\left[\mathsf{verfTx}(tx) = 1 \;\middle|\; \begin{array}{l} p \leq \sum_{i\,:=\,0}^{i\,<\,n}(sp\mathcal{C}_i.v) \\ (ptx, sp\mathcal{C}_A^*, (sk_A, n_A)) \leftarrow \mathsf{spendCoins}([sp\mathcal{C}], p, \perp) \\ (ptx^*, sp\mathcal{C}_B^*, \tilde{\sigma_B}) \leftarrow \mathsf{aptRecvCoins}(ptx, p, x) \\ \langle \tilde{\sigma_{AC}}, tx \rangle \leftarrow \mathbf{dAptFinTx}\langle (ptx, sk_A, n_A, X), (ptx, sk_C, n_C, \tilde{\sigma_B}) \rangle \end{array} \right] = 1.$$

5.2 Instantiation

In this section we will provide an instantiation of the transaction scheme definitions found in definition 5.2, definition 5.4 and definition 5.6. The instantiations can be implemented in a Cryptocurrency based on the Mimblewimble protocol such as Beam and Grin.

5.2.1 Mimblewimble Transaction Scheme

First we provide an instantiation of the simplest form of a transaction in which a sender wants to transfer some value p to a receiver. For the execution of the protocol we assume to have access to a homomorphic commitment scheme such as Pedersen Commitment COM as defined in definition definition 3.7. Furthermore we require a range proof system Π_{RP} as defined in section 3.3.2 and a two-party signature scheme Φ_{MP} as defined in definition 4.1.

To make the pseudocode for the transaction protocol easier to read we first introduce two auxiliary functions createCoin and createTx. The coin creation function will take as input a value v and a blinding factor r. It will create and output a new spendable coin $sp\mathcal{C}$ already containing a range proof π attesting to the statement that the coins value v is within the valid range as defined for the blockchain. The transaction creation algorithm createTx takes as input a message m, a list of input coins $[\mathcal{C}_{inp}]$, a list of output coins $[\mathcal{C}_{out}]$, a list of rangeproofs $[\pi]$, a signature context Λ, a list of commitments C, a signature σ, and a lock time t and will collect the input data into a transaction object.

$\mathsf{createCoin}(v, r)$	$\mathsf{createTx}(m, [\mathcal{C}_{inp}], [\mathcal{C}_{out}], [\pi], \Lambda, [C], \sigma)$
1 : $C \leftarrow \mathsf{commit}(v, r)$	1 : **return** (
2 : $\pi \leftarrow \mathsf{ranPrf}(\mathcal{C}, v, r)$	$m := m,$
3 : **return** (C, r, v, π)	$inp := [\mathcal{C}_{inp}],$
	$out := [\mathcal{C}_{out}],$
	$\Pi := [\pi],$
	$\Lambda := \Lambda,$
	$com := [C],$
	$\sigma := \sigma,$
	$t := t)$

In figure fig. 5.1 we provide an instantiation of the Mimblewimble Transction Scheme using the auxiliary functions provided before.

In the spendCoins function the sender creates his change output coin, which is the difference between the value stored in his input coins and the value which should be transferred to a receiver. He sets up the signature context with his parameters and gets a pre-transaction ptx, newly created spendable output coin $sp\mathcal{C}_A$, as well as a signing key sk_A and secret nonce n_A as output. The pre-transaction can then be sent to a receiver. Note that, as we have already explained earlier, our instantiation differs from the one described by Fuchsbauer et al. [7] in that the sender does not yet sign the transaction during spendCoins, because we are using a Two-Party Signature Scheme definition 4.1 instead of a aggregateable signature scheme definition 5.1.

In recvCoins the receiver of a pre-transaction will verify the senders proof π_B, create his output coin $\mathcal{C}^B_{out}$, add his parameters to the signature context and then create his partial signature $\tilde{\sigma_B}$. The function returns an updated version of the pre-transaction ptx which can be sent back to the sender, as well as the newly created spendable output $sp\mathcal{C}_B$.

Now in finTx the original sender will validate the updated pre-transcation ptx sent to him by the receiver. If he finds it as valid, he will only now create his partial signature and finally finalize the two partial signatures into the final composite one, with which he can then build the final transaction.

$\mathsf{spendCoins}([sp\mathcal{C}], p, t)$

$v \leftarrow \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.v)$

$\textbf{if } p > v \textbf{ return } \bot$

$\textbf{if } \exists\, i \neq j : sp\mathcal{C}[i] = sp\mathcal{C}[j] \textbf{ return } \bot$

$m := \{0,1\}^*$

$(r_A^*, n_A) \leftarrow\$\, \mathbb{Z}_p^*$

$sp\mathcal{C}_A^* \leftarrow \mathsf{createCoin}(v - p, r_A^*)$

$\{\mathcal{C}_{out}^A, r_A^*, v_A, \pi_A\} \leftarrow sp\mathcal{C}_A^*$

$sk_A := r_A^* - \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.r)$

$\Lambda := \{pk := 1_p, R := 1_p\}$

$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$

$ptx \leftarrow \mathsf{createTx}(m, sp\mathcal{C}.C, [\mathcal{C}_{out}^A], [\pi_A], \Lambda, [g^{sk_A}], \emptyset)$

$\textbf{return } (ptx, sp\mathcal{C}_A^*, (sk_A, n_A))$

$\mathsf{recvCoins}(ptx, p)$

$(m, inp, out, \Pi, \Lambda, com, \emptyset, t) \leftarrow ptx$

$\textbf{if } \mathsf{vrfRanPrf}(\Pi[0], out[0]) = 0$

$\quad \textbf{return } \bot$

$(r_B^*, n_B) \leftarrow\$\, \mathbb{Z}_p^*$

$sp\mathcal{C}_B^* \leftarrow \mathsf{createCoin}(p, r_B^*)$

$\{\mathcal{C}_{out}^B, r_B^*, v_B, \pi_B\} \leftarrow sp\mathcal{C}_B^*$

$sk_B := r_B^*$

$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$

$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$

$ptx \leftarrow \mathsf{createTx}(m, inp, out \,||\, \mathcal{C}_{out}^B, \Pi \,||\, \pi_B, \Lambda, com \,||\, g^{sk_B}, \tilde{\sigma_B})$

$\textbf{return } (ptx, sp\mathcal{C}_B^*)$

$\mathsf{finTx}(ptx, sk_A, n_A)$

$(m, inp, out, \Pi, \Lambda, com, \tilde{\sigma_B}, t) \leftarrow ptx$

$\textbf{if } \mathsf{vrfRanPrf}(\Pi[1], out[1]) = 0$

$\quad \textbf{return } \bot$

$\textbf{if } \mathsf{vrfPt}(\tilde{\sigma_B}, m, com[1]) = 0$

$\quad \textbf{return } \bot$

$\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$

$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$

$tx \leftarrow \mathsf{createTx}(m, inp, out, \Pi, \Lambda, com, \sigma_{fin})$

$\textbf{return } tx$

$\mathsf{verfTx}(tx)$

$(m, inp, out, \Pi, \Lambda, com, \sigma, t) \leftarrow tx$

$\mathcal{E} = \sum(out) - \sum(inp)$

$\textbf{return } (\forall\, i \neq j : inp[i] \neq inp[j] \wedge out[i] \neq out[j]) \text{ and}$
$\quad inp \cup out = \emptyset \text{ and } (\forall\, i : \mathsf{vrfRanPrf}(\Pi[i], out[i])) \text{ and } \mathsf{verf}(m, \sigma, \mathcal{E})$

Figure 5.1: Instantiation of Mimblewimble Transaction Scheme.

5.2.2 Extended Mimblewimble Transaction Scheme

Figure fig. 5.2 shows an instantiation of the **dSpendCoins** function of the Extended Mimblewimble Transaction Scheme. We have an array of spendable input coins which keys are shared between two parties Alice and Carol. We use Carol here to not confuse this party with the receiver, which we previously called Bob. Although Carol and Bob could be the same person, they not necessarily have to be.

The protocol starts with both Alice and Carol creating her change outputs with values v_A and v_C. Alice then creates the initial pre-transaction ptx and sends it to Carol who verifies Alice's output, adds her outputs and parameters and sends back ptx, which Alice verifies. The protocol returns ptx to both parties, which can then be transmitted to the receiver by any of the two parties, as well as the secret signing information (sk_A, n_A), (sk_C, n_C).

dSpendCoins$\langle([psp\mathcal{C}_A], p, t), ([psp\mathcal{C}_C], p)\rangle$

Alice		*Carol*
$v \leftarrow \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.v)$		$v \leftarrow \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.v)$
if $p > v$		**if** $p > v$
return $\bot$		**return** $\bot$
if $\exists\, i \neq j : psp\mathcal{C}_A[i] = psp\mathcal{C}_A[j]$		**if** $\exists\, i \neq j : psp\mathcal{C}_C[i] = psp\mathcal{C}_C[j]$
return $\bot$		**return** $\bot$
$v_{rem} = v - p$		
$v_A, v_C \leftarrow \{0, v_{rem}\}$ s.t. $v_A + v_C = v_{rem}$		
	$\xrightarrow{v_C}$	
$m := \{0,1\}^*$		
$(r_A^*, n_A) \leftarrow\$\, \mathbb{Z}_p^*$		$(r_C^*, n_C) \leftarrow\$\, \mathbb{Z}_p^*$
$sp\mathcal{C}_A^* \leftarrow \mathsf{createCoin}(v_A, r_A^*)$		$sp\mathcal{C}_C^* \leftarrow \mathsf{createCoin}(v_C, r_C^*)$
$\{\mathcal{C}_{out}^A, r_A^*, v_A, \pi_A\} \leftarrow sp\mathcal{C}_A^*$		$\{\mathcal{C}_{out}^C, r_C^*, v_C, \pi_C\} \leftarrow sp\mathcal{C}_C^*$
$sk_A := r_A^* - \sum[r_A]$		$sk_C := r_C^* - \sum[r_C]$
$\Lambda := \{pk := 1_p, R := 1_p\}$		
$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
$ptx \leftarrow$ $\mathsf{createTx}(m, [\mathcal{C}_{inp}], [\mathcal{C}_{out}^A], [\pi_A], \Lambda, [g^{n_A}], \emptyset)$		
	$\xrightarrow{ptx}$	
		$(m, inp, out, \Pi, \Lambda, com, t) \leftarrow ptx$
		if $\mathsf{vrfRanPrf}(\Pi[0], out[0]) = 0$
		return $\bot$
		$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_C}, g^{n_C})$
		$ptx' \leftarrow \mathsf{createTx}(m, inp, out \mid\mid \mathcal{C}_{out}^C, \pi \mid\mid \pi_C, \Lambda, com \mid\mid g^{n_C}, \emptyset)$
	$\xleftarrow{ptx'}$	
if $\mathsf{vrfRanPrf}(ptx'.\Pi[1], ptx'.out[1]) = 0$		
return $\bot$		
return $(ptx', sp\mathcal{C}_A^*, (sk_A, n_A))$		**return** $(ptx', sp\mathcal{C}_C^*, (sk_C, n_C))$

Figure 5.2: Extended Mimblewimble Transaction Scheme - **dSpendCoins**

Figure fig. 5.3 shows an instantiation of the **dSpendCoins** function of the Extended Mimblewimble Transaction Scheme. Calling this protocol, two receivers Bob and Carol want to create a receiving shared coin $\mathcal{C}^{sh}_{out}$ with value p and key shares (r_A, r_C). The protocol starts by both receivers verifing the senders output(s). Bob starts by creating a coin with fund value p and his share of the newly created blinding factor and sends it over to Carol. Carol finalizes the shared coin by adding a commitment to her blinding factor to the coin and sends it back, together with the commitment. Bob verifies validity of the updated shared coin after which the two parties engage in two two-party protocols to create their partial signature and coin rangeproof. Finally they create the updated pre-transaction ptx which can be sent back to the transaction sender.

dRecvCoins$\langle (ptx, p), () \rangle$

Bob		*Carol*
$(m, inp, out, \Pi, \Lambda, com, \emptyset, t) \leftarrow ptx$		
foreach out *as* $(i => \mathcal{C}_{out})$		**foreach** out *as* $(i => \mathcal{C}_{out})$
if $\mathsf{vrfRanPrf}(\Pi[i], \mathcal{C}_{out}[i]) = 0$		**if** $\mathsf{vrfRanPrf}(\Pi[i], \mathcal{C}_{out}[i]) = 0$
return $\perp$		**return** $\perp$
$(r_B^*, n_B) \leftarrow\$ \mathbb{Z}_p^*$		
$(\mathcal{C}_{out}^{sh}, \cdot, \cdot, \cdot) \leftarrow \mathsf{createCoin}(p, r_B^*)$		
$sk_B := r_B^*$		
	$\xrightarrow{\quad ptx, \mathcal{C}_{out}^{sh} \quad}$	
		$(r_C^*, n_C) \leftarrow\$ \mathbb{Z}_p^*$
		$sk_C := r_C^*$
		${\mathcal{C}_{out}^{sh}}' := \mathcal{C}_{out}^{sh} \cdot g^{r_C}$
		$ptx' := ptx$
		$ptx'.out[] := {\mathcal{C}_{out}^{sh}}'$
	$\xleftarrow{\quad ptx', g^{sk_C} \quad}$	
$\{\cdots {\mathcal{C}_{out}^{sh}}'\} \leftarrow ptx'.out$		
if ${\mathcal{C}_{out}^{sh}}' \neq \mathcal{C}_{out}^{sh} \cdot g^{sk_C}$		
return $\perp$		
$\pi_{BC} \leftarrow \mathbf{dRanPrf}({\mathcal{C}_{out}^{sh}}', p, sk_B)$		$\pi_{BC} \leftarrow \mathbf{dRanPrf}({\mathcal{C}_{out}^{sh}}', p, sk_C)$
$psp\mathcal{C}_B^* := \{\mathcal{C}_{out}^{sh}, p, r_B^*, \pi_{BC}\}$		$psp\mathcal{C}_C^* := \{\mathcal{C}_{out}^{sh}, p, r_C^*, \pi_{BC}\}$
$(\sigma_{\tilde{BC}}, pk_{BC}) \leftarrow \mathbf{dSign}(m, sk_B, n_B)$		$(\sigma_{\tilde{BC}}, pk_{BC}) \leftarrow \mathbf{dSign}(m, sk_C, n_C)$
$(\cdot, \cdot, \Lambda^*) \leftarrow \sigma_{\tilde{BC}}$		$(\cdot, \cdot, \Lambda^*) \leftarrow \sigma_{\tilde{BC}}$
$\Lambda' \leftarrow \mathsf{setupCtx}(\Lambda, \Lambda^*.pk, \Lambda^*.R)$		$\Lambda' \leftarrow \mathsf{setupCtx}(\Lambda, \Lambda^*.pk, \Lambda^*.R)$
	$ptx^* \leftarrow \mathsf{createTx}(m, inp, out \parallel {\mathcal{C}_{out}^{sh}}', \Pi \parallel \pi_{BC}, \Lambda', com \parallel pk_{BC}, \sigma_{\tilde{BC}})$	
return $(ptx^*, psp\mathcal{C}_B^*)$		**return** $(ptx^*, psp\mathcal{C}_C^*)$

Figure 5.3: Extended Mimblewimble Transaction Scheme - **dRecvCoins**

dFinTx$\langle(ptx, sk_A, n_A), (ptx, sk_C, n_C)\rangle$

Alice	*Carol*
$(m, inp, out, \Pi, \Lambda, com, \tilde{\sigma_B}, t) \leftarrow ptx$	$(m, inp, out, \Pi, \Lambda, com, \tilde{\sigma_B}, t) \leftarrow ptx$
if $\mathsf{vrfRanPrf}(\Pi[1], out[1]) = 0$	**if** $\mathsf{vrfRanPrf}(\Pi[1], out[1]) = 0$
return $\perp$	**return** $\perp$
if $\mathsf{vrfPt}(\tilde{\sigma_B}, m, com[1]) = 0$	**if** $\mathsf{vrfPt}(\tilde{\sigma_B}, m, com[1]) = 0$
return $\perp$	**return** $\perp$
$\tilde{\sigma_{AC}} \leftarrow \mathbf{dSign}(m, sk_A, n_A)$	$\tilde{\sigma_{AC}} \leftarrow \mathbf{dSign}(m, sk_C, n_C)$
$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_B}, \tilde{\sigma_{AC}})$	$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_B}, \tilde{\sigma_{AC}})$
$tx \leftarrow \mathsf{createTx}(m, inp, out, \Pi, \Lambda, com, \sigma_{fin})$	$tx \leftarrow \mathsf{createTx}(m, inp, out, \Pi, \Lambda, com, \sigma_{fin})$
return tx	**return** tx

Figure 5.4: Extended Mimblewimble Transaction Scheme - **dFinTx**

Finally, fig. 5.4 shows the implementation of the **dFinTx** protocol. Running this protocol the two transaction senders, each owning a share of the input coins keys, will cooperate to produce a signature share valid under their input coins and change outputs, after which they can combine the partial signatures into the final one and finalize the transaction.

5.2.3 Adapted Extended Mimblewimble Transaction Scheme

Figure fig. 5.5 shows an instantiation of the aptRecvCoins algorithm. Before updating the pre-transaction *ptx* Bob adapts his partial signature with the witness value x. The procedure then returns the pre-transaction *ptx* containing Bob's adapted partial signature, and the statement X which is a commitment to the witness value x.

In figure fig. 5.6 we show the updated distributed version of the transaction finalization protocol. Again Alice verifies the pre-transaction *ptx* received by Bob and then cooperates with Bob in the **dSign** protocol to build the partial signature for their shared coin. Note that at this point Alice is not able to finalize the signature (and consequently the transaction) as she only knows Bob's adapted partial signature ($\hat{\sigma_B}$), but not the original one ($\tilde{\sigma_B}$), which is needed for the finSig function. Therefore, Bob completes the transaction and outputs it, while Alice outputs $\tilde{\sigma_{AB}}$ with which she can then retrieve x.

5.3 Protocols

In this section we specify three protocols to build Mimblewimble transactions from the definitions found in section 5.1. Later in section section 5.4 we will prove the security of these protocols and finally in section section 5.5 we will utilize them to build our Atomic Swap.

$\mathsf{aptRecvCoins}(ptx, p, x)$

$(m, inp, out, \Pi, \Lambda, com, \emptyset, t) \leftarrow ptx$
if $\mathsf{vrfRanPrf}(\Pi[0], out[0]) = 0$
  return $\perp$
$(r_B^*, n_B) \leftarrow\$ \mathbb{Z}_p^*$
$(\mathcal{C}_{out}^B, \pi_B) \leftarrow \mathsf{createCoin}(p, r_B^*)$
$sk_B := r_B^*$
$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, \Lambda.pk, \Lambda.R)$
$\hat{\sigma_B} \leftarrow \mathsf{adaptSig}(\tilde{\sigma_B}, x)$
$ptx \leftarrow \mathsf{createTx}(m, inp, out \mathbin{||} \mathcal{C}_{out}^B, \Pi \mathbin{||} \pi_B, \Lambda, com \mathbin{||} g^{n_B}, \hat{\sigma_B})$
return $(ptx, (\mathcal{C}_{out}^B, r_B^*), \tilde{\sigma_B})$

Figure 5.5: Adapted Extended Mimblewimble Transaction Scheme - $\mathsf{aptRecvCoins}$.

dAptFinTx$\langle(ptx, sk_A, n_A, X), (ptx, sk_B, n_B, \tilde{\sigma_B})\rangle$

Alice	*Bob*
$(m, inp, out, \Pi, \Lambda, com, \hat{\sigma_B}, t) \leftarrow ptx$	$(m, inp, out, \Pi, \Lambda, com, \hat{\sigma_B}, t) \leftarrow ptx$
if $\mathsf{vrfRanPrf}(\Pi[1], out[1]) = 0$	
return $\perp$	
if $\mathsf{vrfAptSig}(\tilde{\sigma_B}, m, com[1], X) = 0$	
return $\perp$	
$\tilde{\sigma_{AB}} \leftarrow \mathbf{dSign}(m, sk_A, n_A)$	$\tilde{\sigma_{AB}} \leftarrow \mathbf{dSign}(m, sk_B, n_B)$
	$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_{AC}}, \tilde{\sigma_B})$
	$tx \leftarrow \mathsf{createTx}(m, inp, out, \Pi, \Lambda, com, \sigma_{fin})$
return $\tilde{\sigma_{AB}}$	**return** tx

Figure 5.6: Adapted Extended Mimblewimble Transaction Scheme - **dAptFinTx**.

dBuildMWTx$\langle([spC], p, t), (p)\rangle$

$$
\begin{array}{llcl}
 & \textit{Alice} & & \textit{Bob} \\
1: & (ptx, spC^*_A, (sk_A, n_A)) & & \\
 & \quad \leftarrow \ \mathsf{spendCoins}([spC], p, t) & & \\
2: & & \xrightarrow{\quad ptx \quad} & \\
3: & & & (ptx', spC^*_B) \ \leftarrow \ \mathsf{recvCoins}(ptx, p) \\
4: & & \xleftarrow{\quad ptx' \quad} & \\
5: & tx \ \leftarrow \ \mathsf{finTx}(ptx', sk_A, n_A) & & \\
6: & & \xrightarrow{\quad tx \quad} & \\
7: & \mathbf{return}\ (tx, spC^*_A) & & \mathbf{return}\ (tx, spC^*_B)
\end{array}
$$

Figure 5.7: **dBuildMWTx** two-party protocol to build a new transaction

5.3.1 Simple Mimblewimble Transaction - dBuildMWTx

dBuildMWTx is a protocol between a sender and receiver which builds a Mimblewimble transaction transferring a value p from the sender to a receiver for a Mimblewimble Transaction scheme as defined in definition 5.2. It takes as input a list of spendable coins $[spC]$, a transaction value p, and an optional timelock t from the sender, the same transaction value p from the receiver and uses the functions defined earlier to output a valid transaction tx as well as the newly spendable coins to both parties.

$$\langle(tx, spC^*_A), (tx, spC^*_B)\rangle \ \leftarrow \ \mathbf{dBuildMWTx}\langle(spC^*, p, t), (p)\rangle$$

Figure fig. 5.7 shows the implementation of the **dBuildMWTx**.

5.3.2 Shared Output Mimblewimble Transaction - dsharedOutMWTx

dsharedOutMWTx is a protocol between a sender and a receiver. It builds a Mimblewimble transaction transferring value from a sender for the Extendend Mimblewimble Transaction Scheme in definition 5.4. However, instead of simply sending value to a receiver it sends it to a shared coin, for which both the sender and receiver know one part of the opening. As input it again takes a list of spendable coins $[spC]$, a transaction value p and an optional timelock t from the sender and the same transaction value p from the receiver. It outputs the final transaction tx to both parties, Alice will receive her spendable change output spC^*_A and both parties will receive their part of the shared spendable coin $pspC^*_A$, $pspC^*_B$.

$$\langle(tx, spC^*_A, pspC^*_A), (tx, pspC^*_B)\rangle \ \leftarrow \ \mathbf{dsharedOutMWTx}\langle([spC], p, t), ()\rangle$$

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\textbf{dsharedOutMWTx}\langle([sp\mathcal{C}], p, t), ()\rangle} \\
\hline
 & \textit{Alice} & & \textit{Bob} \\
1: & (ptx, sp\mathcal{C}_A^*, (sk_A, n_A)) & & \\
 & \quad \leftarrow \mathsf{spendCoins}([sp\mathcal{C}], p, t) & & \\
2: & & \xrightarrow{\quad ptx \quad} & \\
3: & (ptx', psp\mathcal{C}_A^*) & & (ptx', psp\mathcal{C}_B^*) \\
 & \quad \leftarrow \textbf{dRecvCoins}(ptx, p) & & \quad \leftarrow \textbf{dRecvCoins}() \\
4: & tx \leftarrow \mathsf{finTx}(ptx', sk_A, n_A) & & \\
5: & & \xrightarrow{\quad tx \quad} & \\
6: & \textbf{return } (tx, sp\mathcal{C}_A^*, psp\mathcal{C}_A^*) & & \textbf{return } (tx, psp\mathcal{C}_B^*)
\end{array}
$$

Figure 5.8: **dsharedOutMWTx** two-party protocol to build a new transaction with a shared output

One use case of this transaction protocol is to lock funds between two users, which can then be redeemed by both parties cooperating.

Figure fig. 5.8 shows the implementation of the protocol.

5.3.3 Shared Input Mimblewimble Transaction dsharedInpMWTx

dsharedInpMWTx is a protocol between a sender and a receiver. It builds a Mimblewimble transaction transferring value from a coin shared between the sender and receiver to a receiver again for the Extended Mimblewimble Transaction Scheme outlined in definition 5.4 As input it takes a list of partial spendable coins $[psp\mathcal{C}_A]$, a transaction value p and an optional timelock t from the sender, and the other part of the shared spendable coins $psp\mathcal{C}_B$ as well as the same transaction value p from the receiver. It outputs a final transaction tx to both parties, as well as the new outputs $sp\mathcal{C}_A^*, sp\mathcal{C}_B^*$ to the respective owner.

$$\langle(tx, sp\mathcal{C}_A^*), (tx, sp\mathcal{C}_B^*)\rangle \leftarrow \textbf{dsharedInpMWTx}\langle([psp\mathcal{C}_A], p, t), ([psp\mathcal{C}_B], p)\rangle$$

The protocol can be used to redeem funds which are locked created with the **dsharedInpMWTx** protocol.

Figure fig. 5.9 shows the implementation of the protocol.

dsharedInpMWTx$\langle([psp\mathcal{C}_A], p, t), ([psp\mathcal{C}_B], p)\rangle$

	Alice		*Bob*
1 :	$(ptx, sp\mathcal{C}_A^*, (sk_A, n_A))$ $\leftarrow$ **dSpendCoins**$([psp\mathcal{C}_A], p, t)$		$(ptx, (sk_B, n_B))$ $\leftarrow$ **dSpendCoins**$([psp\mathcal{C}_B], p, t)$
2 :			$(ptx', sp\mathcal{C}_B^*) \leftarrow \mathsf{recvCoins}(ptx, p)$
3 :		$\xleftarrow{ptx'}$	
4 :	$tx \leftarrow$ **dFinTx**(ptx', sk_A, n_A)		$tx \leftarrow$ **dFinTx**(ptx', sk_B, n_B)
5 :	**return** $(tx, sp\mathcal{C}_A^*)$		**return** $(tx, sp\mathcal{C}_B^*)$

Figure 5.9: **dsharedOutMWTx** two-party protocol to build a new transaction from a shared output

5.3.4 Contract Mimblewimble Transaction - dcontractMWTx

dcontractMWTx is a protocol between a sender and a receiver for the Contract Mimblewimble Transaction Scheme defined in definition 5.6. Similar to the **dsharedInpMWTx** it spends an input coin which is shared between the sender and receiver. Additionally, we utilize the adapted signature protocol from definition 4.2 to let the receiver hide a secret witness value x in the transaction signature which the sender can extract from the final transaction, thereby allowing the construction of primitive contracts.

$$\langle(tx, sp\mathcal{C}_A^*, x), (tx, sp\mathcal{C}_B^*)\rangle \leftarrow \textbf{dcontractMWTx}\langle([psp\mathcal{C}_A], p, t, X)([psp\mathcal{C}_B], p, x)\rangle$$

Figure fig. 5.10 shows the implementation of the protocol.

A note on rogue-key attacks: In section 4.1 we mentioned that in a Two-Party Signature Scheme we need to take special care in the key generation phase not to be vulnerable against rogue-key attacks in which one of the parties public key is computed as a function of the other. We see that in all of the protocols layed out in this section we do not take this into account, as for the receiving party it will always be possible to generate his keypair as a function of the senders public key. We now show how attempting a rogue-key attack in Mimblewimble would play out and why it would not threaten the security of our scheme:
Imagine we have an attacker $\mathcal{A}$ who knows the value v of some coin $\mathcal{C} = g^r \cdot h^v$ that is present in the unspent output list of the blockchain. He could then compute $pk_A = \mathcal{C} \cdot (h^v)^{-1}$. For the rogue-key attack to succeed $\mathcal{A}$ would now create a transaction spending $\mathcal{C}$ and choose his output coin pubkey as $pk_B = pk_A^{-1}$ with the attempt of cancelling out Alice's key. However, recalling the structure of Mimblewimble transactions the participants sign the Excess value $\mathcal{E} = inp - out$, where inp and out is the list of

dcontractMWTx$\langle([psp\mathcal{C}_A], p, t, X)([psp\mathcal{C}_B], p, x)\rangle$

Alice		*Bob*
$(ptx, sp\mathcal{C}^*_A, (sk_A, n_A)) \leftarrow$ **dSpendCoins**$([psp\mathcal{C}_A], p, t)$		$(ptx, (sk_B, n_B)) \leftarrow$ **dSpendCoins**$([psp\mathcal{C}_B], p, t)$
		$(ptx', sp\mathcal{C}^*_B, \tilde{\sigma_B}) \leftarrow \mathsf{aptRecvCoins}(ptx, p, x)$
	$\xleftarrow{ptx', X'}$	
if $X \neq \bot \wedge X \neq X'$		
return $\bot$		
$\hat{\sigma_B} \leftarrow ptx'.\sigma$		
$\tilde{\sigma_{AB}} \leftarrow$ **dAptFinTx**(ptx', sk_A, n_A, X)		$tx \leftarrow$ **dAptFinTx**$(ptx', sk_B, n_B, \tilde{\sigma_B})$
	$\xleftarrow{tx}$	
$x \leftarrow \mathsf{extWit}(tx.\sigma, \tilde{\sigma_{AB}}, \hat{\sigma_B})$		
return $(tx, sp\mathcal{C}^*_A, x)$		**return** $(tx, sp\mathcal{C}^*_B)$

Figure 5.10: **dcontractMWTx** two-party protocol to build a primitive contract transaction

input and output coins. Therefore to make the public keys cancel out $\mathcal{A}$ would instead have to choose his key as $pk_B := pk_A$. Given this setup (a transaction which spends the coin $\mathcal{C} = pk_A \cdot h^v$ to $\mathcal{C}^* = pk_B \cdot h^v$), the Excess value $\mathcal{E}$ would calculate like $pk_A \cdot {pk_B}^{-1}$ which by the definition of pk_B is $pk_A \cdot {pk_A}^{-1}$ which would cancel out and allow the adversary to forge a signature. However, since we chose pk_B as simply pk_A and $pk_A = g^r$ (from the original petersen commitment) the new output coin $\mathcal{C}^*$ would in fact be identical to the input coin $\mathcal{C}$ and the transaction would therefore simply spend a coin to itself. Recalling the instantiation of the transaction verification algorithm verfTx defined by Fuchsbauer et al. [7] which we layed out in fig. 5.1 we see that the union between input and output coin list must be empty, otherwise the transaction will not verify. Therefore, even though the attacker could create a forged signature for this transaction, it would still be invalid as by the definition of the transaction verification algorithm. We further consider the case in which the attacker would try to add a fee f to the transaction, to effectivly steal value from a coin. In this case the newly created output coin would be $pk_B \cdot h^{v-f}$. Now the output coin is no longer identical to the input coin, yet the input and output values still cancel out due to the fee and by the definition of pk_B the two public keys must as well still cancel out allowing for a forged signature. In this scenario $\mathcal{A}$ is however faced with the problem that he does not have a valid range proof for this new output coin. To compute such a proof he would need to know the original r of $pk_A = g^r$, which he doesn't, therefore it is again impossible for him the create a valid transaction, even though he would be able to forge the transaction signature. We conclude that all possible rogue-key attacks on Mimblewimble are prevented by means of transaction verification and we therefore do not have to take further special care to prevent them.

5.4 Security & Correctness

In this section we will prove the correctness and security of the instantiation described in section 5.2. We start by proving *Transaction Scheme Correctness*, *Extended Transaction Scheme Correctness* and *Adapted Transaction Scheme Correctness* for the three outlined transaction schemes MW, MW_{ext} and MW_{apt}. We then continue by showing that all protocols described in section 5.3 are secure in the malicious models as defined in definition 3.8.

5.4.1 Correctness

We will argue *Transaction Scheme Correctness* follows from the correctness of the commitment scheme COM, two-party signature scheme Φ as well as the correctness of the range proof system Π_{RP} used in the transaction protocol. If the transaction was constructed correctly (that is by calling the procedures $\mathsf{spendCoins}, \mathsf{recvCoins}, \mathsf{finTx}$, the distributed variants **dSpendCoins**, **dRecvCoins**, **dFinTx** or the adapted ones $\mathsf{aptRecvCoins}$, **dAptFinTx** with valid inputs) it must follow that the final transaction has correct commitments, rangeproofs and a valid signature and verfTx will therefore return 1. We construct the following theorem:

Theorem 3. *Transaction Scheme Correctness*, *Extended Transaction Scheme Correctness* or *Adapted Transaction Scheme Correctness* for a transaction system $MW[COM, \Phi, \Pi_{RP}]$, $MW_{ext}[COM, \Phi, \Pi_{RP}]$ or $MW_{apt}[COM, \Phi, \Pi_{RP}]$ holds if the underlying Commitment Scheme COM, Two-Party Signature Scheme Φ_{MP} and range proof system Π_{RP} are correct.

Proof. We assume there are two honest participants Alice and Bob, there exists a list of input coins $[\mathcal{C}_{inp}]$ with blinding factors $[r_i]$ and values $[v_i]$ wrapped inside a list $[sp\mathcal{C}]$ known to Alice, and some amount p which Alice wants to transfer to Bob. For *Transaction Scheme Correctness* to hold $\mathsf{verfTx}(tx)$ must return 1 with overwhealming probability for the two parties creating the transaction tx in the following three steps:

1. $(ptx, (sk_A, n_A)) \leftarrow \mathsf{spendCoins}([sp\mathcal{C}], p, \perp)$
2. $ptx^* \leftarrow \mathsf{recvCoins}(ptx, p)$
3. $tx \leftarrow \mathsf{finTx}(ptx^*, sk_A, n_A)$

We recall the conditions for $\mathsf{verfTx}(tx)$ to return 1 found in definition 5.2 and show that each of them must hold:

Condition 1 and 2 both must hold if the participants are honest that is compute their coins as given by protocol definition and provide input parameters that are valid. In the case that the sending party provides duplicate inputs the check at the beginning of the $\mathsf{spendCoins}$ procedure will fail and return $\perp$ and thereby halting the protocol. The blinding factors to the output coins created in $\mathsf{spendCoins}$ and $\mathsf{recvCoins}$ are generated randomly, which means a duplication can only appear with negligible probability.

Condition 3 follows from the implementation of the $\mathsf{createCoin}$ function called in $\mathsf{spendCoins}$ as well as $\mathsf{recvCoins}$. In the function a range proof is computed for the new coin $\mathcal{C}$ with value v and blinding factor r as $\pi \leftarrow \mathsf{ranPrf}(\mathcal{C}, v, r)$. Given that our range proof system Π_{RP} system has to be correct $\mathsf{vrfRanPrf}(\pi, \mathcal{C}) = 1$ must hold for all coins created with the $\mathsf{createCoin}$ routine. Therefore Condition 3 must hold if the transaction is computed honestly.

For condition 4 we must look at how the secret keys sk_A and sk_B are constructed. From the instantiation of $\mathsf{spendCoins}$ we can see that Alice's share will be $sk_A := r_A^* - \sum_{i=0}^{n}[r_A]$, where r_A^* is the blinding factor to her output and $[r_A]$ are the blinding factors to her input coins. Bobs secret key is constructed like $sk_B := r_B^*$, so it corresponds to the blinding factor of his output. From the construction of the two-party signature scheme in definition 4.1 we know that therefore the final signature will be valid under the following public key:

$$\mathcal{E}' := g^{sk_A} \cdot g^{sk_B}$$

Given how the secret keys are constructed we arrive at:

$$\mathcal{E}' := g^{r_A^*} \cdot \sum_{i=0}^{n} [g^{-r_A}] \cdot g^{r_B}$$

If we can show that the excess value $\mathcal{E}$ computed in verfTx is the same as above, $\mathsf{verf}(m, \sigma, pk) = 1$ must hold and therefore condition 3 would be proven. We show this by a stepwise conversion of the initial equation computing $\mathcal{E}$ until we arrive at the equation for $\mathcal{E}'$:

$$\mathcal{E} = \mathcal{E}' \tag{5.1}$$

$$\sum_{i=0}^{n} out - \sum_{i=0}^{n} inp - h^f = g^{r_A^*} \cdot \sum_{i=0}^{n} [g^{-r_A}] \cdot g^{r_B} - h^f \tag{5.2}$$

$$\mathcal{C}_{out}^A \cdot \mathcal{C}_{out}^B \cdot \sum_{i=0}^{n} [(\mathcal{C}_{inp})^{-1}] \cdot h^{-f} = \tag{5.3}$$

$$(g^{r_A^*} \cdot h^{v-p}) \cdot (g^{r_B^*} \cdot h^p) \cdot \sum_{i=0}^{n} [(g^{-r_A}, h^{-v_i})] \cdot h^{-f} = \tag{5.4}$$

$$g^{r_A^*} \cdot g^{r_B^*} \cdot \sum_{i=0}^{n} g^{-r_A} = g^{r_A^*} \cdot g^{r_B^*} \cdot \sum_{i=0}^{n} g^{-r_A} \tag{5.5}$$

$$1 = 1 \tag{5.6}$$

From step 5.3 to 5.4 we replace every coin $\mathcal{C}$ by its instantiation for a pedersen commitment $\mathcal{C} = g^r \cdot h^v$.

From step 5.4 to 5.5 we rely on the fact that if Alice is honest $v = \sum v_i + f$, therefore also $(v - p) + p = \sum v_i$ must hold. From that we can infer that $h^{v-p} \cdot h^p \cdot \sum h^{-v_i} \cdot h^f$ must cancel out, otherwise the transaction would either create or burn value, which is not allowed and in which case verfTx should again return 0.

We have managed to show that condition 1-4 must hold for a valid transaction and can conclude that *Transaction Scheme Correctness* holds for $MW[COM, \Pi_{RP}, \Phi_{MP}]$.

We will now argue that the same deriviation holds for *Extended Transaction Scheme Correctness* and *Adapted Transaction Scheme Correctness*.

Condition 1-2 again follow trivially from the construction of **dSpendCoins** and **dRecvCoins** for the same reasons we have already layed out in the previous proof.

dSpendCoins, **dRecvCoins**, aptRecvCoins all rely on the same createCoin routine to create output coins, thereby condition 3 also holds for valid transactions with the same argument as for the previous proof.

In the case of *Extended Transaction Scheme Correctness* the blinding factors for the input coins $[\mathcal{C}_{inp}]$ are shared. However, we can easily reduce this case to the proof for the regular case: In **dSpendCoins** Alice and Carol construct their secret keys as follows:

$$sk_A := r_A^* - \sum_{i=0}^{n} r_A \tag{5.7}$$

$$sk_C := r_C^* - \sum_{i=0}^{n} r_C \tag{5.8}$$

sk_A and sk_C are then inputs to **dFinTx** in which a partial signature $\sigma_{\tilde{AC}}$ is calculated, by both Alice and Carol signing with their secret key. Recall the case we have proven

above, in which we have a single secret key sk_A: We can split sk_A into arbitrarily chosen shares $(sk_A)_1, (sk_A)_2$ with $sk_A = (sk_A)_1 + (sk_A)_2$. By the definition of Two-Party Signatures definition 4.1 the combined signature from $(sk_A)_1, (sk_A)_2$ will be valid under g^{sk_A}. Thereby we can treat sk_A and sk_C from $\mathsf{spendCoins}$ as arbitrary shares of a combined sk_{AC}. It follows from the additive homomorphic property of the elliptic curve that a signature valid under $g^{sk_{AC}}$ must also be valid under $g^{sk_A} \cdot g^{sk_C}$. The case of two receivers calling **dRecvCoins** is symmetric. From this we can conclude that condition 4 must also hold for the *Extended Transaction Scheme*.
Now for the *Adapted Extended Transaction Scheme* the same argument holds. The only difference in this scheme is that in **dAptFinTx** Bob (instead of Alice) will call finSig, as only he knows his unadapted partial signature $\tilde{\sigma_B}$. However, the construction of the signature remains unchanged, therefore the reduction we provided before must hold for the same reasons.

We have thereby proven that if COM, Π_{RP}, Φ_{MP} are correct and the participants behave honestly (that is by providing valid inputs and calling the respective routines in the given order) $\mathsf{verfTx}(tx)$ will return 1 for the resulting transaction tx and therefore theorem 3 holds. □

5.4.2 Security

We now want to prove security in the malicious setting as defined in definition 3.8 for the protocols defined in section 5.3. Again we show that the distributed protocols are secure in the hybrid f_{zk}^{R}-model as already explained in section 4.4.2. We start by proving security of the simple transaction protocol **dBuildMWTx**.

Hybrid functionalities: The parties have access to a trusted third party computing the zero-knowledge proof of knowledge functionalities $\mathsf{f}_{zk}^{R_1}$, $\mathsf{f}_{zk}^{R_2}$ and $\mathsf{f}_{zk}^{R_2*}$. R_1 is the relation between a secret key sk and its public key $pk = g^{sk}$ for the elliptic curve generator point g. R_2 is the relation between two secret inputs r, v and its pedersen commitment $C = g^r \cdot h^v$ for two adjacent generators g, h as defined in definition 3.7. We shorten the call by the prover to just provide $sp\mathcal{C}$ because it is a wrapper that contains the coin commitment, as well as its openings. R_2* is the same as R_2 just for a list of secrets inputs $[(r, v)]$ and its list of commitments $[C]$. Again to shorten the calls by the prover we simplify the call to $\mathsf{f}_{zk}^{R_2*}([sp\mathcal{C}])$.

Proof Idea: We extend the protocol **dBuildMWTx** instantiated in section 5.3 with the following calls to the zero-knowledge proof of knowledge functionalities as depicted in fig. 5.11.

Theorem 4. Let COM be a correct and secure pedersen commitment scheme, Π_{RP} be a correct and secure range proof system and Φ_{MP} be a secure and correct two-party signature scheme, then **dBuildMWTx** securely computes a Mimblewimble transaction transferring the value p from a sender (denoted as Alice) to a receiver (denoted as Bob) in the hybrid $\mathsf{f}_{zk}^{R_1}, \mathsf{f}_{zk}^{R_2}, \mathsf{f}_{zk}^{R_2*}$-model.

dBuildMWTx $\langle([sp\mathcal{C}], p, t), (p)\rangle$

Alice		*Bob*
$\mathsf{f}_{zk}^{R_2*}([sp\mathcal{C}])$		
$(ptx, sp\mathcal{C}_A^*, (sk_A, n_A))$		
$\leftarrow \mathsf{spendCoins}([sp\mathcal{C}], p, t)$		
$\mathsf{f}_{zk}^{R_1}((sk_A, g^{sk_A}))$		
$\mathsf{f}_{zk}^{R_1}((n_A, g^{n_A}))$		
$\mathsf{f}_{zk}^{R_2}(sp\mathcal{C}_A^*)$		
	$\xrightarrow{\quad ptx \quad}$	
		if $\mathsf{f}_{zk}^{R_2*}(tx.inp) = 0$
		return $\perp$
		if $\mathsf{f}_{zk}^{R_1}(tx.\Lambda.pk) = 0$
		return $\perp$
		if $\mathsf{f}_{zk}^{R_1}(tx.\Lambda.R) = 0$
		return $\perp$
		if $\mathsf{f}_{zk}^{R_2}(tx.out[0]) = 0$
		return $\perp$
		$(ptx', sp\mathcal{C}_B^*) \leftarrow \mathsf{recvCoins}(ptx, p)$
		$\mathsf{f}_{zk}^{R_2}(sp\mathcal{C}_B^*)$
	$\xleftarrow{\quad ptx' \quad}$	
if $\mathsf{f}_{zk}^{R_2}(tx.out[1]) = 0$		
return $\perp$		
$tx \leftarrow \mathsf{finTx}(ptx', sk_A, n_A)$		
	$\xrightarrow{\quad tx \quad}$	
return $(tx, sp\mathcal{C}_A^*)$		**return** $(tx, sp\mathcal{C}_B^*)$

We need to decide what version of the protocols you want to put finally in the thesis. So far you are writing 2 versions: (i) the version as it is in Grin; and (ii) the version that you are able to prove secure (i.e., adding the calls to the zkp system). So, one question that you might get is: what is the "good" version, that is, if I create another implementation of Mimblewimble, which one should I take? And another question could be: the fact that you have 2 versions means that the current implementation is broken? Think about the answers and we should discuss them. Regarding the 2 versions, I see 2 possibilities: (a) keep both and clearly say their difference; or (b) keep only the one that you prove secure and say that it is slightly different from that of Mimblewimble.

Figure 5.11: Extension of **dBuildMWTx** (fig. 5.7) in the hybrid Model

Proof. We proof the security of **dBuildMWTx** by constructing a simulator $\mathcal{S}$ with access to a TTP computing the protocol in the ideal setting upon receiving the inputs from the participants. For this the simulator has to extract the inputs used by the adversary. The TTP returns the outputs $(tx, sp\mathcal{C}^*_A)$ (resp. $(tx, sp\mathcal{C}^*_B)$) from which he has to construct a transcript that is indistinguishable from the protocol transcript in the real world. The simulator uses the calls to $\mathsf{f}^{R_1}_{zk}, \mathsf{f}^{R_2}_{zk}, \mathsf{f}^{R_2*}_{zk}$ to achieve this. We proof that the transcript is indistinguishable in the cases that either Alice or Bob is corrupt and controlled by a deterministic polynomial adversary $\mathcal{A}$.

Alice is corrupt: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ and once it calls $\mathsf{f}^{R_2*}_{zk}, \mathsf{f}^{R_1}_{zk}\ \mathsf{f}^{R_2}_{zk}$ saves the values $[sp\mathcal{C}], sk_A, n_A, sp\mathcal{C}^*_A$ to its memory.
2. $\mathcal{S}$ calculates the transaction value p as follows:

$$v = \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.v)$$
$$p = v - sp\mathcal{C}^*_A.v$$

3. $\mathcal{S}$ receives ptx from $\mathcal{A}$ and checks for every transaction input i if $ptx.inp[i] = sp\mathcal{C}[i].\mathcal{C}$, and that $tx.out = [sp\mathcal{C}^*_A.\mathcal{C}]$. He also compares $tx.\Lambda.pk = g^{sk_A}$, $tx.\Lambda.R = g^{n_A}$, $tx.\pi[0] = sp\mathcal{C}^*_A.\pi$ and $tx.com[0] = g^{sk_A}$. If any of the equalities are invalid $\mathcal{S}$ sends abort to the TTP computing **dBuildMWTx** and returns whatever $\mathcal{A}$ returns. Otherwise he extracts $t = tx.t$ and sends the inputs $([sp\mathcal{C}], p, t)$ to the TTP and receives back the outputs $(tx, sp\mathcal{C}^*_A)$.
4. The simulators task is it now to construct ptx' which he can achieve in the following steps:
   a) He takes the signature context Λ and final signature σ_{fin} from the final transaction $\Lambda = tx.\Lambda$ and $\sigma_{\mathit{fin}} = tx.\sigma$.
   b) He computes the adversaries partial signature as $\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$
   c) He further computes

$$pk \leftarrow \Lambda.pk$$
$$pk_A = g^{sk_A}$$
$$(s_A, R_A, \Lambda) \leftarrow \tilde{\sigma_A}$$
$$(s, R) \leftarrow \sigma_{\mathit{fin}}$$
$$s_B = s - s_A$$
$$R_B = R \cdot R_A^{-1}$$
$$pk_B = pk \cdot pk_A^{-1}$$
$$\tilde{\sigma_B} = (s_B, R_B, \Lambda)$$

d) He takes further values from the final transaction:

$$\begin{aligned} \mathcal{C}_{out}^{B} &= tx.out[1] \\ \pi_B &= tx.\pi[1] \\ C_B &= tx.com[1] \end{aligned}$$

e) Now he can compute $ptx' \leftarrow \mathsf{createTx}(m, inp, out \mid\mid \mathcal{C}_{out}^{B}, \Pi \mid\mid \pi_B, \Lambda, com \mid\mid C_B, \tilde{\sigma_B})$

Finally $\mathcal{S}$ will send ptx' as if coming from Bob and sends `continue` to the TTP.

5. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_2}$ he checks equality to $\mathcal{C}_{out}^{B}$ and returns either 0 or 1.
6. Eventually $\mathcal{A}$ will send a tx' after which the simulator will output whatever $\mathcal{A}$ outputs.

Next we need to proof that the transcript produced by $\mathcal{S}$ is indistinguishable from a real one in every phase of the protocol. We separate between the following three phases: **Phase 1**: Alice sends her input coins, signing key and nonce as well as her change output coin to $\mathsf{f}_{zk}^{R_1}$ and $\mathsf{f}_{zk}^{R_2}$ and sends the pre-transaction ptx to Bob. **Phase 2**: Bob calls $\mathsf{f}_{zk}^{R_1}$ and $\mathsf{f}_{zk}^{R_2}$ as the verifier, after which he calls $\mathsf{f}_{zk}^{R_2}$ as the prover and proceeds by sending the updated pre-transaction ptx' to Alice. **Phase 3**: Alice calls $\mathsf{f}_{zk}^{R_2}$ as the verifier and sends back the final transaction tx to Bob which they then both output. Finally the output produced by $\mathcal{S}$ needs to be indistinguishable from that of $\mathcal{A}$ in a real execution.

- *Phase 1*: Due to the deterministic nature of $\mathcal{A}$ we can conclude that this phase has to be indistinguishable as there is not yet any simulation required.
- *Phase 2*: If any of the values that $\mathcal{A}$ sends to the trusted party computing the zero-knowledge proofs of knowledge are different from the value that $\mathcal{A}$ sends in the pre-transaction the equality checks done by $\mathcal{S}$ will fail in which case he will halt the simulated protocol and return whatever $\mathcal{A}$ outputs, which is what would be expected in a real execution. We further argue that the updated pre-transaction ptx' is identical to pre-transaction that would be expected in a real execution by Bob. The signatures $\tilde{\sigma_A}$ and $\tilde{\sigma_B}$ have to add up to σ_{fin} which is the final signature. $\mathcal{S}$ can read σ_{fin} from the transaction in the output he received from the TTP, he can further calculate the adversaries' signature because he knows their signing secrets. From those two values he can then compute the value that $\tilde{\sigma_B}$ must have such that it will complete to σ_{fin} when added to Alice's share of the signature. All further values $\mathcal{S}$ needs to build ptx' he can simply read from the final transaction tx. Therefore ptx' is identical to one that would be expected in a real execution.
- *Phase 3*: When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_2}$ as the verifier, $\mathcal{S}$ can simply check equality with the correct value and return 0 or 1, which is what would be expected in a real execution.

We have managed to show that in the case that Alice is corrupted the simulated transcript is indistinguishable from a transcript that would be produced in a real execution.

Bob is corrupt: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ computes one (or multiple) input coins as follows:

$$r, v \leftarrow\$ \mathbb{Z}_p^*$$
$$sp\mathcal{C} \leftarrow \mathsf{createCoin}(r, v)$$

He chooses p randomly and sets $t = \perp$. Now he can call $\mathsf{spendCoins}$ and get:

$$(ptx, sp\mathcal{C}_A^*, (sk_A, n_A)) \leftarrow \mathsf{spendCoins}([sp\mathcal{C}], p, t)$$

2. The simulator invokes $\mathcal{A}$ and sends ptx as if coming from Alice.

3. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_1}, \mathsf{f}_{zk}^{R_2}, \mathsf{f}_{zk}^{R_2*}$ as the verifier $\mathcal{S}$ simply checks equality with the values he sent earlier and returns either 0 or 1. The adversary proceeds by calling $\mathsf{f}_{zk}^{R_2}(sp\mathcal{C}_B^*)$, $\mathcal{S}$ saves $sp\mathcal{C}_B^*$ and extracts $p = sp\mathcal{C}_B^*.v$ He then calls the TTP computing **dBuildMWTx** with the input p and receives $(tx, sp\mathcal{C}_B^*)$.

4. Next $\mathcal{A}$ sends an updated pre-transaction ptx'. $\mathcal{S}$ verifies that the output coin added by $\mathcal{A}$ matches with $sp\mathcal{C}_B^*$, if it does not he sends abort to the TTP and outputs whatever $\mathcal{A}$ outputs. Otherwise $\mathcal{S}$ computes the following values from the signature context Λ provided in the final transaction and Λ' provided by $\mathcal{A}$:

$$pk_B = \Lambda'.pk \cdot g^{sk_A -1}$$
$$R_B = \Lambda'.R \cdot g^{n_A -1}$$
$$pk_A = \Lambda.pk \cdot pk_B^{-1}$$
$$R_A = \Lambda.R \cdot R_B^{-1}$$

5. Next the simulator rewinds to the first step of the simulation, but instead of choosing the values for the pre-transacion now he uses $tx.inp$ as the pre-transaction input values, $tx.out[0]$ as the single output value, $tx.\Pi[0]$ as the single range proof value and $tx.com[0]$ as the single value in the commitment field. Furthermore he constructs the initial signature context as given by protocol specification:

$$\Lambda := \{pk = 1, R = 1\}$$
$$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, pk_A, R_A)$$

And again sends the pre-transaction to $\mathcal{A}$ as if coming from Alice.

6. The simulator repeats the steps until step 5 where he rewinded earlier, now instead of rewinding $\mathcal{S}$ sends $\mathsf{continue}$ to the TTP and sends tx as if coming from Alice, and finally outputs whatever $\mathcal{A}$ outputs.

Again we now claim that the simulation is indistinguishable from a real execution. Note that due to the rewinding step we need to consider both the message sent before and after the rewind.

- *Phase 1:* In the first iteration the simulator constructs the input values $[sp\mathcal{C}]$ from random values and also chooses a random transaction value p. $\mathcal{S}$ constructs the pre-transaction using those chosen value rather than the real ones. We claim that the adversary cannot distinguish the chosen from the real coin commitments (except with neglible probability). If we assume that he would be able to do so, that means he could distinguish for two pedersen commitments $C_1 = g^{r_1} \cdot h^v, C_2 = g^{r_2} \cdot h^{v'}$ which one commits to v, from which follows that he could break the hiding property of perdersen commitments. Not being able to extract the coin values, the adversary has no chance of knowing if they are correct at this point. For the same reasons the pre-transaction sent by $\mathcal{S}$ after the rewind will be indistinguishable from a real one. However, as this time the pre-transaction is constructed from the real tx which $\mathcal{S}$ received from the TTP, the pre-transaction is in fact identical to a pre-transaction that would be expected in a real execution. The calls to $\mathsf{f}_{zk}^{R_1}, \mathsf{f}_{zk}^{R_2}$ and $\mathsf{f}_{zk}^{R_2*}$ also behave identically as what would be expected in a real execution.
- *Phase 2:* This phase will be identical to the real execution due to the fact that the adversary is deterministic.
- *Phase 3:* The transaction sent to $\mathcal{A}$ in this phase is the one received from the TTP and is therefore identical to what would have been sent in a real execution, given $\mathcal{A}$ sends correct values. (Otherwise the execution would have halted). We like to emphasize that in the case that we wouldn't have done the rewind step, $\mathcal{A}$ would be able to distinguish the transcript from the real one because he can identify differences in the inputs, outputs, proofs and commitment, as well as the signature context of the final transaction tx and the pre-transaction ptx sent in the first phase. For instance inputs which are spent in the final transaction are not present in the pre-transaction. However, due to the rewinding step $\mathcal{S}$ manages to construct the correct pre-transaction which will finalize into tx such that $\mathcal{A}$ again has no chance of distinguishing the two transcripts.
- Regarding protocol outputs if the the adversary misbehaves at any point by sending invalid (or no) values, the simulator will notice, halt the protocol and output whatever $\mathcal{A}$ outputs. If $\mathcal{A}$ behaves honestly instead $\mathcal{S}$ would run the protcol simulation until the end and then again output whatever $\mathcal{A}$ outputs. In both cases the output would be the same as would be expected from $\mathcal{A}$ in a real execution.

We have manged to show that the transcripts produced by $\mathcal{S}$ in the case that Alice and in the case that Bob is corrupt are indistinguishable from the transcript of a real execution and can therefore conclude that the protocol is secure and theorem 4 holds.

$\square$

Before we can continue to proof the security of the three other protocols **dsharedInpMWTx**, **dsharedOutMWTx**, **dcontractMWTx** we first have to proof that all the protocols which are run as part of those executions are secure too. That is we have to show security for **dSpendCoins**, **dRecvCoins**, **dFinTx**, **dAptFinTx**.

We start with the proof for **dSpendCoins** which is called inside **dsharedInpMWTx** as well as **dcontractMWTx**.

Hybrid functionalities: For this proof we need to extend our hybrid model. As previously the parties have access to a trusted third party computing the zero-knowledge proof of knowledge functionalities $\mathsf{f}_{zk}^{R_1}$,$\mathsf{f}_{zk}^{R_2}$ and $\mathsf{f}_{zk}^{R_2*}$. Additionally we introduce $\mathsf{f}_{zk}^{R_3}$, whereas R_3 is the relation between a value v, two secrets r_A, r_C and the commitment $\mathcal{C} = h^v \cdot g^{r_A} \cdot g^{r_C}$. This means that for R_3 we have two provers, one of them having to provide r_A, the other r_C. Both will have to provide the commitment $\mathcal{C}$ and the value v. Both parties can then call the protocol again as the verifier providing the commitment $\mathcal{C}^*$ and receiving 1 if $\mathcal{C}^* = \mathcal{C}_A = \mathcal{C}_C$ (whereas $\mathcal{C}_A$ is the commitment received from Bob as the prover, resp. for Carol) $v_A = v_C$ and $\mathcal{C}^* = h^{v.} \cdot g^{r_A} \cdot g^{r_C}$. A proof system that would support such a relation is for instances SNARKS as can be seen in [?]. To simplify the call made by the prover we just write $\mathsf{f}_{zk}^{R_3}(psp\mathcal{C})$ as $psp\mathcal{C}$ is, just like $sp\mathcal{C}$, a wrapper around $\mathcal{C}$, r, v. As for R_2 we again allow to call the protocol with an array of inputs by calling $\mathsf{f}_{zk}^{R_3*}$

Proof Idea: We extend the protocol **dSpendCoins** instantiated in section 5.2 with the following calls to the zero-knowledge proof of knowledge functionalities as can be seen in fig. 5.12.

Theorem 5. Let COM be a correct and secure pedersen commitment scheme, Π_{RP} be a correct and secure range proof system and Φ_{MP} be a secure and correct two-party signature scheme, then **dSpendCoins** securely computes a Mimblewimble pre-transaction ptx' spending a coin $\mathcal{C}_{out}^{sh}$ owned by the two parties in the hybrid $\mathsf{f}_{zk}^{R_1}$,$\mathsf{f}_{zk}^{R_2}$,$\mathsf{f}_{zk}^{R_3}$-model.

Proof. The proof strategy is the same as already mentioned in section 5.4.2.

Alice is corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ and saves $[psp\mathcal{C}_A]$, $sp\mathcal{C}_A^*$, sk_A, n_A when he calls $\mathsf{f}_{zk}^{R_{1,2,3}}$
2. The simulator then receives ptx from $\mathcal{A}$ and compares the input coins, output coin and proof, signature context value with what he has stored in the first step. If any of those are not equal $\mathcal{S}$ sends abort to the TTP and outputs $\perp$. Otherwise he extracts $p := \sum[psp\mathcal{C}_A.v] - sp\mathcal{C}_A^*.v$ as well as $t := ptx.t$ and sends the inputs $([psp\mathcal{C}_A], p, t)$ to the TTP and receives the outputs $(ptx', sp\mathcal{C}_A^*, (sk_A, n_A))$.
3. $\mathcal{S}$ sends ptx' to $\mathcal{A}$ as if coming from Carol and sends $\mathsf{continue}$ to the TTP to make $\mathcal{A}$ receive the outputs in the ideal setting.

dSpendCoins$\langle([psp\mathcal{C}_A], p, t), ([psp\mathcal{C}_C], p)\rangle$

Alice		*Carol*
$\mathsf{f}_{zk}^{R_{3*}}([psp\mathcal{C}_A])$		$\mathsf{f}_{zk}^{R_{3*}}([psp\mathcal{C}_C])$
$\cdots$		
$\mathsf{f}_{zk}^{R_2}(sp\mathcal{C}_A^*)$		
$\mathsf{f}_{zk}^{R_1}((sk_A, g^{sk_A}))$		
$\mathsf{f}_{zk}^{R_1}((n_A, g^{n_A}))$		
	$\xrightarrow{\quad ptx \quad}$	
		if $\mathsf{f}_{zk}^{R_{3*}}(ptx.inp) = 0$
		return $\perp$
		$\mathsf{f}_{zk}^{R_2}(ptx.out[0]) = 0$
		return $\perp$
		if $\mathsf{f}_{zk}^{R_1}(\Lambda.pk) = 0$
		return $\perp$
		if $\mathsf{f}_{zk}^{R_1}(\Lambda.R) = 0$
		return $\perp$
		$\cdots$
		$\mathsf{f}_{zk}^{R_2}(sp\mathcal{C}_C^*)$
		$\mathsf{f}_{zk}^{R_1}((sk_C, g^{sk_C}))$
		$\mathsf{f}_{zk}^{R_1}((n_C, g^{n_C}))$
	$\xleftarrow{\quad ptx' \quad}$	
if $\mathsf{f}_{zk}^{R_{3*}}(ptx'.inp) = 0$		
return $\perp$		
if $\mathsf{f}_{zk}^{R_2}(ptx'.out[1]) = 0$		
return $\perp$		
$\{pk, R\} \leftarrow ptx'.\Lambda$		
if $\mathsf{f}_{zk}^{R_1}(pk \cdot pk_A^{-1}) = 0$		
return $\perp$		
if $\mathsf{f}_{zk}^{R_1}(R \cdot R_A^{-1}) = 0$		
return $\perp$		
return $(ptx', sp\mathcal{C}_A^*, (sk_A, n_A))$		**return** $(ptx', sp\mathcal{C}_C^*, (sk_C, n_C))$

Figure 5.12: Extension of **dSpendCoins** (fig. 5.2) in the hybrid model

4. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_{1,2,3}}$ as the verifier he compares the values to what he has sent in ptx' and returns either 0 or 1.

5. Finally the simulator outputs whatever $\mathcal{A}$ outputs.

We separate between the following three phases: **Phase 1**: Alice sends her partially owned inputs coins, newly created output coins, as well as her signing secrets to $\mathsf{f}_{zk}^{R_{1,2,3}}$ and sends ptx. Carol sends her partiall owned input coins to f_{zk}^{R3} **Phase 2**: Carol calls $R_{1,2,3}$ as the verifier constructs her output coin and signing secrets, now calls $R_{1,2}$ as the prover and sends the updated ptx' to Alice. **Phase 3**: Alice calls $R_{1,2,3}$ as the verifier. Again the output returned by $\mathcal{S}$ must as well be indistinguishable from that of $\mathcal{A}$ in a real execution.

We now argue why each phase is indistinguishable from a real execution in the case that Alice is corrupted.

- *Phase 1*: No simulation is required in this phase, we therefore conclude that is indistinguishable from a real execution due to the deterministic nature of $\mathcal{A}$.
- *Phase 2*: If $\mathcal{A}$ tried to cheat by providing invalid values in ptx the equalities that $\mathcal{S}$ checks will fail and will lead him halting the protocol and returning which is the same as would be expected in a real execution. $\mathcal{S}$ then sends ptx' to $\mathcal{A}$ which he received from the TTP and therefore has to be identically distributed as in a real execution.
- *Phase 3*: Again if $\mathcal{A}$ tries to cheat by sending an invalid value, he will receive a 0 bit, which would also happen in the real execution.
- In the case that $\mathcal{A}$ would deviate from the protocol specification, as well as in the case that he follows it, $\mathcal{S}$ will always output whatever $\mathcal{A}$ outputs, which has to be indistinguishable from what is expected in a real execution.

As the transcript is identically distributed to a transcript of a real protocol execution we conclude that the simulation in this case is perfect.

Carol is corrupt: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ and saves $[psp\mathcal{C}_C]$ when the adversary calls $\mathsf{f}_{zk}^{R_{3*}}$

2. The simulator then chooses $r_A, r_A^*, p \leftarrow\$\, \mathbb{Z}_p^*$ and sets $psp\mathcal{C}_A := \{\mathcal{C} := psp\mathcal{C}_C.\mathcal{C}, r := r_A, v := psp\mathcal{C}_C.v\}$. He then proceeds by building ptx as given by the protocol specification with the chosen values and $[psp\mathcal{C}_A]$ and sends it to $\mathcal{A}$ as if coming from Alice.

3. When Carol calls $\mathsf{f}_{zk}^{R_{1,2,3}}$ as the verifier $\mathcal{S}$ checks the passed values for equality and returns either 0 or 1. As soon as Carol calls $\mathsf{f}_{zk}^{R_2}(sp\mathcal{C}_C)$ $\mathcal{S}$ will extract $psp\mathcal{C}_C^*.v$ and finally call the TTP with inputs $([psp\mathcal{C}_C], p)$ to receive ptx', $sp\mathcal{C}_C^*$, (sk_C, n_C).

4. Now the simulator rewinds to step 1 and constructs the actual ptx from ptx' as follows:

$$\begin{aligned} \{m, inp, out, \Pi, \Lambda, com, \emptyset, t\} \;&\leftarrow\; ptx' \\ pk_A \;&:=\; ptx'.\Lambda.pk \;\cdot\; g^{{sk_C}^{-1}} \\ R_A \;&:=\; ptx'.\Lambda.R \;\cdot\; g^{{n_C}^{-1}} \\ \Lambda^* \;&:=\; \{pk \;:=\; pk_A, R \;:=\; R_A\} \\ ptx \;&:=\; \mathsf{createTx}(m, inp, out[0], \Pi[0], \Lambda^*, com[0], \emptyset) \end{aligned}$$

   he then sends again ptx Carol and continues as before

5. When $\mathcal{A}$ sends ptx' he compares its inputs, outputs, proofs and signature context to ptx' received from the trusted third party and outputs $\perp$ and sends abort to the TTP if any do not match. Otherwise he sends $\mathsf{continue}$ to the TTP and outputs whatever $\mathcal{A}$ outputs.

We again show that in each phase the transcript produced by the simulator is computationally indistinguishable from a real transcript.

- *Phase 1*: In the first iteration (before the rewind) the pre-transaction that is send to $\mathcal{A}$ will be constructed from randomly chosen values except for the transaction inputs which are given by the commitments in $[psp\mathcal{C}_C]$. Due to the hiding property of the pedersen commitment the adversary cannot determine if the correct value p has been used to construct the output coin, even though he in fact knows the correct value for p, but does not know the blinding factor r_A^*. $\mathcal{A}$ does know the correct values for the input coins from $[psp\mathcal{C}_C]$ thereby it is critical that $\mathcal{S}$ uses the commitments extracted from $[psp\mathcal{C}_C]$ to build the transaction. Otherwise the simulation could be detected. In the second iteration (after the rewind) $\mathcal{S}$ sends the same ptx which would be sent in a real execution which is therefore identical.
- *Phase 2*: When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_{1,2,3}}$ he will receive 0 or 1 identically to the real execution.
- *Phase 3*: If $\mathcal{A}$ sends invalid input, output, proof or context values is the final pre-transaction ptx' the simulator detects this and ouputs $\perp$, otherwise the protocol concludes, which is the same that would happen in the real exeuction.

We have managed to show that the simulator $\mathcal{S}$ can produce an indistinguishable transcript both in the case that Alice and that Carol is corrupted and can thereby conclude that **dSpendCoins** is secure in the $\mathsf{f}_{zk}^{R_1}$,$\mathsf{f}_{zk}^{R_2}$,$\mathsf{f}_{zk}^{R_3}$-model and theorem 5 holds. $\square$

dRecvCoins$\langle (ptx, p), () \rangle$

Bob		*Carol*
$\cdots$		
$\mathsf{f}_{zk}^{R_2}((\mathcal{C}_{out}^{sh}, (p, r_B^*)))$		
	$\xrightarrow{ptx, \mathcal{C}_{out}^{sh}}$	
		if $\mathsf{f}_{zk}^{R_2}(\mathcal{C}_{out}^{sh}) = 0$
		return $\perp$
		$\cdots$
		$\mathsf{f}_{zk}^{R_1}((sk_C, g^{sk_C}))$
	$\xleftarrow{ptx', g^{sk_C}}$	
if $\mathsf{f}_{zk}^{R_1}(g^{sk_C}) = 0$		
return $\perp$		
$\cdots$		$\cdots$
$\pi_{BC} \leftarrow$ **dRanPrf**$({\mathcal{C}_{out}^{sh}}', p, sk_B)$		$\pi_{BC} \leftarrow$ **dRanPrf**$({\mathcal{C}_{out}^{sh}}', p, sk_C)$
$(\sigma \tilde{_{BC}}, pk_{BC}) \leftarrow$ **dSign**(m, sk_B, n_B)		$(\sigma \tilde{_{BC}}, pk_{BC}) \leftarrow$ **dSign**(m, sk_C, n_C)
$\cdots$		$\cdots$
return $(ptx^*, psp\mathcal{C}_B^*)$		**return** $(ptx^*, psp\mathcal{C}_C^*)$

Figure 5.13: **dRecvCoins** in the hybrid model

We continue by proofing security of the **dRecvCoins** which is called inside the **dsharedOutMWTx** protocol.

Hybrid functionalities: Again the parties have access to a trusted third party computing the zero-knowledge proof of knowledge functionalities $\mathsf{f}_{zk}^{R_1}$, $\mathsf{f}_{zk}^{R_2}$ and $\mathsf{f}_{zk}^{R_2*}$. For this proof we do not need R_3 as defined in the previous proof, however we extend the model with two further protocols which have already been proven secure. We extend our model by including the **dSign** protocol for which security has been proven in section 4.4 and the **dRanPrf** for which a secure protocol can be found in [11]

Proof idea: We extend the protocol **dRecvCoins** instantiated in section 5.2 with the following calls to the zero-knowledge proof of knowledge functionalities as outlined in fig. 5.13.

Theorem 6. Let *COM* be a correct and secure pedersen commitment scheme, Π_{RP-MP} be a correct and secure multiparty range proof system and Φ_{MP} be a secure and correct two-party signature scheme, then **dRecvCoins** securely updates a mimbewimble pre-transaction by creating a new output coin ${\mathcal{C}_{out}^{sh}}'$ for which the key is shared between two parties Bob and Carol in the $\mathsf{f}_{zk}^{R_1}$,$\mathsf{f}_{zk}^{R_2}$,**dSign**,**dRanPrf**-model.

Proof. As before we proof security by construction of a simulator $\mathcal{S}$ with access to a trusted third party (TTP) computing **dRecvCoins** in the ideal model upon receiving inputs from the two parties. The simulators task is to extract the inputs of the adversary $\mathcal{A}$, send the inputs to the TTP and construct a protocol transcript indistinguishable from a real one. We first look at the case in which Bob is corrupted and then when Carol is corrupted.

Bob is corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ and saves $(\mathcal{C}_{out}^{sh}, (p, r_B^*))$ when the adversary calls $\mathsf{f}_{zk}^{R_2}$.
2. $\mathcal{A}$ sends $(ptx, \mathcal{C}_{out}^{sh})$. The simulator then compares $\mathcal{C}_{out}^{sh}$ with the values saved in its memory and sends abort to the TTP, halts the protocol and outputs whatever $\mathcal{A}$ if they don't match. Otherwise, he sends (ptx, p) to the TTP computing $\mathsf{recvCoins}$ and receives the outputs $(ptx^*, psp\mathcal{C}_B^*)$.
3. $\mathcal{S}$ proceeds by taking the last output ${\mathcal{C}_{out}^{sh}}'$ from $ptx^*.out$ and computes $g^{sk_C} := {\mathcal{C}_{out}^{sh}}' \cdot {\mathcal{C}_{out}^{sh}}^{-1}$. The simulator computes ptx' by adding ${\mathcal{C}_{out}^{sh}}'$ to ptx and sends it together with g^{sk_C} to $\mathcal{A}$ as if coming from Carol and sends $\mathsf{continue}$ to the TTP.
4. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_1}$ as the verifier $\mathcal{S}$ check equality with the correct value and returns either 0 or 1.
5. When the adversary calls **dRanPrf** the simulator saves sk_B to its memory and returns the last element of $ptx^*.\Pi$ as received from the TTP.
6. For the call to **dSign** the simulator returns the $ptx.\sigma$ as the signature and $g^{sk_B} \cdot g^{sk_C}$ as the public key.
7. $\mathcal{S}$ concludes by outputting whatever $\mathcal{A}$ outputs.

We find the following phases: **Phase 1**: Bob calls $\mathsf{f}_{zk}^{R_2}$ and sends ptx to Carol. **Phase 2**: Carol calls $\mathsf{f}_{zk}^{R_2}$ as the verifier adds her public key to the commitment and sends back an updated pre-transaction and her public key. **Phase 3**: Bob calls $\mathsf{f}_{zk}^{R_1}$ as the verifier and the parties call the trusted third parties computing **dRanPrf** and **dSign**.

We argue that in this case the simulation is perfect, that is the transcript produced by $\mathcal{S}$ is identical to the transcript of the real execution.

- *Phase 1*: No simulation is done during this phase, and the transcript is therby indistinguishable by the deterministic nature of $\mathcal{A}$.

- *Phase 2*: In case $\mathcal{A}$ sends an invalid value for $\mathcal{C}_{out}^{sh}$ the execution will stop with output $\perp$ which is identical to what would happen in the real execution. The simulator can then send the updated pre-transaction as the honest Carol would do and the extracted real value for g^{sk_C}.
- *Phase 3*: $\mathcal{A}$ will receiver 0 or 1 to the call to $\mathsf{f}_{zk}^{R_1}$ as in the real execution. The simulator further manages to reconstruct the real output values for **dRanPrf** and **dSign** again making the transcript identical in this phase.

Carol is corrupted: The simulator works as follows:

1. Since Carol does not have any inputs in this protocol $\mathcal{S}$ can simply send $\emptyset$ to the TTP and receives $(ptx^*, sp\mathcal{C}_C^*)$ from which he extracts Carols bliding factor (and secret key) as $sk_C := sp\mathcal{C}_C^*.r$. He can now create the initial shared coin $\mathcal{C}_{out}^{sh}$ by taking the last output of $ptx^*.out$ as $\mathcal{C}_{out}^{sh\prime}$ and calculating $\mathcal{C}_{out}^{sh} := \mathcal{C}_{out}^{sh\prime} \cdot g^{sk_C{}^{-1}}$. he can further create the initial pre-transaction by removing the last entry of the output coin list, last entry of the proof list and signature from ptx^*.
2. $\mathcal{S}$ invokes $\mathcal{A}$ and send $ptx, \mathcal{C}_{out}^{sh}$ (as calculated in step 1) as if coming from Bob.
3. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_2}$ as the verifier the simulator checks for equality with what he send in the last step and returns either 0 or 1.
4. The adversary then sends the updated ptx' which the simulator validates by checking if the last entry in $ptx'.out$ equals $\mathcal{C}_{out}^{sh\prime}$. If they don't $\mathcal{S}$ will output $\perp$ and send abort to the TTP halting the execution, otherwise he will send $\mathsf{continue}$.
5. Upon the adversary calling **dRanPrf** the simulator will return the proof at the last position in the proofs array of $ptx^*.\Pi$ received from the TTP.
6. The simulator then extracts $p := psp\mathcal{C}_C^*$ and computes $pk_B := \mathcal{C}_{out}^{sh} \cdot h^{v-1}$ and returns $ptx^*.\sigma$ and $sk_C^* \cdot pk_B$ when $\mathcal{A}$ calls **dSign**.
7. The simulation completes with $\mathcal{S}$ outputting whatever $\mathcal{A}$ outputs.

We now argue why in each of the three phases the transcript produced by $\mathcal{S}$ is indistinguishable from a real transcript.

- *Phase 1*: Because $\mathcal{S}$ as able to call the TTP already in the first step he is able to receive the protocol outputs. The simulator can then extracts carols secret key sk_C from Carols $psp\mathcal{C}_C^*$ output, which must also be her blinding factor in $\mathcal{C}_{out}^{sh\prime}$. He therefore can reconstruct $\mathcal{C}_{out}^{sh}$ which must be sent by Bob in this phase, simply by subtracting Carols part from the output which is present in $ptx^*.out$. $\mathcal{S}$ is further able to reconstruct the ptx which must be sent by Bob in this phase simply by removing the values from ptx^* which get added at a later point in the protocol. The transcript is therefore identical to a real one in this phase.

- *Phase 2*: If $\mathcal{A}$ tries do cheat by sending an invalid value to $\mathsf{f}_{zk}^{R_2}$ as the verifier he will receive 0 as a response and 1 otherwise, which is identical to the real case. Similarity the execution will halt with $\perp$ if $\mathcal{A}$ sends invalid values as ptx' and g^{sk_C}, again identical to a real execution.

- *Phase 3*: $\mathcal{S}$ is able to read the output values for π_{BC} and $\sigma_{\tilde{BC}}$ from ptx^*, he further is able to calculate pk_{BC} as he knows g^{sk_C} and is further able to reconstruct pk_B from $\mathcal{C}_{out}^{sh}$. Therefore the simulation again is perfect in this phase.

Both in the case the Bob and Carol is corrupted $\mathcal{S}$ is able to produce a transcript indistinguishable from a transcript produced on a real execution we can therefore conclude that the protocol is secure in the $\mathsf{f}_{zk}^{R_1}$,$\mathsf{f}_{zk}^{R_2}$,**dSign**,**dRanPrf**-model and theorem 6 holds. $\square$

We claim that the security of the protocols **dFinTx** and **dAptFinTx** can be reduced to the security of **dSign** as all interaction between the two parties happens in the call to **dSign**. We have already proven the security of **dSign** in section 4.4 and can reuse the simulator constructed there for the protcols **dFinTx** and **dAptFinTx**.

We can now continue to proof security of the protocols found in section 5.3. We start with **dsharedOutMWTx**.

Hybrid functionalities: For this proof we again assume the access to a trusted third party computing the zero-knowledge proof of knowledge functionalities $\mathsf{f}_{zk}^{R_1}$, $\mathsf{f}_{zk}^{R_2}$ and $\mathsf{f}_{zk}^{R_3*}$, with the three relations defined as in previous proofs. We further require a trusted third party computing **dRecvCoins**, which we have already proven to be secure in the hybrid model.

We extend the protocol dsharedOutMWTx instantiated in section 5.3 with the following calls to the zero-knowledge proof of knowledge functionalities shown in fig. 5.14.

Theorem 7. Let *COM* be a correct and secure pedersen commitment scheme, Π_{RP} be a correct and secure range proof system and Φ_{MP} be a secure and correct two-party signature scheme, then **dsharedOutMWTx** securely computes a Mimblewimble transaction with a output coin ${\mathcal{C}_{out}^{sh}}'$ which spending secret is shared between Alice and Bob.

Proof. We proof security of the protocol in the malicious setting by constructing a simulator $\mathcal{S}$ with access to a trusted third party (TTP) computing **dsharedOutMWTx** in the ideal model upon receiving inputs from the two parties. The simulators task is to extract the adversaries inputs, send them to the TTP to receive the protocol outputs and construct a transcript indistinguishable from a transcript produced in a real execution. We separately look at the case in which Alice and in which Bob is corrupted.

Alice is corrupted: Simulator $\mathcal{S}$ works as follows:

dsharedOutMWTx$\langle([sp\mathcal{C}], p, t), ()\rangle$

Alice		*Bob*
$\mathsf{f}_{zk}^{R_{3*}}([sp\mathcal{C}])$		
$\cdots$		
$\mathsf{f}_{zk}^{R_2}(sp\mathcal{C}_A^*)$		
$\mathsf{f}_{zk}^{R_1}((sk_A, g^{sk_A}))$		
$\mathsf{f}_{zk}^{R_1}((n_A, g^{n_A}))$		
	$\xrightarrow{ptx}$	
		if $\mathsf{f}_{zk}^{R_{3*}}(ptx.inp) = 0$
		return $\perp$
		if $\mathsf{f}_{zk}^{R_2}(ptx.out[0]) = 0$
		return $\perp$
		if $\mathsf{f}_{zk}^{R_1}(ptx.\Lambda.pk) = 0$
		return $\perp$
		if $\mathsf{f}_{zk}^{R_1}(ptx.\Lambda.R) = 0$
		return $\perp$
$(ptx', psp\mathcal{C}_A^*) \leftarrow$ **dRecvCoins**(ptx, p)		$(ptx', psp\mathcal{C}_B^*) \leftarrow$ **dRecvCoins**$()$
$tx \leftarrow \mathsf{finTx}(ptx', sk_A, n_A)$		
	$\xrightarrow{tx}$	
return $(tx, sp\mathcal{C}_A^*, psp\mathcal{C}_A^*)$		**return** $(tx, psp\mathcal{C}_B^*)$

Figure 5.14: **dsharedOutMWTx** in the hybrid model

1. $\mathcal{S}$ invokes $\mathcal{A}$ and saves $[sp\mathcal{C}]$, sk_A, n_A and $sp\mathcal{C}_A^*$ to its memory.

2. $\mathcal{A}$ sends ptx from which $\mathcal{S}$ extracts $t := ptx.t$. He further extracts $p := \sum sp\mathcal{C}_i.v - sp\mathcal{C}_A^*.v$. $\mathcal{S}$ verifies that the values $ptx.inp$, $ptx.out$, $ptx.\pi$ and $ptx.\Lambda$ correspond to what he has saved to its memory. In case this verification failes he sends abort to the TTP and outputs $\perp$.

3. $\mathcal{S}$ sends $([sp\mathcal{C}], p, t)$ to the TTP and receives $(tx, sp\mathcal{C}_A^*, psp\mathcal{C}_A)$.

4. When $\mathcal{A}$ calls **dRecvCoins** $\mathcal{S}$ verifies that ptx and p passed by $\mathcal{A}$ are correct and only then forwards them to the TTP to receive $(ptx', psp\mathcal{C}_A^*)$ which he then returns to $\mathcal{A}$. Otherwise he returns $\perp$ to $\mathcal{A}$ and sends abort to the TTP and halts the protocol

5. $\mathcal{S}$ sends continue to TTP. Eventually $\mathcal{A}$ sends tx after which $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

It is easy to see that the simulation is perfect as every simulated message exchanged between the party is identical to what would be exchanged in a real execution. Also if the adversary cheats (by sending an invalid ptx) this is noticed by the simulator who then outputs $\perp$ which is identical as well to the real case.

Bob is corrupted: Simulator works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ and sends () to the TTP to receive the outputs (tx, $psp\mathcal{C}_B^*$)
2. $\mathcal{S}$ now has the following challenge: $\mathcal{A}$ first expects the first pre-transaction ptx coming from Alice, which should not have any signature, only one output (Alice change output) and only a partially setup signature context Λ. To achieve this $\mathcal{S}$ clones tx into ptx, removes the last output coin (and proof), and sets the signature field to $\emptyset$. The simulator can now construct the partially setup signature context as follows:

$$sk_A, n_A \leftarrow\$\, \mathbb{Z}_p^*$$
$$\Lambda' := \{pk := g^{sk_A}, g^{n_A}\}$$

   He then sets $ptx.\Lambda := \Lambda'$ and sends ptx to $\mathcal{A}$ as if coming from Alice.
3. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_{1,2,3}}$ as the verifier $\mathcal{S}$ compares the values with what he sent in step 1 in ptx and returns either 0 or 1.
4. $\mathcal{A}$ will call **dRecvCoins** upon which $\mathcal{S}$ calls the TTP computing **dRecvCoins** to receive ptx', $psp\mathcal{C}_B^*$ which $\mathcal{S}$ then returns to $\mathcal{A}$.
5. Finally $\mathcal{S}$ sends tx as if coming from Alice and outputs whatever $\mathcal{A}$ outputs.

It is easy to see that tx sent by $\mathcal{S}$ in the last step must be identical to the value from a real execution as it has been computed by the TTP. Also when $\mathcal{A}$ tries to cheat by sending invalid values to $\mathsf{f}_{zk}^{R_{1,2,3}}$ will he receive 0, as would be the case in a real execution. ptx' must be identical to a real execution as computed by the trusted third party computing **dRecvCoins**. Therefore the only thing that remains to show is that ptx constructed by the simulator is indistinguishable from a ptx exchanged in a real transcript. We note that sk_A and n_A in a real execution are uniformly distributed values in $\mathbb{Z}_p^*$. Consequently g^{sk_A} and g^{n_A} are uniformly distributed in $\mathbb{G}$. By construction of $\mathcal{S}$ this must also hold in the simulated case. Therefore the signature context constructed in step 2 for ptx must be indistinguishable from a real one, which also means that the ptx is indistinguishable, as the rest of the values are taken from tx as computed by the TTP. We must also note that even when $\mathcal{A}$ receives ptx' and tx later in the protocol, he has no way of realizing that tx was constructed by $\mathcal{S}$. This follows from the fact that the final R and pk in the final signature context of ptx' and tx is composed of three values each: $\Lambda = pk_{A1} \cdot pk_{A2} \cdot pk_B$ (similar for R). $\mathcal{A}$ only learns one of Alice's public keys (from step 2) and knows his own, but does not know anything about Alice second keypair. Therefore he has no way of learning that the final pk is not constructed correctly. The same argument holds for R.

We have shown that the simulator $\mathcal{S}$ is able to produce an indistinguishable transcrip both in the case that Alice and that Bob is corrupted and can thereby conclude that **dsharedOutMWTx** is secure in the $\mathsf{f}_{zk}^{R_1}$,$\mathsf{f}_{zk}^{R_2}$,$\mathsf{f}_{zk}^{R_3}$,**dRecvCoins**-model and consequently theorem 7 holds. □

Next we proof security for **dsharedInpMWTx**.

Hybrid functionalities: For this proof it is enough to give the parties access to a trusted third party computing the **dSpendCoins** and the **dFinTx** protocol. Further calls to a zero-knowledge proof of knowledge functionality are not needed. This means that we do not have to extend to original protocol instantiation any further.

Theorem 8. Let COM be a correct and secure pedersen commitment scheme, Π_{RP-MP} a correct and secure multiparty range proof system and Φ_{MP} be a secure and correct two-party signature scheme, then **dRecvCoins** securely computes a mimewimble transaction spending an input coin $\mathcal{C}_{out}^{sh}$ shared between Alice and Bob in the hybrid **dSpendCoins**, **dFinTx**-model

Proof. We proof security by construction of simulator $\mathcal{S}$ with access to a trusted third party (TTP) computing **dsharedInpMWTx** in the ideal model upon receiving inputs from the two parties, as well as a trusted third party computing the functionality of the hybrid model. The simulators task is to extract the adversaries inputs, send them to the TTP and construct a protocol transcript indistinguishable from a real one. We again look separately at the case in in which Alice, and in which Bob is corrupted.

Alice is corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ and saves $[psp\mathcal{C}_A]$, p and t when $\mathcal{A}$ calls **dSpendCoins**.
2. He then forwards those values as the inputs to the TTP computing **dSpendCoins** and receives $(ptx,\ sp\mathcal{C}_A^*, (sk_A, n_A))$ which he returns to $\mathcal{A}$. He proceeds by sending the inputs $([psp\mathcal{C}_A], p, t)$ to the TTP computing **dsharedInpMWTx** and receives $(tx, sp\mathcal{C}_B^*)$.
3. The simulator now has the challenge to construct a ptx' which is partially signed. The final signature is composed of A + B1 + B2, where B2 is the signature from Bobs output coins and A + B1 are the signatures from the shared input coin. ptx' has to contain the partial signature B2, such that the partial signature verification algorithm verifies and such that when combined with the signatures A and B1 it will complete into the final signature $tx.\sigma$. Therefore the only way for the simulator to create a valid simulation is to calculate the actual value for

the B2 signature, which is challenging since he does not know sk_B, n_B. However, he knows the final siganture $\sigma_{fin} := tx.\sigma$ and he can create the signature A as $\tilde{\sigma_A} \leftarrow \mathsf{signPt}(tx.m, sk_A, n_A, tx.\Lambda)$. $\mathcal{S}$ is able to recompute the value for the B2 signature as follows:

a) $\mathcal{S}$ choses $(sk_B', n_B') \leftarrow\$ \mathbb{Z}_p^*$

b) He then computes a temporary $\tilde{\sigma_B}' \leftarrow \mathsf{signPt}(tx.m, sk_B', n_B', tx.\Lambda)$

c) He clones tx into ptx' and sets $ptx'.\sigma := \tilde{\sigma_B}'$

d) Now the simulator calls the TTP computing **dFinTx** with the inputs ptx', sk_A, n_A to receive tx'

e) Note that the signature in tx' now contains a signature composed of A + B1 + B2', where B2' is the partial signature computed in step b. Therefore now it is possible to recompute the value of the partial signature for B1 as follows:

$$
\begin{aligned}
(s', R') &\leftarrow tx' \\
(s_A, R_A, \Lambda) &\leftarrow \tilde{\sigma_A} \\
(s_B', R_B', \Lambda) &\leftarrow \tilde{\sigma_B}' \\
s_{B1} &:= s' - s_A - s_B' \\
R_{B1} &:= R' \cdot R_A{}^{-1} \cdot R_B'^{-1} \\
\tilde{\sigma_{B1}} &:= \{s_{B1}, R_{B1}, \Lambda\}
\end{aligned}
$$

f) $\mathcal{S}$ now has the correct values for the signatures A and B1 and can therefore recompute the correct value for the partial signature B2 from $tx.\sigma$ with the same calculation as shown in the previous step

4. $\mathcal{S}$ is now able to construct ptx' by again cloning tx and setting $ptx'.\sigma := \tilde{\sigma_{B2}}$. The simulator will rewind the call to the TTP computing **dFinTx** and send ptx' to $\mathcal{A}$ as if coming from Bob.

5. When $\mathcal{A}$ calls **dFinTx** $\mathcal{S}$ will forward the inputs to the TTP party computing **dFinTx**, return the TTP outputs back to $\mathcal{A}$ and finally output whatever $\mathcal{A}$ outputs.

As only ptx' is constructed by $\mathcal{S}$, it is the only value for which we have to prove indistinguishability. We have already shown that the final signature in tx is composed of three parts A, B1 and B2. Through the calculations layed out the simulator is able to recompute the real value of B2, which must make ptx' identical to the one send by Bob in the real execution.

Bob is corrupted: Simulation in this case is trivial, as there is no message sent from Alice to Bob and $\mathcal{S}$ doesn't need to extract any inputs. A perfect simulation is therefore achieved simply by forwarding the inputs sent by $\mathcal{A}$ to the TTP computing **dSpendCoins** and **dFinTx** and finally outputting whatever $\mathcal{A}$ outputs.

We have managed to construct a simulator in the case the Alice as well as that Bob is corrupted which produced a protocol transcript indistinguishable from a real one and can therefore conclude that **dsharedInpMWTx** is secure in the **dSpendCoins**,**dFinTx**-model and theorem 8 must hold. □

We now move to the final proof, proving security of **dcontractMWTx**:

Hybrid functionalities: We proof the security of **dcontractMWTx** in the hybrid model in which the participants have access to a trusted third party computing **dSpendCoins** and **dAptFinTx**. We also again require access to a trusted third party computing the zero-knowledge proof of knowledge functionality $\mathsf{f}_{zk}^{R_1}$, with R_1 being defined equally in previous proofs.

Proof idea: We extend the original **dcontractMWTx** with a single call to $\mathsf{f}_{zk}^{R_1}$ from each Alice and Bob. On Bobs side we extend the protocol with the following call at the beginning of the protocol: $\mathsf{f}_{zk}^{R_1}((x, g^x))$. On Alice side we add the following verification at line 2 of the protocol: **If** $\mathsf{f}_{zk}^{R_1}(X) = 0$ **return** $\perp$.

Theorem 9. Let *COM* be a correct and secure pedersen commitment scheme, Π_{RP-MP} be a correct and secure multiparty range proof system and Φ_{MP} be a secure and correct two-party signature scheme, then **dcontractMWTx** securely computes a mimlewimble transaction transferring value from a shared input coin $\mathcal{C}_{out}^{sh}$ to Bob, while at the same time revealing a secret witness value x to Alice for which she knows the X for which $X = g^x$.

Proof. We proof security by constructing a simulator $\mathcal{S}$ with access to a trusted third party (TTP) computing **dcontractMWTx** in the ideal setting upon receiving inputs from the two parties. The simulators task is to extract the adversaries inputs, send them to the TTP and construct a transcript indistinguishable from a transcript produced during a real protocol execution. We separately look at the case in which Alice and in which Bob is corrupted.

Alice is corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ and saves the inputs $[psp\mathcal{C}_A]$, p and t when the adversary calls **dSpendCoins**.

2. He forwards the inputs received in step 1 to the TTP computing **dSpendCoins** to receive the outputs $(ptx,\ sp\mathcal{C}_A^*,\ (sk_A, pk_A))$, which he then forwards to $\mathcal{A}$ as the protocol results.

3. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_1}$ as the verifier $\mathcal{S}$ saves X to his memory and sends the inputs $([psp\mathcal{C}_A],\ p,\ t,\ X)$ to the TTP computing **dcontractMWTx** to receive the outputs $(tx,\ sp\mathcal{C}_A^*,\ x)$.

4. As in the previous proof the simulator now has the task to construct a pre-transaction ptx' with a partial signature B2 of A, B1, B2. The simulator can compute $\tilde{\sigma_B}$ in the same way as we layed out in the previous proof. Note however that in this case $\mathcal{A}$ expects an adapted signature $\hat{\sigma_B}$ which will verify with the adapted partial signature verification routine passing X. $\mathcal{S}$ can easily calculate the adapted signature by computing $\hat{\sigma_B} \leftarrow \mathsf{adaptSig}(\tilde{\sigma_B}, x)$ and constructing ptx' by cloning tx and setting the signature field to $\hat{\sigma_B}$. Finally $\mathcal{S}$ sends (ptx', X) to $\mathcal{A}$ as if coming from Bob.

5. When $\mathcal{A}$ calls **dAptFinTx** $\mathcal{S}$ forwards the inputs to the TTP computing **dAptFinTx** and returns the TTP outputs to $\mathcal{A}$. If the output returned to the adversary was not $\perp$ the simulator will send tx to $\mathcal{A}$ as if coming from Bob, send continue to the TTP computing **dcontractMWTx** and output whatever $\mathcal{A}$ outputs.

In this proof only ptx' and X sent in the first message from Bob to Alice is constructed by $\mathcal{S}$. All other values are direclty forwarded from a trusted third party and must therefore trivially be indistinguishable from the real execution. As $\mathcal{S}$ knows x, constructing the real value of X is simply calculating g^x. That ptx' is identical to the value sent in a real execution must hold for the same reasons as outlined in the previous proof and by the fact that $\mathcal{S}$ knows x and can therefore call adaptSig as it is called in a real execution by Bob.

Bob is corrupted: Again finding a perfect simulator is trivial in this case as there are no messages send directly from Alice to Bob and $\mathcal{S}$ doesn't need to extract any inputs. Whenever $\mathcal{A}$ calls one of the trusted third parties to compute a hybrid functionality $\mathcal{S}$ externally forwards the call to the TTP and returns whatever was the result to $\mathcal{A}$.

We have managed to construct a simulator producing a transcript indistinguishable from a real one both in the case that Alice and that Bob is corrupted and controlled by an adversary $\mathcal{A}$ and can therefore conclude that **dcontractMWTx** is secure in the **dSpendCoins**, **dAptFinTx**, $\mathsf{f}_{zk}^{R_1}$-model and theorem 9 must hold. □

5.5 Atomic Swap protocol

With the outlined Adapted Mimblewimble Transaction Scheme from definition 5.6 and protocols from 5.3 we can now construct an Atomic Swap protocol with another Cryptocurrency. In this thesis we will explain a swap with Bitcoin, as at present Bitcoin and Bitcoin-like cryptocurrencies are the most widely adopted. We will generally refer to the "Bitcoin side" and the "Mimblewimble side" of the swap to be most generic. Upon implementation one has to decide for a specific implementation, for example BTC on the Bitcoin side and Grin on the Mimblewimble side.

not sure I would claim that Bitcoin-like cryptocurrencies are also widely adopted

Here, I would write sth like: To abstract away from the details of the different Bitcoin implementations, we define here the minimal DPT functions that we require for our atomic swap. These functionalities are inherent to the Bitcoin functionality and thus supported in all implementations.

On the Bitcoin side we define three DPT functions ($\mathsf{lockBtcScript}, \mathsf{verifyLock}, \mathsf{spendBtc}$).

- $(spk) \leftarrow \mathsf{lockBtcScript}(pk_A, pk_B, X, t)$: The locking script function lets Bob construct a Bitcoin script only spendable by Alice if she receives the discrete logarithm x of X with $X = g^x$. Additionally, the function requires Bobs public key pk_B and a timelock t (given as a block number) as input which allows Bob to reclaim his funds after some time if the atomic swap was not completed successfully. The function will create and return a Bitcon script spk to which Bob can send funds using a P2SH transaction. To spend this output Alice will have to provide a signature under her public key pk_A and X, which she is able to provide, once acquired x. This construction is similar although simpler to the locking mechanism described by Malavolta et al.

  Question that arises here: Why a simpler locking mechanism suffices here? What is the extra functionality in Malavolta et al. that you do not require here?

  For a in-depth security analysis of this concept we refer the interested reader to their paper [14]. For a concrete Bitcoin Script realizing this functionality see section 6.

- $\{1, 0\} \leftarrow \mathsf{verifyLock}(pk_A, pk_B, X, v, t, \psi_{lock})$: The lock verification algorithm takes as input Alices, Bobs public keys and the statement X and the UTXO ψ_{lock}. The function will compute the Bitcoin lock script spk as created by $\mathsf{lockBtcScript}$ check equality with ψ_{lock} and if the value locked under the UTXO equals v. Upon successful verification the function returns 1, otherwise 0.

- $tx \leftarrow \mathsf{spendBtc}(inp, out, sk)$: The spend Bitcoin functionality is a wrapper around the $\mathsf{buildTransaction}, \mathsf{signTransaction}$ defined in 3.2.1. It constructs and signs a transaction spending the UTXOs given in inp and creates the fresh UTXOs in out. It returns a signed transaction which then can be broadcast.

I assume that the phases required in atomic swap will be explained earlier in the thesis?

5.5.1 Setup phase

We assume Alice owns Mimblewimble coins $[sp\mathcal{C}]$ with the total value v_{mw} and Bob owns Bitcoin locked in some UTXO ψ with a value of v_{btc} and secret spending key sk_{btc}. Before the protocol can start the two parties must agree on the value they want to swap, the exchange rate of the currencies and a time after which the swap should be canceled. After coming to an agreement the following variables are defined and known by both Alice and Bob:

- 1^n A security parameter.
- a_{btc} The amount of Bitcoin Bob will swap to Alice.
- a_{mw} The amount of the Mimblewimble coin Alice will swap to Bob.
- t_{btc} The locktime as a blockheight for the Bitcoin side.
- t_{mw} The locktime as a blockheight for the Mimblewimble side.

the symbol for *varSwpState* is overloaded. You have used this symbol already many times before for other purpose (e.g., the adversary?)

We collect this shared variables in an initial swap state $\mathcal{A}$:

$$\mathcal{A} \ := \ \{1^n, a_{btc}, a_{mw}, t_{btc}, t_{mw}\}$$

In practice, we need to consider that exchange rates might fluctuate, furthermore timeouts have to be calculated separately for each chain. The problems with cross chain payments are discussed by Tairi et al. in [23], they propose to use a fixed exchange rate for each day and to use a real world timeout like one day and then calculate the specific block numbers by taking the average block time of the blockchain into account. In our setup we can also fix the exchange rate at the beginning of the protocol, which stays unchanged during protocol execution. If the exchange rate fluctuates and one party is negatively impacted he or she could still decide to stop being cooperative which means the coins would be returned to the original owners after the timeout.

There is furthermore the problem of transaction fees, which we do not consider for this formalization. Depending on the current network load the participants need to agree on a fee that they are willing to pay for each network. It needs to be considered that if fees are picked to low, it might take time for transactions to be confirmed, and the swap will take longer, if they are picked high the participants will lose funds.

We formalize the protocol **lockSwp** in figure 5.15. The protocol takes as input the shared swap state $\mathcal{A}$ from both parties. From Alice her Mimblewimble input coins $[sp\mathcal{C}]$ with the summed up value v_{mw} is furthermore required as an input. From Bob we require a list of UTXO's $[\psi]$ he wants to spend, he also needs to provide their spending keys $[sk_{btc}]$ and their summed of total value v_{btc}, although this could also be read from the blockchain.

The protocol starts by both parties creating and exchanging keys. Bob now creates two new Bitcoin outputs ψ_{lock} and ψ_B, of which one is the locked Bitcoins which Alice might retrieve later (or Bob after time t_{btc} has passed), and the other Bobs change output.

(Difference between what is stored in the input UTXO and what should be sent to Alice). After Bob has published the transaction sending value to the new outputs, he will provide Alice with the statement X under which the Bitcoins' are locked together with Alice's public key. Alice can now verify that the funds on Bitcoin side are indeed correctly locked. After that she will collaborate with Bob to spend her Mimblewimble coins into an output shared by both parties. Immediately after, both parties collaborate again to spend this shared coin back to Alice with a timelock of t_{mw}. It is immanent that Alice does not publish the first transaction (A -> AB) before the timelocked refund transaction (AB -> A) was signed, otherwise her funds are locked in the shared output without the possibility of refund if Bob refuses to cooperate. The setup protocol concludes with the funds locked up in both chains and ready to be swapped and outputs the updated swap state $\mathcal{A}$ to both parties. Additionally, it outputs Alice's part $psp\mathcal{C}_A^*$ of the locked Mimblewimble coin, her change output on the Mimblewimble side $sp\mathcal{C}_A^*$, her secret key sk_A for the Bitcoin side and $sp\mathcal{C}_A{}'$, which is refund coin, only valid after t_{mw}. For Bob it furthermore outputs his part $psp\mathcal{C}_B^*$ of the locked Mimblewimble coin, his change output on the Bitcoin side ψ_B and the secret witness value x, which shall be revealed to Alice in the execution phase.

$\mathbf{lockSwp}\langle(\mathcal{A}, [sp\mathcal{C}], v_{mw})(\mathcal{A}, [\psi], [sk_{btc}], v_{btc})\rangle$

Alice		*Bob*
$(sk_A, pk_A) \leftarrow \mathsf{keyGen}(1^n)$		$(sk_B, pk_B) \leftarrow \mathsf{keyGen}(1^n)$
		$(x, X) \leftarrow \mathsf{keyGen}(1^n)$
	$\xrightarrow{pk_A}$	
	$\xleftarrow{pk_B}$	
		$spk \leftarrow \mathsf{lockBtcScript}(pk_A, X, pk_B, t_{btc})$
		$\psi_{lock} \leftarrow \mathsf{createUTXO}(a_{btc}, spk)$
		$\psi_B \leftarrow \mathsf{createUTXO}(v_{btc} - a_{btc}, pk_B)$
		$tx_{btc} \leftarrow \mathsf{spendBtc}([\psi], [\psi_{lock}, \psi_B], [sk_{btc}])$
		$\mathsf{publish}_{\mathsf{BTC}}([tx_{btc}])$
		$\mathcal{A} := \mathcal{A} \cup (X, \psi_{lock})$
	$\xleftarrow{X, \psi_{lock}}$	
if $\mathsf{verifyLock}(pk_A, pk_B, X, a_{btc}, t_{btc}, \psi_{lock}) = 0$		
return $\perp$		
$\mathcal{A} := \mathcal{A} \cup (X, \psi_{lock})$		
$({tx_{mw}}^{fnd}, sp\mathcal{C}^*_A, psp\mathcal{C}^*_A) \leftarrow \mathbf{dsharedOutMWTx}([sp\mathcal{C}], a_{mw}, \perp)$		$(psp\mathcal{C}^*_B) \leftarrow \mathbf{dsharedOutMWTx}(a_{mw})$
$({tx_{mw}}^{rfnd}, sp\mathcal{C}_A{}') \leftarrow \mathbf{dsharedInpMWTx}(psp\mathcal{C}^*_A, a_{mw}, t_{mw})$		${tx_{mw}}^{rfnd} \leftarrow \mathbf{dsharedInpMWTx}(psp\mathcal{C}^*_B, a_{mw})$
$\mathsf{publish}_{\mathsf{MW}}([{tx_{mw}}^{fnd}, {tx_{mw}}^{rfnd}])$		
return $(\mathcal{A}, psp\mathcal{C}^*_A, sp\mathcal{C}^*_A, sk_A, sp\mathcal{C}_A{}')$		**return** $(\mathcal{A}, psp\mathcal{C}^*_B, \psi_B, x)$

You need to add at some point that Alice (and Bob) parse the state in $\mathcal{A}$ to extract parameters they need later.
I think we need a way to differentiate better between local operations and 2-party protocols (e.g., dsharedInMWTx).

Figure 5.15: Atomic Swap - **lockSwp**.

5.5.2 Execution Phase

First we need to define an additional auxiliary function verfTime with the following interface:

$$\{0,1\} \leftarrow \mathsf{verfTime}(C, t)$$

This function will verify that there is sufficient time to execute the atomic swap protcol. As input it takes a chain parameter C (in our case this could be either BTC or MW) and a block height t. The routine will verify that the current height of the blockchain is marginally below t. If this is the case it will return 1, or 0 otherwise. How much time exactly should be left for the function to return 1 is implementation specific, and could be set to for instance one day. We now define a protocol **execSwap** to execute the Atomic Swap between some amount a_{btc} on the Bitcoin side and some amount on the Mimblewimble side a_{mw}. We assume the participants have successfully run the **lockSwp** protocol and both know the updated swap state $\mathcal{A}$ as returned by the setup protocol. Both parties need to provide their part of the locked Mimblewimble coins as input to the protocol. Additionally, Alice needs to provide her secret key for the Bitcoin side sk_A and Bob the secret witness value x. The protocol starts with both parties checking that there is enough time left to complete the protocol. After the check they will run the **dcontractMWTx** protocol in which they spend the locked Mimblewimble output to Bob, while at the same time revealing x to Alice. Either one of the parties can now publish the transaction to the Mimblewimble network, which concludes the swap on the mimbewimble side, as Bob is now in full control of the funds. Alice, knowing x, creates now a new UTXO where she then sends the funds from the Bitcoin lock. After publishing this transaction to the Bitcoin network, Alice is in full possession of the swapped funds on the Bitcoin side and the Atomic Swap is completed. The protocol outputs their newly created output/coin to each party.

execSwap$\langle(\mathcal{A}, psp\mathcal{C}^*_A, sk_A), (\mathcal{A}, psp\mathcal{C}^*_B, x)\rangle$

Alice	*Bob*
$(a_{mw}, a_{btc}, t_{mw}, t_{btc}, \psi_{lock}, X) \leftarrow \mathcal{A}$	$(a_{mw}, a_{btc}, t_{mw}, t_{btc}) \leftarrow \mathcal{A}$
if $\mathsf{verfTime}(BTC, t_{btc}) = 0 \vee \mathsf{verfTime}(MW, t_{mw}) = 0$	**if** $\mathsf{verfTime}(BTC, t_{btc}) = 0 \vee \mathsf{verfTime}(MW, t_{mw}) = 0$
return $\perp$	**return** $\perp$
$(tx_{mw}, \emptyset, x)$	$(tx_{mw}, sp\mathcal{C}^*_B)$
$\leftarrow$ **dcontractMWTx**$(psp\mathcal{C}^*_A, a_{mw}, \perp, X)$	$\leftarrow$ **dcontractMWTx**$(psp\mathcal{C}^*_B, a_{mw}, x)$
$\mathsf{publish}_{\mathsf{MW}}(tx_{mw})$	$\mathsf{publish}_{\mathsf{MW}}(tx_{mw})$
$(sk_A{}', pk_A{}') \leftarrow \mathsf{keyGen}(1^n)$	
$\psi_A \leftarrow \mathsf{createUTXO}(a_{btc}, pk_A{}')$	
$tx_{btc} \leftarrow \mathsf{spendBtc}([\psi_{lock}], [\psi_A], [sk_A, x])$	
$\mathsf{publish}_{\mathsf{BTC}}(tx_{btc}{}^*)$	
return (ψ_A)	**return** $(sp\mathcal{C}^*_B)$

Here both publish the mimblewimble tx?

Figure 5.16: Atomic Swap - **lockSwp**.

5.5.3 Refunding

If one party refused to cooperate or goes offline the coins can be returned to the original owner. On the Bitcoin side this is the case as Bob can simply spend the locked output with his private key sk_B after the timeout t_{btc} has passed. He then can simply construct and sign a transaction spending the output to a new UTXO which is in his full possession. He even could prepare this transaction upfront and broadcast it, once the the blocknumber hits t_{btc} the transaction will become valid and get mined. Again we stress the importance of using appropriate timeouts, if a timeout is too short the swap might get cancelled if there are some delays, if the timeout is too long the funds might be locked for an unnessary amount of time.

On the Mimblewimble side the second transaction spending the shared output back to Alice guarantees that her funds are returned to her after the timeout t_{mw} hits. For this reason it is so important that Alice publishes both the fund and refund transaction at the same time. If she would publish the funding transaction first, Bob could refuse to cooperate for the refund transaction, in which case the funds would stay in the locking output only retrievable if both parties cooperate. If the swap executes successful the refund transaction would get discarded by miners, as it then is no longer valid even after the timeout t_{mw}.

CHAPTER 6

Implementation

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] S. Al-Kuwari, J. H. Davenport, and R. J. Bradford. Cryptographic hash functions: recent design trends and security notions. *IACR Cryptology ePrint Archive*, 2011:565, 2011.

[2] A. M. Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc.", 2014.

[3] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostakova, M. Maffei, P. Moreno-Sanchez, and S. Riahi. Generalized bitcoin-compatible channels. Cryptology ePrint Archive, Report 2020/476, 2020. `https://eprint.iacr.org/2020/476`.

[4] B. B ü nz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[5] G. Betarte, M. Cristi á , C. Luna, A. a. n. Silveira, and D. Zanarini. Towards a formally verified implementation of the mimblewimble cryptocurrency protocol. *arXiv preprint arXiv:1907.01688*, 2019.

[6] S. Delgado-Segura, C. Pérez-Sola, G. Navarro-Arribas, and J. Herrera-Joancomartí. Analysis of the bitcoin utxo set. In *International Conference on Financial Cryptography and Data Security*, pages 78–91. Springer, 2018.

[7] G. Fuchsbauer, M. Orr ù , and Y. Seurin. Aggregate cash systems: a cryptographic investigation of mimblewimble. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 657–689. Springer, 2019.

[8] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[9] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 321–340. Springer, 2010.

[10] T. E. Jedusor. Mimblewimble, 2016.

[11] D. Klinec and V. Matyas. Privacy-friendly monero transaction signing on a hardware wallet, extended version. *IACR Cryptol. ePrint Arch.*, 2020:281, 2020.

[12] Y. Lindell. A note on constant-round zero-knowledge proofs of knowledge. *Journal of cryptology*, 26(4):638–654, 2013.

[13] Y. Lindell. How to simulate it–a tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer, 2017.

[14] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *NDSS*, 2019.

[15] G. Maxwell. Coinjoin: Bitcoin privacy for the real world. In *Post on Bitcoin forum*, 2013.

[16] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Designs, Codes and Cryptography*, 87(9):2139–2164, 2019.

[17] S. Micali, K. Ohta, and L. Reyzin. Accountable-subgroup multisignatures. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 245–254, 2001.

[18] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.

[19] A. Poelstra. Mimblewimble, 2016.

[20] T. Ristenpart and S. Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 228–245. Springer, 2007.

[21] C.-P. Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[22] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[23] E. Tairi, P. Moreno-Sanchez, and M. Maffei. A2l: Anonymous atomic locks for scalability and interoperability in payment channel hubs. *IACR Cryptol. ePrint Arch.*, 2019:589, 2019.