TU WIEN Informatics

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

I wish to show appreciation to my advisors Dr. Pedro Moreno-Sanchez and Univ. Prof. Dr. Matteo Maffei, for providing me with valueable guidance throughout this thesis. I would also like to thank Andrew Poelstra and the Grin Developers (specifically Jasper) for helping me with technical issues that arose during the proof of concept implementation. Special thanks further go to Bernhard Abfalter for proofreading.

Abstract

Since the inception of Bitcoin in 2008, we have witnessed continuous growth in the Bitcoin and blockchain space. As the number of individual cryptocurrencies rises, interoperability between them becomes a critical topic, for instance, to allow for decentralized coin exchange. By utilizing smart contracts or script constructs available on most blockchain systems, connecting two cryptocurrencies is possible via so-called Atomic Swaps. However, for the currencies focusing on privacy enhancements, no such capabilities exist.
This thesis explores the Mimblewimble protocol, a construction of an exceptionally efficient Privacy-enhancing cryptocurrency. By building on other authors' previous work, we formalize different kinds of Mimblewimble transactions that allow for shared coin ownership and simple contracts and prove their security and correctness. We improve on the protocol's security model by identifying and resolving a weakness in prior formalizations. Utilizing our advanced transaction protocols, we manage to design an Atomic Swap protocol for Mimblewimble based systems built solely with cryptographic primitives. We further formalize an Atomic Swap protocol between Bitcoin and Grin, a Mimblewimble based cryptocurrency. We then implement a proof of concept in the programming language Rust, which we successfully deploy and evaluate on the Bitcoin and Grin testnets.

Contents

CHAPTER 1

Introduction

Since the original release of the Bitcoin whitepaper [**?**] on October 31, 2008, by the anonymous entity Satoshi Nakamoto, we have seen a continuous rise in interest in Bitcoin and other cryptocurrencies. The Bitcoin protocol allows for a P2P (peer to peer) exchange of the Bitcoin currency without trusted intermediaries. This is made possible by the distributed consensus protocol of proof-of-work in which so-called miners compete in adding new transactions to the Bitcoin ledger for which they are rewarded by newly created currency and by transaction fees. Bitcoin works with the so-called UTXO (Unspent transaction output) model, in which all funds are stored in transaction outputs. Every output can only ever be spent once by its owner. Therefore, the list of UTXOs denotes all currently spendable coins in the network.

Coin Exchanges. According to CoinMarketCap[1] at the time of writing, there are almost 9000 different cryptocurrencies available with a combined market cap of $1.8 trillion. Most of these currencies try to improve upon various Bitcoin protocol limitations, for instance, by making it either more expressive, more private, or more efficient. Trading between Fiat (government-issued currency) and cryptocurrencies, and between individual cryptocurrencies is a popular topic among retail investors. It has recently started to become attractive as well for institutions and publicly listed companies such as Microstrategy, Grayscale, or Tesla.[2] Most of the trading is currently happening on centralized exchanges such as Binance[3] or Coinbase[4]. Users can deposit, exchange, and withdraw cryptocurrencies or Fiat currency on these platforms, while the service provider controls the funds available to its users.

[1] https://coinmarketcap.com/
[2] https://bitcointreasuries.org/
[3] https://www.binance.com/en
[4] https://www.coinbase.com/

Although used in practice, centralized exchanges have several drawbacks as the following ones: We have seen instances [5,6,7] in which hackers managed to steal large quantities of funds from such platforms. Exchanges of this sort are usually required by law to collect KYC information, which acts as proof of the customers' identity to prevent illegal activity on their platforms. This requirement can create a barrier of entry for people that lack identification documents or are unwilling to give away this sort of data to the provider. Consequently, decentralized exchanges such as Uniswap[8] or Bisq[9] have emerged and are gaining rapidly in popularity. A decentralized exchange allows users to exchange cryptocurrencies directly in a P2P fashion without the need for a trusted intermediary. Smart contracts that reside on a blockchain such as Ethereum[10] and allow for trustless swaps between currencies make decentralized exchange possible. Interoperability between cryptocurrencies is critical to enable such coin swaps. A trustless protocol allowing trades of two individual cryptocurrencies is called an Atomic Swap protocol [**?**]. In such a protocol, funds are locked up on both sides of the trade and settled such that each party gains access to the locked funds on the other side after successful protocol execution. Usually, such a protocol is built utilizing scripting or smart contract capabilities of the individual blockchains. However, cryptocurrencies that focus on improving user privacy that we will discuss in section 2.3 usually lack these capabilities, making the construction of a secure Atomic Swap protocol particularly challenging.

Mimblewimble. The Mimblewimble protocol was introduced in 2016 by an anonymous author Tom Elvis Jedusor [**?**] and represents an outline for a new Privacy-enhancing cryptocurrency with shallow space requirements of the ledger. The author's name and the protocol's name are references to the Harry Potter franchise. [11] In Harry Potter, Mimblewimble is a tongue-tying curse that reflects the protocol's design goal, improving the user's privacy. Later, Andrew Poelstra took up the original writing's ideas and published his understanding of the protocol [**?**]. Mimblewimble gained popularity in the community and was implemented in the Grin[12] and Beam[13] cryptocurrencies that both launched in early 2019. In the same year, two papers [**?**, **?**] were published, which successfully defined and proved security properties for Mimblewimble. Compared to Bitcoin, there are some differences in the Mimblewimble protocol:

- Transaction values are hidden from a blockchain observer, which is not the case in Bitcoin.

[5] https://tinyurl.com/yrcpp5jm
[6] https://tinyurl.com/5f8wkv7v
[7] https://tinyurl.com/a9hxmftm
[8] https://uniswap.org/
[9] https://bisq.network/
[10] https://ethereum.org/
[11] https://harrypotter.fandom.com/wiki/Tongue-Tying_Curse
[12] https://grin.mw/
[13] https://beam.mw/

- Coin ownership is given by a single private key of a so-called coin Commitment. In Mimblewimble, there are no addresses or scripting capabilities that do exist in Bitcoin.
- The nodes constantly purge spent coins from the ledger such that only unspent transaction outputs remain, and the ledger's space requirements remain low, but public verifiability of the blockchain is not lost.
- Transactions are continuously merged together to achieve a degree of transaction indistinguishability, further improving the user's privacy.

Motivation.

I think the introduction is missing at this point a motivation to answer the questions: "why did you do this work?", "why is interesting for society and/or blockchain community?", "what are the technical challenges?". Given that section 2 is already formalisms, I would even try to put here a sketch of your approach and 1-2 sentences about why you did it like that and why is challenging.

Our contribution. In this thesis, we will first describe a new variant of a Two-Party Signature Scheme, which can be used to build primitive contracts on a Mimblewimble based cryptocurrency. We will formalize this construction and prove its correctness and security in chapter 3. In chapter 4, we then formalize four different kinds of transaction protocols that can be conducted between a sender and a receiver in a Mimblewimble based cryptocurrency, of which one uses the Signature Scheme introduced in the previous chapter. We show that all four transaction types are secure and correct as of the definitions given by Fuchsbauer et al. in [**?**] and finally construct an Atomic Swap protocol that allows for a trustless exchange between Bitcoin and a Mimblewimble based cryptocurrency. In chapter 5, we showcase our working proof of concept implementation of the protocol, which was tested and evaluated on the Bitcoin and Grin testnets.
Our thesis contributes to making Privacy-enhancing cryptocurrencies, specifically those built on the Mimblewimble protocol, more interoperable, allowing them to be listed on decentralized exchanges. The following request for funding[14] shows the developers' and the community's interest in implementing a production-grade version of such a protocol. By implementing our formalization and taking our proof of concept as a reference, developers can build and deploy an Atomic Swap protocol that securely swaps Grin and Bitcoin funds without intermediaries.

[14] https://tinyurl.com/4jkbkccu

CHAPTER 2

Preliminaries

This chapter will lay down the general notations and definitions required for the later parts of the thesis. In section 2.1, we will define several cryptographic primitives which are necessary for our constructions. Section 2.2 will describe definitions around Bitcoin, mainly its transaction structure. After that, in section 2.3, we will discuss the notion of Privacy-enhancing cryptocurrencies and range proofs in section 2.3.2. Both are essential to understand the Mimblewimble protocol addressed in section 2.4. Finally, we explain the concept of Scriptless Scripts and Adaptor signatures in section 2.5, which are relevant building blocks for the constructions found in this thesis.

2.1 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 regarding g is not known. Exponentiation stands for repeated application of the group operation. We define the group operation between two curve points as $g^a \cdot g^b = g^{a+b}$. We write procedures as $\mathsf{function}(\cdot)$ and protocols (executed between multiple parties) as $\mathbf{protocol}(\cdot)$

Visually, the notation for function and protocol are not easy to distinguish. Is it easy to change to make it more different?

When you want to put a reference to where you get a definition from, I would just put it at the side of the name of the definition

Definition 2.1 (Hard Relation [**?**])**.** Given a language $L_R := \{X \mid \exists x \text{ s.t. } (x, X) \in R\}$ then the relation R is considered hard if the following three properties hold:

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

2. Relation R is poly-time decidable.

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

Definition 2.2 (Discrete Logarithm)**.** We define the discrete logarithm in a group $\mathbb{G}$ of a point h as the integer n such that for the groups generator g the following holds:

$$g^n = h$$

Discrete logarithm assumption: Given the tuple (h, g) such that $h = g^n$ one can compute n only with negligible probability which makes (h, n) a hard relation as of definition 2.1.

The discrete logarithm is not a hard relation. The hard relation is given by the discrete logarithm assumption (also called discrete logarithm problem). That is, an adversary A given the tuple (g, h) such that $h = g^n$ can compute n only with negligible probability.

Definition 2.3 (Signature Scheme)**.** A Signature Scheme Φ is a tuple of algorithms $(\mathsf{keyGen},\ \mathsf{sign},\ \mathsf{verf})$ defined as follows: [**?**]

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

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

- $\sigma \leftarrow \mathsf{sign}(m, sk)$: The signing algoritm creates a signature under the message m, which can be verified with the respective public key pk. 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 validity of the signature.

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: $\mathsf{verf}(pk, \mathsf{sign}(sk, m), m) = 1$

- Unforgeability ($\mathsf{EUF} - \mathsf{CMA}$): Informally the existential unforgeability under chosen message attacks holds if attacker $\mathcal{A}$ with access to a signing oracle $\mathcal{O}_s$ cannot forge a valid signature for a chosen message. More formally, for a polytime adversary $\mathcal{A}$, message m, and public key pk, with access to the oracle $\sigma \leftarrow \mathcal{O}_s(m, pk)$ that stores all signed messaged in a list $\{m_1, ..m_n\}$ the following must hold:

$$\sigma \leftarrow \mathcal{A}(m^*, pk),\ m^* \notin \{m_1, ..m_n\},\ P(\mathsf{verf}(m, \sigma, pk) = 1) \leq \mathsf{negl}(\cdot)$$

Definition 2.4 (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 [**?**]. 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 $\mathsf{H}(I) = \mathsf{H}(I^*)$.

The relation between the input I and the output $\mathsf{H}(I)$ is a hard relation as defined in definition 2.1.

Definition 2.5 (Commitment Scheme)**.** A cryptographic Commitment Scheme *COM* is defined by a pair of functions $(\mathsf{setupCom}, \mathsf{commit})$ [**?**].

- $rs \leftarrow \mathsf{setupCom}(1^n)$: 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 $\mathbb{C}_{PP}$.

- $C \leftarrow \mathsf{commit}(I, r)$ The commit routine is DPT function that takes an arbitrary input $I \in \mathbb{I}_{PP}$, a random nonce value $r \in \mathbb{K}_{PP}$ 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 $r \in \mathbb{K}_{PP}$ the probabilty of finding a $I^* \neq I$ and a r^* with $\mathsf{commit}(I, r) = \mathsf{commit}(I^*, r^*)$ is negligible.

- *Hiding:* For a PPT adversary $\mathcal{A}$, Commitment inputs $I_0, I_1 \in \mathbb{I}_{PP}$ randomness $r \in \mathbb{K}_{PP}$ and a Commitment output $C := \mathsf{commit}(I_b, r,)$ the probabilty of the adversary choosing the correct b out of $\{0,1\}$ must not be higher then $\frac{1}{2} + \mathsf{negl}(\cdot)$.

Definition 2.6 (Additive Homomorphic Commitment Scheme)**.** A Commitment Scheme as defined in definition 2.5 is said to be addtive homomorphic if the following holds [**?**]

$$\mathsf{commit}(I_1, r_1) \cdot \mathsf{commit}(I_2, r_2) = \mathsf{commit}(I_1 + I_2, r_1 + r_2)$$

Definition 2.7 (Pedersen Commitment Scheme)**.** A *Pedersen Commitment Scheme* is an instance of a Commitment Scheme as defined in definition 2.5 that has the additive homomorphic property as defined in definition 2.6.

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:

$$rs \leftarrow \mathsf{setupCom}(g_1, g_2) := g := g_1, h := g_1$$
$$C \leftarrow \mathsf{commit}(I, r) := g^r h^I$$

An instantiation of the Pedersen Commitment Scheme must pick two adjacent generators g_1, g_2 for the setup routine 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 definition 2.2. So no x is known such that $h = g^x$. In practice, this is often achieved by hashing g with a special hash function that outputs a group element as h.

Here you are switching the topic a little, going from primitives to your security analysis technique. I would introduce this change in text or even create another subsection

To prove our protocols' security, 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 explain two terms: IDEAL, the execution in the ideal model and REAL, and the real model's execution. The following definitions are based on a tutorial paper on simulation proofs by Yehuda Lindell [**?**].

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 (TTP) 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 that 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 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 as x to proceed with the protocol execution. The decision is made by $\mathcal{A}$ and may depend on the input or auxiliary 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 corrupted party's index. 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 corrupted party's index. 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 2.8 (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 $z = \mathsf{poly}(|\mathsf{x}|)$ and $\equiv_c$ means computationally indistinguishable [**?**].

2.2 Bitcoin

In this section, we will discuss the basics of the Bitcoin transaction protocol. We will find definitions that we will use later in section 4.5 to construct an Atomic Swap protocol. The primary reference of this section is the book Mastering Bitcoin by Andreas Antonopoulos [**?**].

2.2.1 Bitcoin Transaction Protocol

A *Bitcoin Transaction* is a data structure that 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 employed. A UTXO is an output constructed in a previous transaction that 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 the output has not yet been spent, and its funds are available to be redeemed by the owner. To unlock this value, one must provide a script fulfilling the locking condition, referred to as scriptSig. In the most common case, the lock condition will fulfill by giving a valid signature under a public key. This type of construction is referred to as a P2PK or P2PKH output, which we will see in more detail in section section 2.2.1. However, more complex conditions, such we shall see in section 2.2.1, are possible.

Definition 2.9 (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 thesis, 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 a UTXO. Note that we use verf as a generalization of a verification function. In practice, verification of a spent UTXO will most of the time correspond to the digital signature verification. However, as we shall see in section 2.2.1, 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** $\mathsf{verf}(spk, \sigma, v)$

A complete transaction consists of one or many UTXOs as inputs and one or many new 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. An output value lower than the combined input value is allowed. Then the miner of the transaction gets to collect the

```
{
  "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 2.1: A decoded Bitcoin transaction

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 verify after a certain block number in the Bitcoin blockchain was mined. Figure 2.1 shows a decoded Bitcoin transaction. [1]

Definition 2.10 (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:

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

A transaction is valid if the following conditions are fulfilled:

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

Definition 2.11 (Bitcoin Transaction Scheme)**.** We define a Bitcoin Transaction Scheme as a tuple of two DPT functions ($\mathsf{buildTransaction}, \mathsf{verfTransaction}$) and the PPT function $\mathsf{signTransaction}$.

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

- $tx_{btc} \leftarrow \mathsf{buildTransaction}(\Psi_{inp}, \Psi_{out}, vs, t)$: The transaction building algorithm is a DPT function that takes as input a set of unspent transaction outputs Ψ_{inp}, a collection of newly created transaction outputs Ψ_{out} a version number vs and an optional locking time t. The algorithm will output an unsigned transaction tx_{btc}.

- ${tx_{btc}}^* \leftarrow \mathsf{signTransaction}(tx_{btc}, [\sigma])$: The transaction signing algorithm is a PPT function that takes as input an unsigned Bitcoin transaction tx_{btc} and an array of unlocking scripts $[\sigma]$ for all transaction inputs. The algorithm outputs a signed Bitcoin transaction, which the sender can now 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} outputting 1 on a successful verification or 0 otherwise. The function will check the well-balancedness of the transaction, verify the unlocking scripts, locktime and scan through the blockchain if all inputs are indeed unspent. Note that any public verifier with access to the blockchain ledger and tx_{btc} is able to perform the verification.

We now 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 the former 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 al. found in their paper Analysis of the Bitcoin UTXO set [**?**] from 2017 that more than 80% of the UTXO set consisted of P2PKH transactions, whereas about 17% were P2SH and 0.12% P2PK outputs. P2PKH outputs can be encoded into a Bitcoin address using base58 encoding. These 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 straightforward manner. 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 and the unlocking requirements for the script itself. Upon verification, the provided script's hash will be computed and compared with the value given in the locking condition. If those match, the actual script will be executed. The advantage of using this approach over just handcrafting a custom locking script is that the locking scripts are relatively

short, making the transactions smaller, reducing fees, or shifting them from the sender to the output owner. 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.

2.3 Privacy-enhancing Cryptocurrencies

As seen in section 2.2 in Bitcoin funds are stored in UTXOs, which an address can identify. The value being transferred in a transaction is given in plain text; therefore, by the nature of Bitcoin's public blockchain, anyone can look up the amount stored in a given address. So-called block explorers[2] make such a lookup straightforward. As demonstrated, for instance, in [?] or [?], it is further possible to link multiple addresses by analyzing transactions, further weakening the system's anonymity and allowing for the extraction of sensitive metadata. Attempts such as CoinJoin [?] or its successor CoinShuffle [?], CoinShuffle++ [?] introduced protocols that can mitigate this likability issue in Bitcoin.

The goal of privacy-enhancing cryptocurrencies is to improve upon Bitcoin's anonymity by the use of cryptographic techniques such as Zero-Knowledge Proofs section 2.3.1 and homomorphic commitments definition 2.6 to achieve Transaction Unlinkability as well as Confidential Transactions (first mentioned by Adam Back in [?]) in which the transferred values are hidden in homomorphic Commitments.

Definition 2.12 (Transaction Unlinkability)**.** Given are the two related transactions tx_a which sends a value from $A \to B$, tx_b sending from $B \to C$ and unrelated tx_c sending from $X \to Y$. For an attacker $\mathcal{A}$ that is given tx_a, tx_b, tx_c with the task of finding the linking transactions and without having any additonal knowledge than what can be inferred from the public ledger, the following must hold for Transaction Unlinkability to be fulfilled:

$$P(\mathcal{A}(tx_a, tx_b, tx_c) \;=\; (tx_a, tx_b)) \;=\; \frac{1}{3} \;+\; \mathsf{negl}(\cdot)$$

Definition 2.13 (Confidential transactions)**.** Given a transaction tx that transfers amount a, and a set of possible transaction amounts $[1, \cdots, n]$ for an attacker $\mathcal{A}$ trying to extract the correct transaction amount, the following for transactions to be considered confidential:

$$P(\mathcal{A}(tx) \;=\; a) \;=\; \frac{1}{n} \;+\; \mathsf{negl}(\cdot)$$

I think this sentence does not make much sense. Rewrite

Where did you get this definition of confidential transaction from? Or did you crafted on your own? I think this rarely holds in practice. For instance, I know that the probability that amount is 1 satoshi (or 1M Bitcoins) is way lower than "normal" amounts. Perhaps I would define it as, having two possible transaction values a and b, the adversary, looking at the transaction does not know whether the transaction is sending a or b coins.

[2] https://blockstream.info/

Notable examples of such constructions are Zerocash [?], Monero [?], and Mimblewimble [?]. Zerocoin proposed using one-way accumulators [?], allowing the minting of of unlinkable Zerocoins from regular Bitcoins. However, their proposal had some limitations, such as only allowing Zerocoins of a fixed denomination and the inefficient construction. In Zerocash, the authors improved upon Zerocoin by utilizing Zero-Knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs) [?] instead of an accumulator and managed to address the limitations. However, the protocol requires an initial trusted setup. A prominent implementation of the Zerocash protocol is the Zcash cryptocurrency.[3] Monero utilizes Ring Signatures to achieve Transaction Unlinkability and further uses homomorphic Commitments definition 2.6 together with range proofs section 2.3.2 to hide transaction amounts as initially proposed by Adam Back. The Monero protocol has been implemented in the Monero cryptocurrency.[4] Mimblewimble, being the main topic of this thesis, will be discussed in detail in section 2.4.

2.3.1 Zero-Knowledge Proofs

Zero-Knowledge proofs were first defined in 1988 by Fiat, Fiege, and Shamir and are essential for building cryptocurrencies. The proofs allow a prover to convince a verifier that he owns a witness value without revealing the value itself. Initially, the protocol was presented as an interactive proof between a prover and verifier. However, by utilizing the Fiat Shamir Heuristic [?], the proofs can be converted into a non-interactive protocol, making the proof publicly verifiable. Digital signatures (definition 2.3) such as Schnorr Signatures are a prominent instantiation of a Zero-Knowledge proof protocol. Zero-Knowledge proofs such as Bulletproofs [?], or zk-SNARKS [?], have essential applications in privacy-Enhancing cryptocurrencies.

I see that you have not defined the notion of zero-knowledge proof. Don't you need it afterwards?

2.3.2 Range Proofs

A range proof testifies that a secret value that was encrypted or committed to lies within a specific valid range of values. The proofs are Zero-Knowledge in that they do not leak any information about the secret value other than that it lies in the given interval [?]. Range proofs can be implemented using ring signatures [?], which was the original implementation used by Monero, later replaced by the more efficient Bulletproofs [?], which are also used in the two most prominent Mimblewimble based cryptocurrencies, Beam and Grin. We define a Range Proof System as follows:

Definition 2.14 (Range Proof System)**.** A Range Proof System $\Pi_{RP}[COM]$ regarding a homomorphic Commitment Scheme COM consists of a tuple of functions ($\mathsf{ranPrfSetup}$, $\mathsf{ranPrf}, \mathsf{vrfRanPrf}$).

[3] https://z.cash/
[4] https://www.getmonero.org/

- $ps \leftarrow \mathsf{ranPrfSetup}(1^n, i, j)$: The range proof setup algorithm takes as input a security parameter 1^n and two numbers i and j to define the lower and upper bound (lb, ub) of the range proof protocol.

- $\pi \leftarrow \mathsf{ranPrf}(C, v, r)$: The proof algorithm is a PPT function that takes as input a Commitment C, a value v and a blinding factor r. It will output a proof π attesting 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 that verifies the validity of the proof π concerning the Commitment C. It will output 1 upon a successful verification or 0 otherwise.

An efficient instantion of a Range Proof System is the bulleproof [**?**] protocol that is currently used in the Monero and Mimblewimble-based cryptocurrencies.

We also define a Two-Party Range Proof System as an extension to a regular Range Proof System. Two parties collaborate to compute a Zero-Knowledge proof attesting that a secret value of a specific Commitment or encrypted value is within a given interval. A construction of such a protocol was done, for instance, by Klinec et al. in [**?**].

Definition 2.15 (Two-Party Range Proof System)**.** A Two-Party Range Proof System $\Pi_{RP-MP}[COM]$ regarding a homomorphic Commitment Scheme COM is an extension to the regular Range Proof System 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 to produce a valid range proof π without a party learning the blinding factor share from the other party.

What are the security/privacy properties that you require from range proofs and distributed range proofs?

2.4 Mimblewimble

This section will outline the fundamental properties of the protocols employed in Mimblewimble, which are relevant for the thesis and particularly the Atomic Swap protocol constructed in chapter 4. The Mimblewimble protocol is an outline for a privacy-enhancing cryptocurrency that achieves Confidential Transactions as of definition 2.13. If the protocol also achieves Transaction Unlinkability definition 2.12, is not fully clear. One can argue that Transaction merging (see section 2.4) together with Cut-through section 2.4 should make transactions unlinkable. However, one author managed to link a large amount of Mimblewimble transactions in a practical setting using well-connected network nodes that attempt to sniff original transactions before they are merged[5].

putting the section without any parenthesis or context breaks the reading flow. What about adding (see section x.y)?

[5] https://github.com/bogatyy/grin-linkability

Then what? why are you using Mimblewimble? Should we just not care about this as it is not privacy-preserving? (I am just asking this because could be one of the questions of the reviewer :P)

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 2.16 (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 [**?**] 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 prominent 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 2.17 (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 2.18 (Transaction validity)**.** A transaction is valid if:

- The transaction is well-balanced as defined in definition 2.17
- $\forall\ (\mathcal{C}_i, \pi_i) \in \mathcal{C}_{out}\ \mathsf{vrfRanPrf}(\pi_i, \mathcal{C}_i) = 1$

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 transaction's well-balancedness 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 value given e. Therefore, knowledge of the discrete logarithm to $\mathcal{E}$ is proven by providing a valid signature under $\mathcal{E}$ as the 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 be set to 0. A complete Mimblewimble transaction is of the 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 $\{\sigma\}$ a set of signatures [**?**]. Even though usually a transaction would only require a single excess and signature, we will see in the next section that these fields always have to be lists instead of just a single value.

Transaction Merging

An intriguing property of the Mimblewimble protocol is that two transactions can easily be merged into a single one, essentially a non-interactive version of the CoinJoin protocol on Bitcoin [**?**]. 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 2.18, 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 the public key. The same must hold for tx_1.

Then it follows 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.

What does the symbol , means? You have not defined it before

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 was 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 Mimblewimble 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, public verification is still possible. Now every time two transactions are merged with the method laid out previously, the two individual offset values o_0 and 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 [**?**].

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 let's recall the equation given for transaction well-balancedness in definition 2.17:

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

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 spent, 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 and provide a notion of unlinkability, as transaction histories are continuously erased.

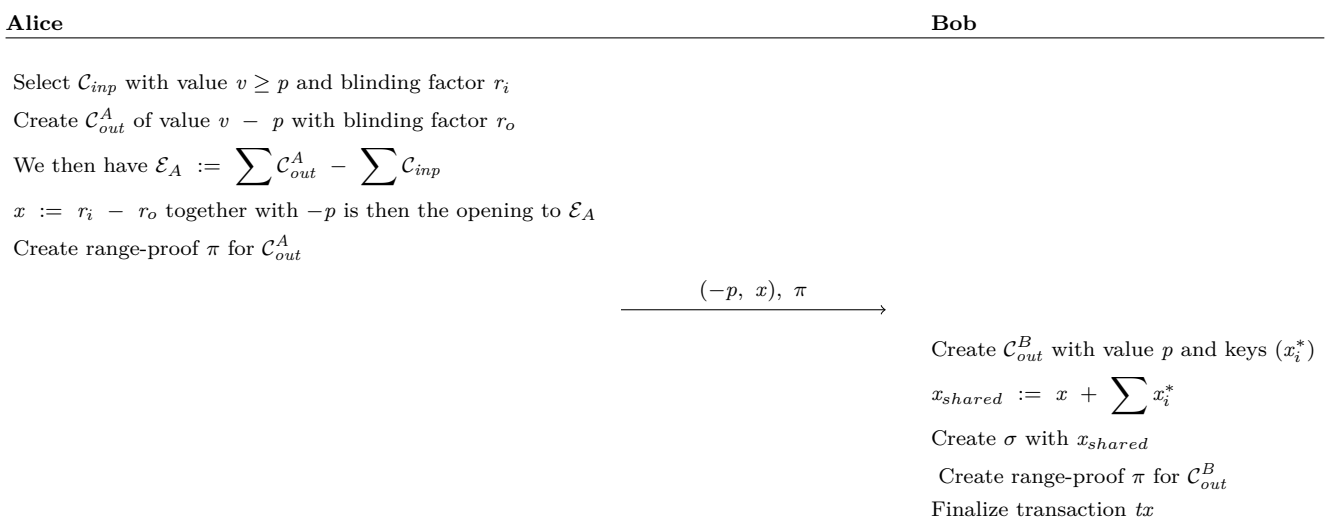

Figure 2.2: Original transaction building process

Transaction Building

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. 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: [**?**]

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 again be multiple) with the remainder of her input value substracted by the value sent 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 from Alice, he can now create a signature to $\mathcal{E}$ and finalize the transaction

Figure 2.2 depicts the original transaction flow.

This protocol, however, turned out to be insecure as shown by Fuchsbauer et al. in [**?**]. 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 this:

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

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 that has already existed on the ledger with the same blinding factor and value, meaning the proof will still be valid.

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 3.

Alternatively, Fuchsbauer et al. [**?**] proposed another way to build transactions that 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 chosen 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 therefore will be discarded.

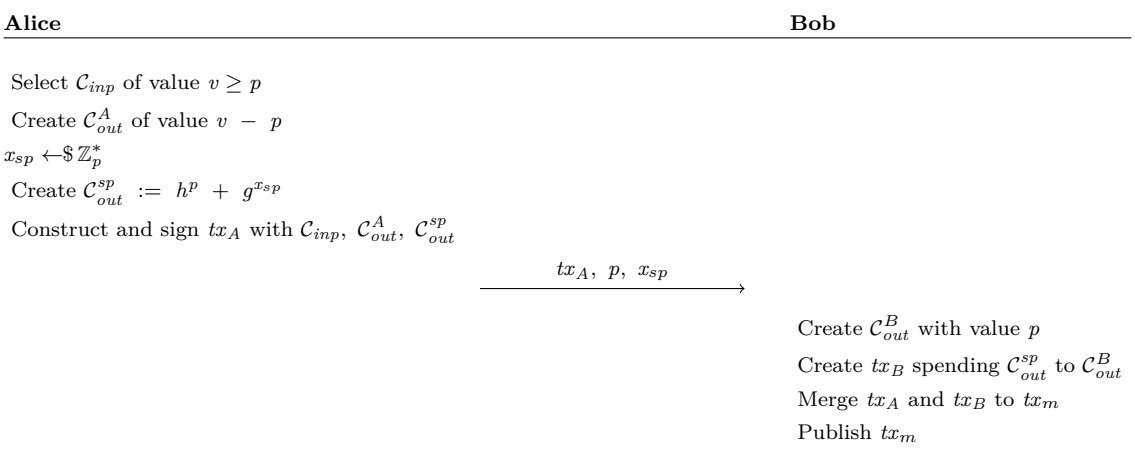

Figure 2.3: Salvaged transaction protocol by Fuchsbauer et al. [**?**]

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 4, at least one additional round of interaction will be required. A figure showing the protocol flow is depicted in fig. 2.3.

Mimblewimble Ledger

In Mimblewimble, the ledger itself is a transaction of the form defined in section 2.4 with a set of input and outputs which initially start out empty [**?**]. The list of outputs as given in the ledger is the list of spendable coins, similar to the list of UTXOs (unspent transaction outputs) in Bitcoin. When publishing a new transaction, it will be merged with the ledger itself, as seen in section 2.4, after which a cut-through as seen in section 2.4 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 ever 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 [**?**]. 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.

2.5 Scriptless Scripts

In 2017 Andrew Poelstra presented a concept called Scriptless Scripts that achieves the execution of primitive contracts only by the use of standard cryptographic tools such as digital signatures. [**?**] The most prominent Scriptless Script is the Adaptor Signature Scheme which has been fully formalized and proven to be secure for Schnorr and ECDSA by Aumayr et al. in [**?**]. Scriptless Scripts are helpful for cryptocurrencies that lack scripting functionality, as is the case in Mimblewimble based systems. It also helps to replace script-based approaches with improvements in privacy as well as efficiency, shown in a paper by Christoph Egger et al. [**?**]. In chapter 3, we will leverage the Adaptor Signature Scheme concept together with a Two-Party Signature Scheme to build a scriptless Atomic Swap protocol applicable for Mimblewimble based cryptocurrencies. The Adaptor Signature Scheme is a two-step process: A signer first computes a pre-signature that can be completed only by a party knowing a certain secret witness value x from a hard relation $(x, X) \in R$. After the pre-signature is completed into the final one, it must then be possible to extract x given the final and the pre-signature.

We here repeat the definition of an Adaptor Signature Scheme found by Aumayr et al. in [**?**].

Definition 2.19 (Adaptor Signature Scheme)**.** An Adaptor Signature Scheme wrt. a hard relation R as of definition 2.1 and a Signature Scheme Φ as of definition 2.3 consists of four algorithms:

- $\tilde{\sigma} \leftarrow \mathsf{pSign}(sk, m, X)$ is a PPT algorithm that on input secret key sk, message $m \in \{0,1\}^*$ and statement X outputs a pre-signature $\tilde{\sigma}$.
- $\{1, 0\} \leftarrow \mathsf{pVrfy}(pk, m, X, \tilde{\sigma})$ is DPT algorithm that on input a public key, message $m \in \{0,1\}^*$, statement X and pre-signature $\tilde{\sigma}$, outputs either 1 or 0.
- $\sigma \leftarrow \mathsf{Adapt}(\tilde{\sigma}, x)$ is a DPT algorithm that on input a pre-signature $\tilde{\sigma}$ and witness value x, outputs a signature σ.
- $x \leftarrow \mathsf{Ext}(\sigma, \tilde{\sigma}, X)$ is a DPT algorithm that on input a signature σ, pre-signature $\tilde{\sigma}$ and statement X, outputs a witness x such that $(x, X) \in R$, or $\perp$.

I would add here the security notions as well so that the reader can relate them with the one that you define later for your 2-party adaptor signature

CHAPTER 3

Two-Party Fixed Witness Adaptor Signatures

This chapter will define a variant of the Adaptor Signature Scheme as shown in definition 2.19. This new variant is explicitly tailored to meet the requirements of being applicable in the scenario of two-party signature protocols that one can construct for Signature Schemes such as Schnorr [**?**]. 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 create a signature cooperatively, without revealing their key share to the other party. The advantage of our Adaptor Signature Scheme in comparison to the original definition is that we do not need to introduce an additional pre-signature step in the two-party scenario mentioned above. 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 more straightforward protocol. In particular, our protocol will allow one of the two parties to mask his signature share with a witness value x of $(x, X) \in R$ (where R is a hard relation as of definition 2.1). The second party (knowing X, but not x) can verify that x is indeed contained in the peer's partial signature. To complete the final signature, the party knowing x has to first replace his partial signature (masked with x) with the original unmasked version, which corresponds to the adapting step of the original Adaptor Signature Scheme definition 2.19. Having previously received the masked partial signature, the second party can now extract x from the final signature, his partial signature, and the other party's masked partial signature. One can then leverage this feature to build an Atomic Swap protocol, as shown in chapter 4.

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 protocol's final signatures fulfill the same properties as regular Schnorr signatures seen in [**?**] and prove its correctness. From this two-party protocol, we then derive the adapted variant already mentioned before. We

start by defining our extended Signature Scheme in section 3.1, proceed by providing a Schnorr-based instantiation of the protocol in section 3.2 and finally prove its correctness and security in section 3.4.

3.1 Definitions

A Two-Party Signature Scheme is an extension of a Signature Scheme shown in definition 2.3, 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 revealing 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 3.1 (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. It 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 and update parameters with each other during the protocol execution. Note that this context always has to be consistent between the two parties. If Alice were to update Λ, she has to send the updated version to Bob to continue the 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 (similar for Bob), and 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 participant's public key pk_A (pk_B for Bob's 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 2.8. 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 [**?**].

From definition 3.1, we now derive a Two-Party Adaptor Signature Scheme Φ_{Apt}, allowing one of the participants to hide a secret witness value inside his partial signature.

Definition 3.2 (Two-Party Fixed Witness Adaptor Signature Scheme)**.** Given a pair $(x, X) \in R$, (where R is a hard relation as of definition 2.1) a Two-Party Fixed Witness Adaptor Signature Scheme Φ_{Apt} is an extension to Φ_{MP} with the following algorithms.

$$\Phi_{Apt} \;:=\; (\Phi_{MP} \;||\; \mathsf{mskSig} \;||\; \mathsf{vrfMskSig} \;||\; \mathsf{extWit})$$

- $\hat{\sigma_A} \leftarrow \mathsf{mskSig}(\tilde{\sigma_A}, x)$: The mask 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 a masked partial signature $\hat{\sigma_A}$ that can be verified to contain x using the $\mathsf{vrfMskSig}$ function without revealing x.

- $\{1, 0\} \leftarrow \mathsf{vrfMskSig}(\hat{\sigma_A}, m, pk_A, X)$: The masked signature verification algorithm is a DPT function that takes as input a masked partial signature $\hat{\sigma_A}$, the other participant's public key pk_A and a statement X. The function will verify the partial signature's validity and that it was masked with 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 and the final composite signature σ_{fin}. Consequently, only protocol participants knowing the partial signatures exchanged during the protocol can run this algorithm.

Similar to how it is defined in [**?**], additionally to regular Correctness, as described in definition 2.3, we require our Signature Scheme to satisfy Adaptor Signature Correctness. This property is given when one can complete every masked partial signature generated by mskSig into a final signature for all pairs $(x,\ X) \in R$. And it will then be possible to extract the witness computing extWit with the required parameters.

Definition 3.3 (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 $(X,\ x) \leftarrow \mathsf{genRel}(1^n)$ it must hold that:

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

$\mathsf{keyGen}(1^n)$	$\mathsf{sign}(m, sk)$	$\mathsf{verf}(m, \sigma, pk)$
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 3.1: Schnorr Signature Scheme as first defined in [**?**]

3.2 Schnorr-based Instantiation

We start by providing a general instantiation of a Signature Scheme (see definition 2.3): 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 2.4, and $m \in \{0,1\}^*$ is a message.

The reader can see a concrete implementation in fig. 3.1. The Signature Scheme is called Schnorr Signature Scheme, first defined in [**?**] 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 their 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 follows: As shown in fig. 3.1, 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 3.1. Note that this two-party variant of the scheme 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 an auxiliary function $\mathsf{setupCtx}$ to use for the instantiation:

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

$\mathsf{setupCtx}(\Lambda, pk_A, R_A)$

1 : $\Lambda.pk := \Lambda.pk \cdot pk_A$
2 : $\Lambda.R := \Lambda.R \cdot R_A$
3 : **return** Λ

This function helps the participants setting up and updating the signature context shared between them. In fig. 3.2, we show a concrete instantiation of the protocol and procedures.

In dKeyGen, Alice and Bob will each randomly chose their secret key and nonce. They further require to create a zero-knowledge proof attesting that they had generated their key before they exchanged any message. These proofs are 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 [**?**]. Another secure key generation setup for a Schnorr-based multi-signature protocol was found by Micali et al. in [**?**]. 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 set up the signature context and send it to Bob, together with her public key and zk-proof of knowledge. Bob verifies the proof and then proceeds by adding his parameters to the shared signature context and sending it back to Alice, together with his parameters, which Alice will verify.

We note here that this is only one possible way of securely computing the parties' keypairs and nonce values and setting up the shared context. Alternative methods 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 protocols. Another scenario is that both the keypairs and nonce values were generated by a protocol similar to dKeyGen beforehard. Still, the shared context Λ is not yet set up. In this case, the setupCtx can be incorporated into the signing protocol, as we shall see in section 3.3. 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 ordinary 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 adjustment affects that the signature shares themselfes are not yet a valid signature (neither under pk nor under pk_A or pk_B), and other means that signing can only start after the context Λ has been fully setup. This property is because to be valid under pk, the signature shares are missing the other participant's s value. 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.

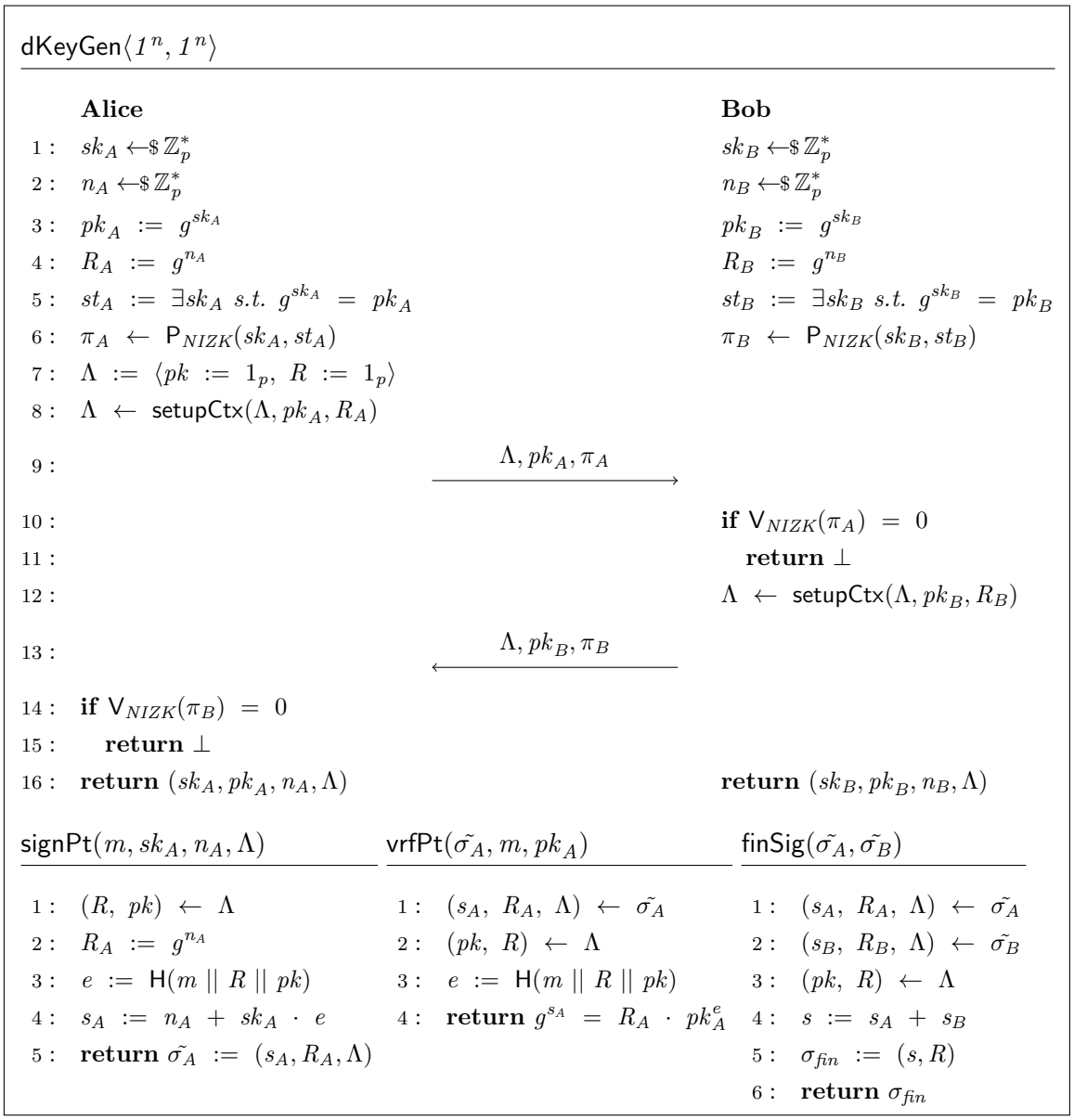

Figure 3.2: Two-Party Schnorr Signature Scheme

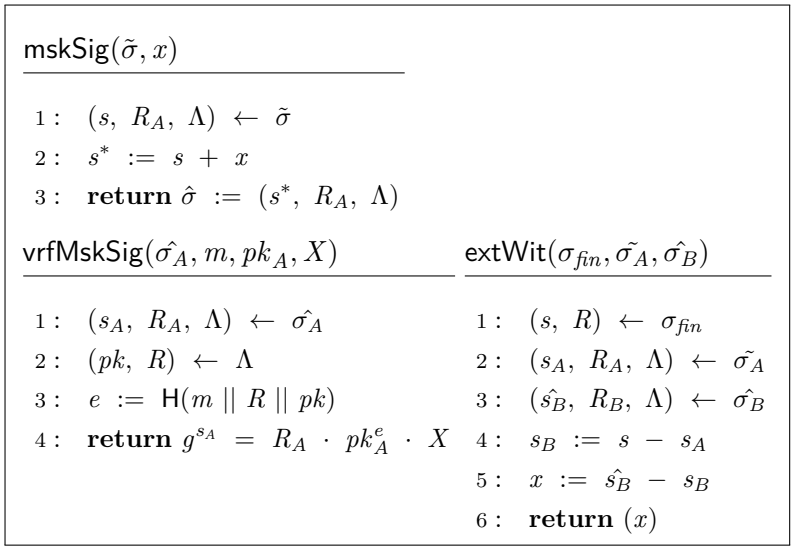

Figure 3.3: Fixed Witness Adaptor Schnorr Signature Scheme

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. [**?**].

In fig. 3.3, we further provide a Schnorr-based instantiation for the Fixed Witness Two-Party Adaptor Signature Scheme as defined in definition 3.2.

mskSig will add a secret witness x to the s value of the signature. Changing the partial signature this way means that it can't be verified using vrfPt any longer. Therefore, we introduce vrfMskSig, which takes as an additional parameter the statement X, which will be included in the verifier's equation. Now the function verifies the partial signature's validity and that it indeed has been masked with the witness value x, being the discrete logarithm of X. After obtaining σ_{fin}, we can then cleverly unpack the secret x, shown in the extWit function.

3.3 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. 3.2. **dAptSign** additionally uses the functionality of fig. 3.3 allowing one party to mask 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 and 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

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

	Alice		*Bob*
1 :	$\Lambda := \{pk := 1_p, R := 1_p\}$		
2 :	$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
3 :		$\xrightarrow{\Lambda, pk_A := g^{sk_A}}$	
4 :			$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
5 :			$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
6 :		$\xleftarrow{\tilde{\sigma_B}, \Lambda, pk_B := g^{sk_B}}$	
7 :	**if** $\mathsf{vrfPt}(\tilde{\sigma_B}, m, pk_B) = 0$		
8 :	**return** $\perp$		
9 :	$\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$		
10 :		$\xrightarrow{\tilde{\sigma_A}}$	
11 :			**if** $\mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A) = 0$
12 :			**return** $\perp$
13 :	$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$		$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$
14 :	$pk \leftarrow \Lambda.pk$		$pk \leftarrow \Lambda.pk$
15 :	**return** (σ_{fin}, pk)		**return** (σ_{fin}, pk)

Figure 3.4: Instantiation of the **dSign** protocol.

the signature context Λ has not yet been set up between the parties. The reason for this is that we are faced with precisely this scenario in the Mimblewimble transaction protocols, which we shall see later in section 4.2. Both parties input the shared message m as well as their secret keys and secret nonces. The reader can see the instantiation of the protocol in fig. 3.4. 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. 3.1.

We now define the **dAptSign** protocol between Alice and Bob, creating a signature σ_{fin} under the composite public key $pk := pk_A \cdot pk_B$. Bob will hide his secret x, which Alice can extract after completing the signing process. The concrete instantiation can be

seen in fig. 3.5. In this protocol, only Bob can call the signature finalization algorithm finSig for computing the final signature, which is different from the previous protocol, in which both could do so. The reason for this is that the function requires Bob's unmasked partial signature $\tilde{\sigma_B}$ as input, which Alice does not know. (Note that this corresponds to the adapting step of the original definition 2.19) 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.

3.4 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 definition 2.8.

3.4.1 Adaptor Signature Correctness

To prove that Adaptor Signature Correctness holds, we have three statements to prove as given by definition 3.3. First we demonstrate that $\mathsf{verf}(m, \sigma_{fin}, \Lambda.pk) = 1$ holds in our Schnorr-based instantiation of the Signature Scheme, where Λ is set up such that $pk = pk_A \cdot pk_B$.

Proof. For this proof, we assume the setup already specified in definition 3.3. The proof is by showing the equality of the equation checked by the verifier of the final signature by continuous substitutions on the left side of the equation:

$$g^s = R \cdot pk^e \tag{3.1}$$
$$g^{s_A} \cdot g^{s_B} \tag{3.2}$$
$$g^{n_A + e \cdot sk_A} \cdot g^{n_B + e \cdot sk_B} \tag{3.3}$$
$$g^{n_A} \cdot pk_A^e \cdot g^{n_B} \cdot pk_B^e \tag{3.4}$$
$$R_A \cdot pk_A^e \cdot R_B \cdot pk_B^e \tag{3.5}$$
$$R \cdot pk^e = R \cdot pk^e \tag{3.6}$$
$$1 = 1 \tag{3.7}$$

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

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

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

	Alice		*Bob*
1 :	$\Lambda := \{pk := 1_p, R := 1_p\}$		
2 :	$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
3 :	$pk_A := g^{sk_A}$		
4 :		$\xrightarrow{\Lambda, pk_A}$	
5 :			$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
6 :			$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
7 :			$\hat{\sigma_B} \leftarrow \mathsf{mskSig}(\tilde{\sigma_B}, x)$
8 :			$pk_B := g^{sk_B}$
9 :			$X := g^x$
10 :		$\xleftarrow{\hat{\sigma_B}, \Lambda, pk_B, X}$	
11 :	**if** $\mathsf{vrfMskSig}(\tilde{\sigma_B}, m, pk_B, X) = 0$		
12 :	**return** $\perp$		
13 :	$\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$		
14 :		$\xrightarrow{\tilde{\sigma_A}}$	
15 :			**if** $\mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A) = 0$
16 :			**return** $\perp$
17 :			$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$
18 :		$\xleftarrow{\sigma_{fin}}$	
19 :	$pk \leftarrow \Lambda.pk$		$pk \leftarrow \Lambda.pk$
20 :	**if** $\mathsf{verf}(m, \sigma_{fin}, pk) = 0$		
21 :	**return** $\perp$		
22 :	$x \leftarrow \mathsf{extWit}(\sigma_{fin}, \tilde{\sigma_A}, \hat{\sigma_B})$		
23 :	**return** $(x, (\sigma_{fin}, pk))$		**return** (σ_{fin}, pk)

Figure 3.5: Instantiation of the **dAptSign** protocol.

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

$$g^{\hat{\sigma_B}} = R_B \cdot pk_B^e \cdot X \tag{3.8}$$
$$g^{\tilde{\sigma_B} + x} \tag{3.9}$$
$$g^{n_B + sk_B \cdot e + x} \tag{3.10}$$
$$g^{n_B} \cdot g^{sk_B \cdot e} + g^x \tag{3.11}$$
$$R_B \cdot pk_B^e \cdot X = R_B \cdot pk_B^e \cdot X \tag{3.12}$$
$$1 = 1 \tag{3.13}$$

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 masked partial signature

$$x = \hat{s_B} - (s - s_A) \tag{3.14}$$
$$\hat{s_B} - ((s_A + s_B) - s_A) \tag{3.15}$$
$$s_B + x - (s_B) \tag{3.16}$$
$$x = x \tag{3.17}$$
$$1 = 1 \tag{3.18}$$

□

3.4.2 Security

We have shown that the outlined Signature Scheme is correct. Next, we have to prove its security. Our goal is to prove security in the malicious setting (as shown in definition 2.8) 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 Yehuda Lindell laid out in [**?**]. 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 statement 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 [**?**]. A secure zero-knowledge proof must fulfill Completeness, Soundness and Zero-Knowledge properties, which are defined, for instance, in [**?**].

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 first to send X_1 and then X_2.

Proof idea: 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. 3.6 with the newly added lines marked in blue. We note here that by making those adjustments to the original protocol, we now no longer prove security of the original protocol but rather the adjusted one. This circumstance does not mean that the original protocol is insecure. Still, if one wants to implement a version of the protocol proven to be secure, it should include the calls added in the adjusted protocol. The same argument holds for all of the modified protocols from this section.

In the adjusted protocol, both Alice and Bob will verify the validity of the other party's public key and nonce Commitments. They will stop protocol execution in case the peer sent an invalid value. 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 set up securely, 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 the protocol's security 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 Alice and Bob's inputs. The simulator's task will be to extract the inputs used by $\mathcal{A}$ such that he can call the TTP and receive the outputs. From this output, the simulator $\mathcal{S}$ has to construct a transcript that is indistinguishable from a protocol transcript in the real world in which a deterministic polynomial adversary $\mathcal{A}$ controls the corrupted party. The simulator uses the calls to f_{zk}^{R} 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: 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}.

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

	Alice		*Bob*
1 :	$\Lambda := \{pk := 1_p, R := 1_p\}$		
2 :	$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
3 :	$\mathsf{f}^R_{zk}((sk_A, pk_A))$		
4 :	$\mathsf{f}^R_{zk}((n_A, R_A))$		
5 :		$\Lambda, pk_A, R_A \longrightarrow$	
6 :			$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
7 :			$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
8 :			**if** $\mathsf{f}^R_{zk}(pk_A) = 0 \vee$
			$\mathsf{f}^R_{zk}(\Lambda.R) = 0$
9 :			**return** $\perp$
10 :			$\mathsf{f}^R_{zk}((sk_B, pk_B))$
11 :			$\mathsf{f}^R_{zk}((n_B, R_B))$
12 :		$\tilde{\sigma_B}, \Lambda, pk_B \longleftarrow$	
13 :	**if** $\mathsf{f}^R_{zk}(pk_B) = 0 \vee$		
	$\mathsf{f}^R_{zk}(\Lambda.R \cdot R_A^{-1}) = 0$		
14 :	**return** $\perp$		
15 :	**if** $\mathsf{vrfPt}(\tilde{\sigma_B}, m, pk_B) = 0$		
16 :	**return** $\perp$		
17 :	$\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$		
18 :		$\tilde{\sigma_A} \longrightarrow$	
19 :			**if** $\mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A) = 0$
20 :			**return** $\perp$
21 :	$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$		$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$
22 :	$pk \leftarrow \Lambda.pk$		$pk \leftarrow \Lambda.pk$
23 :	**return** (σ_{fin}, pk)		**return** (σ_{fin}, pk)

Figure 3.6: Adjustment to the **dSign** protocol seen in fig. 3.4

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).

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 is identical to what one would expect in an actual 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 $\mathcal{A}$ 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, the simulator will notice it, halt, and output whatever $\mathcal{A}$ outputs. If $\mathcal{A}$ behaves correctly, $\mathcal{S}$ will play the protocol until the end and finally output whatever

$\mathcal{A}$ outputs. So both in the case that $\mathcal{A}$ acts honestly and if he does not, $\mathcal{A}$'s and $\mathcal{S}$'s outputs will be indistinguishable.

We have shown that the distributions of transcript messages are indistinguishable in every phase of the protocol if 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 $\mathsf{H}(m \;||\; R_A \;\cdot\; R_B \;||\; pk_A \;\cdot\; pk_B)$ 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 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 $\tilde{\sigma_A} \;\leftarrow\; \mathsf{signPt}(m, sk_A, n_A, \Lambda)$ as defined in the protocol 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 laid 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 extent 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 is the correct one (other than guessing).

- *Phase 2:* When $\mathcal{A}$ calls f_{zk}^R with the proper parameters sent to him, he will still receive 1 back, or 0 otherwise, which is the same as would be expected in an actual

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 Alice's and Bob's input values of which at least Alice input is chosen randomly by $\mathcal{S}$. As dependent on random tape, the calculation output will 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 hash function's Pre-image Resistance property. The remaining messages are identical to what would be expected in a real execution due to the deterministic nature of $\mathcal{A}$.

- *Phase 3:* The simulator will verify the values sent to him by $\mathcal{A}$ and will halt and output $\perp$ if 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 modeled by $\mathcal{S}$ sending abort to the TTP. Otherwise, $\mathcal{S}$ will calculate his part of the partial signature as defined by the protocol. Therefore, it will be found to be valid by $\mathcal{A}$ and will complete to σ_{fin} with finSig, because of the fixed, calculated Schnorr challenge $\mathcal{S}$ computed 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 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 adversary's 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. Therefore, we can 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. 3.7.

Theorem 2. Assume we have two key pairs (sk_A, pk_A) and (sk_B, pk_B) which were set up 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.

Proof. We proof the security of **dAptSign** by constructing a simulator $\mathcal{S}$ which is given the output (σ_{fin}, pk) (resp. $(x, (\sigma_{fin}, pk))$) from a TTP that securely computes the protocol in the ideal world after receiving the inputs from Alice and Bob. The simulator's task

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

	Alice		*Bob*
1 :	$\mathsf{f}^R_{zk}((sk_A, pk_A))$		
2 :	$\mathsf{f}^R_{zk}((n_A, R_A))$		
3 :	$\Lambda := \{pk := 1_p, R := 1_p\}$		
4 :	$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
5 :		$\xrightarrow{\Lambda, pk_A, R_A}$	
6 :			**if** $\mathsf{f}^R_{zk}(pk_A) = 0 \vee$ $\mathsf{f}^R_{zk}(\Lambda.R) = 0$
7 :			**return** $\perp$
8 :			$\mathsf{f}^R_{zk}((sk_B, pk_B))$
9 :			$\mathsf{f}^R_{zk}((n_B, R_B))$
10 :			$\mathsf{f}^R_{zk}((x, X))$
11 :			$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
12 :			$\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
13 :			$\hat{\sigma_B} \leftarrow \mathsf{mskSig}(\tilde{\sigma_B}, x)$
14 :			$pk_B := g^{sk_B}$
15 :			$X := g^x$
16 :		$\xleftarrow{\tilde{\sigma_B}, \Lambda, pk_B, X}$	
17 :	**if** $\mathsf{vrfMskSig}(\tilde{\sigma_B}, m, pk_B, X) = 0$		
18 :	**return** $\perp$		
19 :	**if** $\mathsf{f}^R_{zk}(pk_B) = 0 \vee$ $\mathsf{f}^R_{zk}(\Lambda.R \cdot R_A^{-1}) = 0 \vee$ $\mathsf{f}^R_{zk}(X) = 0 \vee$		
20 :	**return** $\perp$		
21 :	$\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$		
22 :		$\xrightarrow{\tilde{\sigma_A}}$	
23 :			**if** $\mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A) = 0$
24 :			**return** $\perp$
25 :			$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$
26 :		$\xleftarrow{\sigma_{fin}}$	
27 :	$pk \leftarrow \Lambda.pk$		$pk \leftarrow \Lambda.pk$
28 :	**if** $\mathsf{verf}(m, \sigma_{fin}, pk) = 0$		
29 :	**return** $\perp$		
30 :	$x \leftarrow \mathsf{extWit}(\sigma_{fin}, \tilde{\sigma_A}, \hat{\sigma_B})$		
31 :	**return** $(x, (\sigma_{fin}, pk))$		**return** (σ_{fin}, pk)

Figure 3.7: Adjustments to the **dAptSign** protocol seen in fig. 3.5

again is to extract the adversary's inputs and send them to the trusted third party to receive the protocol outputs and construct a transcript that is indistinguishable from the protocol transcript in the real world from this output. The simulator uses the calls to f_{zk}^R 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 prove 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 the simulator compares 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 3.4.2, 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 simulator's protocol output is equivalent to $\mathcal{A}$'s expected 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 this phase's transcript must be indistinguishable from a real transcript.

- *Phase 2:* In this phase, $\mathcal{S}$ sends $X := g^x$ to $\mathcal{A}$ for which the simulator received x 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 laid out in Phase 1.

- *Phase 4:* Now, $\mathcal{A}$ expects to receive σ_{fin}, from which he can extract the witness x. Indeed he will receive σ_{fin} as $\mathcal{S}$ has received from the TTP, which would have been expected in an actual execution. Furthermore, it must 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. Similarly, 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, $\mathcal{S}$'s simulated transcript 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 described 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 he 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 3.4.2, in this phase, the adversary will receive some public, nonce Commitment, and signature context. As he does not know Alice's actual inputs, he has no way of knowing if the values received are the correct ones fitting with Alice's inputs, other than by guessing.

- *Phase 2:* As argued before, with the hash function modeled in the random oracle, its output is expected to be randomly distributed. As $\mathcal{S}$'s computation to create the hash output relies 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:* $\mathcal{S}$ will verify the equality of $\mathcal{A}$'s values with the variables saved to its memory before. In case he sent invalid values $\mathcal{A}$ should not receive the final outputs (σ_{fin}, pk), which is modeled by sending abort to the TTP. The same behavior 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 output whatever $\mathcal{A}$ outputs, 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 follows it, $\mathcal{S}$ will output whatever $\mathcal{A}$ outputs, therefore being equal to the expected output from $\mathcal{A}$ in an actual 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 2 therefore holds. □

CHAPTER 4

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

This section 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 [**?**]. 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, the notions of cut through, transaction merging, coin minting (coinbase transactions), and publishing transactions to the ledger, all discussed in section 2.4 and formalized by Fuchsbauer et al. in [**?**], will not be the 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 keys instead of just one, thereby enabling coins owned by multiple parties, which is similar to a mutlisig address in Bitcoin [**?**]. 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 to provide the functionality that is later needed to build the final Atomic Swap protocol, which we will introduce in section 4.5.

We will proceed by providing an instantiation of the three transaction schemes in sec-

tion 4.2, which can be implemented and deployed on a Mimblewimble based cryptocurrency such as Beam or Grin. In section 4.3, we define two-party protocols from the outlined schemes to construct Mimblewimble transactions. Section 4.4 shows the proofs that the formalizations are correct and the protocols secure in the malicious setting as shown in definition 2.8. Finally, in section 4.5, we describe an 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.

4.1 Definitions

As we have already discussed in section 2.4 for creating 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 described in definition 3.1, a Range Proof System as shown in definition 2.14 such as Bulletproofs, as described in section 2.3.2 and a homomorphic Commitment Scheme COM as defined in definition 2.6 such as Pedersen Commitments seen in definition 2.7.

Fuchsbauer et al. have defined three procedures, Send, Rcv, and Ldgr, regarding creating a transaction. Send called by the sender will create a pre-transaction, Rcv takes the pre-transaction and adds the receiver's 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 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. A sender owning a set of input coins calls Send to create an initial pre-transaction signed already by the sender and then forwarded to the fund receiver. The receiver then calls Rcv to add his output coins with the correct value. His signature is then aggregated with the sender's 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 algorithms already laid out by Fuchsbauer et al. The main reason is that we are using the notion of two-party signatures as of definition 3.1 instead of aggregatable signatures, which are employed in their paper. While aggregatable signatures are a similar concept to the two-party signatures, we can find some essential differences. Ultimately, the two-party

signatures is, as we shall see, the more appropriate and secure choice for the formalization. First of all, we need to define the notion of an Aggregatable Signature Scheme:

Definition 4.1 (Aggregatable Signature Scheme)**.** A Signature Scheme Φ can be called aggregatable 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 an aggregated signature σ_a valid for the same message m under the composite public key $pk_a = pk_1 \cdot pk_2$

In the Schnorr Signature Scheme, we can only aggregate signatures by primitively 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 $pk_a = pk_1 \cdot pk_2$ [**?**].
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. s is calculated as $s = n + e \cdot sk$ where $e = \mathsf{H}(m \mid\mid R \mid\mid pk)$. As we have changed the nonce Commitment R and the public key pk in our aggregated signature the Schnorr challenge e will be different from the one used by the individual signers, thereby making the verification algorithm return 0. We can fix this issue by having the individual signers use the final composite R and pk for their Schnorr challenge calculation, which is exactly what we are doing in the Schnorr-based instantiation of the Two-Party Signature Scheme in fig. 3.2. This detail, however, introduces the necessity for an initial setup phase in which the parties exchange messages to compute R and pk from their shares. 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 and disregard the public key shares. We also note that the two-party version is currently implemented in Grin and Beam in practice.[1] Finally, there is another critical advantage that comes with the two-party Schnorr approach. For the peers to start the signing process, the final composite pk and nonce Commitment R need to be known. That also entails that the flow pointed out in [**?**], 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 receiver's turn. We need to introduce a third round. The sender receives the partially signed pre-transaction from the receiver, adds his partial signature and only now can finalize the signature and thereby the transaction. While having to add an additional round would seem like an inconvenience at first, we discover that we avoid being vulnerable to a *Transaction Sniff Attack* by doing so.

For the following attack to be possible, we need to assume that the channel between the sender (Alice) and receiver (Bob) has been compromised therefore can no longer be considered secure. We show that under this assumption, the formalization laid out by Fuchsbauer et al. would be vulnerable to the *Transaction Sniff Attack*. In contrast our formalization would still be secure.

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

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 Bob's service. 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 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. In the meantime, the malicious attacker managed to sniff ptx sent by Alice. Already containing Alice's signature, all the attacker has to do is guess the value Alice might want to send, create an output coin with that value, add his 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 spent and are now in possession of the attacker.

Starting the signing process only at the receiver's turn and introducing a third-round solves this issue because Alice adds the signature for her input coins only at the last step. Using the Two-Party Signature Scheme instead of an Aggregatable Signature Scheme forces us to make this change because of the additional setup phase required. Even if the attacker were able to sniff one of the pre-transactions exchanged between the parties, because Alice will only ever add the signature for her input coins at the 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 $sp\mathcal{C}$ that represents a spendable coin and contains a reference to the coin Commitment C, range proof π, and its (secret) spending information 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 4.2 (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^* and Bob's new spendable output coin $sp\mathcal{C}_B^*$, 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 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 the same as defined in the paper by Fuchsbauer et al. [**?**], we still add it here for completeness. Note that in their work, one can find it under the name MW.Ver. We rename 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 algorithm 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 the public key, which is calculated by summing up the output coins and subtracting the input coins. (See section 2.4)

We say a Mimblewimble Transaction Scheme is correct if the verification algorithm verfTx returns 1 upon providing a transaction that is well balanced and contains a valid signature. More formally:

Definition 4.3 (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:

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

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

Definition 4.4 (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 distributed protocols:

$$MW_{ext}[COM, \Phi_{MP}, \Pi_{RP-MP}] := \\ MW[COM, \Phi_{MP}, \Pi_{RP-MP}] \;||\; (\mathbf{dSpendCoins}, \mathbf{dRecvCoins}, \mathbf{dFinTx})$$

Note that for this scheme, we require a Two-Party Range Proof System Π_{RP-MP} as shown in definition 2.15. Specifically, we need the 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 both Alice and Carol own a coin $\mathcal{C}$, 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 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. $sp\mathcal{C}_C^*$), and 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 an 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 [**?**].

- $\langle tx, tx \rangle \leftarrow \mathbf{dFinTx}\langle (ptx, sk_A, n_A), (ptx, sk_C, n_C) \rangle$: The distributed finalized transaction protocol has to be used to create 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 similarly to the standard scheme:

Definition 4.5 (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:

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

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

Definition 4.6 (Contract Mimblewimble Transaction Scheme)**.** The contract version of the Mimblewimble Transaction Scheme updates the Extended Mimblewimble Transaction Scheme by providing a modified version of the single party receive routine and the distributed finalize transaction protocol.

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

• $(ptx^*, sp\mathcal{C}_B^*, \tilde{\sigma_B}) \leftarrow \mathsf{aptRecvCoins}(ptx, p, x)$: The contract variant of the receive function takes an additional input, a secret witness value x, hidden in the transaction signature and extracted by the other party after the completion of the protocol. Note that the routine also returns Bob's unmasked partial signature. The reason for this is that we later need the unmasked version to complete the signature and finalze the transaction. By not sharing this unmasked signature with Alice, Bob is the one who gets to finalize the transaction, which is different from the simpler protocol and is a crucial feature necessary for our Atomic Swap protocol. We want to stress here that aptRecvCoins is only a single-party algorithm. We can only use it if we're going to create an output coin owned by a single receiver. It would, of course, be conceivable also to 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.

• $\langle \tilde{\sigma_{AB}}, tx \rangle \leftarrow \mathbf{dAptFinTx}\langle (ptx^*, sk_A, n_A, X), (ptx^*, sk_B, n_B, \tilde{\sigma_B}) \rangle$: The finalize transaction algorithm's contract variant 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 unmasked version of his partial signature as it is required for transaction completion. This protocol could also be implemented as a three-party protocol with two senders controlling a shared coin and a third receiver. However, in our case, which we will describe later in section 4.3, 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 can finalize the transaction, which is different from finTx and **dFinTx**. The reason for that is 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 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. That is why the protocol does not return the final transaction tx to Alice. Instead, the protocol will output the sender's partial signature, which Alice can later use to extract the final transaction's witness value.

Similar to before, we define Correctness for the adapted scheme:

Definition 4.7 (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:

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

4.2 Instantiation

This section will provide an instantiation of the Transaction Scheme definitions found in definition 4.2, definition 4.4, and definition 4.6. One can implement the instantiations in a cryptocurrency based on the Mimblewimble protocol such as Beam and Grin.

4.2.1 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 protocol's execution we assume to have access to a homomorphic Commitment Scheme such as Pedersen Commitment COM, shown in definition 2.7. Furthermore, we require a Range Proof System Π_{RP} as described in definition 2.14 and a Two-Party Signature Scheme Φ_{MP} as of definition 3.1.

To make the pseudocode for the transaction protocol easier to read, we first introduce two auxiliary functions $\mathsf{createCoin}$ and $\mathsf{createTx}$. 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 $\mathsf{createTx}$ 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 range proofs $[\pi]$, 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 fig. 4.1 and fig. 4.2, we provide an instantiation of the Mimblewimble Transaction 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 transferred to a receiver. He sets up the signature context with his parameters and gets a pre-transaction ptx, newly created spendable output coin $sp\mathcal{C}_A$, and 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. [**?**] in that the sender does not yet sign the transaction during spendCoins, because we are using a Two-Party Signature Scheme definition 3.1 instead of an aggregatable signature scheme definition 4.1.

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 that the receiver can send back to the sender, and the newly created spendable output $sp\mathcal{C}_B$.

Now in finTx, the original sender will validate the updated pre-transaction 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.

4.2.2 Extended Mimblewimble Transaction Scheme

Figure 4.3 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 do 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).

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

1 : $v \leftarrow \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.v)$
2 : **if** $p > v$ **return** $\perp$
3 : **if** $\exists\, i \neq j : sp\mathcal{C}[i] = sp\mathcal{C}[j]$ **return** $\perp$
4 : $m := \{0,1\}^*$
5 : $(r_A^*, n_A) \leftarrow\$\, \mathbb{Z}_p^*$
6 : $sp\mathcal{C}_A^* \leftarrow \mathsf{createCoin}(v - p, r_A^*)$
7 : $\{\mathcal{C}_{out}^A, r_A^*, v_A, \pi_A\} \leftarrow sp\mathcal{C}_A^*$
8 : $sk_A := r_A^* - \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.r)$
9 : $\Lambda := \{pk := 1_p, R := 1_p\}$
10 : $\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$
11 : $ptx \leftarrow \mathsf{createTx}(m, sp\mathcal{C}.C, [\mathcal{C}_{out}^A], [\pi_A], \Lambda, [g^{sk_A}], \emptyset)$
12 : **return** $(ptx, sp\mathcal{C}_A^*, (sk_A, n_A))$

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

1 : $(m, inp, out, \Pi, \Lambda, com, \emptyset, t) \leftarrow ptx$
2 : **if** $\mathsf{vrfRanPrf}(\Pi[0], out[0]) = 0$
3 : **return** $\perp$
4 : $(r_B^*, n_B) \leftarrow\$\, \mathbb{Z}_p^*$
5 : $sp\mathcal{C}_B^* \leftarrow \mathsf{createCoin}(p, r_B^*)$
6 : $\{\mathcal{C}_{out}^B, r_B^*, v_B, \pi_B\} \leftarrow sp\mathcal{C}_B^*$
7 : $sk_B := r_B^*$
8 : $\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
9 : $\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, n_B, \Lambda)$
10 : $ptx \leftarrow \mathsf{createTx}(m, inp, out \,||\, \mathcal{C}_{out}^B, \Pi \,||\, \pi_B, \Lambda, com \,||\, g^{sk_B}, \tilde{\sigma_B})$
11 : **return** $(ptx, sp\mathcal{C}_B^*)$

Figure 4.1: Instantiation of Mimblewimble Transaction Scheme part 1.

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

1 : $(m, inp, out, \Pi, \Lambda, com, \tilde{\sigma_B}, t) \leftarrow ptx$
2 : **if** $\mathsf{vrfRanPrf}(\Pi[1], out[1]) = 0$
3 : **return** $\perp$
4 : **if** $\mathsf{vrfPt}(\tilde{\sigma_B}, m, com[1]) = 0$
5 : **return** $\perp$
6 : $\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$
7 : $\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$
8 : $tx \leftarrow \mathsf{createTx}(m, inp, out, \Pi, \Lambda, com, \sigma_{fin})$
9 : **return** tx

$\mathsf{verfTx}(tx)$

1 : $(m, inp, out, \Pi, \Lambda, com, \sigma, t) \leftarrow tx$
2 : $\mathcal{E} = \sum(out) - \sum(inp)$
3 : **return** $(\forall\, i \neq j : inp[i] \neq inp[j] \wedge out[i] \neq out[j])$ and
$inp \cup out = \emptyset$ and $(\forall\, i : \mathsf{vrfRanPrf}(\Pi[i], out[i]))$ and $\mathsf{verf}(m, \sigma, \mathcal{E})$

Figure 4.2: Instantiation of Mimblewimble Transaction Scheme part 2.

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

	Alice		*Carol*
1 :	$v \leftarrow \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.v)$		$v \leftarrow \sum_{i\,:=\,0}^{i\,<\,n} (sp\mathcal{C}_i.v)$
2 :	**if** $p > v \vee \exists\, i \neq j : psp\mathcal{C}_A[i] = psp\mathcal{C}_A[j]$		**if** $p > v \vee \exists\, i \neq j : psp\mathcal{C}_A[i] = psp\mathcal{C}_A[j]$
3 :	**return** $\perp$		**return** $\perp$
4 :	$v_{rem} = v - p$		
5 :	$v_A, v_C \leftarrow \{0, v_{rem}\}$ s.t. $v_A + v_C = v_{rem}$		
6 :		$\xrightarrow{\quad v_C \quad}$	
7 :	$m := \{0,1\}^*$		
8 :	$(r_A^*, n_A) \leftarrow\!\$\, \mathbb{Z}_p^*$		$(r_C^*, n_C) \leftarrow\!\$\, \mathbb{Z}_p^*$
9 :	$sp\mathcal{C}_A^* \leftarrow \mathsf{createCoin}(v_A, r_A^*)$		$sp\mathcal{C}_C^* \leftarrow \mathsf{createCoin}(v_C, r_C^*)$
10 :	$\{\mathcal{C}_{out}^A, r_A^*, v_A, \pi_A\} \leftarrow sp\mathcal{C}_A^*$		$\{\mathcal{C}_{out}^C, r_C^*, v_C, \pi_C\} \leftarrow sp\mathcal{C}_C^*$
11 :	$sk_A := r_A^* - \sum[r_A]$		$sk_C := r_C^* - \sum[r_C]$
12 :	$\Lambda := \{pk := 1_p, R := 1_p\}$		
13 :	$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{n_A})$		
14 :	$ptx \leftarrow$ $\mathsf{createTx}(m, [\mathcal{C}_{inp}], [\mathcal{C}_{out}^A], [\pi_A], \Lambda, [g^{n_A}], \emptyset)$		
15 :		$\xrightarrow{\quad ptx \quad}$	
16 :			$(m, inp, out, \Pi, \Lambda, com, t) \leftarrow ptx$
17 :			**if** $\mathsf{vrfRanPrf}(\Pi[0], out[0]) = 0$
18 :			**return** $\perp$
19 :			$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_C}, g^{n_C})$
20 :			$ptx' \leftarrow \mathsf{createTx}(m, inp, out \,||\, \mathcal{C}_{out}^C, \pi \,||\, \pi_C, \Lambda, com \,||\, g^{n_C}, \emptyset)$
21 :		$\xleftarrow{\quad ptx' \quad}$	
22 :	**if** $\mathsf{vrfRanPrf}(ptx'.\Pi[1], ptx'.out[1]) = 0$		
23 :	**return** $\perp$		
24 :	**return** $(ptx', sp\mathcal{C}_A^*, (sk_A, n_A))$		**return** $(ptx', sp\mathcal{C}_C^*, (sk_C, n_C))$

Figure 4.3: Extended Mimblewimble Transaction Scheme - **dSpendCoins**

Figure 4.4 shows an instantiation of the **dRecvCoins** function of the Extended Mimblewimble Transaction Scheme. Calling this protocol, two receivers, Bob and Carol, want to create a receiving shared coin $\mathcal{C}_{out}^{sh}$ with value p and key shares (r_A, r_C). The protocol starts by both receivers verifying the sender's output(s). Bob begins 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 the updated shared coin's validity, after which the two parties engage in two two-party protocols to create their partial signature and coin range proof. Finally, they make the updated pre-transaction *ptx* which can be sent back to the transaction sender.

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

	Bob		*Carol*
1 :	$(m, inp, out, \Pi, \Lambda, com, \emptyset, t) \leftarrow ptx$		
2 :	**foreach** *out as* $(i => \mathcal{C}_{out})$		**foreach** *out as* $(i => \mathcal{C}_{out})$
3 :	**if** $\mathsf{vrfRanPrf}(\Pi[i], \mathcal{C}_{out}[i]) = 0$		**if** $\mathsf{vrfRanPrf}(\Pi[i], \mathcal{C}_{out}[i]) = 0$
4 :	**return** $\perp$		**return** $\perp$
5 :	$(r_B^*, n_B) \leftarrow\$ \mathbb{Z}_p^*$		
6 :	$(\mathcal{C}_{out}^{sh}, \cdot, \cdot, \cdot) \leftarrow \mathsf{createCoin}(p, r_B^*)$		
7 :	$sk_B := r_B^*$		
8 :		$\xrightarrow{ptx, \mathcal{C}_{out}^{sh}}$	
9 :			$(r_C^*, n_C) \leftarrow\$ \mathbb{Z}_p^*$
10 :			$sk_C := r_C^*$
11 :			$\mathcal{C}_{out}^{sh\,\prime} := \mathcal{C}_{out}^{sh} \cdot g^{r_C}$
12 :			$ptx' := ptx$
13 :			$ptx'.out[] := \mathcal{C}_{out}^{sh\,\prime}$
14 :		$\xleftarrow{ptx', g^{sk_C}}$	
15 :	$\{\cdots \mathcal{C}_{out}^{sh\,\prime}\} \leftarrow ptx'.out$		
16 :	**if** $\mathcal{C}_{out}^{sh\,\prime} \neq \mathcal{C}_{out}^{sh} \cdot g^{sk_C}$		
17 :	**return** $\perp$		
18 :	$\pi_{BC} \leftarrow \mathbf{dRanPrf}(\mathcal{C}_{out}^{sh\,\prime}, p, sk_B)$		$\pi_{BC} \leftarrow \mathbf{dRanPrf}(\mathcal{C}_{out}^{sh\,\prime}, p, sk_C)$
19 :	$psp\mathcal{C}_B^* := \{\mathcal{C}_{out}^{sh}, p, r_B^*, \pi_{BC}\}$		$psp\mathcal{C}_C^* := \{\mathcal{C}_{out}^{sh}, p, r_C^*, \pi_{BC}\}$
20 :	$(\sigma_{\tilde{BC}}, pk_{BC}) \leftarrow \mathbf{dSign}(m, sk_B, n_B)$		$(\sigma_{\tilde{BC}}, pk_{BC}) \leftarrow \mathbf{dSign}(m, sk_C, n_C)$
21 :	$(\cdot, \cdot, \Lambda^*) \leftarrow \sigma_{\tilde{BC}}$		$(\cdot, \cdot, \Lambda^*) \leftarrow \sigma_{\tilde{BC}}$
22 :	$\Lambda' \leftarrow \mathsf{setupCtx}(\Lambda, \Lambda^*.pk, \Lambda^*.R)$		$\Lambda' \leftarrow \mathsf{setupCtx}(\Lambda, \Lambda^*.pk, \Lambda^*.R)$
23 :	$ptx^* \leftarrow$		$ptx^* \leftarrow$
24 :	$\mathsf{createTx}(m, inp, out \mid\mid \mathcal{C}_{out}^{sh\,\prime}, \Pi \mid\mid \pi_{BC}, \Lambda', com \mid\mid pk_{BC}, \sigma_{\tilde{BC}})$		$\mathsf{createTx}(m, inp, out \mid\mid \mathcal{C}_{out}^{sh\,\prime}, \Pi \mid\mid \pi_{BC}, \Lambda', com \mid\mid pk_{BC}, \sigma_{\tilde{BC}})$
25 :	**return** $(ptx^*, psp\mathcal{C}_B^*)$		**return** $(ptx^*, psp\mathcal{C}_C^*)$

Figure 4.4: Extended Mimblewimble Transaction Scheme - **dRecvCoins**

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

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

Figure 4.5: Extended Mimblewimble Transaction Scheme - **dFinTx**

Finally, fig. 4.5 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. They can combine the partial signatures into the final one and finalize the transaction.

4.2.3 Contract Mimblewimble Transaction Scheme

Figure 4.6 shows an instantiation of the aptRecvCoins algorithm. Before updating the pre-transaction *ptx*, Bob masks his partial signature with the witness value x. The procedure then returns the pre-transaction *ptx* containing Bob's masked partial signature and the statement X, a Commitment to the witness value x.

In fig. 4.7, 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 Alice cannot finalize the signature (and consequently the transaction) as she only knows Bob's masked signature share ($\hat{\sigma_B}$), 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 $\tilde{\sigma_{AB}}$ to retrieve x.

4.3 Protocols

This section specifies three protocols to build Mimblewimble transactions from the definitions found in section 4.1. Later in section 4.4, we will prove the security of these protocols, and finally, in section 4.5, we will utilize them to build the Atomic Swap.

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

1 : $(m, inp, out, \Pi, \Lambda, com, \emptyset, t) \leftarrow ptx$
2 : **if** $\mathsf{vrfRanPrf}(\Pi[0], out[0]) = 0$
3 : **return** $\perp$
4 : $(r_B^*, n_B) \leftarrow\$ \mathbb{Z}_p^*$
5 : $(\mathcal{C}_{out}^B, \pi_B) \leftarrow \mathsf{createCoin}(p, r_B^*)$
6 : $sk_B := r_B^*$
7 : $\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_B}, g^{n_B})$
8 : $\tilde{\sigma_B} \leftarrow \mathsf{signPt}(m, sk_B, \Lambda.pk, \Lambda.R)$
9 : $\hat{\sigma_B} \leftarrow \mathsf{mskSig}(\tilde{\sigma_B}, x)$
10 : $ptx \leftarrow \mathsf{createTx}(m, inp, out \,||\, \mathcal{C}_{out}^B, \Pi \,||\, \pi_B, \Lambda, com \,||\, g^{n_B}, \hat{\sigma_B})$
11 : **return** $(ptx, (\mathcal{C}_{out}^B, r_B^*), \tilde{\sigma_B})$

Figure 4.6: Contract Mimblewimble Transaction Scheme - aptRecvCoins.

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

	Alice	*Bob*
1 :	$(m, inp, out, \Pi, \Lambda, com, \hat{\sigma_B}, t) \leftarrow ptx$	$(m, inp, out, \Pi, \Lambda, com, \hat{\sigma_B}, t) \leftarrow ptx$
2 :	**if** $\mathsf{vrfRanPrf}(\Pi[1], out[1]) = 0$	
3 :	**return** $\perp$	
4 :	**if** $\mathsf{vrfMskSig}(\tilde{\sigma_B}, m, com[1], X) = 0$	
5 :	**return** $\perp$	
6 :	$\sigma_{\tilde{AB}} \leftarrow$ **dSign**(m, sk_A, n_A)	$\sigma_{\tilde{AB}} \leftarrow$ **dSign**(m, sk_B, n_B)
7 :		$\sigma_{fin} \leftarrow \mathsf{finSig}(\sigma_{\tilde{AC}}, \tilde{\sigma_B})$
8 :		$tx \leftarrow \mathsf{createTx}(m, inp, out, \Pi, \Lambda, com, \sigma_{fin})$
9 :	**return** $\sigma_{\tilde{AB}}$	**return** tx

Figure 4.7: Adapted Extended Mimblewimble Transaction Scheme - **dAptFinTx**.

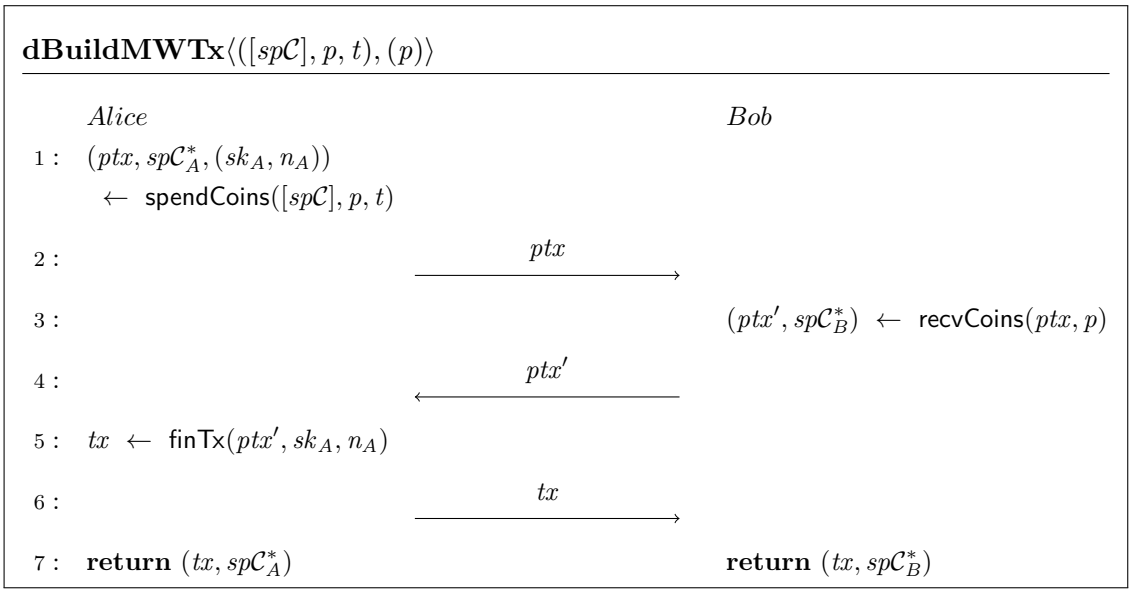

Figure 4.8: **dBuildMWTx** two-party protocol to build a new transaction

4.3.1 Simple Mimblewimble Transaction - dBuildMWTx

dBuildMWTx is a protocol between a sender and receiver that builds a Mimblewimble transaction transferring the value p from the sender to a receiver for a Mimblewimble Transaction Scheme as described in definition 4.2. 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.

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

Figure 4.8 shows the implementation of the protocol.

4.3.2 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 Extended Mimblewimble Transaction Scheme from definition 4.4. 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 $pspC^*_A$, $pspC^*_B$.

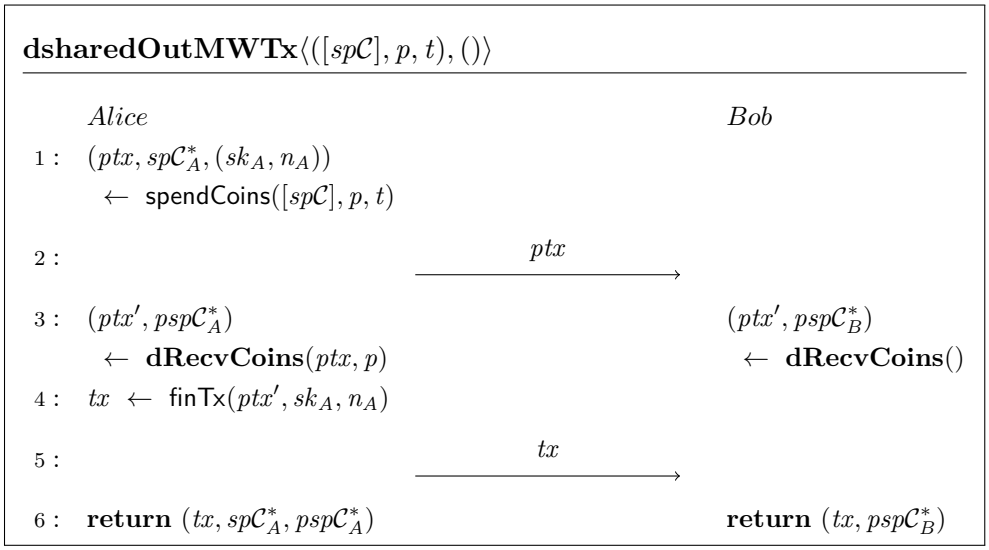

Figure 4.9: **dsharedOutMWTx** two-party protocol to build a new transaction with a shared output

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

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

Figure 4.9 shows the implementation of the protocol.

4.3.3 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 two parties to a receiver again for the Extended Mimblewimble Transaction Scheme outlined in definition 4.4 As input, it takes a list of partial spendable coins $[pspC_A]$, a transaction value p, an optional timelock t from the sender, and the other part of the shared spendable coins $pspC_B$. It takes the same transaction value p from the receiver. It outputs a final transaction tx to both parties and the new outputs spC^*_A, spC^*_B to the respective owner.

$$\langle (tx, spC^*_A), (tx, spC^*_B) \rangle \leftarrow \textbf{dsharedInpMWTx}\langle ([pspC_A], p, t), ([pspC_B], p) \rangle$$

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

Figure 4.10 shows the implementation of the protocol.

dsharedInpMWTx$\langle([pspC_A], p, t), ([pspC_B], p)\rangle$

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

Figure 4.10: **dsharedOutMWTx** two-party protocol to build a new transaction from a shared output

4.3.4 Contract Mimblewimble Transaction - dcontractMWTx

dcontractMWTx is a protocol between a sender and a receiver for the Contract Mimblewimble Transaction Scheme shown in definition 4.6. 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 3.2 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 execution of a primitive contract.

$$\langle(tx, spC^*_A, x), (tx, spC^*_B)\rangle \leftarrow \textbf{dcontractMWTx}\langle([pspC_A], p, t, X)([pspC_B], p, x)\rangle$$

Figure 4.11 shows the implementation of the protocol.

A note on rogue-key attacks: In section 3.1, we mentioned that we need to take special care in the key generation phase in a Two-Party Signature Scheme. Otherwise the protocol might be vulnerable against rogue-key attacks in which one of the party's public keys is computed as a function of the other. We see that we do not take this into account in all of the protocols laid out in this section. As for the receiving party, it will always be possible to generate his keypair as a function of the sender's public key. We now show how attempting a rogue-key attack in Mimblewimble would play out and why it does not threaten the security of our scheme:
Imagine we have an attacker $\mathcal{A}$ who knows the value v of some coin $\mathcal{C} = g^r \cdot h^v$ 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 canceling out Alice's key. However, recalling the structure of Mimblewimble transactions the participants sign

dcontractMWTx$\langle([pspC_A], p, t, X)([pspC_B], p, x)\rangle$

	Alice		*Bob*
1 :	$(ptx, spC_A^*, (sk_A, n_A))$		$(ptx, (sk_B, n_B))$
	$\leftarrow$ **dSpendCoins**$([pspC_A], p, t)$		$\leftarrow$ **dSpendCoins**$([pspC_B], p, t)$
2 :			$(ptx', spC_B^*, \tilde{\sigma_B})$
			$\leftarrow$ aptRecvCoins(ptx, p, x)
3 :		$\xleftarrow{ptx', X'}$	
4 :	**if** $X \neq \perp \wedge X \neq X'$		
5 :	**return** $\perp$		
6 :	$\hat{\sigma_B} \leftarrow ptx'.\sigma$		
7 :	$\sigma_{\tilde{AB}}$		tx
	$\leftarrow$ **dAptFinTx**(ptx', sk_A, n_A, X)		$\leftarrow$ **dAptFinTx**$(ptx', sk_B, n_B, \tilde{\sigma_B})$
8 :		$\xleftarrow{tx}$	
9 :	$x \leftarrow$ extWit$(tx.\sigma, \sigma_{\tilde{AB}}, \hat{\sigma_B})$		
10 :	**return** (tx, spC_A^*, x)		**return** (tx, spC_B^*)

Figure 4.11: **dcontractMWTx** two-party protocol to build a primitive contract transaction

the excess value $\mathcal{E} = inp - out$, where inp and out is the input and output coins list. Therefore, making 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 $pk_A \cdot {pk_B}^{-1}$. By the prior definition of pk_B as pk_A the excess calculation would look like $pk_A \cdot {pk_A}^{-1}$, 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 Pedersen Commitment) the new output coin $\mathcal{C}^*$ would be identical to the input coin $\mathcal{C}$, and the transaction therefore spend a coin to itself. Recalling the instantiation of the transaction verification algorithm verfTx defined by Fuchsbauer et al. [**?**], which we laid out in fig. 4.2, 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 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 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. However,

in this scenario, $\mathcal{A}$ is 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 through transaction verification, and we, therefore, do not have to take other special care in the key generation phase to avoid them.

4.4 Correctness & Security

In this section, we will prove the orrectness and security of the instantiation described in section 4.2. 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 show that all protocols described in section 4.3 are secure in the malicious models as defined in definition 2.8.

4.4.1 Correctness

We will argue *Transaction Scheme Correctness* follows from Correctness of the Commitment Scheme COM, Two-Party Signature Scheme Φ and 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 $\mathbf{dSpendCoins}, \mathbf{dRecvCoins}, \mathbf{dFinTx}$ or the adapted ones $\mathsf{aptRecvCoins}, \mathbf{dAptFinTx}$ with valid inputs) it must follow that the final transaction has correct commitments, range proofs, 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 overwhelming 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 4.2 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 valid input parameters. If the sending party provides duplicate inputs, the check at the beginning of the spendCoins procedure will fail and return $\perp$ and thereby halting the protocol. The blinding factors to the output coins created in spendCoins and recvCoins are generated randomly, which means a duplication can only appear with negligible probability.

Condition 3 follows from the implementation of the createCoin function called in spendCoins as well as recvCoins. In the procedure, 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 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 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. Bob's secret key is constructed like $sk_B := r_B^*$, so it corresponds to his output's blinding factor. From the construction of the Two-Party Signature Scheme in definition 3.1, 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 $\mathcal{E}'$:

$$\mathcal{E} = \mathcal{E}' \tag{4.1}$$

$$\sum_{i=0}^{n} out - \sum_{i=0}^{n} inp - h^f = g^{r_A^*} \cdot \sum_{i=0}^{n} [g^{-r_A}] \cdot g^{r_B} - h^f \tag{4.2}$$

$$\mathcal{C}_{out}^A \cdot \mathcal{C}_{out}^B \cdot \sum_{i=0}^{n} [(\mathcal{C}_{inp})^{-1}] \cdot h^{-f} = \tag{4.3}$$

$$(g^{r_A^*} \cdot h^{v-p}) \cdot (g^{r_B^*} \cdot h^p) \cdot \sum_{i=0}^{n} [(g^{-r_A}, h^{-v_i})] \cdot h^{-f} = \tag{4.4}$$

$$g^{r_A^*} \cdot g^{r_B^*} \cdot \sum_{i=0}^{n} g^{-r_A} = g^{r_A^*} \cdot g^{r_B^*} \cdot \sum_{i=0}^{n} g^{-r_A} \tag{4.5}$$

$$1 = 1 \tag{4.6}$$

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. 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 shown that conditions 1-4 must hold for a valid transaction and conclude that *Transaction Scheme Correctness* holds for $MW[COM, \Pi_{RP}, \Phi_{MP}]$.

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

$$sk_A := r_A^* - \sum_{i=0}^{n} r_A \tag{4.7}$$

$$sk_C := r_C^* - \sum_{i=0}^{n} r_C \tag{4.8}$$

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 the Two-Party Signature Scheme definition 3.1, 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 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 unmasked partial signature $\tilde{\sigma_B}$. However, the signature's construction 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, providing valid inputs and calling the respective routines in the given order), verfTx(tx) will return 1 for the resulting transaction tx and therefore theorem 3 holds. □

4.4.2 Security

We now want to prove security in the malicious setting as described in definition 2.8 for the protocols defined in section 4.3. Again we show that the distributed protocols are secure in the hybrid f_{zk}^{R}-model as already explained in section 3.4.2. 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 private inputs r, v, and its Pedersen Commitment $C = g^r \cdot h^v$ for two adjacent generators g, h as defined in definition 2.7. We shorten the prover's call to provide $sp\mathcal{C}$ because it is a wrapper containing the coin Commitment, and 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 prover's calls we simplify the call to $\mathsf{f}_{zk}^{R_2*}([sp\mathcal{C}])$.

Proof Idea: We extend the protocol **dBuildMWTx** instantiated in section 4.3 with the following calls to the zero-knowledge proof of knowledge functionalities as depicted in fig. 4.12. Again as already pointed out in the previous section, we must make adjustments to the protocol to prove their security. That, in turn, does not mean that the original protocols are insecure. We add calls to make a security proof of this sort possible. When one implements these protocols with security in mind, one should also implement the additional calls added with the adjusted versions. The same holds for all modified protocols throughout this section.

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. **dBuildMWTx** securely computes a Mimblewimble transaction

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

	Alice		*Bob*
1 :	$\mathsf{f}_{zk}^{R_2*}([spC])$		
2 :	$(ptx, spC_A^*, (sk_A, n_A))$		
3 :	$\leftarrow$ spendCoins$([spC], p, t)$		
4 :	$\mathsf{f}_{zk}^{R_1}((sk_A, g^{sk_A}))$		
5 :	$\mathsf{f}_{zk}^{R_1}((n_A, g^{n_A}))$		
6 :	$\mathsf{f}_{zk}^{R_2}(spC_A^*)$		
7 :		$ptx \longrightarrow$	
8 :			**if** $\mathsf{f}_{zk}^{R_2*}(tx.inp) = 0$
9 :			**return** $\perp$
10 :			**if** $\mathsf{f}_{zk}^{R_1}(tx.\Lambda.pk) = 0$
11 :			**return** $\perp$
12 :			**if** $\mathsf{f}_{zk}^{R_1}(tx.\Lambda.R) = 0$
13 :			**return** $\perp$
14 :			**if** $\mathsf{f}_{zk}^{R_2}(tx.out[0]) = 0$
15 :			**return** $\perp$
16 :			$(ptx', spC_B^*) \leftarrow$ recvCoins(ptx, p)
17 :			$\mathsf{f}_{zk}^{R_2}(spC_B^*)$
18 :		$ptx' \longleftarrow$	
19 :	**if** $\mathsf{f}_{zk}^{R_2}(tx.out[1]) = 0$		
20 :	**return** $\perp$		
21 :	$tx \leftarrow$ finTx(ptx', sk_A, n_A)		
22 :		$tx \longrightarrow$	
23 :	**return** (tx, spC_A^*)		**return** (tx, spC_B^*)

Figure 4.12: Extension of **dBuildMWTx** (fig. 4.8) in the hybrid model

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.

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 participants' inputs. 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 an indistinguishable transcript from a real world's protocol transcript. The simulator uses the calls to $\mathsf{f}_{zk}^{R_1}, \mathsf{f}_{zk}^{R_2}, \mathsf{f}_{zk}^{R_2*}$ to achieve this. We prove that the transcript is indistinguishable when 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 $\mathsf{f}_{zk}^{R_2*}, \mathsf{f}_{zk}^{R_1}\ \mathsf{f}_{zk}^{R_2}$ 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 simulator's task is 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 $\sigma_{fin} = tx.\sigma$.

b) He computes the adversaries partial signature as $\tilde{\sigma_A} \leftarrow \mathsf{signPt}(m, sk_A, n_A, \Lambda)$

c) He further computes:

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

d) He other takes 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 prove that $\mathcal{S}$'s transcript is indistinguishable from a real one in every protocol phase. We separate between the following three phases: **Phase 1**: Alice sends her input coins, signing key and nonce, and 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 output. Finally, $\mathcal{S}$'s output needs to be indistinguishable from that of $\mathcal{A}$ in a real execution.

- *Phase 1*: Due to $\mathcal{A}$'s deterministic nature, 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 $\mathcal{S}$'s equality checks will fail. In this case, he will halt the simulated protocol and return whatever $\mathcal{A}$ outputs, which would be expected in a real execution. We further argue that the updated pre-transaction ptx' is identical to the pre-transaction 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' signatures 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 other values $\mathcal{S}$ needs to build ptx' he can 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 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 the protocol would produce 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, $\mathcal{S}$ 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 $\mathcal{A}$:

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

5. The simulator rewinds to the first step of the simulation. Instead of choosing the values for the pre-transaction, he now 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 field storing the commitments. 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 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 we need to consider the message sent before and after the rewind.

- *Phase 1:* In the first iteration, the simulator constructs the input values $[spC]$ from random values and chooses a random transaction value p. $\mathcal{S}$ creates the pre-transaction using those computed values rather than the real ones. We claim that the adversary cannot distinguish the chosen from the real coin commitments (except with negligible probability). If we assume that he would be able to do so, that means he could distinguish for two 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 the Perdersen Commitment. 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 $\mathcal{S}$'s pre-transaction after the rewind will be indistinguishable from a real one. However, this time the pre-transaction is constructed from the real tx that $\mathcal{S}$ received from the TTP. Therefore the pre-transaction is 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 is identical to what would be expected in a real execution because 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, $\mathcal{A}$ could distinguish the transcript from the real one. That is because he can identify differences in the inputs, outputs, proofs, and commitments and 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 last 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 adversary misbehaves at any point by sending invalid (or no) values, the simulator will notice, halt the protocol, and return whatever $\mathcal{A}$ returns. If $\mathcal{A}$ behaves honestly, $\mathcal{S}$ would run the protocol simulation until the end and then again output whatever $\mathcal{A}$ outputs. In both cases, the output would be the same as expected from $\mathcal{A}$ in a real execution.

We have managed to show that the transcripts produced by $\mathcal{S}$ in the case that Alice and in the case that Bob is corrupt are both indistinguishable from a transcript produced during

a real execution and can therefore conclude that the protocol is secure and theorem 4 holds.

□

Before we can continue to prove the security of the three other protocols **dsharedInpMWTx**, **dsharedOutMWTx**, **dcontractMWTx**, we first have to prove that all the protocols run as part of those executions are secure. 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 give 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 $\mathcal{C}^* = h^{v\cdot} \cdot g^{r_A} \cdot g^{r_C}$. A proof system that would support such a relation is zk-SNARKS as can be seen in [**?**]. To simplify the call made by the prover, we 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 calling the protocol with an array of inputs by calling $\mathsf{f}_{zk}^{R_{3*}}$

Proof Idea: We extend the protocol **dSpendCoins** instantiated in section 4.2 with the following calls to the zero-knowledge proof of knowledge functionalities, as shown in fig. 4.13.

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 the previous proof for theorem 4.

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, proof and signature context values 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 to receive the outputs $(ptx', sp\mathcal{C}_A^*, (sk_A, n_A))$.

dSpendCoins$\langle([pspC_A], p, t), ([pspC_C], p)\rangle$

	Alice		*Carol*
1 :	$\mathsf{f}_{zk}^{R_{3*}}([pspC_A])$		$\mathsf{f}_{zk}^{R_{3*}}([pspC_C])$
2 :	$\cdots$		
3 :	$\mathsf{f}_{zk}^{R_2}(spC_A^*)$		
4 :	$\mathsf{f}_{zk}^{R_1}((sk_A, g^{sk_A}))$		
5 :	$\mathsf{f}_{zk}^{R_1}((n_A, g^{n_A}))$		
6 :		$ptx \longrightarrow$	
7 :			**if** $\mathsf{f}_{zk}^{R_{3*}}(ptx.inp) = 0$
8 :			**return** $\perp$
9 :			$\mathsf{f}_{zk}^{R_2}(ptx.out[0]) = 0$
10 :			**return** $\perp$
11 :			**if** $\mathsf{f}_{zk}^{R_1}(\Lambda.pk) = 0$
12 :			**return** $\perp$
13 :			**if** $\mathsf{f}_{zk}^{R_1}(\Lambda.R) = 0$
14 :			**return** $\perp$
15 :			$\cdots$
16 :			$\mathsf{f}_{zk}^{R_2}(spC_C^*)$
17 :			$\mathsf{f}_{zk}^{R_1}((sk_C, g^{sk_C}))$
18 :			$\mathsf{f}_{zk}^{R_1}((n_C, g^{n_C}))$
19 :		$ptx' \longleftarrow$	
20 :	**if** $\mathsf{f}_{zk}^{R_{3*}}(ptx'.inp) = 0$		
21 :	**return** $\perp$		
22 :	**if** $\mathsf{f}_{zk}^{R_2}(ptx'.out[1]) = 0$		
23 :	**return** $\perp$		
24 :	$\{pk, R\} \leftarrow ptx'.\Lambda$		
25 :	**if** $\mathsf{f}_{zk}^{R_1}(pk \cdot pk_A^{-1}) = 0$		
26 :	**return** $\perp$		
27 :	**if** $\mathsf{f}_{zk}^{R_1}(R \cdot R_A^{-1}) = 0$		
28 :	**return** $\perp$		
29 :	**return** $(ptx', spC_A^*, (sk_A, n_A))$		**return** $(ptx', spC_C^*, (sk_C, n_C))$

Figure 4.13: Extension of **dSpendCoins** (fig. 4.3) in the hybrid model

3. $\mathcal{S}$ sends ptx' to $\mathcal{A}$ as if coming from Carol and sends continue to the TTP to make $\mathcal{A}$ receive the outputs in the ideal setting.

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 partially 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 be indistinguishable from that of $\mathcal{A}$ in a real execution.

We now argue why each phase is indistinguishable from a real execution if Alice is corrupted.

- *Phase 1*: No simulation is required in this phase. We, therefore, conclude that it 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 to halt the protocol and return, which is the same as 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, and 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 as if coming from 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 sends abort to the TTP and returns whatever $\mathcal{A}$ returns if any do not match. Otherwise, he sends continue to the TTP and again outputs whatever $\mathcal{A}$ outputs.

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

- *Phase 1*: In the first iteration (before the rewind), the pre-transaction that is sent to $\mathcal{A}$ will be constructed from randomly chosen values except for the transaction inputs 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 knows the right 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 $\mathcal{S}$ must use the Commitments extracted from $[psp\mathcal{C}_C]$ to build the transaction. Otherwise, one could detect the simulation. In the second iteration (after the rewind), $\mathcal{S}$ sends the same ptx, which would be expected in a real execution which is therefore identically distributed.

- *Phase 2*: When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_{1,2,3}}$ he will receive 0 or 1 again identically to what is expected in a real execution.

- *Phase 3*: If $\mathcal{A}$ sends invalid input, output, proof, or context values, in the final pre-transaction ptx' the simulator detects this and returns. Otherwise the protocol concludes.

- Both in the case in which $\mathcal{A}$ behaves as of protocol specification and where he deviates, $\mathcal{S}$ will always output whatever $\mathcal{A}$ outputs, making the simulator's output indistinguishable from what would be expected in a real scenario.

We have managed to show that the simulator $\mathcal{S}$ can produce an indistinguishable transcript both in the case that Alice and in that Carol is corrupted and 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. □

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

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

	Bob		*Carol*
1 :	$\cdots$		
2 :	$\mathsf{f}_{zk}^{R_2}((\mathcal{C}_{out}^{sh}, (p, r_B^*)))$		
3 :		$\xrightarrow{ptx, \mathcal{C}_{out}^{sh}}$	
4 :			**if** $\mathsf{f}_{zk}^{R_2}(\mathcal{C}_{out}^{sh}) = 0$
5 :			**return** $\perp$
6 :			$\cdots$
7 :			$\mathsf{f}_{zk}^{R_1}((sk_C, g^{sk_C}))$
8 :		$\xleftarrow{ptx', g^{sk_C}}$	
9 :	**if** $\mathsf{f}_{zk}^{R_1}(g^{sk_C}) = 0$		
10 :	**return** $\perp$		
11 :	$\cdots$		$\cdots$
12 :	$\pi_{BC} \leftarrow$ **dRanPrf**$(\mathcal{C}_{out}^{sh\,\prime}, p, sk_B)$		$\pi_{BC} \leftarrow$ **dRanPrf**$(\mathcal{C}_{out}^{sh\,\prime}, p, sk_C)$
13 :	$(\sigma_{\tilde{BC}}, pk_{BC}) \leftarrow$ **dSign**(m, sk_B, n_B)		$(\sigma_{\tilde{BC}}, pk_{BC}) \leftarrow$ **dSign**(m, sk_C, n_C)
14 :	$\cdots$		$\cdots$
15 :	**return** $(ptx^*, psp\mathcal{C}_B^*)$		**return** $(ptx^*, psp\mathcal{C}_C^*)$

Figure 4.14: Extension of **dRecvCoins** (fig. 4.4) in the hybrid model

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 other protocols which have already been proven secure. We extend our model including the **dSign** protocol, proven to be secure in section 3.4 and the **dRanPrf** for which one can find a secure protocol in [**?**].

Proof idea: We extend the protocol **dRecvCoins** instantiated in section 4.2 with the following calls to the zero-knowledge proof of knowledge functionalities as outlined in fig. 4.14.

Theorem 6. Let *COM* be a correct and secure Pedersen Commitment Scheme, Π_{RP-MP} be a correct and secure Two-Party Range Proof System, and Φ_{MP} be a secure and correct Two-Party Signature Scheme, then **dRecvCoins** securely updates a Mimblewimble pre-transaction by creating a new output coin $\mathcal{C}_{out}^{sh\,\prime}$ 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. The proof strategy again is as specified in the proof for theorem 4.

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 $(ptx, \mathcal{C}_{out}^{sh})$ to Alice. 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 $\mathcal{A}$ outputs if they don't match. Otherwise, he sends (ptx, p) to the TTP computing recvCoins and receives the results $(ptx^*, psp\mathcal{C}_B^*)$.

3. $\mathcal{S}$ proceeds by taking the last output $\mathcal{C}_{out}^{sh\,\prime}$ from $ptx^*.out$ and computes $g^{sk_C} := \mathcal{C}_{out}^{sh\,\prime} \cdot \mathcal{C}_{out}^{sh\,-1}$. The simulator computes ptx' by adding $\mathcal{C}_{out}^{sh\,\prime}$ to ptx sending it together with g^{sk_C} to $\mathcal{A}$ as if coming from Carol and sends continue to the TTP.

4. When $\mathcal{A}$ calls $\mathsf{f}_{zk}^{R_1}$ as the verifier $\mathcal{S}$ checks 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**. Finally, we again have to show that the simulator's output is indistinguishable from that of $\mathcal{A}$ in a real execution.

We argue that in this case, the simulation is perfect. That is the transcript produced by $\mathcal{S}$ is identically distributed as a transcript produced during a real execution.

- *Phase 1*: No simulation is done during this phase, and the transcript is thereby indistinguishable from a real one simply 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, which is the same as would happen in a real execution. The simulator proceeds by sending the updated pre-transaction and the extracted value for g^{sk_C}, exactly as would be expected from honest Carol.

- *Phase 3*: $\mathcal{A}$ will receive 0 or 1 to the call to $\mathsf{f}_{zk}^{R_1}$ as in the real execution, depending on if he sends a valid or invalid value. In the case that $\mathcal{A}$ behaves dishonestly, $\mathcal{S}$ will notice and halt the protocol. If he instead acts honestly, the simulator will simulate the protocol until the end. $\mathcal{S}$ will return whatever $\mathcal{A}$ returns making the output indistinguishable from an output produced by $\mathcal{A}$ in a real execution.

Carol is corrupted: The simulator works as follows:

1. Since Carol does not have any inputs in this protocol, $\mathcal{S}$ can send $\emptyset$ to the TTP to receive $(ptx^*, sp\mathcal{C}_C^*)$, from which he extracts Carol's blinding 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 sends $ptx, \mathcal{C}_{out}^{sh}$ (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 equality with what he sent in the last step and returns 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 send abort to the TTP halting the execution and returning whatever $\mathcal{A}$ returns. Otherwise, he will send 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 $\mathcal{S}$ can call the TTP already in the first step, he can receive the protocol outputs instantly. The simulator can then extract Carol's secret key sk_C from Carol's $psp\mathcal{C}_C^*$ output, which must also be her blinding factor in $\mathcal{C}_{out}^{sh\,\prime}$. He, therefore, can reconstruct $\mathcal{C}_{out}^{sh}$ which would have been sent by Bob in a real execution, by subtracting Carol's part from the output, which is present in $ptx^*.out$. $\mathcal{S}$ is further able to reconstruct the ptx, which would have been sent by Bob in a real execution by removing the values from ptx^* which get added at a later point in the protocol. Therefore the transcript in this phase is exactly how it would be expected in a real execution.

- *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 what would happen in a real execution. Similarly, the execution will halt if $\mathcal{A}$ sends invalid values as ptx' and g^{sk_C}, again how it would happen in 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 can reconstruct pk_B from $\mathcal{C}_{out}^{sh}$. Therefore, the simulation again is perfect also in this phase.

- Regarding protocol outputs, the simulator will again detect if $\mathcal{A}$ deviates from the protocol specification at any point and will output whatever $\mathcal{A}$ outputs in any case, making the protocol output indistinguishable from one produced during a real execution.

Both in the case, Bob and Carol are corrupted, $\mathcal{S}$ can produce a transcript indistinguishable from a transcript produced with a real execution. Therefore, we can 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. □

We claim that the security proof of the protocols **dFinTx** and **dAptFinTx** can be reduced to the proof for **dSign** as all interaction between the two parties happens in the call to **dSign**. We have already proven the security of **dSign** in section 3.4 and can reuse the simulator for the protocols **dFinTx** and **dAptFinTx**.

We can now continue to prove the security of the protocols found in section 4.3. We start with **dsharedOutMWTx**.

Hybrid functionalities: For this proof, we again assume 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 4.3 with the following calls to the zero-knowledge proof of knowledge functionalities shown in fig. 4.15.

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 $\mathcal{C}_{out}^{sh\,\prime}$ which spending secret is shared between Alice and Bob.

Proof. The proof strategy is again as defined in the proof for theorem 4.

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

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 fails he sends abort to the TTP and outputs whatever $\mathcal{A}$ outputs.

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 sends to $\mathcal{A}$. Otherwise, he returns $\perp$ to $\mathcal{A}$ and sends abort to the TTP, and halts the protocol returning whatever $\mathcal{A}$ returns.

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 is expected in a real execution. Also, if the

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

	Alice		*Bob*
1 :	$\mathsf{f}_{zk}^{R_{3*}}([spC])$		
2 :	$(ptx, spC_A^*, (sk_A, n_A))$		
	$\leftarrow$ spendCoins$([spC], p, t)$		
3 :	$\mathsf{f}_{zk}^{R_2}(spC_A^*)$		
4 :	$\mathsf{f}_{zk}^{R_1}((sk_A, g^{sk_A}))$		
5 :	$\mathsf{f}_{zk}^{R_1}((n_A, g^{n_A}))$		
6 :		$ptx \longrightarrow$	
7 :			**if** $\mathsf{f}_{zk}^{R_{3*}}(ptx.inp) = 0$
8 :			**return** $\perp$
9 :			**if** $\mathsf{f}_{zk}^{R_2}(ptx.out[0]) = 0$
10 :			**return** $\perp$
11 :			**if** $\mathsf{f}_{zk}^{R_1}(ptx.\Lambda.pk) = 0$
12 :			**return** $\perp$
13 :			**if** $\mathsf{f}_{zk}^{R_1}(ptx.\Lambda.R) = 0$
14 :			**return** $\perp$
15 :	$(ptx', pspC_A^*)$		$(ptx', pspC_B^*)$
	$\leftarrow$ **dRecvCoins**(ptx, p)		$\leftarrow$ **dRecvCoins**$()$
16 :	$tx \leftarrow$ finTx(ptx', sk_A, n_A)		
17 :		$tx \longrightarrow$	
18 :	**return** $(tx, spC_A^*, pspC_A^*)$		**return** $(tx, pspC_B^*)$

Figure 4.15: Extension of **dsharedOutMWTx**(fig. 4.9) in the hybrid model

adversary cheats (by sending an invalid *ptx*), this is noticed by the simulator, who then halts the protocol and outputs whatever $\mathcal{A}$ outputs, which is again what would be expected in a real protocol execution.

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 set up 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 set up 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 $\mathcal{S}$'s *tx* sent in the last step is exactly what would be expected in a real execution as the TTP has computed it. When $\mathcal{A}$ tries to cheat by sending invalid values to $\mathsf{f}_{zk}^{R_{1,2,3}}$ he will receive 0, as would be the case in a real execution. ptx' must resemble ptx^* from a real execution because it is 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 built in step 2 for *ptx* must be indistinguishable from a real one, which 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: $\Lambda = pk_{A1} \cdot pk_{A2} \cdot pk_B$ (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's second key pair. Therefore he has no way of learning that the final pk was not computed as of protocol specification. The same argument holds for R.

We have shown that the simulator $\mathcal{S}$ can produce an indistinguishable transcript both in the case that Alice and Bob are corrupted and 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. That means that we do not have to extend to original protocol instantiation seen in fig. 4.10 any further.

Theorem 8. Let *COM* be a correct and secure Pedersen Commitment Scheme, Π_{RP-MP} a correct and secure Two-Party 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. The proof strategy is as defined in the proof for theorem 4.

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 share from Bob's output coins and A + B1 are the signature shares 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 and n_B. However, he knows the final signature $\sigma_{fin} := tx.\sigma$ and he can create the signature A as $\tilde{\sigma_A} \leftarrow \mathsf{signPt}(tx.m, sk_A, n_A, tx.\Lambda)$. $\mathcal{S}$ can recompute the value for the B2 signature as follows:

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

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 right value for the partial signature B2 from $tx.\sigma$ with the same calculation as shown in the previous step

4. $\mathcal{S}$ can now 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 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 tx's final signature is composed of three parts A, B1, and B2. Through the calculations laid out the simulator can recompute the actual value of B2, as it would be in real execution, which must make ptx' identical to what would be sent by an honest Bob in a 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. Therefore, a perfect simulation is achieved 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 and Bob are corrupted, which produced a protocol transcript indistinguishable from a real one and 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 prove the security of **dcontractMWTx** in the hybrid model in which the participants have access to a trusted third party computing **dSpendCoins** and **dAptFinTx**. We 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 Alice and Bob. On Bob's 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 Two-Party Range Proof System, and Φ_{MP} be a secure and correct Two-Party Signature Scheme, then **dcontractMWTx** securely computes a Mimblewimble 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 $X = g^x$.

Proof. The proof strategy is as defined in the proof for theorem 4.

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 laid out in the previous proof, we still lay it out here again for completeness:

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

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 right value for the partial signature B2 from $tx.\sigma$ with the same calculation as shown in the previous step

Note, however, that in this case, $\mathcal{A}$ expects a masked partial signature $\hat{\sigma_B}$ which will verify with the masked partial signature verification routine passing X. $\mathcal{S}$ can easily calculate the masked signature by running $\hat{\sigma_B} \leftarrow \mathsf{mskSig}(\tilde{\sigma_{B1}}, x)$ 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 $\mathsf{continue}$ to the TTP computing **dcontractMWTx**, and output whatever $\mathcal{A}$ outputs.

In this proof, only ptx' and X sent in Bob's first message to Alice is constructed by $\mathcal{S}$. All other values are directly 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 the same as expected in a real execution must hold because the simulator was able to reconstruct the original signature shares from the final signature and by the fact that $\mathcal{S}$ knows x and can therefore call mskSig as given by the protocol specification.

Bob is corrupted: Again finding a perfect simulator is trivial in this case as there are no messages are sent 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 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 Bob are corrupted and controlled by a deterministic adversary $\mathcal{A}$. Therefore, we conclude that **dcontractMWTx** is secure in the $\mathbf{dSpendCoins}, \mathbf{dAptFinTx}, \mathsf{f}_{zk}^{R_1}$-model and theorem 9 must hold. □

4.5 Atomic Swap Protocol

With the outlined Contract Mimblewimble Transaction Scheme from definition 4.6 and protocols from section 4.3, we can now construct an Atomic Swap protocol with another cryptocurrency. This thesis will explain a swap with Bitcoin, as at present, Bitcoin is the most widely adopted cryptocurrency. To abstract away from the details of 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. We define the following 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 procedure requires Bob's public key pk_B and a timelock t (given as a block number) as input, allowing Bob to reclaim his funds after some time if the Atomic Swap was not completed successfully. The function will create and return a Bitcoin script *spk* to which Bob can send funds using a P2SH transaction. To spend this output, Alice will have to provide a multi-signature under her public key pk_A and X, which she can provide, once acquired x. Malavolta et al. in [**?**] described an alternative secure way of constructing a locking mechanism on Bitcoin. In their construction the two parties cooperate to build an initial signature for the spending transaction which is not yet valid as it is missing some witness value x, only known to one of the two parties. Once the second party gets hold of the witness value they can complete the signature and finalize the transaction. Comparing their solution to the more primitive multi-signature script, it achieves greater privacy (from the outside, the lock output looks like a standard P2PKH output), and needs only a single signature, therefore less space for the unlocking transaction. However, the construction is slightly more complex in ECDSA signatures, which are at present the only Signature Scheme available on Bitcoin. Even though the structure by Malavolta et al. would also be applicable in our case, because of the additional complexity involved and since the focus of this thesis is the Mimblewimble side of the swap, we decided to implement the more straightforward script-based locking mechanism in our proof of concept implementation.

- $\{1, 0\} \leftarrow \mathsf{verifyLock}(pk_A, pk_B, X, v, t, \psi_{lock})$: The lock verification algorithm takes as input Alice and Bob public keys and the statement X and the UTXO ψ_{lock}. The function will compute the Bitcoin lock script *spk* as created by 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 buildTransaction, signTransaction defined in section 2.2.1. 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.

In the following, we describe the phases of an Atomic Swap protocol executed between two parties, one owning funds on a Bitcoin-like cryptocurrency and the other on a Mimblewimble based one. As both of the currently most prominent implementations of the Mimblewimble protocol operate on the secp256k1 elliptic curve (which is also the curve that Bitcoin uses), we, therefore, assume a secp256k1-based implementation of the Mimblewimble protocol. For a Mimblewimble based implementation that operates on a different curve, additional considerations would have to be made to guarantee the protocol's security. In the setup phase section 4.5, the two parties agree on the swap parameters: the exchange rates, the amount being swapped and the timeout for the refunding. In the locking phase section 4.5.1, the goal is to lock up the funds on both chains, such that they can either be redeemable by the other party in case the swap was successful or be refunded to the original owners in case the trade has failed. The precondition for running the locking phase is that the parties have completed the setup phase. In the execution phase section 4.5.2, the two parties cooperate to

redeem the funds locked by the other parties. The peers can only enter this phase after completing the locking step. When the funds are redeemed on both sides, the swap is considered successfully completed. In case the execution phase fails, for instance, if one party stops cooperating, the exchange is considered failed, and we enter the refunding phase section 4.5.3. A unique security requirement here is that the funds are refunded to their original owners on both sides only in case of failure. If the swap completes on one side but then can't be completed on the other side, one party would lose all of their value, therefore we must make sure that this case is an impossibility.

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 we should cancel the swap. After agreeing, 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 block height for the Bitcoin side.
- t_{mw} The locktime as a block height for the Mimblewimble side.

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

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

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 [**?**]. They propose using a fixed exchange rate for each day and using real-world timeout like one day and then calculating the specific block numbers by taking blockchain's average block time into account. Alternatively, if the chains allow it, we could use a real-world Unix timestamp as a timeout instead of a block height. In our setup, we can also fix the exchange rate at the beginning of the protocol, which stays unchanged during protocol execution. Suppose the exchange rate fluctuates and one party is negatively impacted. In that case, they could still decide stop cooperating, 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 too 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 value.

4.5.1 Locking phase

We formalize the protocol **lockSwp** in fig. 4.16. The protocol takes as input the shared swap state $\mathcal{W}$ 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 sum of total value v_{btc}, although one could also read this from the blockchain.

The protocol starts with both parties creating and exchanging keys. Bob now makes two new Bitcoin outputs ψ_{lock} and ψ_B, one of these is the locked Bitcoins that Alice might retrieve later (or Bob after time t_{btc} has passed), and the other Bob's 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 the 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. Both parties immediately 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 time-locked refund transaction (AB -> A) was signed. Otherwise, funds are locked in the shared output without the possibility of a refund if Bob refuses to cooperate. The locking protocol concludes with the funds locked up in both chains and ready to be swapped and outputs the updated swap state $\mathcal{W}$ 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 a refund coin, only valid after t_{mw}. Bob also 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.

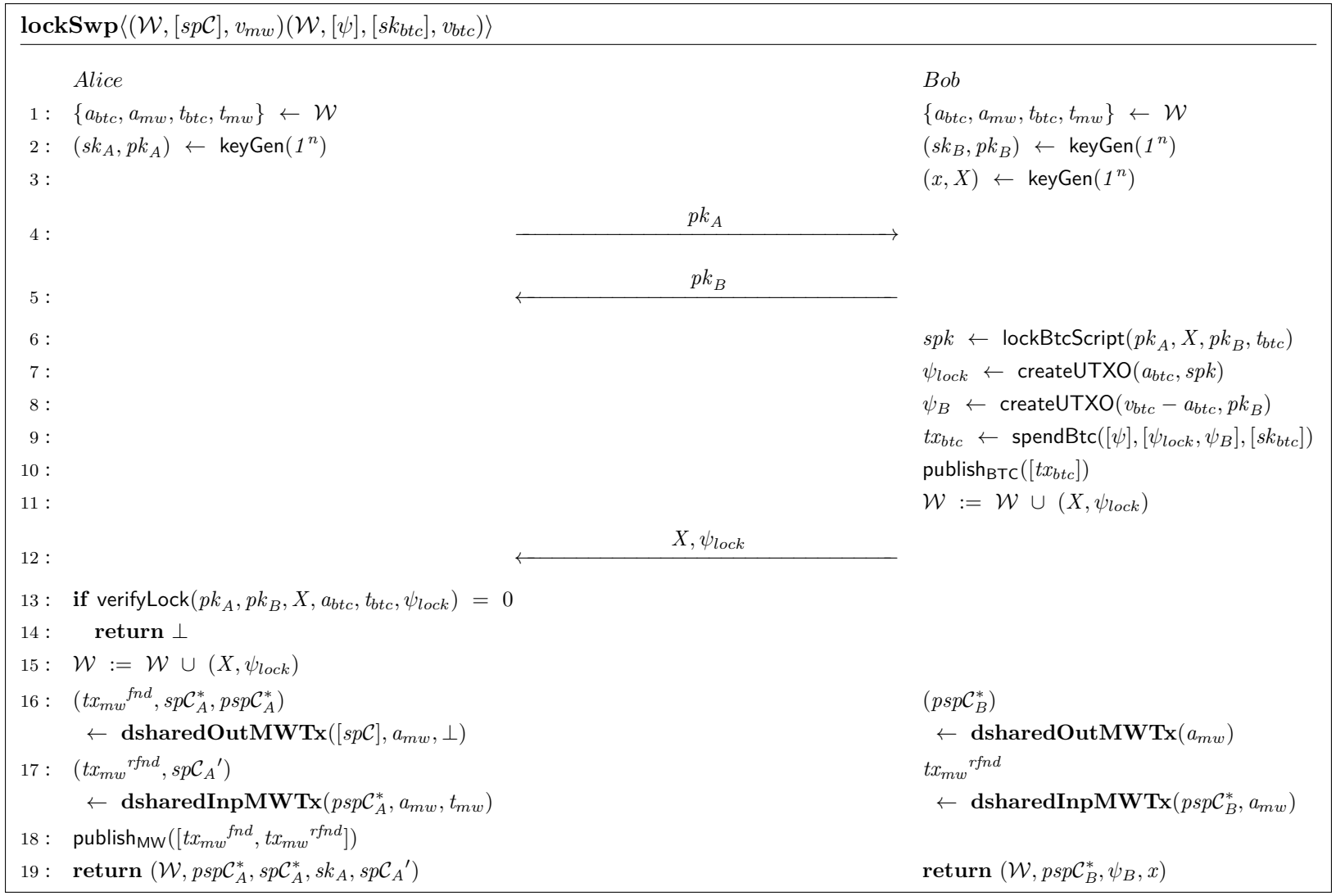

Figure 4.16: Atomic Swap - **lockSwp**.

4.5.2 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 left to execute the Atomic Swap protocol. 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}. The reader can find an instantiation of the protocol in fig. 4.17. We assume the participants have successfully run the **lockSwp** protocol, and both know the updated swap state $\mathcal{W}$ 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 private 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. Alice can now publish the transaction to the Mimblewimble network, concluding the swap on the Mimblewimble side, as Bob is now in complete control of the funds. Knowing x, Alice now creates 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 Bitcoin side's swapped funds, and the Atomic Swap is completed. The protocol outputs their newly created output/coin to each party. We note here that after completion of the swap on the Mimblewimble side, Alice is possible to redeem her Bitcoin. However, she still has to construct the transaction and get it mined on the network. Otherwise, if she would take too long and the timeout block height is reached, Bob could still try to refund his coins, even though he already received the funds on the Mimblewimble side. Therefore, it is crucial to pick long enough timeouts and check how much time is left again before running the execution protocol.

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

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

Figure 4.17: Atomic Swap - **execSwap**.

4.5.3 Refunding phase

If one party refused to cooperate, or goes offline the coins can be returned to the original owner. Bob can spend the locked output with his private key sk_B on the Bitcoin side after the timeout t_{btc} has passed. He can then construct and sign a transaction spending the output to a new UTXO in his full possession. He even could prepare this transaction upfront and broadcast it. Once the block number 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 canceled if there are some delays. If the timeout is too long, the funds might be locked for an unnecessary 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 vital that Alice publishes both the fund and refund transaction at the same time. If she would post 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 successfully, the refund transaction would get discarded by miners, as it is no longer valid even after the timeout t_{mw}.

CHAPTER 5

Implementation

We have developed a prototype implementation[1] of the Atomic Swap protocol outlined in section 4.5 in the Rust programming language between Bitcoin and the Grin cryptocurrency. On the Grin side, we have been using the official grin-wallet[2] library, on the Bitcoin side, the rust-bitcoin[3] library for transaction creation, signing, etc. We choose the programming language Rust because the official libraries available for the Grin cryptocurrency are also written in Rust. We discovered two shortcomings in the grin-wallet library during development, so we used our local forked version of this library to address those shortcomings. We then submitted both changes to the library as pull requests.[4,5] Our implementation sends JSON-RPC[6] requests to a locally running Bitcoin Core and Grin Core node to query blockchain state, submit, and verify transactions. The developed program is executable via the command line and accepts the following commands:

- **init**: The init command is run during the setup phase of the Atomic Swap. It creates a new swap slate consisting of a public file containing information about the offered currency, exchange rate, connection details, and other general parameters and a private file in which secret keys and nonce values are stored. The creator can share the public file with an interested party to execute a swap.

- **import**: The import command is run during the Atomic Swap setup phase and allows importing of funds both on the Bitcoin and the Grin side. Note that both parties need to import funds with a value of at least the desired value to be swapped before the protocol can continue.

[1] https://github.com/jafalter/mw-btc-swap
[2] https://github.com/mimblewimble/grin-wallet
[3] https://github.com/rust-bitcoin/rust-bitcoin
[4] https://github.com/mimblewimble/grin-wallet/pull/565
[5] https://github.com/mimblewimble/grin-wallet/pull/557
[6] https://www.jsonrpc.org/

- **accept**: An interested party executes the accept command during the setup phase. It imports the public slate file shared by the offering party and creates the individual private slate file, to which funds can be imported.

- **listen**: The listen command concludes the setup phase on the offerer's side and starts a TCP server to which peers can connect. The precondition for this command to execute is that the offered funds have been successfully imported into the swap slate.

- **lock**: The lock command starts the locking phase of the Atomic Swap. It is executed by the accepting party, while the offering party must already be listening on a TCP server. Again the command will verify that enough funds have been imported and will otherwise exit with an error. During this phase, funds will be locked up on both chains as specified in section 4.5.1. Private and public slate files will be updated during this command to allow swap execution to be initiated.

- **execute**: The execute command runs the execution phase of the Atomic Swap section 4.5.2, unlocking the funds locked during the locking phase and transferring the coins to their new owners. Again it has to be run by the accepting party while the offering party is still listening on a TCP server. Before the execution starts, the program will check on both chains if enough time is left to finish the swap before the timeout expires. If not enough time is left in at least one of the two blockchains, the program will exit with an error.

- **cancel**: The cancel command returns funds that have been locked during the locking phase to the original owners. The accepting party will execute the command while the offering party is listening on a TCP server. Running the cancel protocol is only allowed as soon as the timeout has been hit on both blockchains. Trying to run the command earlier will result in an error.

5.1 Implementation Bitcoin side

On the Bitcoin side, we have used a P2SH address to implement the locking mechanism. In a practical setting, we note that it would be recommended to use a more efficient P2WSH[7] (Pay to Witness Script Hash) address instead. Because of the implementation just being a prototype, we went with the slightly simpler P2SH version. However, one can also use the same script with a P2WSH address. Figure 5.1 shows the locking script used on the Bitcoin side. The script has two execution paths, depending on the number on the stack when the *OP_IF* command executes. To make it evaluate to true, one would have to push any number other than 0 to the stack. To make it evaluate to false, one would have to push the number 0 to the stack. If *OP_IF* evaluates to true, we execute the refunding code. *OP_CHECKLOCKTIMEVERIFY* will cause the script to fail unless the nLockTime on the transaction is equal or greater than the value passed in *<refund_time>*. This condition makes sure that this part of the script will only be executable after a specific time, such that refunding is not possible prematurely (Note

[7] `https://bitcoincore.org/en/segwit_wallet_dev/`

```
1 : OP_IF
2 :     < refund_time >
3 :   OP_CHECKLOCKTIMEVERIFY
4 :   OP_DROP
5 :     < refund_pub_key >
6 :   OP_CHECKSIGVERIFY
7 : OP_ELSE
8 :   2 < recv_pub_key > < X > 2 OP_CHECKMULTISIGVERIFY
9 : OP_ENDIF
```

Figure 5.1: Bitcoin locking script.

here that one has to set the nLockTime of the transaction, which guarantees that it can only be mined after a particular time). If the OP_IF evaluates to false, we execute the script's redemption path in which Alice needs to provide a valid signature under her public key and the statement X to Bobs witness value x.

5.2 Implementation Grin side

On the Grin side, we have implemented the three transaction protocols specified in section 4.3. Communication between the transacting parties happens via a TCP channel. For implementing two-party bulletproof range proofs, we used the implementation of grinswap[8], which runs a three-round protocol between the two parties using the secp256k1-zkp[9] library for bulletproof computation.

As already laid out in section 4.5, to lock the Mimblewimble side's funds, the two parties first engage in a **dsharedOutMWTx**, and then a time-locked **dsharedInpMWTx** transaction refunding the coins to the original owner. The timelock on the Grin side can be achieved by setting the transaction's respective kernel feature. The program saves the refunding transaction to the slate file for later use. In the execution phase the **dcontractMWTx** transaction protocol is run. The result is broadcast to the network sending the Grin to its new owner while revealing the witness value x to Alice, who is can now build a valid spending script for the Bitcoin side.

5.3 Evaluation

We successfully managed to run the full protocol between Alice owning Grin and Bob owning Bitcoin on the Bitcoin and Grin testnets. In transaction

[8]https://github.com/vault713/grinswap
[9]https://github.com/mimblewimble/secp256k1-zkp

10536404873e6ae133afde600b5630d6a00f3be0b9dde01a248c6f13a00b3a4b[10], which was mined in block 1937148[11] of the Bitcoin testnet 0.000016 BTC were locked in the lock address 2NCJDq4YRQ9C83fgvepMqU2D9kE4x7h36Ji[12]. On the Grin side our lock transaction locking 0.1 GRN was mined in block 718594[13] sending funds to the lock Commitment 08c2e1a98f5fd328cc67b7df5ab9fdee9cf0c1c1f166d5d08a02a578945fdf6076[14]. In the execution phase the locked funds on the Grin side were sent to Bob (09ef66334dc2e4c74732dafda8af3c32494eed5b23beb483d29d7ef32bf5c3ebb8[15]) in a contract transaction mined in block 718596[16]. Executing the contract allowed Alice to unlock her funds on the Bitcoin side in the transaction aa2ab77482841571b6413c68de681830c61527bc6a90ef1781d6208d151fea10[17], which was mined in block 1937150[18] on the Bitcoin testnet. We can reduce the protocol's execution time to the time it takes for the individual transactions to be mined on the networks. Therefore depends on the current network load and the miner's fee included with the transaction, as transactions are prioritized according to their fees [**?**]. It is unavoidable to estimate fees according to the current network load and the Atomic Swap timeout in a practical setting. Otherwise, a redeeming transaction may stay unconfirmed until the timeout expires. Then the second party could try to refund the locked funds with a higher fee, outspending the redeeming transaction. Another time-consuming process that we noticed when running our implementation is the validation of locked funds on the Bitcoin side. The reason for this comes from the way Bitcoin wallets handle the importing of new funds. When importing a new address into a Bitcoin Core node, one has to rescan the blockchain, a process that can take up to one hour or longer. it would be possible to import the address without performing a rescan. However, if the funding transaction was mined in a block before we execute the import command, we would miss the transaction, and the address would therefore appear empty. A faster solution would be, for instance, to use the API of a block explorer such as Blockstream.info[19]. However, this would introduce a trusted third party to the protocol, which is generally to be avoided.

[10] https://tinyurl.com/pend7sdk
[11] https://tinyurl.com/7jmryv47
[12] https://tinyurl.com/2sz8munw
[13] https://floonet.grinscan.net/block/718594
[14] https://tinyurl.com/y7ed5za5
[15] https://tinyurl.com/ychtyf8v
[16] https://floonet.grinscan.net/block/718596
[17] https://tinyurl.com/45ysh9e4
[18] https://tinyurl.com/fddeahk8
[19] https://blockstream.info/

I think this section is well written. But it is a pity that it does not show all the work that you had to do to make the overall implementation work. Perhaps you want to go in a bit more detail about what was implemented in the libraries itself and what you had to implement yourself. For the latter point, maybe you can explain also what challenges you had and how you overcame them. Additionally, you might want to say that the implementation is open source, that you engaged with the community, etc...

CHAPTER 6

Conclusion

This thesis aimed to improve interoperability between Privacy-enhancing cryptocurrencies and regular cryptocurrencies, allowing for the listing of more Privacy-enhancing currencies on decentralized exchanges such as Uniswap.[1] We have achieved this goal by constructing an Atomic Swap protocol between Bitcoin and a Mimblewimble based cryptocurrency. Similar to Joël Gugger, who constructed an Atomic Swap protocol between Bitcoin and the Monero cryptocurrency in [**?**], we leveraged a Scriptless Scripts approach that was derived from Andrew Poelstra's inital work [**?**] and the Adaptor Signature Scheme [**?**] by Auymayr et al. We further managed to prove Correctness and Security in the malicious setting for four different types of Mimblewimble transactions that allow for the distribution of Mimblewimble coins between multiple keys and the execution of simple contracts, based on our own secure and correct two-party version of the Adaptor Signature Scheme. We pointed out a weakness in the security analysis of the Mimblewimble protocol done by Fuchsbauer et al. in [**?**], which we have fixed by introducing the notion of Two-Party Signature Schemes into their model.

Pedro: I would like to go with you again through this. Please remind me in the next call

While two-party signatures are prone to Rogue-key attacks, we demonstrate the infeasibility of such an attack in the Mimblewimble protocol.

Related work. Maurice Herlihy [**?**] describes how hashed timelock contracts can be leveraged to build Atomic Swaps between cryptocurrencies that have at least some basic scripting capabilities. In a hashed timelock contract, the hash pre-image x of $h = \mathsf{H}(x)$, where $\mathsf{H}(\cdot)$ is a hash function, serves as the secret contract value that makes the swap possible. Fuchsbauer et al. conduct a complete security analysis of the Mimblewimble

[1] `https://uniswap.org/`

protocol, which will form the basis for chapter 3. Betarte et al. [?] work towards making implementations of the Mimblewimble protocol formally verifiable. In 2016 Andrew Poelstra published his specification of the Mimblewimble protocol in [?] and has managed to expand on the ideas posted in the original writing [?] by Jedusor. In the following year, Poelstra further introduced the notion of Scriptless Scripts [?], showing how one can build primitive contracts just by using cryptography alone. Poelstra's ideas were formalized as Adaptor signatures as a standalone cryptographic primitive by Aumayr et al. in [?]. Poelstra's insights on Scriptless Scripts and the formalization by Aumayr et al. together form the basis for our construction of the Two-Party Fixed Witness Adaptor Signature Scheme in chapter 3. Joël Gugger constructs a cross-chain Atomic Swap between Monero and Bitcoin in [?]. Just like Mimblewimble based crypto coins, Monero[2] is a Privacy-enhancing cryptocurrency that lacks scripting capabilities and even lacks timelocking capabilities (which do exist in Mimblewimble). In Gugger's protocol, the Monero funds are locked in an address shared between the two trading parties. On the Bitcoin side, funds are locked so that spending them will reveal the missing key for the Monero side, therefore unlocking the Monero funds for the second party. The author made some special considerations since the two cryptocurrencies operate on two different elliptic curves.

Future Research. Erkan Tairi et al. have demonstrated in [?] how to construct Payment channel hubs as a solution to scalability issues of blockchain technologies. With their approach relying solely on Scriptless Scripts, it would be interesting to explore the feasibility of such a construction on a Mimblewimble based cryptocurrency by utilizing the contract transactions outlined in our research. A. Faz-Hernandez et al. examine the possibility of hash functions that output an elliptic curve point for multiple elliptic curves, including secp256k1. [?] Based on this work on could research if more elaborate contracts such as hashed timelock contracts are made possible on Mimblewimble with such a hash function.

Interesting, I would like to talk to you about this.

Finally, another interesting research topic would be the construction of a cross-chain Atomic Swap between two cryptocurrencies both lacking scripting functionality. For instance by combining our protocol with the construction by Gugger [?] to build Atomic Swaps between Monero and a Mimblewimble based cryptocurrency.

[2] https://www.getmonero.org/

List of Figures