



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

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Jakob Abfalter, BSc

Registration Number 01126889

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Matteo Maffei

Assistance: Dr. Pedro Moreno-Sanchez

Vienna, 6th April, 2020

Jakob Abfalter

Matteo Maffei

Erklärung zur Verfassung der Arbeit

Jakob Abfalter, BSc

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

Wien, 6. April 2020

Jakob Abfalter

Acknowledgements

Enter your text here.

Abstract

Enter your text here.

Contents

Abstract	vii
Contents	ix
1 Introduction	1
2 Motivation & Objectives	5
3 Preliminaries	7
3.1 General Notation and Definitions	7
3.2 Bitcoin	11
3.3 Privacy-enhancing Cryptocurrencies	14
3.4 Mimblewimble	15
3.5 Scriptless Scripts	20
3.6 Adaptor Signatures	20
4 Two Party Fixed Witness Adaptor Signatures	23
4.1 Definitions	23
4.2 Schnorr-based instantiation	25
4.3 Correctness & Security	30
5 Adaptor Signature Based Atomic Swaps Between Bitcoin and a Mimblewimble Based Cryptocurrency	41
5.1 Definitions	42
5.2 Instantiation	45
5.3 Security & Correctness	56
5.4 Atomic Swap protocol	64
6 Implementation	71
6.1 Implementation Bitcoin side	71
6.2 Implementation Grin side	71
6.3 Performance Evaluation	71
List of Figures	73
	ix

List of Tables	75
List of Algorithms	77

Introduction

Pedro: We need to discuss a structure for the introduction. Proposal:

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

Mimblewimble The Mimblewimble protocol was introduced in 2016 by an anonymous entity named Jedusor, Tom Elvis [jedusor2016mimblewimble]. The author's name, as well as the protocols name, are references to the Harry Potter franchise. ¹ In Harry Potter, Mimblewimble is a tongue-typing curse which reflects the goal of the protocol's design, which is improving the user's privacy. Later, Andrew Poelstra took up the ideas from the original writing and published his understanding of the protocol in his paper [poelstra2016mimblewimble]. The protocol gained increasing interest in the community and was implemented in the Grin ² and Beam ³ Cryptocurrencies, which both launched in early 2019. In the same year, two papers were published, which successfully defined and proved security properties for Mimblewimble [fuchsbauer2019aggregate, betarte2019towards].

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

¹https://harrypotter.fandom.com/wiki/Tongue-Tying_Curse

²<https://grin.mw/>

³<https://beam.mw/>

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

Compared to Bitcoin, there are some differences in Mimblewimble:

- Use of Pedersen commitments instead of plaintext transaction values

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

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

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

- Spend outputs are purged from the ledger such that only unspent transaction outputs remain.
- No scripting features.

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

Pedro: An intuition of these two terms is required here

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

By utilizing Pedersen commitments in the transactions, we hide the amounts transferred in a transaction, improving the systems user privacy, but also requiring additional range proofs, attesting to the fact that actual amounts transferred are in between a valid range. Not having any addresses enables transaction merging and transaction cut through, which we will explain in section ??.

However, this comes with the consequence that building transactions require active interaction between the sender and receiver, which is different than in constructions more similar to Bitcoin, where a sender can transfer funds to any address without requiring active participation by the receiver. Through transaction merging and cut-through and some further protocol features, which we will see later in this section, we gain the third mentioned property of being able to delete transaction outputs from the Blockchain, which have already been spent before. This ongoing purging in the Blockchain makes it particularly space-efficient as the space required by the ledger only grows in the number of UTXOs, in contrast to Bitcoin, in which space requirement increases with the number of overall mined transactions. Saving space is especially relevant for Cryptocurrencies employing confidential transactions because the size of the range proofs attached to outputs can be significant.

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

Another advantage of this property is that new nodes joining the system do not have to verify the whole history of the Blockchain to validate the current state, making it much easier to join the network. Another limitation of Mimblewimble- based Cryptocurrencies

is that at least the current construction does not allow scripts, such as they are available in Bitcoin or similar systems. Transaction validity is given solely by a single valid signature plus the balancedness of inputs and outputs. This shortcoming makes it challenging to realize concepts such as multi signatures or conditional transactions which are required for Atomic Swap protocols. However, as we will see in ?? there are ways we can still construct the necessary transactions by merely relying on cryptographic primitives [fuchsbauer2019aggregate].

CHAPTER 2

Motivation & Objectives

TODO

Preliminaries

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

3.1 General Notation and Definitions

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

Definition 3.1 (Hard Relation). Given a language $L_R := \{A \mid \exists a \text{ s.t. } (A, a) \in R\}$ then the relation R is considered hard if the following three properties hold: [aumayr2020bitcoinchannels]

1. $\text{genRel}((1^n))$ is a *PPT* sampling algorithm which outputs a statement/witness of the form $(A, a) \in R$.
2. Relation R is poly-time decidable.
3. For all *PPT* adversaries \mathcal{A} the probability of finding a given A is negligible.

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

$$g^m = n$$

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

Definition 3.3 (Signature Scheme). A signature scheme Φ is a tuple of algorithms (keyGen, sign, verf) defined as follows: [goldwasser1988digital]

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

- $(sk, pk) \leftarrow \text{keyGen}(1^n)$: The keygen function creates a keypair (sk, pk) , the public key can be distributed to the verifier(s) and the secret key has to be kept private.
- $\sigma \leftarrow \text{sign}(m, sk)$: The signing function creates a signature consisting of a variable s and R which is a commitment to the secret nonce k used during the signing process. As an input it takes a message m and the secret key sk of the signer.
- $\{1, 0\} \leftarrow \text{verf}(m, \sigma, pk)$: The verification function allows a verifier knowing the signature σ , message m and the provers public key pk to verify the signatures validity.

A valid signature scheme has to fulfill two security properties:

- Correctness: For all messages m and valid keypairs (sk, pk) the following must hold with overwhelming probability: $\text{verf}(pk, \text{sign}(sk, m), m) \stackrel{?}{=} 1$
- Unforgeability (EUF – CMA): Informally the existential unforgeability under chosen message attacks holds if an attacker \mathcal{A} is unable to forge a valid signature for a chosen message. A formalization of the property can be found in section 4.3.2

Definition 3.4 (Cryptographic Hash Function). A cryptographic hash function H is defined as $H(I) \rightarrow \{0, 1\}^n$ for some fixed number n and some input I [al2011cryptographic]. 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 $H(I_1) := H(I_2)$ 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 $H(I)$. 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 $H(I) \stackrel{?}{=} H(I^*)$.

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

Definition 3.5 (Commitment Scheme). A cryptographic Commitment Scheme COM is defined by a pair of functions ($\text{keyGen}, \text{commit}$) [bunz2018bulletproofs].

- $rs \leftarrow \text{setupCom}(1^n)$: 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} .
- $C \leftarrow \text{commit}(I, k)$ The commit routine is DPT function that takes an arbitrary input $I \in \mathbb{I}_{PP}$, a random value $k \in \mathbb{K}_{PP}$ 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 $k \in \mathbb{K}_{PP}$ the probability of finding a $I^* \neq I$ and a k^* with $\text{commit}(I, k) = \text{commit}(I^*, k^*)$ is negligible.
- *Hiding*: For a PPT adversary \mathcal{A} , commitment inputs $I_0, I_1 \in \mathbb{I}_{PP}$ randomness $k \in \mathbb{K}_{PP}$ and a commitment output $C := \text{commit}(I_b, k)$ the probability of the adversary choosing the correct b out of $\{0, 1\}$ must not be higher than $\frac{1}{2} + \text{negl}(P)$.

Definition 3.6 (Additive Homomorphic Commitment). A Commitment Scheme as defined in 3.5 is said to be additive homomorphic if the following holds [bunz2018bulletproofs]

$$\text{commit}(I_1, k_1) \cdot \text{commit}(I_2, k_2) = \text{commit}(I_1 + I_2, k_1 + k_2)$$

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

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

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

An instantiation of the pedersen commitment scheme must pick two adjacent generators g, h for the setup to be secure in terms of hiding and binding. Formally adjacent means

that there exists a hard relation between g and h in terms of the discrete logarithm 3.2. That means no x is known such that $h = g^x$. In practice this is often achieved by hashing g and using the hash output as h .

To prove the security of our protocols we define the notion of security in the presence of malicious adversaries, which may deviate from the protocol arbitrarily. To construct the definition we must first define two terms, **IDEAL** the execution in the ideal model and **REAL**, the execution in the real model. The following definitions are based on a tutorial paper on simulation proofs by Yehuda Lindell. [[lindell2017simulate](#)]

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 $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$. The adversary \mathcal{A} either controls P_1 or P_2 . The ideal execution **IDEAL** relies on the assumption that we have access to a trusted third party and proceeds in the following steps:

1. **Inputs:** The input of P_1 is x and the input of P_2 is y . Both parties get an additional auxiliary input z . We note that we can generalize the concept to functions which require multiple inputs or even functions which do not require any input. In the case of multiple inputs the inputs of P_1 would then be a list $[x_i]$ and the inputs of P_2 a list $[y_i]$. For the case of simplicity we here describe the case with one single parameter provided by each party.
2. **Send Inputs:** The honest party (the one which is not controlled by \mathcal{A}) sends its input x (resp. y) to the trusted third party. The malicious party can either abort the execution by sending the symbol **abort** to the trusted third party, send its input x (resp. y), or send an arbitrarily chosen string k with the same length to x to proceed with the protocol execution. The decision is made by \mathcal{A} and may depend on the input or 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 $f(x^*, y^*) = (f_1(x^*, y^*), f_2(x^*, y^*))$ and proceeds by sending $f_1(x^*, y^*)$ (respective $f_2(x^*, y^*)$) 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 **continue** which means the output $f_1(x^*, y^*)$ (respective $f_2(x^*, y^*)$) will be delivered to the honest party.

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

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

Execution in the Real Model Again let \mathcal{A} be a non-uniform PPT adversary and $i \in \{1, 2\}$ be the index of the corrupted party. In this model a real two-party protocol γ is executed but the adversary \mathcal{A} sends all 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 $\text{REAL}_{f, P(z), i}(x, z)$ and is defined as the output pair of the honest party and the adversary \mathcal{A} from the real execution of γ .

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

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

where $|x| = |y|$ and $z = \text{poly}(|x|)$. [lindell2017simulate]

3.2 Bitcoin

In this section we will discuss the basics of the Bitcoin transaction protocol. We will find definitions which we will use later in section 5.4 to construct an atomic swap protocol. The main reference of this section is the book Mastering Bitcoin by Andreas Antonopoulos [antonopoulos2014mastering].

3.2.1 Bitcoin Transaction Protocol

A *Bitcoin Transaction* is a data structure which allows transferring value between participants of the network. In Bitcoin there are no user balances or user accounts, instead the UTXO model (unspent transaction outputs) is employed. An UTXO is a output constructed in a previous transaction which holds value in the form of an amount expressed in Bitcoin (more precisely in Satoshis, which is the smallest unit of Bitcoin) and a locking condition (referred to as scriptPubKey). Unspent means that this output has not been spent yet in a transaction and its funds are therefore available to be redeemed by a participant capable of unlocking the output. To unlock this value one has to provide a script fulfilling the locking condition, referred to as scriptSig. In the most common

case the lock condition will be to provide a valid signature under a public key. This is referred to as a P2PK or P2PKH output which we will see in more detail in section 3.2.1. However, more complex conditions, such we shall see in section 3.2.1 are possible.

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

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

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

<div style="border-bottom: 1px solid black; margin-bottom: 10px;"> createUTXO(v, spk) </div> <div style="margin-bottom: 10px;"> 1 : return $\psi := \{v := v, spk := spk, \sigma := \emptyset\}$ </div> <div style="border-bottom: 1px solid black; margin-bottom: 10px;"> spendUTXO(ψ, σ) </div> <div style="margin-bottom: 10px;"> 1 : $\{v, spk\} \leftarrow \psi$ 2 : return $\psi := \{v := v, spk := spk, \sigma := \sigma\}$ </div> <div style="border-bottom: 1px solid black; margin-bottom: 10px;"> verfUTXO(ψ) </div> <div> 1 : $\{v, spk, \sigma\} \leftarrow \psi$ 2 : return verf(spk, σ, v) </div>
--

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

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

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

```

{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig": "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298c2",
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}

```

Figure 3.1: A decoded Bitcoin transaction¹

A transaction is valid if the following conditions are fulfilled:

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

Definition 3.11 (Bitcoin Transaction Scheme). We define a Bitcoin Transaction scheme as a tuple of three DPT functions (buildTransaction , signTransaction , verfTransaction).

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

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

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

P2PK, P2PKH

P2PK stands for Pay-to-Public-Key and P2PKH for Pay-to-Public-Key-Hash. In this type of output spk will be constructed such that its value unlocks if a correct signature is provided in σ for a corresponding public key pk . P2PKH is an update to this script in which the spk contains a hashed version of the public key pk , instead of the public key itself. To spend a P2PKH output one has to provide the unhashed public key in addition to a valid signature. This type of output, is the most commonly used output in the Bitcoin blockchain to transfer value from one participant to another. Delgado et al. found in their paper Analysis of the Bitcoin UTXO set from 2017 that more than 80% of the UTXO set at that time consisted of P2PKH transactions, whereas about 17% were P2SH and 0.12% P2PK outputs. [delgado2018analysis] 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 simple 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 as well as the unlocking requirements for the script itself. Upon verification the hash of the provided script will be computed and compared with the value given in the locking condition, if those match the actual script will be executed. The advantages of using this approach over just handcrafting a custom locking script is that the locking scripts are rather short making the transactions smaller and therefore reducing fees, or rather shifting the fees from the sender to the owner of the output. Additionally this type of output can be encoded again into a Bitcoin address similar to a P2PKH output, making it easy to request a payment.

3.3 Privacy-enhancing Cryptocurrencies

3.3.1 Zero Knowledge Proofs

3.3.2 Range Proofs

Definition 3.12 (Rangeproof System). A Rangeproofs system $\Pi[COM]$ with regards to a homomorphic commitment scheme COM consists of a tuple of functions $(\text{ranPrfSetup}, \text{ranPrf}, \text{vrfRanPrf})$.

- $ps \leftarrow \text{ranPrfSetup}(1^n, i, j)$: The rangeproof setup algorithm takes as input a security parameter 1^n as well as two numbers i and j which are treated as exponents of 2 to define the lower and upper bound of the rangeproof protocol.
- $\pi \leftarrow \text{ranPrf}(C, v, r)$: The proof algorithm is a DPT function which takes as input a commitment C a value v and a blinding factor r . It will output a proof π attesting to the statement that the value v of commitment C is in between the range $\langle lb, ub \rangle$ as defined during the ranPrfSetup function.
- $\{1, 0\} \leftarrow \text{vrfRanPrf}(\pi, C)$: The proof verification algorithm is a DPT function which verifies the validity of the proof π with regards to the commitment C . It will output 1 upon a successful verification or 0 otherwise.

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

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

For MP proofs [klinec2020privacy]

3.4 Mimblewimble

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

Transaction Structure

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

Definition 3.14 (Mimblewimble Coin). For two adjacent elliptic curve generators g and h a coin in Mimblewimble is a tuple of the form (C, π) , where $C := g^v \cdot h^k$ is a Pedersen Commitment [pedersen1991non] to the value v with blinding factor k . 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 practice $\langle 0, 2^{64} - 1 \rangle$ is used in the most prominent implementations.

A Mimblewimble transaction consists of $\mathcal{C}_{inp} := (C_1, \dots, C_n)$ input coins and $\mathcal{C}_{out} := (C'_1, \dots, C'_n)$ output coins.

Definition 3.15 (Transaction well-balancedness). A transaction is considered *well-balanced* iff $\sum v'_i - \sum v_i = 0$ so the sum of all output values subtracted from the sum of input values has to be 0. (Not taking transaction fees into account)

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

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

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

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

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

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

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

$$\mathcal{E} := g^{(\sum k'_i - \sum k_i)} \cdot h^s$$

Finally a Mimblewimble transaction is of form:

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

where s is the transaction supply amount, \mathcal{C}_{inp} is the list of input coins, \mathcal{C}_{out} is the list of output coins and K is the transaction Kernel. The Kernel consists of $\{\pi\}$ which is a set of all output coin range proofs, $\{\mathcal{E}\}$ a set of excess values and finally $\{\sigma\}$ a set of signatures [fuchsbauer2019aggregate]. Even though normally a transaction would only require a single excess value and signature, for reasons we will see in the next section these fields always have to be lists instead of just a single value.

Transaction Merging

An intriguing property of the Mimblewimble protocol is that two transactions can easily be merged into a single one, which is essentially a non-interactive version of the Coin-Join protocol on Bitcoin [maxwell2013coinjoin]. Assume we have the following two

transactions:

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

Then we can build a single merged transaction:

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

We can easily deduce that if tx_0 and tx_1 are valid, it must follow that tx_m is valid:

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

By the rules of arithmetic it then must also hold that

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

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

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

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

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, σ_0 or \mathcal{E}_1, σ_1 . Thereby the attacker was able to reconstruct one of the original transactions from which tx_m was constructed. Following this method he might be able to uncover all original transactions from the merged one.

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

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

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

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

Since o is now also included in the transaction kernel and therefore known to the verifier, the public verification is still possible. Now every time two transactions are merged with the method layed out previously, the two individual offset values o_0, o_1 are combined into a single value o_m . If offsets are picked truly randomly, and the possible range of values is broad enough, the probability of recovering the uncombined offsets from a merged one becomes negligible, making it infeasible to recover original transactions from a merged one [poelstra2016mimblewimble].

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 :

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

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

$$\sum \mathcal{C}_{out} - \sum \mathcal{C}_{inp} = \sum (g^{k'_i}) - \sum (g^{k_i})$$

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

Transaction Building

As already pointed out, building transactions in Mimblewimble is an interactive process between the sender and receiver of funds. Jedusor, Tom Elvis originally envisioned the following two-step process to build a transaction: [jedusor2016mimblewimble]

Assume Alice wants to transfer coins of value p to Bob.

1. Alice first selects an input coin \mathcal{C}_{inp} (or potentially multiple) in her control with total stored value v with $v \geq p$. She then creates change coin outputs \mathcal{C}_{out}^A (could again be multiple) with the remainder of her input value substracted by the value send to Bob. For her newly created output coins and her input coins she calculates her part of discrete logarithm x (her part of the key) to the final \mathcal{E} and sends all this information to Bob as a pre-transaction.
2. Bob creates himself additional output coins \mathcal{C}_{out}^B of total value p and similar to Alice creates his share x^* of the discrete logarithm of \mathcal{E} . Together with the share received by Alice he can now create a signature to \mathcal{E} and finalize the transaction



Figure 3.2: Original transaction building process

Figure 3.2 depicts the original transaction flow.

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

In detail this reverting transaction would look like:

$$tx_{rv} := (0, \mathcal{C}_{out}^A \parallel \mathcal{C}_{out}^B, \mathcal{C}_{inp}, (\pi_{rv}, \mathcal{E}_{rv}, \sigma_{rv}))$$

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:

$$\mathcal{E}_{rv} := \sum \mathcal{C}_{out}^A \parallel \mathcal{C}_{out}^B - \mathcal{C}_{inp}$$

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

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

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

In essence this means Bob spends the newly created outputs and sends them back to the original input coins, chosen by Alice. It might at first seem unclear why Bob would do

that. An example situation could be if Alice pays Bob for some good which Bob is selling. Alice decides to pay in advance, but then Bob discovers that he is already out of stock of the good that Alice ordered. To return the funds to Alice, he reverses the transaction instead of participating in another interactive process to build a new transaction with new outputs. If Alice already deleted the keys to her initial coins, the funds are now lost. The problem was solved in the Grin and Beam Mimblewimble implementations by making the signing process itself a two-party process which will be explained in more detail in chapter 4.

Alternatively Fuchsbauer et al. [fuchsbauer2019aggregate] proposed another way to build transactions which would not be vulnerable to this problem:

1. Alice constructs a full-fledged transaction tx_A spending her input coins \mathcal{C}_{inp} and creates her change coins \mathcal{C}_{out}^A , plus a special output coin $\mathcal{C}_{out}^{sp} := h^p \cdot g^{x_{sp}}$, 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}_{out}^{sp} to create an output only he controls \mathcal{C}_{out}^B 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}_{out}^{sp} will be both in the coin output and input list of the transaction and therefore will be discarded.

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

3.5 Scriptless Scripts

3.6 Adaptor Signatures

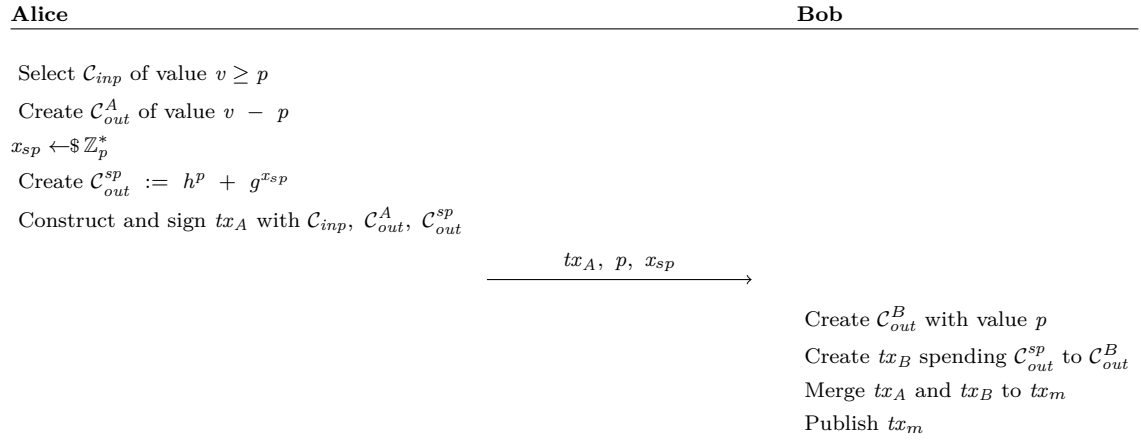


Figure 3.3: Salvaged transaction protocol by Fuchsbauer et al. [fuchsbauer2019aggregate]

Two Party Fixed Witness Adaptor Signatures

In this chapter, we will define a variant of the adaptor signature scheme as explained in section 3.6. The main difference in the protocol outlined in this thesis is that one of the two parties does know the fixed secret witness before the start of the protocol. The aim of the protocol will then be that the other person is able to extract the witness from the final signature. This feature can then be leveraged to build an Atomic Swap protocol as we will show in 5.

First we will define the general two-party signature creation protocol as it is currently implemented in Mimblewimble-based Cryptocurrencies. We reduce the generated signatures to the general case [schnorr1989efficient] and prove its correctness. From this two-party protocol, we then derive the adapted variant, which allows hiding a fixed witness value in the signature, which can be revealed only by the other party after attaining the final signature.

We start by defining our extended signature scheme in section 4.1, proceed by providing a schnorr-based instantiation of the protocol in section 4.2 and finally prove its security in section 4.3.

4.1 Definitions

A two-party signature scheme is an extension of a signature scheme as defined in definition 3.3, which allows us to distribute signature generation for a composite public key shared between two parties Alice and Bob. Alice and Bob want to collaborate to generate a signature valid under the composite public key $pk := pk_A \cdot pk_B$ without having to reveal their secret keys to each other.

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

- $((sk_A, pk_A, k_A, \Lambda), (sk_B, pk_B, k_B, \Lambda)) \leftarrow \text{dKeyGen}\langle 1^n, 1^n \rangle$: The distributed key generation protocol takes as input the security parameter from both Alice and Bob and returns the tuple $(sk_A, pk_A, k_A, \Lambda)$ to Alice (similar to Bob) where (sk_A, pk_A) is a pair of private and corresponding public keys, k_A a secret nonce and Λ is the signature context containing parameters shared between Alice and Bob. We introduce Λ for the participants to share as well as update parameters with each other during the protocol execution.
- $(\tilde{\sigma}_A) \leftarrow \text{signPt}(m, sk_A, k_A, \Lambda)$: The partial signing algorithm is a DPT function that takes as input the message m and the share of the secret key sk_A and nonce k_A (similar for Bob) as well as the shared signature context Λ . The procedure outputs $(\tilde{\sigma}_A)$, that is, a share of the signature to a participant.
- $\{1, 0\} \leftarrow \text{vrfPt}(\tilde{\sigma}_A, m, pk_A)$: The share verification algorithm is a DPT function that takes as input a signature share $\tilde{\sigma}_A$, a message m , and the other participants public key pk_A (similar pk_B for Bobs partial signature). The algorithm returns 1 if the verification was successful or 0 otherwise.
- $\sigma_{fin} \leftarrow \text{finSig}(\tilde{\sigma}_A, \tilde{\sigma}_B)$: 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 shared public key pk .

We require the two party signature scheme to be correct as well as secure as of definition 3.8. For the correctness of the distributed key-generation protocol dKeyGen , special care needs to be taken to guarantee a uniformly random distribution of generated keys as pointed out by Lindell and Yehuda in [lindell2017fast].

Definition 4.2 (Two Party Fixed Witness Adaptor Schnorr Signature Scheme). From the definition 4.1, we now derive an adapted signature scheme Φ_{Apt} , which allows one of the participants to hide the discrete logarithm x of a statement $X := g^x$ chosen at the beginning of the protocol. Again we extend our previously defined signature scheme with new functions:

$$\Phi_{Apt} := (\Phi_{MP} \parallel \text{adaptSig} \parallel \text{vrfAptSig} \parallel \text{extWit})$$

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

- $\{1, 0\} \leftarrow \text{vrfAptSig}(\hat{\sigma}_A, m, pk_A, X)$: The verification algorithm is a DPT function that takes as input an adapted partial signature $\hat{\sigma}$, the other participants public keys and a statement X . The function will verify the partial signature's validity as well that it contains the secret witness x .
- $x \leftarrow \text{extWit}(\sigma_{fin}, \tilde{\sigma}_A, \hat{\sigma}_B)$: The witness extraction algorithm is a DPT function that lets Alice extract the secret witness x from the final composite signature. Note that to extract the witness x the partial signatures shared between the participants beforehand and the statement X needs to be provided as inputs. This means that for executing this function one needs to first learn the partial signatures exchanged between the parties.

Note that our definition of the adaptor signature scheme is different from the definition seen in 3.6. This has the reason that we require our scheme to be able to hide a secret chosen before the signing protocol has been started. One of the participants will be able to hide this secret during the distributed signing protocol which the other party can extract after completion of the protocol. This feature is a requirement for our signature scheme such that we can build the atomic swap protocol which will be laid out at a later point in the thesis.

Similar to how it is defined in [aumayr2020bitcoinchannels] additionally to regular Correctness as defined in 3.3 we require our signature scheme to satisfy Adaptor Signature Correctness. This property is given when every adapted partial signature generated by adaptSig can be completed into a final signature for all pairs $(x, X) \in R$, from which it will be possible to extract the witness computing extWit with the required parameters.

Definition 4.3 (Adaptor Signature Correctness). More formally *Adaptor Signature Correctness* is given if for every security parameter $n \in \mathbb{N}$, message $m \in \{0, 1\}^*$, keypairs $((sk_A, pk_A, k_A, \Lambda), (sk_B, pk_B, k_B, \Lambda)) \leftarrow \text{dKeyGen}(1^n, 1^n)$ with their composite public key $\Lambda.pk$ and every statement/witness pair $(X, x) \in \text{genRel}(1^n)$ in a relation R it must hold that:

$$\Pr \left[\begin{array}{l} \text{verf}(m, \sigma_{fin}, \Lambda.pk) = 1 \\ \wedge \\ \text{vrfAptSig}(\hat{\sigma}_B, m, pk_B, X) = 1 \\ \wedge \\ (X, x^*) \in R \end{array} \middle| \begin{array}{l} \tilde{\sigma}_A \leftarrow \text{signPt}(m, sk_A, k_A, \Lambda) \\ \tilde{\sigma}_B \leftarrow \text{signPt}(m, sk_B, k_B, \Lambda) \\ \hat{\sigma}_B \leftarrow \text{adaptSig}(\tilde{\sigma}_B, x) \\ \sigma_{fin} \leftarrow \text{finSig}(\tilde{\sigma}_A, \tilde{\sigma}_B) \\ x^* \leftarrow \text{extWit}(\sigma_{fin}, \tilde{\sigma}_A, \hat{\sigma}_B) \end{array} \right] = 1.$$

4.2 Schnorr-based instantiation

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

A concrete implementation can be seen in figure 4.1. The signature scheme is called schnorr signature scheme, first defined in [schnorr1989efficient] and is widely employed

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

Figure 4.1: Schnorr Signature Scheme as first defined in [schnorr1989efficient]

in many cryptography systems. Correctness of the scheme is easy to derive. As s is calculated as $k + e \cdot sk$, when generator g is raised to s , we get $g^{k + e \cdot sk}$ which we can transform into $g^k \cdot g^{sk \cdot e}$, and finally into $R \cdot pk^e$ which is the same as the right side of the equation.

From the regular schnorr signature we now provide an instantiation for the two-party case defined in definition 4.1. 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 will build our adapted scheme.

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

$\text{setupCtx}(\Lambda, pk_A, R_A)$
1 : $\Lambda.pk := \Lambda.pk \cdot pk_A$
2 : $\Lambda.R := \Lambda.R \cdot R_A$
3 : return Λ

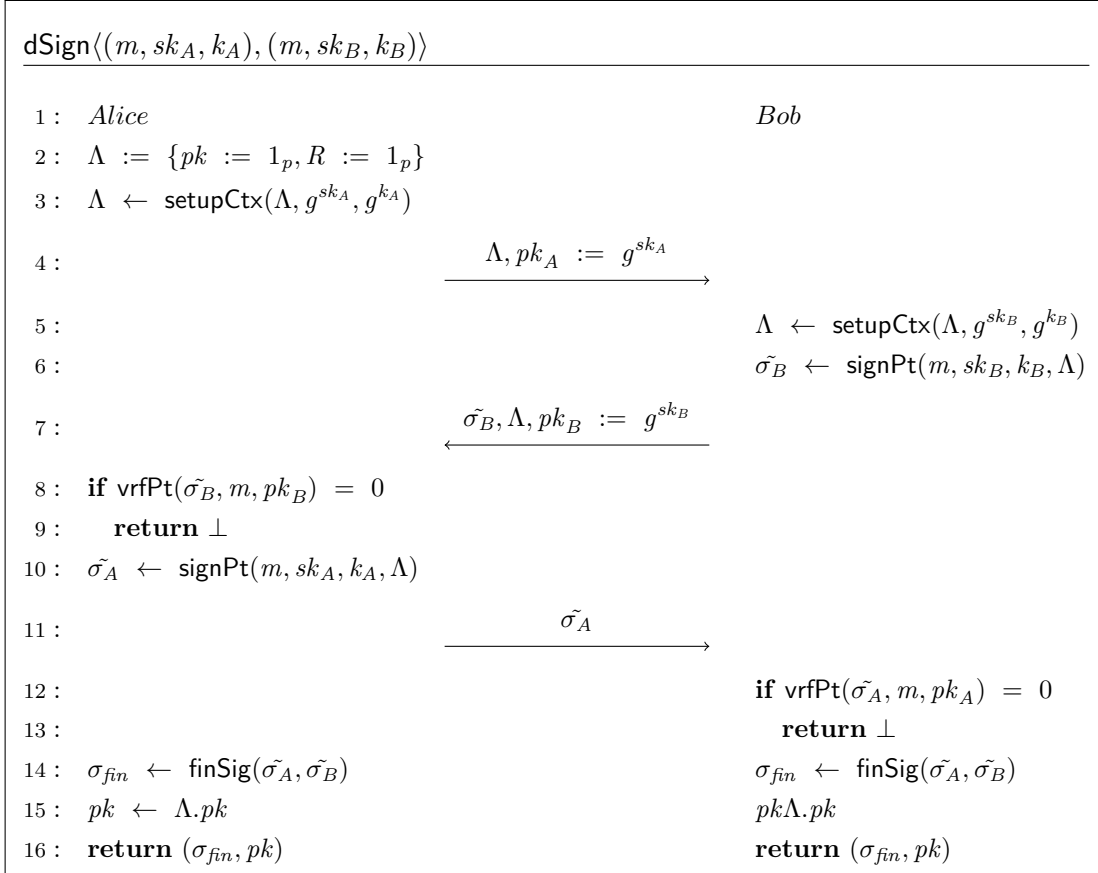
This function helps the participants to setup and update the signature context shared between them. In figure 4.2 we show a concrete instantiation of the protocol and functions. In `dKeyGen` Alice and Bob will each randomly chose their secret key and nonce. They further require to create a zero-knowledge proof attesting to the fact that they have generated their key before any message was exchanged. This is essential for the scheme to achieve EUF-CMA as described by Lindell et al. [lindell2017fast].

In `dKeyGen` Alice will initially setup the signature context and send it to Bob, together with her public and zk-proof. Bob verifies the proof (and exits if it is invalid). He will proceed by adding his parameters to the signature context and send it back to Alice, together with his public key and zk-proof, which Alice will verify.

`signPt` and `vrftPt` are generally similiar to the instantiation of the normal schnorr signature scheme. Note however that for computing the schnorr challenge e the input into the hash

function will be the combined public key pk and combined nonce commitment R , which the participants can read from the context object Λ . This has the effect that the partial signature itself are not yet a valid signature (neither under pk nor under pk_A or pk_B). This is because to be valid under pk the partial signatures are missing the s values from the other participants. They are also not valid under the partial public keys pk_A or pk_B because the schnorr challenge is computed already with the combined values. There we have to introduce the slightly adjusted vrfPt to be able to verify specifically the partial signatures.

We further formalize a protocol dSign which is a protocol between two parties running the partial signature creation outlined before. Note that we assume that the secret keys as well as nonces used in the signatures have already been generated, for example by running the dKeyGen protocol. Both parties input the shared message m as well as their secret keys and secret nonces. The protocol outputs a signature σ_{fin} to the message σ_{fin} , 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 with the composite public key $pk := pk_A \cdot pk_B$. A verifier knowing the signed message m , the final signature σ_{fin}

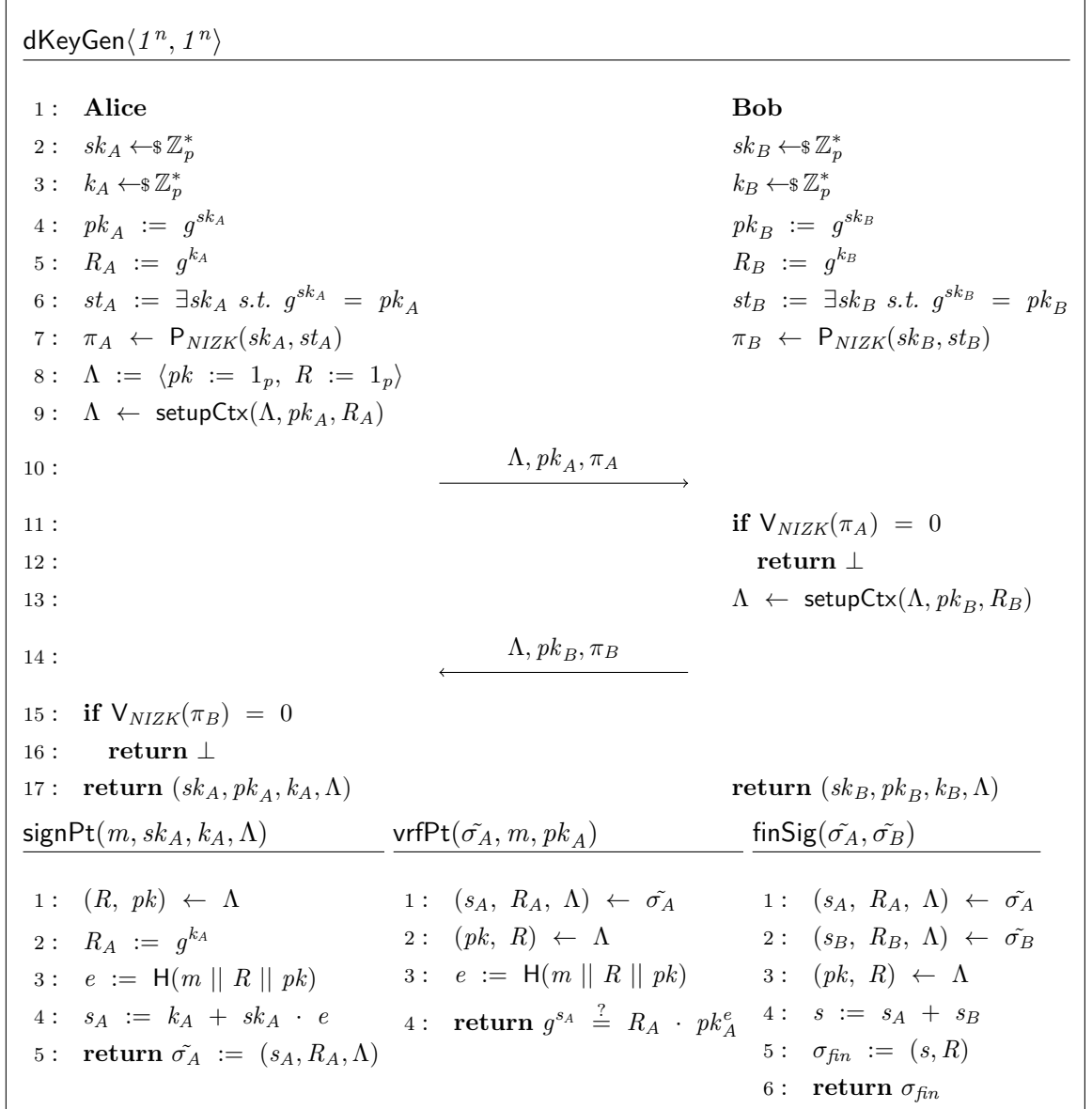


Figure 4.2: Two Party Schnorr Signature Scheme

<u>adaptSig($\tilde{\sigma}, x$)</u>	
1: $(s, R_A, \Lambda) \leftarrow \tilde{\sigma}$	
2: $s^* := s + x$	
3: return $\hat{\sigma} := (s^*, R_A, \Lambda)$	
<u>vrfAptSig($\hat{\sigma}_A, m, pk_A, X$)</u>	<u>extWit($\sigma_{fin}, \tilde{\sigma}_A, \hat{\sigma}_B$)</u>
1: $(s_A, R_A, \Lambda) \leftarrow \hat{\sigma}_A$	1: $(s, R) \leftarrow \sigma_{fin}$
2: $(pk, R) \leftarrow \Lambda$	2: $(s_A, R_A, \Lambda) \leftarrow \tilde{\sigma}_A$
3: $e := H(m \parallel R \parallel pk)$	3: $(\hat{s}_B, R_B, \Lambda) \leftarrow \hat{\sigma}_B$
4: return $g^{s_A} \stackrel{?}{=} R_A \cdot pk_A^e \cdot X$	4: $s_B := s - s_A$
	5: $x := \hat{s}_B - s_B$
	6: return (x)

Figure 4.3: Fixed Witness Adaptor Schnorr Signature Scheme

and the composite public key pk can now verify the signature using the regular `verf` procedure.

Note that this way of computing schnorr signatures is not new. For a proof of its correctness and a more extensive explanation we refer the reader to a paper by Maxwell et al. [maxwell2019simple].

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

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

We now define a protocol `dAptSign` between Alice and Bob creating a signature σ_{fin} for the composite public key $pk := pk_A \cdot pk_B$. Now Bob will hide his secret x which Alice can extract after the signing process has completed. One thing to note is that in this protocol only Bob is able to call `finSig` to create the final signature, which is different to the previous protocol. This is because the function requires Bobs unadapted partial signature $\tilde{\sigma}_B$ as input, which Alice does not know. (She only knows Bobs adapted variant). 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.

$\text{dAptSig}((m, sk_A, k_A), (m, sk_B, k_B, x))$	
<pre> 1 : Alice 2 : $\Lambda := \{pk := 1_p, R := 1_p\}$ 3 : $\Lambda \leftarrow \text{setupCtx}(\Lambda, g^{sk_A}, g^{k_A})$ 4 : 5 : 6 : 7 : 8 : 9 : 10 : 11 : if $\text{vrfAptSig}(\tilde{\sigma}_B, m, pk_B, X) = 0$ 12 : return \perp 13 : $\tilde{\sigma}_A \leftarrow \text{signPt}(m, sk_A, k_A, \Lambda)$ 14 : 15 : 16 : 17 : 18 : 19 : $pk \leftarrow \Lambda.pk$ 20 : if $\text{verf}(m, \sigma_{fin}, pk) = 0$ 21 : return \perp 22 : $x \leftarrow \text{finSig}(\tilde{\sigma}_A, \hat{\sigma}_B)$ 23 : return $(x, (\sigma_{fin}, pk))$ </pre>	<pre> Bob 1 : 2 : 3 : 4 : $\Lambda, pk_A := g^{sk_A}$ 5 : 6 : $\Lambda \leftarrow \text{setupCtx}(\Lambda, g^{sk_B}, g^{k_B})$ 7 : $\tilde{\sigma}_B \leftarrow \text{signPt}(m, sk_B, k_B, \Lambda)$ 8 : $\hat{\sigma}_B \leftarrow \text{adaptSig}(\tilde{\sigma}_B, x)$ 9 : $pk_B := g^{sk_B}$ 10 : $X := g^x$ 11 : 12 : 13 : 14 : $\hat{\sigma}_B, \Lambda, pk_B, X$ 15 : 16 : 17 : 18 : 19 : 20 : 21 : 22 : 23 : 24 : 25 : 26 : 27 : 28 : 29 : 30 : 31 : 32 : 33 : 34 : 35 : 36 : 37 : 38 : 39 : 40 : 41 : 42 : 43 : 44 : 45 : 46 : 47 : 48 : 49 : 50 : 51 : 52 : 53 : 54 : 55 : 56 : 57 : 58 : 59 : 60 : 61 : 62 : 63 : 64 : 65 : 66 : 67 : 68 : 69 : 70 : 71 : 72 : 73 : 74 : 75 : 76 : 77 : 78 : 79 : 80 : 81 : 82 : 83 : 84 : 85 : 86 : 87 : 88 : 89 : 90 : 91 : 92 : 93 : 94 : 95 : 96 : 97 : 98 : 99 : 100 : </pre>

4.3 Correctness & Security

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

4.3.1 Adaptor Signature Correctness

To prove that Adaptor Signature Correctness holds we have 3 statements to prove, first we prove that $\text{verf}(m, \sigma_{fin}, \Lambda.pk) \stackrel{?}{=} 1$ holds in our schnorr-based instantiation of the signature scheme, whereas Λ is setup such that $pk = pk_A \cdot pk_B$.

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

$$g^s = R \cdot pk^e \quad (4.1)$$

$$g^{s_A} \cdot g^{s_B} \quad (4.2)$$

$$g^{k_A + e \cdot sk_A} \cdot g^{k_B + e \cdot sk_B} \quad (4.3)$$

$$g^{k_A} \cdot pk_A^e \cdot g^{k_B} \cdot pk_B^e \quad (4.4)$$

$$R_A \cdot pk_A^e \cdot R_B \cdot pk_B^e \quad (4.5)$$

$$R \cdot pk^e = R \cdot pk^e \quad (4.6)$$

$$1 = 1 \quad (4.7)$$

It remains to prove that with the same setup $\text{vrfAptSig}(\hat{\sigma}_B, m, pk_B, X) \stackrel{?}{=} 1$ and $(X, x^*) \in R$ hold.

$$\text{vrfAptSig}(\hat{\sigma}_B, m, pk_B, X) \stackrel{?}{=} 1$$

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

$$g^{\hat{\sigma}_B} = R_B \cdot pk_B^e \cdot X \quad (4.8)$$

$$g^{\tilde{\sigma}_B + x} \quad (4.9)$$

$$g^{k_B + sk_B \cdot e + x} \quad (4.10)$$

$$g^{k_B} \cdot g^{sk_B \cdot e} \cdot g^x \quad (4.11)$$

$$R_B \cdot pk_B^e \cdot X = R_B \cdot pk_B^e \cdot X \quad (4.12)$$

$$1 = 1 \quad (4.13)$$

We now continue to prove the last equation required:

$$((X, x^*) \in R)$$

We do this by showing that x is calculated correctly in `extWit`:

$$x := \hat{s} - (s - s_A) \quad (4.14)$$

$$\hat{s} - ((s_A + s_B) - s_A) \quad (4.15)$$

$$s_B + x - (s_B) \quad (4.16)$$

$$x := x \quad (4.17)$$

$$(4.18)$$

□

4.3.2 Security

We have shown that the outlined signature scheme is correct, next we have to prove its security. Our goal is to proof security in the malicious setting (as defined in 3.8) that means the adversary might or might not behave as specified by the protocol. For achieving this we will prove security for both the **dSign** and **dAptSign** protocols in the hybrid model which was layed out by Yehuda Lindell in [**lindell2017simulate**]. In particular, we will use the \mathbf{f}_{zk}^R -model in which we assume that we have access to a constant-round protocol \mathbf{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 and a statement X . One party provides the witness statment pair (x, X) , the second the statement X^* . If $X = X^*$ the functionality returns 1, otherwise 0. More formally:

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

That a constant-round zero-knowledge proof of knowledge exists was proven in [**lindell2013note**]. We recall from 3.3.1 that a secure zero-knowledge proof must fulfill Completeness, Soundness and Zero-Knowledge.

Proof idea In order to construct our simulation proof in the hybrid-model we make some adjustments to the **dSign** protocol utilizing the capabilities of the \mathbf{f}_{zk}^R functionality:

$\text{dSign}((m, sk_A, k_A), (m, sk_B, k_B))$

<pre> 1 : Alice 2 : ... 3 : $f_{zk}^{R_1}((sk_A, pk_A))$ 4 : $f_{zk}^{R_2}((k_A, R_A))$ 5 : 6 : 7 : 8 : 9 : 10 : 11 : 12 : ... 13 : if $f_{zk}^{R_3}(pk_B) = 0 \vee$ 14 : $f_{zk}^{R_4}(\Lambda.R \cdot R_A^{-1}) = 0$ 15 : return \perp 16 : ... 17 : return (σ_{fin}, pk) </pre>	$\xrightarrow{\Lambda, pk_A, R_A}$ $\xleftarrow{\tilde{\sigma}_B, \Lambda, pk_B}$	<pre> Bob ... if $f_{zk}^{R_1}(pk_A) = 0 \vee f_{zk}^{R_2}(\Lambda.R) = 0$ return \perp $f_{zk}^{R_3}((sk_B, pk_B))$ $f_{zk}^{R_4}((k_B, R_B))$... return (σ_{fin}, pk) </pre>
--	--	--

That means both Alice and Bob will verify the validity of the public key and nonce commitments of the other party and will stop protocol execution in case an invalid value has been sent. We assume parties have access to a trusted third party computing f_{zk}^R which will return 1 iff $pk_A = pk_A^*$ (where pk_A^* is the public key that Bob received from Alice) and $pk_A = g^{sk_A}$. (The same holds for the reversed case)

Theorem 1. Assume we have two key pairs (sk_A, pk_A) and (sk_B, pk_B) which were setup securely as for instance with the distributed keygen protocol dKeyGen . Then dSign securely computes a signature σ_{fin} under the composite public key $pk := pk_A \cdot pk_B$ in the $(f_{zk}^{R_1}, f_{zk}^{R_2}, f_{zk}^{R_3}, f_{zk}^{R_4})$ -model.

Proof. We proof security of the protocol by constructing a simulator \mathcal{S} who is given output (σ_{fin}, pk) from a TTP (trusted third party) that securely computes the protocol in the ideal world, receiving the inputs from Alice and Bob. From this output the simulator \mathcal{S} will output a transcript which is indistinguishable from the protocol transcript in the real world in which the corrupted party is controlled by a deterministic polynomial adversary \mathcal{A} . The simulator uses the calls to $(f_{zk}^{R_1}, f_{zk}^{R_2}, f_{zk}^{R_3}, f_{zk}^{R_4})$ in order to do this.

Furthermore we assume that the message m is known to both Alice and Bob. We have to proof two cases, one in which Alice 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 (k_A, R_A) that \mathcal{A} sends to $f_{zk}^{R_1}$. In case those values are invalid (not in the specified relation) or the adversary does not call them at all \mathcal{S} sends **abort** to the TTP and outputs \perp .
2. Next \mathcal{S} receives the message (Λ, pk_A^*, R_A^*) 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 \perp , otherwise he will send the inputs (m, sk_A, k_A) and receive back (σ_{fin}, pk) .
3. \mathcal{S} calculates pk_B, R_B and $\tilde{\sigma}_B$ as follows:

$$\begin{aligned}
 (s, R) &\leftarrow \sigma_{fin} \\
 pk_B &:= pk \cdot pk_A^{-1} \\
 R_B &:= R \cdot R_A^{-1} \\
 \Lambda &\leftarrow \text{setupCtx}(\Lambda, pk_B, R_B) \\
 \tilde{\sigma}_A &\leftarrow \text{signPt}(m, sk_A, k_A, \Lambda) \\
 (s_A, R_A, \Lambda) &\leftarrow \tilde{\sigma}_A \\
 s_B &:= s - s_A \\
 \tilde{\sigma}_B &:= (s_B, R_B, \Lambda)
 \end{aligned}$$

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 $f_{zk}^{R_3}$ and f_{zk}^4 (as the verifier) we check equality with pk_B (respective R_B) and thereafter send back either 0 or 1.
6. Eventually \mathcal{S} will receive $\tilde{\sigma}_A^*$ from \mathcal{A} and checks if $\tilde{\sigma}_A = \tilde{\sigma}_A^*$. If they are indeed the same the simulator will send **continue** to the TTP and output (σ_{fin}, pk) , otherwise he will send **abort** and output \perp .

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) to $f_{zk}^{R_1}$, (k_A, R_A) to $f_{zk}^{R_2}$ and (Λ, pk_A, R_A) to Bob **(2)** Bob sends pk_A to $f_{zk}^{R_1}$, Λ, R to $f_{zk}^{R_2}$, (sk_B, pk_B) to $f_{zk}^{R_3}$, (k_B, R_B) to $f_{zk}^{R_4}$ and $(\tilde{\sigma}_B, \Lambda, pk_B)$ to Alice **(3)** Alice sends pk_B to $f_{zk}^{R_3}$, R_B to $f_{zk}^{R_4}$ and finally $\tilde{\sigma}_A$ to Bob.

- *Phase 1* Since \mathcal{A} is required to be deterministic, the distribution is identical to a real execution. Also in the case the Alice does not send a message, or sends invalid values which will lead Bob to output \perp we also output \perp in the simulation, which again is indistinguishable.

- *Phase 2* As \mathcal{S} managed to calculate Bobs $\tilde{\sigma}_B, pk_B, R_B$ from the final (σ_{fin}, pk) and none of the values depend on any random tape we can say that the values sent in the ideal model are identical to those in the real model. As Bob in this case is the honest party, we don't have to consider any deviation from the protocol specification.
- *Phase 3* The messages sent by the deterministic \mathcal{A} again have to be identical to the real execution and finally the output σ_{fin}, pk or \perp if \mathcal{A} made Bob are the identical values to the real protocol.

We have shown that the distributions in each phase are indeed identical, which proves the indistinguishability of the two transcripts in the case Alice is corrupted.

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

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

$$\begin{aligned}\Lambda &:= \{pk := 1, R := 1\} \\ \Lambda &\leftarrow \text{setupCtx}(\Lambda, g^{sk_A}, g^{k_A})\end{aligned}$$

2. \mathcal{S} now invokes \mathcal{A} and sends (Λ, pk_A, R_A) as if coming from Alice.
3. When \mathcal{A} calls $f_{zk}^{R_1}$ and $f_{zk}^{R_2}$ (as verifier) \mathcal{S} checks equality to the parameters he sent in step 1 and returns either 1 or 0. When \mathcal{A} calls $f_{zk}^{R_3}((sk_B, pk_B))$ and $f_{zk}^{R_4}((k_B, R_B))$ the simulator saves those values to its memory. If \mathcal{A} does not call $f_{zk}^{R_3}$ and $f_{zk}^{R_4}$ or sends invalid values which are not in a relation as specified, \mathcal{S} sends **abort** to the TTP and outputs \perp .
4. Now \mathcal{S} externally sends the inputs (m, sk_B, k_B) to the TTP and receives back (σ_{fin}, pk)
5. When \mathcal{A} queries $H(m \parallel R_A \cdot R_B \parallel pk_A \cdot pk_B)$ \mathcal{S} sends back e^* such that:

$$\begin{aligned}\sigma_{fin} &= k_A + sk_A \cdot e^* + k_B + sk_B \cdot e^* \\ e^* &= \frac{\sigma_{fin} - k_A - k_B}{sk_A + sk_B}\end{aligned}$$

6. \mathcal{S} receives $(\tilde{\sigma}_B, \Lambda, pk_B)$ from \mathcal{A} . (In case he does not \mathcal{S} sends **abort** to the TTP and outputs \perp). He verifies the values sent to him by comparing them with pk_B and R_B from its memory, if they are found to be invalid \mathcal{S} sends **abort** to the TTP, otherwise it sends **continue**.

7. \mathcal{S} calculates as defined in the protocol as $\tilde{\sigma}_A \leftarrow \text{signPt}(m, sk_A, k_A, \Lambda)$ and then sends it to \mathcal{A} as if coming from Alice and finally outputs (σ_{fin}, pk) .

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

- **Phase 1:** The values (pk_A, R_A) sent by \mathcal{S} to \mathcal{A} only depend on Alice's input parameters (and to some extent on the public elliptic curve parameters). As \mathcal{A} does not know sk_A or k_A , he has no way of determining for two public keys pk_A, pk_A^* which of the two is the correct one (other than guessing).
- **Phase 2:** If \mathcal{A} calls $f_{zk}^{R_1}$ and $f_{zk}^{R_2}$ with the parameters sent to him he will still receive 1 back, and 0 otherwise, which is again exactly the same as in the real execution. The hash function $H(\cdot)$ is expected to output a random value for the schnorr challenge as defined by the hiding property of the hash function. In the simulated case \mathcal{S} calculates the output value from the final signature and depends on the input values of Alice and Bob of which at least Alice input is chosen randomly by \mathcal{S} . As dependent on randomly chosen inputs the calculation output will as well be distributed uniformly across the possible values and is therefore indistinguishable from a real hash function output. The remaining messages sent by \mathcal{A} are identical to those of the real execution due to the deterministic nature of \mathcal{A} .
- **Phase 3:** The simulator will now verify the values sent to him by \mathcal{A} and will halt and output \perp in the case that he sends something invalid which is identical to the real execution. In this case \mathcal{A} must not receive (σ_{fin}, pk) in the ideal setting which is modelled by \mathcal{S} sending **abort** to the TTP. Otherwise \mathcal{S} will calculate his part of the partial signature as defined by the protocol. It will therefore found to be valid by \mathcal{A} and it will complete to σ_{fin} with **finSig** (which \mathcal{A} might call) because of the fixed, calculated schnorr challenger \mathcal{S} has returned \mathcal{A} in Phase 2. \mathcal{S} proceeds by outputting (σ_{fin}, pk) as he would in a real execution. σ_{fin} would match with the σ_{fin} output by \mathcal{A} if he or she were to compute it honestly, therefore the output values are again a perfect simulation of the real execution.

We have managed to show that in the case that Bob is corrupted the transcript is indistinguishable to a real transcript and even identical for the most part. We can therefore conclude that the transcript output will be indistinguishable from a real one in all cases and have thereby proven that the protocol **dSign** is secure. \square

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 $f_{zk}^{R_5}$, for the pair (x, X) .

$\text{dAptSign}\langle(m, sk_A, k_A), (m, sk_B, k_B, x)\rangle$	
1 : <i>Alice</i>	<i>Bob</i>
2 : ...	
3 : $f_{zk}^{R_1}((sk_A, pk_A))$	
4 : $f_{zk}^{R_2}((k_A, R_A))$	
5 :	$\xrightarrow{\Lambda, pk_A, R_A}$
6 :	...
7 :	if $f_{zk}^{R_1}(pk_A) = 0 \vee f_{zk}^{R_2}(\Lambda.R) = 0$
8 :	return \perp
9 :	$f_{zk}^{R_3}((sk_B, pk_B))$
10 :	$f_{zk}^{R_4}((k_B, R_B))$
11 :	$f_{zk}^{R_5}((x, X))$
12 :	$\xleftarrow{\tilde{\sigma}_B, \Lambda, pk_B, X}$
13 : ...	
14 : if $f_{zk}^{R_3}(pk_B) = 0 \vee$	
15 : $f_{zk}^{R_4}(\Lambda.R \cdot R_A^{-1}) = 0 \vee$	
16 : $f_{zk}^{R_5}(X) = 0 \vee$	
17 : return \perp	
18 : ...	
19 : return $(x, (\sigma_{fin}, pk))$	return (σ_{fin}, pk)

Theorem 2. Assume we have two key pairs (sk_A, pk_A) and (sk_B, pk_B) which were setup securely as for instance with the distributed keygen protocol dKeyGen . 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 $(f_{zk}^{R_1}, f_{zk}^{R_2}, f_{zk}^{R_3}, f_{zk}^{R_4}, f_{zk}^{R_5})$ -model.

Proof. We proof the security of dAptSign by constructing a simulator \mathcal{S} who is given the output (σ_{fin}, pk) (resp. $(x, (\sigma_{fin}, pk))$) from a TTP that securely computes the protocol in the ideal world, receiving the inputs from Alice and Bob. From this output the simulator \mathcal{S} will output a transcript that is indistinguishable from the protocol transcript in the real world. The simulator uses the calls to $(f_{zk}^{R_1}, f_{zk}^{R_2}, f_{zk}^{R_3}, f_{zk}^{R_4}, f_{zk}^{R_5})$ in order to do this. As in the proof before we assume the message m is known to both participants. We proof that the transcript is indistinguishable in case Alice is corrupted as well as in the case Bob is corrupted.

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

1. \mathcal{S} invokes \mathcal{A} . When \mathcal{A} internally calls $\mathbf{f}_{zk}^{R_1}$ and $\mathbf{f}_{zk}^{R_2}$ \mathcal{S} saves (sk_A, pk_A) and (k_A, R_B) to its memory.
2. \mathcal{S} receives Λ, 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 $R_A = g^{k_A}$. If any of those checks fail \mathcal{S} sends **abort** to the TTP and outputs \perp . Otherwise he sends (m, sk_A, k_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 layed out in the proof beforehand in step 3.
4. \mathcal{S} calculates $s_B^* := s_B + x$ (received from the TTP) 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 $\mathbf{f}_{zk}^{R_{3,4,5}}$ we compare the parameters send by \mathcal{A} to the real one, in case he sends an invalid value will return 0, otherwise 1.
7. \mathcal{S} receives $\tilde{\sigma}_A^*$ from \mathcal{A} and checks $\tilde{\sigma}_A = \tilde{\sigma}_A^*$. If the equality holds \mathcal{A} sends **continue** to the TTP and finally sends σ_{fin} to \mathcal{A} as if coming from Bob and outputs (σ_{fin}, pk) .

We reuse the phases defined in the previous proof with two adjustments:

- In *Phase 2* Bob additionally sends X to Alice
- We introduce *Phase 4* in which Bob sends σ_{fin} to Alice

We now again argue why each phase in the simulation is indistinguishable from a real execution

- *Phase 1:* This phase is identical to phase 1 the previous proof, thereby the argument is the same
- *Phase 2:* In this phase \mathcal{S} sends $X := g^x$ to \mathcal{A} for which x was received from the TTP, therefore it will be the same x sent in the real execution by the honest party which makes the simulation perfect in this phase.
- *Phase 3:* Again the messages send in this phase are produced by the deterministic \mathcal{A} which will be identical to the real execution. In contrast to the **dSign** protocol now the adversary does not yet finish the protocol.

- *Phase 4*: Now the \mathcal{A} expects to receive σ_{fin} , from which he is able to extract the witness x . Indeed he will receive a σ_{fin} which is identical to the one sent in a real execution by honest Bob, furthermore he will be able to extract x such that $X = g^x$, which again makes this phase identical to the real execution.

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

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

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

$$\begin{aligned} \Lambda &:= \{pk := 1, R := 1\} \\ \Lambda &\leftarrow \text{setupCtx}(\Lambda, g^{sk_A}, g^{k_A}) \end{aligned}$$

2. \mathcal{S} now invokes \mathcal{A} and sends (Λ, pk_A, R_A) as if coming from Alice.
3. When \mathcal{A} calls $f_{zk}^{R_{1,2}}$ (as the verifier) \mathcal{S} checks for equality with the values sent by him and returns either 0 or 1. Once he sends $(sk_B, pk_B), (k_B, R_B), (x, X)$ internally to $f_{zk}^{R_{1,2,3}}$ \mathcal{S} saves them to his memory.
4. \mathcal{S} sends (m, sk_A, k_A, x) to the TTP and receives (σ_{fin}, pk) .
5. When \mathcal{A} queries $H(\cdot)$ the simulator again sets the output to e^* calculated with the same steps as layed out in the previous proof in step 5.
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 **continue** to the TTP, otherwise sends **abort** and outputs \perp .
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} (if not he outputs \perp) and will verify that $\sigma_{fin}^* = \sigma_{fin}$. If the equality holds he will output $(x, (\sigma_{fin}, pk))$, if not \perp .

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

- *Phase 1*: This phase is identical to phase 1 in the previous proof, thereby the same argumentation holds.

- *Phase 2:* Again this phase is similar to phase 2 in the **dSign** proof with the only difference that \mathcal{A} will make the additional call to $f_{zk}^{R_5}((x, X))$ and send the value X to \mathcal{S} . Both these changes do not require any further interaction from \mathcal{S} thereby the arguments from the previous proof in phase 2 still hold.
- *Phase 3:* In this section \mathcal{S} will verify equality of the values sent by \mathcal{A} with the variables saved prior to its memory and halts (plus output \perp) if any of the values are unequal. In this case \mathcal{A} should not receive the final outputs (σ_{fin}, pk) which is modelled by sending **abort** to the TTP. The same behaviour is expected in a real execution when Alice calls $f_{zk}^{R_{3,4,5}}$ and receives a 0 bit. We have already argued in the prior proof why $\tilde{\sigma}_A$ is indistinguishable from the one calculated by Alice in a real execution and only refer to the argumentation here.
- *Phase 4:* In phase 4 \mathcal{S} is expected to receive σ_{fin}^* from \mathcal{A} which needs to be equal to σ_{fin} received earlier by the TTP. \mathcal{S} will do this simple equality check and if successful output $(x, (\sigma_{fin}, pk))$ which is identical to the output in a real execution. In the other case we would simply output \perp which is identical to the case in which a Bob sends a σ_{fin} that does not verify.

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. \square

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

In this section, we will 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 [fuchsbauer2019aggregate], in particular the final transactions constructed by our protocols should be valid transactions as by the definitions by Fuchsbauer et al. As we will only focus on the transaction protocol (transferring value from one or many parties to one or many parties), the notions of transaction aggregation, coin minting and adding transaction to the main ledger discussed in [fuchsbauer2019aggregate] will not be topic of this formalization, as they are not relevant building blocks for the proposed Atomic Swap protocol.

As an extension to the regular transaction protocol transferring value from one sender to a receiver we will define two further protocols. The first of them titled *Extended Mimblewimble Transaction Scheme* will provide additional functions to create and spend coins owned by two parties instead of just one, thereby enabling coins owned by multiple parties at once, which is similar to a multisig address in Bitcoin [antonopoulos2014mastering]. The second extended definition is called *Script Mimblewimble Transaction Scheme* and will allow the receiver of a coin to hide a secret witness value x in his part if the transaction, in a way that the sender (or the senders) can redeem this secret after the protocol has completed and the final transaction is available.

We will proceed by providing an instantiation of the three transactions schemes in section 5.2 which can be implemented and deployed on a Mimblewimble based Cryp-

tocurrency such as Beam or Grin. Furthermore we provide proofs that the schemes are correct and secure in the malicious setting as defined in 3.8. Finally, we define a Atomic Swap protocol from these building blocks, allowing two parties to securely and trustlessly swap funds from a Mimblewimble blockchain with those on another blockchain, such as Bitcoin in section 5.4.

5.1 Definitions

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

Fuchsbauer et al. have defined three procedures **Send**, **Rcv** and **Ldgr** with regards to the creation of a transaction. **Send** called by the sender will create a pre-transaction, **Rcv** takes the pre-transaction and adds the receivers output and **Ldgr** (again called by the sender) verifies and publishes the final transaction to the blockchain ledger. As we have already pointed out in this thesis we won't discuss the transaction publishing phase therefore we will not cover the publishing functionality of the **Ldgr** procedure, however we will use the verification capabilities of the algorithm. That means the transactions created by our protocol must be compatible with the **ledgr** functionality formalized by Fuchsbauer et al. We redefine the **Send** and **Rcv** functionality in our paper, making small adjustments to the original definitions to fit with our requirements.

To improve the readability of our formalizations we introduce a wrapper spC which represents a spendable coin and contains a reference to the coin commitment and rangeproof as defined in 3.14 as well as its (secret) spending information which consist of the coins value and blinding factor.

$$spC := \{C, v, r, \pi\}$$

If we want to indicate that a spendable coin is a output coin of a transaction we write spC^* .

Definition 5.1 (Mimblewimble Transaction Scheme). A Mimblewimble Transaction Scheme $MW[COM, \Phi_{MP}, \Pi]$ with commitment scheme COM , two-party signature scheme Φ_{MP} , and rangeproof system Π consists of the following tuple of procedures:

$$MW[COM, \Phi_{MP}, \Pi] := (\text{spendCoins}, \text{recvCoins}, \text{finTx}, \text{ledgr})$$

- $(ptx, spC_A^*, (sk_A, k_A)) \leftarrow \text{spendCoins}([spC], p, t)$: 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 send to a receiver, Alice spendable change output coin $sp\mathcal{C}_A^*$ as well as the senders signing key and secret nonce (sk_A, k_A) later used in the transaction signing process.

- $(ptx^*, sp\mathcal{C}_B^*) \leftarrow \text{recvCoins}(ptx, p)$: The receiveCoins algorithm is a DPT routine called by the receiver and takes as input a pre-transaction ptx and a fund value p . It will output a modified pre-transaction ptx^* together with Bobs new spendable output coin $sp\mathcal{C}_B^*$ which has been added to the transaction. At this stage the transaction already has to be partially signed. (by the receiver)
- $tx \leftarrow \text{finTx}(ptx, sk_A, k_A)$: The finalize algorithm is a DPT routine again called by the transaction sender that takes as input a pre-transaction ptx and the senders signing key sk_A and nonce k_A . The function will output a finalized signed transaction tx .
- $\{\chi, \perp\} \leftarrow \text{ledgr}(\chi, tx)$: The ledger algorithm is taken from the Fuchsbauer et al. paper and can be used to verify and publish a valid transaction. If an invalid transaction is passed it will output \perp , otherwise it will output the updated ledger. As input it takes the current state of the ledger χ and the transaction tx . We assume that every protocol participant can access the current ledger χ , as it is publicly available by running a node.

We say a Mimblewimble Transaction Scheme is correct if the ledger algorithm ledgr returns and updated χ if and only if the added transaction is well balanced and its signature is valid. More formally:

Definition 5.2 (Transaction Scheme Correctness). For any transaction fund value p and list of spendable input coins $[sp\mathcal{C}]$ transaction with combined value $v \geq p$ the following must hold:

$$\Pr \left[\text{ledgr}(\chi, tx) \neq \perp \mid \begin{array}{l} p \leq \sum_{i=0}^n (sp\mathcal{C}_i.v) \\ (ptx, \cdot, (sk_A, k_A)) \leftarrow \text{spendCoins}([sp\mathcal{C}], v, \perp) \\ (ptx^*, \cdot) \leftarrow \text{recvCoins}(ptx, p) \\ tx \leftarrow \text{finTx}(ptx^*, sk_A, k_A) \end{array} \right] = 1.$$

Finally we define a two-party protocol dBuildMWTx which takes as input a list of spendable coins $[sp\mathcal{C}]$, a transaction value p and a optional timelock t and uses the functions defined earlier to output a valid transaction tx as well as the newly spendable coins to both parties.

$$\langle (tx, sp\mathcal{C}_A^*), (txsp\mathcal{C}_B^*) \rangle \leftarrow \text{dBuildMWTx}(\langle sp\mathcal{C}^*, p, t \rangle, (p))$$

Definition 5.3 (Extended Mimblewimble Transaction Scheme). An extended Mimblewimble transaction scheme $MW_{ext}[COM, \Phi_{MP}, \Pi]$ is an extension to MW with the following two procedures:

$$MW_{ext}[COM, \Phi_{MP}, \Pi] := MW[COM, \Phi_{MP}, \Pi] \parallel (\text{dSpendCoins}, \text{dRecvCoins}, \text{dFinTx})$$

- $\langle (ptx, spC_A^*, (sk_A, k_A)), (ptx, spC_C^*, (sk_C, k_C)) \rangle \leftarrow \text{dSpendCoins}(\langle [spC_A], p, t \rangle, \langle [spC_C], p, t \rangle)$: The distributed coin spending algorithm takes as input a list of spendable input coins which ownership is shared between Alice and Carol. Note that for each provided input coin Alice and Carol have only a share to the blinding factor. A coins full blinding factor can then be calculated as: $r := r_A + r_C$. 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 pretransaction ptx and change coin for each party, as well as their signing information.
- $\langle (ptx^*, spC_B^*), (ptx^*, spC_C^*) \rangle \leftarrow \text{dRecvCoins}(\langle ptx, p \rangle, \langle ptx, p \rangle)$: The distributed coin receive procedure takes as input a pre-transaction ptx and a value p which should be transferred with the transaction. The distributed algorithm will generate a output coin owned by both Alice and Carol. (each owning a share of the key). The output will be an updated pre-transaction ptx^* , the shared output coin C_{out}^{sh} with the respective shares of the blinding factor added as spC_B^*, spC_C^* . Note that C_{out}^{sh} will only be spendable if both owners cooperate running the dSpendCoins protocol.
- $\langle tx, tx \rangle \leftarrow \text{dFinTx}(\langle ptx, sk_A, k_A \rangle, \langle ptx, sk_C, k_C \rangle)$: The distributed finalized transaction protocol has to be used if we are creating a transaction spending a shared coin (i.e. the transaction was created with the dSpendCoins algorithm). In this case we require signing information from both Alice and Carol.

Correctness is given very similar to the standard scheme:

Definition 5.4 (Extended Transaction Scheme Correctness). For any list of spendable coins $[spC]$ with total value v greater then the transaction fund value p and split blinding factors $([r_A], [r_C])$ the following must hold:

$$\Pr \left[\text{ledgr}(\chi, tx) \neq \perp \mid \begin{array}{l} p \leq \sum_{i=0}^n (spC_i \cdot v) \\ \langle (ptx, \cdot, (sk_A, k_A)), (ptx, (sk_C, k_C)) \rangle \leftarrow \\ \text{dSpendCoins}(\langle [spC_A], p, \perp \rangle, \langle [spC_C], p, \perp \rangle) \\ \langle (ptx^*, \cdot)(ptx^*, \cdot) \rangle \leftarrow \text{dRecvCoins}(\langle ptx, p \rangle, \langle ptx, p \rangle) \\ tx \leftarrow \text{dFinTx}(\langle ptx^*, sk_A, k_A \rangle, \langle ptx^*, sk_C, k_C \rangle) \end{array} \right] = 1.$$

Definition 5.5 (Script Mimblewimble Transaction Scheme). The script version of the extended Mimblewimble Transaction Scheme updates the Extended Mimblewimble Transaction Schme by providing a modified version of the single party receive routine and the distributed finalize transaction protocol.

$$MW_{apt}[COM, \Phi_{MP}, \Pi] := MW_{ext}[COM, \Phi_{MP}, \Pi] \parallel \text{aptRecvCoins}, \text{dAptFinTx}$$

- $(ptx^*, spC_B^*, \tilde{\sigma}_B) \leftarrow \text{aptRecvCoins}(ptx, p, x)$: The script variant of the receive function takes an additional input a secret witness value x which will be hidden in the transactions signature and extractable by the other party after the protocols' completion. Note that the routine also returns Bob's unadapted partial signature. The reason for this is that we later still need the unadapted version to complete the signature und thereby finalize the transaction. By not sharing this unadapted signature with Alice, Bob is the one who gets to finalize the transaction which is different from the simpler protocol and is an important feature for our atomic swap protocol.
- $\langle \sigma_{AB}, tx \rangle \leftarrow \text{dAptFinTx}(\langle ptx^*, sk_A, k_A, X \rangle, \langle ptx^*, sk_B, k_B, \tilde{\sigma}_B \rangle)$: The script variant of the finalize transaction algorithm is a distributed protocol between the sender(s) and receiver. Additionally to the pre-transaction ptx^* the senders need to input their signing information, Bob needs to input the unadapted version of his partial signature as it is needed for transaction completion. This protocol could also be implemented as a three party protocol, two senders controlling a shared coin and a third receiver. However, as in the case we will describe later in 5.4 one of the two senders is also the receiver, we allowed ourselves to model this protocol as being between only two parties to simplify the formalization. In this version of the protocol only Bob will be able to finalize the transaction, which is different to finTx and dFinTx . This has the practical reason that for the atomic swap execution Bob needs to be the one in control of building the final transaction. If Alice were to build the final transaction before Bob, she might be able to extract the Witness value before the transaction has been published, which in the atomic swap scenario would mean she could steal the funds stored on the other chain. This is why the protocol does not return the final transaction tx to Alice, instead the protocol will output the senders partial signature, which Alice can later use to extract a witness value.

Similar as before we define correctness for the adapted scheme:

Definition 5.6 (Adapted Transaction Scheme Correctness). For any transaction fund value p and list of input coins $[spC]$ with combined value $v \geq p$ and any witness value $x \in \mathbb{Z}_*^*$ the following must hold:

$$\Pr \left[\text{ledgr}(\chi, tx) \neq \perp \mid \begin{array}{l} p \leq \sum_{i=1}^n (spC_i.v) \\ (ptx, spC_A^*, (sk_A, k_A)) \leftarrow \text{spendCoins}([spC], p, \perp) \\ (ptx^*, spC_B^*, \tilde{\sigma}_B) \leftarrow \text{aptRecvCoins}(ptx, p, x) \\ \langle \sigma_{AC}, tx \rangle \leftarrow \text{dAptFinTx}(\langle ptx, sk_A, k_A, X \rangle, \langle ptx, sk_C, k_C, \tilde{\sigma}_B \rangle) \end{array} \right] = 1.$$

5.2 Instantiation

In this section we will provide an instantiation of the transaction scheme definitions found in 5.1, 5.3 and 5.5. The instantiations can be implemented in a Cryptocurrency based on the Mumblewimble protocol such as Beam and Grin.

5.2.1 Mimblewimble Transaction Scheme

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

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

<code>createCoin(v, r)</code>	<code>createTx($m, [C_{inp}], [C_{out}], [\pi], \Lambda, [C], \sigma, t$)</code>
1: $C \leftarrow \text{commit}(v, r)$	1: return (
2: $\pi \leftarrow \text{ranPrf}(C, v, r)$	2: $m := m,$
3: return (C, r, v, π)	3: $inp := [C_{inp}],$
	4: $out := [C_{out}],$
	5: $\Pi := [\pi],$
	6: $\Lambda := \Lambda,$
	7: $com := [C],$
	8: $\sigma := \sigma,$
	9: $t := t$)

In figure 5.1 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 which should be transferred to a receiver. He sets up the signature context with his parameters and gets a pre-transaction ptx , newly created spendable output coin spC_A , as well as a signing key sk_A and secret nonce k_A as output. The pre-transaction can then be sent to a receiver. Note that this instantiation differs from the one described by Fuchsbauer et al. [fuchsbauer2019aggregate] in that the sender does not yet sign the transaction during `spendCoins`. This has the reason that in our definition of the Two-Party Signature Scheme 4.1 the signature context Λ requires to be fully setup before a partial signature can be created, therefore signing can only start at the receivers turn, after the signature context has been completed. In the referenced paper it is possible to start the signing

earlier, because instead of using the notion of a two-party signing protocol, they instead rely on an aggregateable signature scheme. The sender and receiver both will create their signatures which will then be aggregated into the final one. However, we find that by using a two-party signature scheme for our formalization we are closer to what is implemented in practice¹. Furthermore by starting the signing process at the receivers turn we avoid a potential problem: If an adversary learns the already signed pre-transaction and transaction value p before the intended receiver, the adversary would be able to steal the coins by creating his malicious output coin together with his signature, which he could then aggregate to the senders pre-transaction.

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

Figure 5.2 show the implementation of a two party protocol using the routines instantiated in 5.1 to build a Mimblewimble transaction transferring p from a sender owning the coins spC^* to a receiver.

5.2.2 Extended Mimblewimble Transaction Scheme

Figure 5.3 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 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, k_A) , (sk_C, k_C) .

¹<https://medium.com/@brandonarvanaghi/grin-transactions-explained-step-by-step-fdceb905a853>.

5. ADAPTOR SIGNATURE BASED ATOMIC SWAPS BETWEEN BITCOIN AND A MIMBLEWIMBLE BASED CRYPTOCURRENCY

spendCoins($[spC], p, t$)

```

1:  $v \leftarrow \sum_{i:=0}^{i < n} (spC_i.v)$ 
2: if  $p > v$ 
3:   return  $\perp$ 
4:  $m := \{0, 1\}^*$ 
5:  $(r_A^*, k_A) \leftarrow \$\mathbb{Z}_p$ 
6:  $spC_A^* \leftarrow \text{createCoin}(v - p, r_A^*)$ 
7:  $\{C_{out}^A, r_A^*, v_A, \pi_A\} \leftarrow spC_A^*$ 
8:  $sk_A := r_A^* - \sum_{i:=0}^{i < n} (spC_i.r)$ 
9:  $\Lambda := \{pk := 1_p, R := 1_p\}$ 
10:  $\Lambda \leftarrow \text{setupCtx}(\Lambda, g^{sk_A}, g^{k_A})$ 
11:  $ptx \leftarrow \text{createTx}(m, spC.C, [C_{out}^A], [\pi_A], \Lambda, [g^{sk_A}], \emptyset, t)$ 
12: return  $(ptx, spC_A^*, (sk_A, k_A))$ 
recvCoins( $ptx, p$ )

```

```

1:  $(m, inp, out, \Pi, \Lambda, com, \emptyset, t) \leftarrow ptx$ 
2: if  $\text{vrfRanPrf}(\Pi[0], out[0]) = 0$ 
3:   return  $\perp$ 
4:  $(r_B^*, k_B) \leftarrow \$\mathbb{Z}_p$ 
5:  $spC_B^* \leftarrow \text{createCoin}(p, r_B^*)$ 
6:  $\{C_{out}^B, r_B^*, v_B, \pi_B\} \leftarrow spC_B^*$ 
7:  $sk_B := r_B^*$ 
8:  $\Lambda \leftarrow \text{setupCtx}(\Lambda, g^{sk_B}, g^{k_B})$ 
9:  $\tilde{\sigma}_B \leftarrow \text{signPt}(m, sk_B, k_B, \Lambda)$ 
10:  $ptx \leftarrow \text{createTx}(m, inp, out \parallel C_{out}^B, \Pi \parallel \pi_B, \Lambda, com \parallel g^{sk_B}, \tilde{\sigma}_B, t)$ 
11: return  $(ptx, spC_B^*)$ 
finTx( $ptx, sk_A, k_A$ )

```

```

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

```

Figure 5.1: Instantiation of Mimblewimble Transaction Scheme.

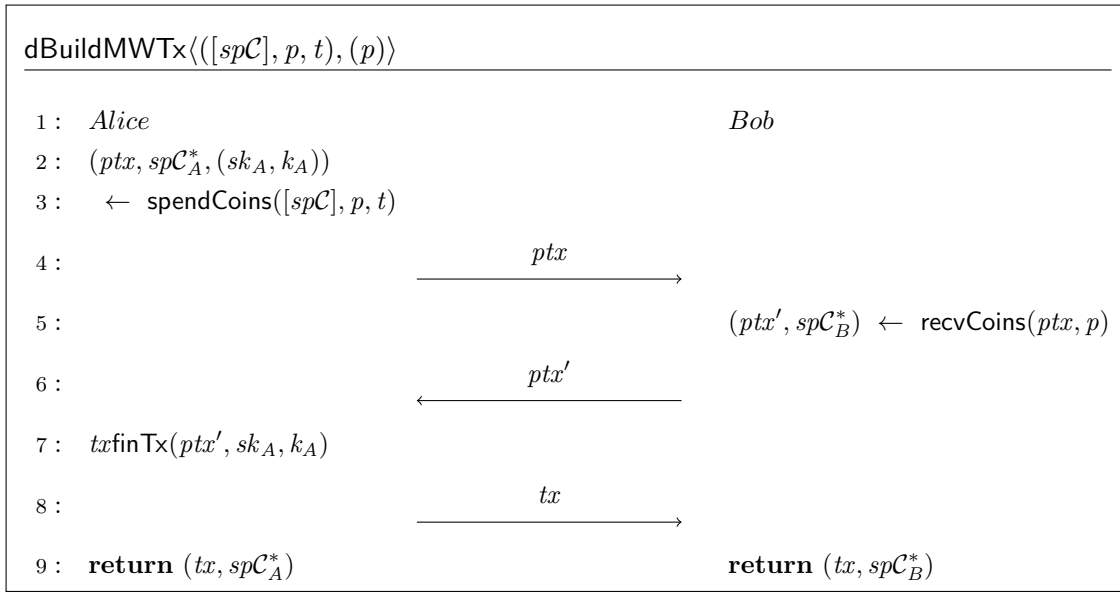


Figure 5.2: dBuildMWTx two-party protocol to build a new transaction

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

1 : *Alice*

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

3 : **if** $p > v$

4 : **return** \perp

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

6 : $(r_A^*, k_A) \leftarrow \$_{\mathbb{Z}_p}$

7 : $sp\mathcal{C}_A \leftarrow \text{createCoin}(v_A, r_A^*)$

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

9 : $sk_A := r_A^* - \sum [r_A]$

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

11 : $\Lambda \leftarrow \text{setupCtx}(\Lambda, g^{sk_A}, g^{k_A})$

12 : $ptx \leftarrow$

13 : $\text{createTx}(m, [\mathcal{C}_{inp}], [\mathcal{C}_{out}^A], [\pi_A], \Lambda, [g^{k_A}], \emptyset, t)$

14 :

\xrightarrow{ptx}

15 :

16 :

17 :

18 :

19 :

20 :

\xleftarrow{ptx}

21 : **if** $\text{vrfRanPrf}(ptx.\Pi[1], ptx.out[1]) = 0$

22 : **return** \perp

23 : **return** $(ptx, sp\mathcal{C}_A, (sk_A, k_A))$

24 :

Carol

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

if $p > v$

return \perp

$(r_C^*, k_C) \leftarrow \$_{\mathbb{Z}_p}$

$sp\mathcal{C}_C \leftarrow \text{createCoin}(v_C, r_C^*)$

$\{\mathcal{C}_{out}^C, r_C^*, v_C, \pi_C\} \leftarrow sp\mathcal{C}_C^*$

$sk_C := r_C^* - \sum [r_C]$

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

if $\text{vrfRanPrf}(\Pi[0], out[0]) = 0 \vee t \neq t^*$

return \perp

$\Lambda \leftarrow \text{setupCtx}(\Lambda, g^{sk_C}, g^{k_C})$

$ptx \leftarrow \text{createTx}(m, inp, out \parallel \mathcal{C}_{out}^C, \pi \parallel \pi_C, \Lambda, com \parallel g^{k_C}, \emptyset, t)$

return $(ptx, sp\mathcal{C}_C, (sk_C, k_C))$

Figure 5.3: Extended Mimblewimble Transaction Scheme - dSpendCoins

Figure 5.4 shows an instantiation of the `recvCoins` 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 senders output(s). Bob starts by creating a coin with fund value p and his share of the newly create blinding factor and sends it over to Carol. Carol finalizes the shared coin by adding a commitment to her blinding factor to the coin and sends it back, together with the commitment. Bob verifies validity of the updated shared coin after which the two parties engage in two two-party protocols to create their partial signature and coin rangeproof. Finally they create the updated pre-transaction ptx which can be sent back to the sender.

$\text{dRecvCoins}\langle (ptx, p), (ptx, p) \rangle$

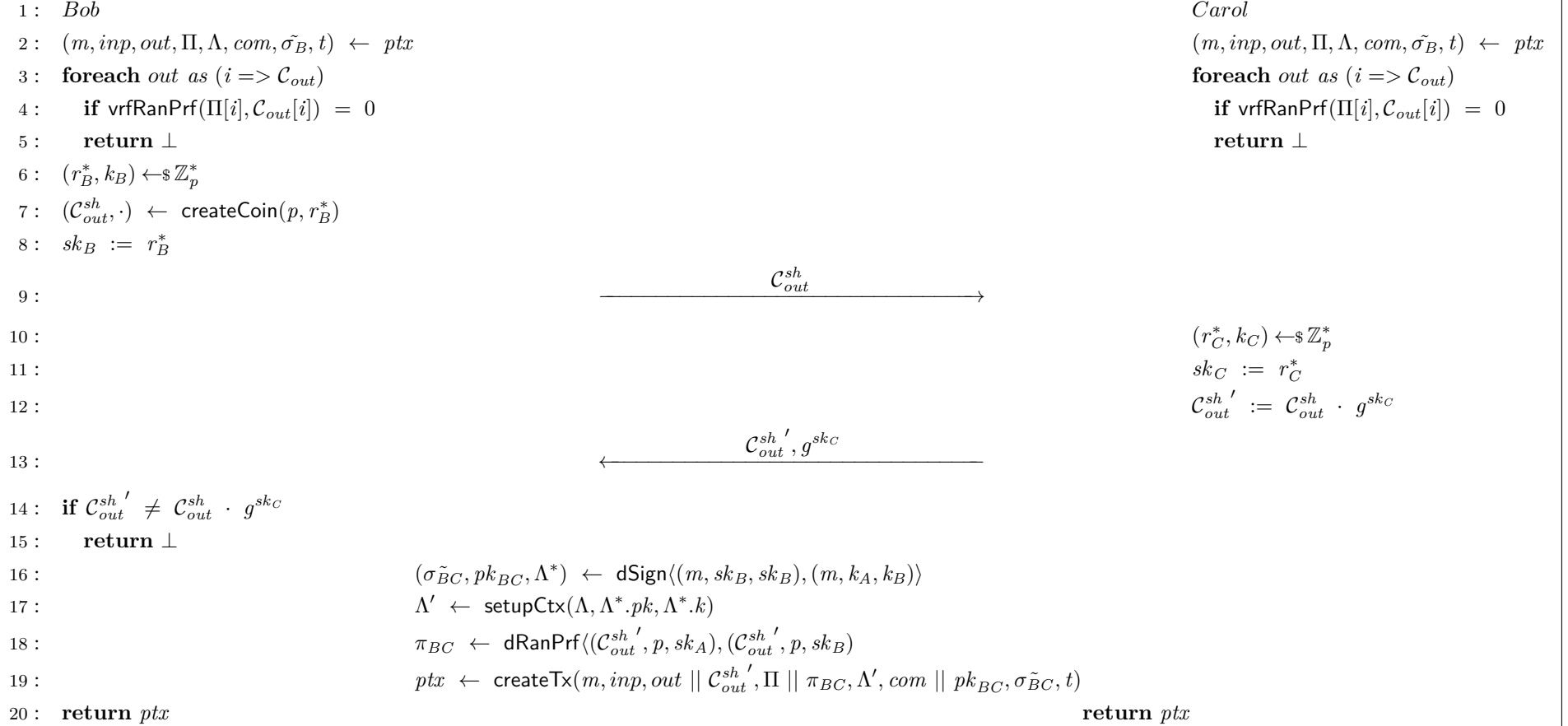


Figure 5.4: Extended Mimblewimble Transaction Scheme - dRecvCoins

$\text{dFinTx}(\langle ptx, sk_A, k_A \rangle, \langle ptx, sk_C, k_C \rangle)$

<pre> 1: Alice 2: (m, inp, out, Π, Λ, com, σ̃_B, t) ← ptx 3: if vrfRanPrf(Π[1], out[1]) = 0 4: return ⊥ 5: if vrfPt(σ̃_B, m, com[1]) = 0 6: return ⊥ 7: 8: σ̃_AC ← dSign(⟨m, sk_A, k_A⟩, ⟨m, sk_C, k_C⟩) 9: σ_fin ← finSig(σ̃_B, σ̃_AC) 10: tx ← createTx(m, inp, out, Π, Λ, com, σ_fin, t) 11: return tx </pre>	<pre> Carol (m, inp, out, Π, Λ, com, σ̃_B, t) ← ptx if vrfRanPrf(Π[1], out[1]) = 0 return ⊥ if vrfPt(σ̃_B, m, com[1]) = 0 return ⊥ σ̃_AC ← dSign(⟨m, sk_A, k_A⟩, ⟨m, sk_C, k_C⟩) σ_fin ← finSig(σ̃_B, σ̃_AC) tx ← createTx(m, inp, out, Π, Λ, com, σ_fin, t) return tx </pre>
--	---

Figure 5.5: Extended Mimblewimble Transaction Scheme - dFinTx

```

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

```

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

5.2.3 Adapted Extended Mimblewimble Transaction Scheme

Figure 5.6 shows an instantiation of the **aptRecvCoins** algorithm. Before updating the pre-transaction ptx Bob adapts his partial signature with the witness value x . The procedure then returns the pre-transaction ptx containing Bobs adapted partial signature, and the statement X which is a commitment to the witness value x .

In figure 5.7 we show the updated distributed version of the transaction finalization protocol. Again Alice verifies the pre-transaction ptx received by Bob and then proceeds by building her own partial signature. Note that at this point Alice is not able to finalize the signature (and consequently the transaction) as she only knows Bobs adapted partial signature, but not the original one, which is needed for the **finSig** function. Therefore in another round of interaction Alice sends her partial signature to Bob, who will verify Alice partial signature and finally calculate the final signature, needed for the transaction. He will send over σ_{fin} which lets both parties construct the valid transaction as well as Alice call **extWit** to extract the secret witness x .

$\text{dAptFinTx}(\langle ptx, sk_A, k_A, X \rangle, \langle ptx, sk_C, k_C, \tilde{\sigma}_B \rangle)$

1: *Alice*

2: $(m, inp, out, \Pi, \Lambda, com, \hat{\sigma}_B, t) \leftarrow ptx$

3: **if** $\text{vrfRanPrf}(\Pi[1], out[1]) = 0$

4: **return** \perp

5: **if** $\text{vrfAptSig}(\tilde{\sigma}_B, m, com[1], X) = 0$

6: **return** \perp

7: $\sigma_{\tilde{A}C} \leftarrow \text{dSign}(\langle m, sk_A, k_A \rangle, \langle m, sk_C, k_C \rangle)$

8:

$\xrightarrow{\sigma_{\tilde{A}C}}$

9:

10:

11:

12:

Bob

$(m, inp, out, \Pi, \Lambda, com, \hat{\sigma}_B, t) \leftarrow ptx$

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

return \perp

if $\text{vrfAptSig}(\tilde{\sigma}_B, m, com[1], X) = 0$

return \perp

$\text{vrfPt}(\sigma_{\tilde{A}C}, m, com[0])$

$\sigma_{fin} \leftarrow \text{finSig}(\sigma_{\tilde{A}C}, \tilde{\sigma}_B)$

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

return tx

Figure 5.7: Adapted Extended Mimblewimble Transaction Scheme - dAptFinTx .

5.3 Security & Correctness

In this section we will prove the correctness and security of the instantiation described in ???. We start by proving *Transaction Scheme Correctness*, *Extended Transaction Scheme Correctness* and *Adapted Transaction Scheme Correctness* for the three outlined transaction schemes MW , MW_{ext} and MW_{apt} . We then continue by showing that all three security definitions (*Inflation-Resistance*, *Theft-Resistance*, *Transaction indistinguishability*) hold again for all three transaction types. The general process will be to find a proof for the basic transaction protocol defined in 5.1 and then reduce the extended and adapted case to the basic one.

5.3.1 Correctness

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

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

Proof. We assume there are two honest participants Alice and Bob, there exists a list of input coins $[C_{inp}]$ with blinding factors $[r_A]$ and total value v known to Alice, and some amount p which Alice wants to transfer to Bob. For *Transaction Scheme Correctness* to hold `verfTx(tx)` must return 1 with overwhelming probability for the two parties creating a transaction tx in the following three steps:

1. $(ptx, (sk_A, k_A)) \leftarrow \text{spendCoins}([spC], p, \perp)$
2. $ptx^* \leftarrow \text{recvCoins}(ptx, p)$
3. $tx \leftarrow \text{finTx}(ptx^*, sk_A, k_A)$

Here are the following three requirements for `verfTx(tx)` to return 1:

1. The list of input coins inp , as well as output coins out must each include at least one element.

2. For each output coin \mathcal{C}_{out} the rangeproof π must verify.
3. The transaction signature under the public key $pk := \sum_{i=0}^n out - \sum_{i=0}^n inp$ must verify.

Condition 1 must hold trivially, as already in `spendCoins` Alice will create the output \mathcal{C}_{out}^A and initialize the pre-transaction with $[\mathcal{C}_{inp}]$ and $[\mathcal{C}_{out}^A]$, thereby already after `spendCoins` both input and output lists are non empty and this condition is fulfilled.

Condition 2 follows from the implementation of the `createCoin` function called in `spendCoins` as well as `recvCoins`. In the function a rangeproof is computed for the new coin \mathcal{C} with value v and blinding factor r as $\pi \leftarrow \text{ranPrf}(\mathcal{C}, v, r)$. Given that our Rangeproof system π system has to be correct $\text{vrfRanPrf}(\pi, \mathcal{C}) = 1$ must hold for all coins created with the `createCoin` routine. Therefore Condition 2 must hold.

For condition 3 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. Bobs secret key is constructed like $sk_B := r_B^*$, so it corresponds to the blinding factor of his output. From the construction of the two-party signature scheme in 4.1 we know that therefore the final signature will be valid under the following public key:

$$pk^* := g^{sk_A} \cdot g^{sk_B}$$

Given how the secret keys are constructed we arrive at:

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

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

$$pk = pk^* \tag{5.1}$$

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

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

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

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

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

5. ADAPTOR SIGNATURE BASED ATOMIC SWAPS BETWEEN BITCOIN AND A MIMBLEWIMBLE BASED CRYPTOCURRENCY

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

From step 5.4 to 5.4 we rely on the fact that if Alice is honest $v = \sum_{i=0}^n v_i$, therefore also $(v - p) + p = \sum_{i=0}^n v_i$ must hold. From that we can infer that $h^{v-p} \cdot h^p \cdot \sum_{i=0}^n h^{-v_i}$ must cancel out. We have managed to show that condition 3 must hold and can conclude that *Transaction Scheme Correctness* holds.

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

Condition 1 again follows trivially from the construction of **dSpendCoins** for the same reasons we have already layed out.

dSpendCoins, **dRecvCoins**, **aptRecvCoins** all rely on the same **createCoin** routine to create output coins, thereby condition 2 will hold for the same reasons as layed out before.

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

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

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

sk_A and sk_C are then inputs to **dFinTx** in which a partial signature σ_{AC} is calculated, by both Alice and Carol signing with their secret key. Assume the general key from before in which we have a single secret key sk_A . We can split sk_A into arbitrarily chosen shares $(sk_A)_1, (sk_A)_2$ with $sk_A = (sk_A)_1 + (sk_A)_2$. By the definition of Two-Party Signatures 4.1 the combined signature from $(sk_A)_1, (sk_A)_2$ will be valid under g^{sk_A} . Thereby we can treat sk_A and sk_C from **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 3 must also hold for the *Extended Transaction Scheme*.

Now for the *Adapted Extended Transaction Scheme* the same argument holds. The only difference in this scheme is that in **dAptFinTx** Bob (instead of Alice) will call **finSig**, as only he knows his unadapted partial signature $\tilde{\sigma}_B$. However, the construction of the signature remains unchanged, therefore the reduction we provided before must hold for the same reasons.

As we have managed to show that all **verfTx(tx) = 1** must hold for all three transaction schemes, assuming the participants are honest and the underlying commitment scheme *COM*, two signature scheme Φ_{MP} and rangeproof system Π are correct theorem 3 must also hold. \square

It remains to show that the commitment scheme, signature scheme and rangeproof system used in the given instantiations of the three transaction schemes are correct.

All three transaction schemes use Pedersen Commitments as the commitment system COM , for which a security and correctness evaluation can be found in [pedersen1991non]. For the correctness of the rangeproof system Π we refer the reader again to the bulletproof paper [bunz2018bulletproofs]. The instantiation of the Mimblewimble Transaction Scheme MW (defined in 5.1 instantiated in figure 5.1) uses a standard Two-Party Signature Scheme Φ_{MP} defined in 4.1 for which a correctness proof is available by Maxwell et al. in [maxwell2019simple].

For the Extended Mimblewimble Transaction Scheme MW_{ext} defined in 5.3 and instantiated in figure 5.3, 5.4 and 5.5 the same signature scheme Φ_{MP} and rangeproof system Π is used, therefore we can make the same argument as for the previous case to show *Extended Transaction Scheme Correctness* holds.

For the Adapted Extended Mimblewimble Transaction Scheme MW_{apt} , which we have defined in 5.5, and have shown an instantiation in figure 5.6, and 5.7, the Two Party Fixed Witness Adaptor Schnorr Signature Scheme Φ_{Apt} , which we have defined in 4.2, is used. We have already shown correctness of this signature scheme in section 4.3 by proofing that *Adaptor Signature Correctness* holds. Furthermore, the scheme used a multiparty version of the bulletproof system Π_{mp} for which we refer the reader to [klinec2020privacy] for a proof of its correctness.

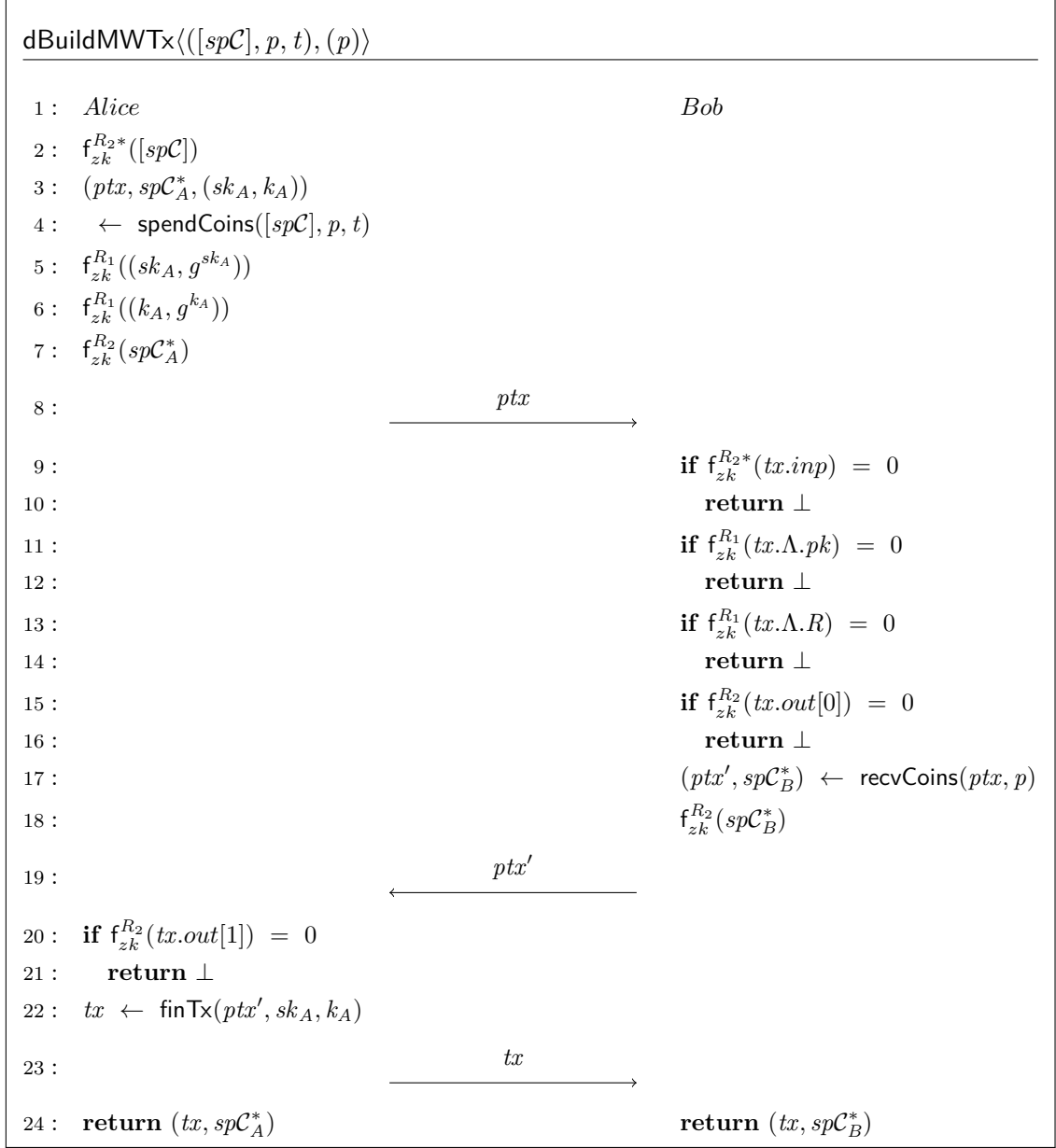
5.3.2 Security

We now prove the security of our transaction schemes in the malicious setting as defined in 3.8. In particular we show that the distributed protocols for constructing transactions are secure in the hybrid \mathbf{f}_{zk}^R -model as already explained in 4.3.2.

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

Proof Idea: We extend the protocol $dBuildMWTx$ defined in section 5.2 with the following calls to the zero-knowledge proof of knowledge functionalities:

5. ADAPTOR SIGNATURE BASED ATOMIC SWAPS BETWEEN BITCOIN AND A MIMBLEWIMBLE BASED CRYPTOCURRENCY



Theorem 4. Let COM be a correct and secure pedersen commitment scheme, Π be a correct and secure rangeproof system and Φ_{MP} be a secure and correct two-party signature scheme, then dBuildMWTx securely computes a Mimbewimble transaction transferring the value p from a sender (denoted as Alice) to a receiver (denoted as Bob) in the hybrid $f_{zk}^{R_1}, f_{zk}^{R_2}$ -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 inputs for the participants. The TTP returns the outputs (tx, spC_A^*) (resp. (tx, spC_B^*)) from which he

has to construct a transcript that is indistinguishable from the protocol transcript in the real world. The simulator uses the calls to $f_{zk}^{R_1}, f_{zk}^{R_2}, f_{zk}^{R_2*}$ to achieve this. We proof that the transcript is indistinguishable in the cases that either Alice or Bob is corrupt and controlled by a deterministic polynomial adversary \mathcal{A} .

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

1. \mathcal{S} invokes \mathcal{A} and once it calls $f_{zk}^{R_2*}, f_{zk}^{R_1}, f_{zk}^{R_2}$ saves the values $[sp\mathcal{C}], sk_A, k_A, sp\mathcal{C}_A^*$.
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 $tx.\Lambda.pk = g^{sk_A}$, $tx.\Lambda.R = g^{k_A}$, $tx.\pi[0] = sp\mathcal{C}_A^*.\pi$ and $tx.com[0] = g^{sk_A}$. If any of the equalities were invalid \mathcal{S} sends **abort** to the TTP computing **dBuildMWTx** and returns \perp . Otherwise he extracts $t = tx.t$ and sends the inputs $([sp\mathcal{C}], p, t)$ to the TTP and receives back the outputs $(tx, sp\mathcal{C}_A^*)$.
4. The simulators task is it now to construct ptx' which he can achieve in the following steps:
 - a) He takes the signature context Λ and final signature σ_{fin} from the final transaction $\Lambda = tx.\Lambda$ and $\sigma_{fin} = tx.\sigma$.
 - b) He computes the adversaries partial signature as $\tilde{\sigma}_A \leftarrow \text{signPt}(m, sk_A, k_A, \Lambda)$
 - c) He further computes

$$pk \leftarrow \Lambda.pk$$

$$pk_A = g^{sk_A}$$

$$(s_A, R_A, \Lambda) \leftarrow \tilde{\sigma}_A$$

$$(s, R) \leftarrow \sigma_{fin}$$

$$s_B = s - s_A$$

$$R_B = R \cdot R_A^{-1}$$

$$pk_B = pk \cdot pk_A^{-1}$$

$$\tilde{\sigma}_B = (s_B, R_B, \Lambda)$$

- d) He takes further values from the final transaction:

$$\mathcal{C}_{out}^B = tx.out[1]$$

$$\pi_B = tx.\pi[1]$$

$$C_B = tx.com[1]$$

e) Now he can compute $ptx' \leftarrow \text{createTx}(m, inp, out \parallel \mathcal{C}_{out}^B, \Pi \parallel \pi_B, \Lambda, com \parallel C_B, \tilde{\sigma}_B, t)$

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

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

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

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

We have managed to show that in the case that Alice is corrupted the simulation is perfect, because the transcript is in fact identical to the transcript of the real execution.

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

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

$$r, v \leftarrow \mathbb{Z}_*^* \\ spC \leftarrow \text{createCoin}(r, v)$$

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

$$(ptx, spC_A^*, (sk_A, k_A)) \leftarrow \text{spendCoins}([spC], p, t)$$

2. The simulator invokes \mathcal{A} and sends ptx as if coming from Alice.
3. When \mathcal{A} calls $f_{zk}^{R_1}, f_{zk}^{R_2}$ as the verifier \mathcal{S} simply checks equality with the values he sent and returns either 0 or one. The adversary proceeds by calling $f_{zk}^{R_2}(spC_B^*)$, \mathcal{S} saves spC_B^* and extracts $p = spC_B^*.v$. He then calls the TTP computing `dBuildMWTx` with the input p and receives (tx, spC_B^*) .
4. Next \mathcal{A} sends an updated pre-transaction ptx' . \mathcal{S} verifies the output coin added by \mathcal{A} matches with spC_B' , if it does not he sends `abort` to the TTP and outputs \perp . Now \mathcal{S} computes the following values from the signature context Λ provided in the final transaction and Λ' provided by \mathcal{A} :

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

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

$$\begin{aligned} \Lambda &:= \{pk = 1, R = 1\} \\ \Lambda &\leftarrow \text{setupCtx}(\Lambda, pk_A, R_A) \end{aligned}$$

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

6. The simulator repeats the steps until 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 tx .

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

- *Phase 1:* In the first iteration the simulator constructs the input values $[spC]$ from random values and also chooses a random transaction value p . \mathcal{S} constructs the pre-transaction using those chosen value rather than the real ones. We claim that

the adversary cannot distinguish the chose 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 pedersen commitments $C_1 = g^{r_1} \cdot h^v$, $C_2 = g^{r_2} \cdot h^{v'}$ which one commits to v , from which follows that he could break the hiding property of pedersen commitments. Not being able to extract the coin values the adversary has no chance of knowing if they are correctly chosen at this point. For the same reasons the pre-transaction sent by \mathcal{S} after the rewind will be indistinguishable from a real one. However, as this time the pre-transaction is constructed from the real value which \mathcal{S} received from the TTP, the pre-transaction is in fact identical to the pre-transaction sent in the real execution. The calls to $f_{zk}^{R_1}$ and $f_{zk}^{R_2}$ also behave identically to the real execution in which the parties have access to a TTP computing those protocols.

- *Phase 2:* This phase will be identical to the real execution due to the fact that the adversary is deterministic.
- *Phase 3:* The transaction sent to \mathcal{A} in this phase is the one received from the TTP and is therefore identical to what would have been sent in the real execution, given \mathcal{A} sends correct values. (Otherwise the execution would have halted with \perp). We like to emphasize that in the case that we wouldn't have done the rewind step \mathcal{A} would be able to distinguish the transcript from the real one because he can identify differences in the inputs, outputs, proofs and commitment, as well as the signature context of the final transaction tx and the pre-transaction ptx sent in the first phase. For instance inputs which are spend in the final transaction where 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.

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

□

5.4 Atomic Swap protocol

With the outlined Adapted Mimblewimble Transaction Scheme from definition 5.5 we can now construct an Atomic Swap protocol with another Cryptocurrency. In this thesis we will explain a swap with Bitcoin, as at present Bitcoin and Bitcoin-like cryptocurrencies are the most widely adopted. We will generally refer to the “Bitcoin side“ and the “Mimblewimble side“ of the swap to be most generic. Upon implementation one has to decide for a specific implementation, for example BTC on the Bitcoin side and Grin on the Mimblewimble side. On the Bitcoin side we construct two DPT functions (`lockBtcScript`, `verifyLock`).

- $(spk) \leftarrow \text{lockBtcScript}(pk_A, pk_B, X, t)$: The locking script function lets Bob construct a Bitcoin script only spendable by Alice if she receives the discrete logarithm x of X with $X = g^x$. Additionally the function requires Bobs public key pk_B and a timelock t (given as a block number) as input which allows Bob to reclaim his funds after some time if the atomic swap was not completed successfully. The function will create and return a Bitcoin script spk to which Bob can send funds using a P2SH transaction. If Alice (knowing her secret key sk_A) acquires x , she can construct the complete secret key simply by calculating $sk := sk_A + x$. This construction is similiar although simpler to the locking mechanism described by Malavolta et al. For a in-depth security analysis of this concept we refer the interested reader to their paper [malavolta2019anonymous]. For a concrete Bitcoin Script realizing this functionality see section 6.
- $\{1, 0\} \leftarrow \text{verifyLock}(pk_A, pk_B, X, v, t)$: The lock verification algorithm takes as input Alices, Bobs public keys and the statement X . The function will compute the Bitcoin lock script spk as created by `lockBtcScript` and then check the Bitcoin blockchain if the value locked under the script equals v . Upon successful verification the function returns 1, otherwise 0.

5.4.1 Setup phase

We assume Alice owns Mimbalewimble coins \mathcal{C}_{inp} with a value of v_{mw} and blinding factor r_A and Bob Bitcoin locked in some UTXO ψ with a value of v_{btc} belonging to him. Before the protocol can start the two parties must agree on the value they want to swap, the exchange rate of the currencies and a time after which the swap should be canceled. After coming to an agreement the following variables are defined and known by both Alice and Bob:

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

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

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

In practice, we need to consider that exchange rates might fluctuate, furthermore timeouts have to be calculated separately for each chain. The problems with cross chain payments are discussed by Tairi et al. in [tairi2019a21], they propose to use a fixed exchange rate

5. ADAPTOR SIGNATURE BASED ATOMIC SWAPS BETWEEN BITCOIN AND A MIMBLEWIMBLE BASED CRYPTOCURRENCY

for each day and to use a real world timeout like one day and then calculate the specific block numbers by taking the average block time of the blockchain into account. In our setup we can also fix the exchange rate at the beginning of the protocol, which stays unchanged during protocol execution. If the exchange rate fluctuates and one party is negatively impacted he or she could still decide to stop being cooperative which means the coins would be returned to the original owners after the timeout.

We formalize the protocol **setupSwp** in figure 5.8. The protocol takes as input the shared swap state \mathcal{A} from both parties. From Alice her Mimbewimble input coin \mathcal{C}_{inp} together with the required spending information r_A and the coins value v_{mw} . From Bob we require the UTXO ψ he wants to spend, similar to Alice he needs to provide spending secret σ and total value stored in the UTXO v_{btc} , although this could also be read from the blockchain.

The protocol starts by both parties creating and exchanging keys. Bob now creates two new Bitcoin outputs ψ_{lock} and ψ_B , of which one is the locked Bitcoins which Alice might retrieve later (or Bob after time t_{btc} has passed), and the other Bobs change output. (Difference between what is stored in the input UTXO and what should be sent to Alice). After Bob has published the transaction sending value to the new outputs he will provide Alice with the statement X under which the Bitcoins' are locked together with Alice's public key. Alice can now verify that the funds on Bitcoin side are indeed correctly locked. After that she will collaborate with Bob to spend her Mimbewimble coins into an output shared by both parties. Immediately after, both parties collaborate again to spend this shared coin back to Alice with a timelock of t_{mw} . It is immanent that Alice does not publish the first transaction (A \rightarrow AB) before the timelocked refund transaction (AB \rightarrow A) was signed, otherwise her funds are locked in the shared output without the possibility of refund. The setup protocol concludes with the funds locked up in both chains and ready to be swapped.

¹spendCoins($[spC]$, a_{mw} , \perp)
²dRecvCoins((ptx, ptx) , (ptx, ptx))
³dSpendCoins($([spC_A]$, a_{mw} , $t_{mw})$, $([spC_B]$, a_{mw} , $t_{mw})$)
⁴dFinTx($(ptx_2^*$, sk_A , $k_A)$, $(ptx_2^*$, sk_B , $k_B)$)

$\text{setupSwp}(\langle \mathcal{A}, \mathcal{C}_{inp}, r_A, v_{mw} \rangle, \langle \mathcal{A}, \psi, \sigma, v_{btc} \rangle)$

<p>1 : <i>Alice</i></p> <p>2 : $(sk_A, pk_A) \leftarrow \text{keyGen}(1^n)$</p> <p>3 :</p> <p>4 :</p> <p>5 :</p> <p>6 :</p> <p>7 :</p> <p>8 :</p> <p>9 :</p> <p>10 :</p> <p>11 :</p> <p>12 :</p> <p>13 :</p> <p>14 : if $\text{verifyLock}(pk_A, pk_B, X, a_{btc}, \emptyset) = 0$</p> <p>15 : return 0</p> <p>16 : $(ptx, (sk_A, k_A)) \leftarrow \text{spendCoins}^1$</p> <p>17 :</p> <p>18 :</p> <p>19 :</p> <p>20 : $tx \leftarrow \text{finTx}(ptx^*, sk_A, k_A)$</p> <p>21 :</p> <p>22 : $ptx_2^* \leftarrow \text{recvCoins}(ptx_2, a_{mw})$</p> <p>23 :</p> <p>24 : $\text{publish}_{\text{MW}}([tx, tx_2])$</p> <p>25 : return \mathcal{A}</p> <p>26 :</p>	<p><i>Bob</i></p> <p>$(sk_B, pk_B) \leftarrow \text{keyGen}(1^n)$</p> <p>$(x, X) \leftarrow \text{keyGen}(1^n)$</p> <p>$spk \leftarrow \text{lockBtcScript}(pk_A, X, pk_B, t_{btc})$</p> <p>$\psi_{lock} \leftarrow \text{createUTXO}(a_{btc}, spk)$</p> <p>$\mathcal{A} := \mathcal{A} \cup \psi_{lock} \cup X$</p> <p>$\psi_B \leftarrow \text{createUTXO}(v_{btc} - a_{btc}, pk_B)$</p> <p>$tx_{btc} \leftarrow \text{buildTransaction}([\psi_{inp}], [\psi_{lock}, \psi_B], 1, \emptyset)$</p> <p>$tx_{btc}^* \leftarrow \text{signTransaction}(tx_{btc}, [\sigma])$</p> <p>$\text{publish}_{\text{BTC}}([tx_{btc}^*])$</p> <p>$X$</p> <p>$\langle (ptx^*, (\mathcal{C}_{out}^{sh}, r_A^*)), (ptx^*, (\mathcal{C}_{out}^{sh}, r_B^*)) \rangle \leftarrow \text{dRecvCoins}^2$</p> <p>$\mathcal{A} := \mathcal{A} \cup \mathcal{C}_{out}^{sh}$</p> <p>$\langle (ptx_2, (sk_A, k_A)), (ptx_2, (sk_B, k_B)) \rangle \leftarrow \text{dSpendCoins}^3$</p> <p>$\langle tx_2, tx_2 \rangle \leftarrow \text{dFinTx}^4$</p> <p>return \mathcal{A}</p>
---	--

Figure 5.8: Atomic Swap - setupSwp .

5.4.2 Execution Phase

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

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

This function will verify that there is sufficient time to execute the atomic swap protocol. As input it takes a chain paramter 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 Mimbalewimble side a_{mw} . We assume the participants have successfully run the `setupSwp` protocol and both know the updated swap state \mathcal{A} as returned by the setup protocol. Alice (Mimbalewimble side) needs to provide her part of the blinding factor r_A^* of the shared Mimbalewimble coin \mathcal{C}_{out}^{sh} as well as the swap state \mathcal{A} as protocol inputs. Bob (Bitcoin side) needs to provide his part of the blinding factor r_B^* of the shared Mimbalewimble coin \mathcal{C}_{out}^{sh} , the secret witness value x and the swap state \mathcal{A} as protocol inputs. The protocol starts with the two parties creating a Mimbalewimble transaction with the goal of spending the shared input coin to Bob. Bob will call the `aptRecvCoins` routine to hide the secret x in his adapted signature $\hat{\sigma}_B$, which Alice can retrieve from the final transaction. Note that at this point only Bob knows his original partial signature $\tilde{\sigma}_B$. Therefore they have to cooperate again in the `dAptFinTx` to compute the final Mimbalewimble transaction transferring value from the shared coin \mathcal{C}_{out}^{sh} to Bob. After Bob has published the transaction and is now in possession of the swapped funds on the Mimbalewimble side, he will send the transaction to Alice (or Alice could retrieve it directly from the blockchain, if Bob would refuse to cooperate). Now Alice knows all the pieces needed to extract the secret witness x :

- The final signature σ_{fin} .
- Bobs adapted signature $\hat{\sigma}_B$ previously verified to contain x .
- Her and Bobs partial signature spending the shared coin σ_{AB} .

She can now call `extWit` to retrieve x and compute the secret key sk_{BTC} needed to unlock the Bitcoin funds. She will now create her own output ψ_A , and will transfer the funds from the lock. After publishing this transaction to the Bitcoin network, Alice is in full possession of the swapped funds on the Bitcoin side. The procedure returns 1 if the swap was successfully executed or 0 otherwise (when not enough time was left to safely execute the swap).

⁵`dSpendCoins` $\langle ([sp\mathcal{C}_A], a_{mw}, \perp), ([sp\mathcal{C}_B], a_{mw}, \perp) \rangle$

⁶`dAptFinTx` $\langle (ptx^*, sk_A, k_A, X), (ptx^*, sk_B, k_B, \tilde{\sigma}_B) \rangle$

execSwap($\langle (\mathcal{A}, r_A^*), (\mathcal{A}, x, r_B^*) \rangle$)

<pre> 1 : Alice 2 : $(a_{mw}, a_{btc}, t_{mw}, t_{btc}, \mathcal{C}_{out}^{sh}, \psi_{lock}, X) \leftarrow \mathcal{A}$ 3 : 4 : 5 : 6 : $(..., \hat{\sigma}_B) \leftarrow ptx^*$ 7 : 8 : 9 : 10 : 11 : 12 : $(..., \sigma_{fin}) \leftarrow tx_{mw}$ 13 : $x \leftarrow \text{extWit}(\sigma_{fin}, \sigma_{AB}, \hat{\sigma}_B)$ 14 : $sk_{BTC} \leftarrow sk_A + x$ 15 : $(sk_A', pk_A') \leftarrow \text{keyGen}(1^n)$ 16 : $\psi_A \leftarrow \text{createUTXO}(a_{btc}, pk_A')$ 17 : $tx_{btc} \leftarrow \text{buildTransaction}([\psi_{lock}^*, [\psi_A], 1, \emptyset)$ 18 : $tx_{btc}^* \leftarrow \text{signTransaction}(tx_{btc}, sk_{BTC})$ 19 : $\text{publish}_{BTC}(tx_{btc}^*)$ 20 : return 1 </pre>	<pre> Bob $(a_{mw}, t_{mw}, t_{btc}, \mathcal{C}_{out}^{sh}) \leftarrow \mathcal{A}$ $(ptx^*, (\mathcal{C}_{out}^B, r_B'), \tilde{\sigma}_B) \leftarrow \text{aptRecvCoins}(ptx, a_{mw}, x)$ $\xleftarrow{ptx^*}$ if $\text{verfTime}(BTC, t_{btc}) = 0 \vee \text{verfTime}(MW, t_{mw}) = 0$ return 0 $\langle \sigma_{AB}, tx_{mw} \rangle \leftarrow \text{dAptFinTx}^6$ $\text{publish}_{MW}(tx_{mw})$ $\xleftarrow{tx_{mw}}$ return 1 </pre>
---	---

Figure 5.9: Atomic Swap - setupSwp.

5.4.3 Refunding

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

On the Mumblewimble 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. Because of this line number 24 in the setup phase is very important. Alice here published both the locking of the funds and the refund at the same time. If she would publish the locking transaction first, Bob could refuse to cooperate, in which case the funds would stay in the locking output only retrievable if both parties cooperate. If the swap executes successful the refund transaction would get discarded by miners, as it then is no longer valid even after the timeout t_{mw} .

CHAPTER 6

Implementation

- 6.1 Implementation Bitcoin side
- 6.2 Implementation Grin side
- 6.3 Performance Evaluation

List of Figures

3.1	A decoded Bitcoin transaction ¹	13
3.2	Original transaction building process	19
3.3	Salvaged transaction protocol by Fuchsbauer et al. [fuchsbauer2019aggregate]	21
4.1	Schnorr Signature Scheme as first defined in [schnorr1989efficient] . . .	26
4.2	Two Party Schnorr Signature Scheme	28
4.3	Fixed Witness Adaptor Schnorr Signature Scheme	29
5.1	Instantiation of Mimblewimble Transaction Scheme.	48
5.2	dBuildMWTx two-party protocol to build a new transaction	49
5.3	Extended Mimblewimble Transaction Scheme - dSpendCoins	50
5.4	Extended Mimblewimble Transaction Scheme - dRecvCoins	52
5.5	Extended Mimblewimble Transaction Scheme - dFinTx	53
5.6	Adapted Extended Mimblewimble Transaction Scheme - aptRecvCoins. .	54
5.7	Adapted Extended Mimblewimble Transaction Scheme - dAptFinTx. . . .	55
5.8	Atomic Swap - setupSwp.	67
5.9	Atomic Swap - setupSwp.	69

¹<https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch06.asciidoc>.

List of Tables

List of Algorithms