TU WIEN Informatics

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

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Jakob Abfalter, BSc
Registration Number 01126889

to the Faculty of Informatics

at the TU Wien

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

Vienna, 6th April, 2020

Jakob Abfalter | Matteo Maffei

Erklärung zur Verfassung der Arbeit

Jakob Abfalter, BSc

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

Wien, 6. April 2020

Jakob Abfalter

Acknowledgements

Enter your text here.

Abstract

Enter your text here.

Contents

CHAPTER 1

Introduction

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

Mimblewimble The Mimblewimble protocol was introduced in 2016 by an anonymous entity named Jedusor, Tom Elvis [**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

[1] https://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

CHAPTER 3

Preliminaries

In this chapter we will lay down the general notations and definitions required for the later parts of the thesis. In section 3.1 we will define several cryptographic primitives which are required for our constructions. Section 3.2 will describe several definitions around Bitcoin, particularly its transaction structure. After that in secion 3.3 we will discuss the notion of privacy enhancing cryptocurrencies, and then range proofs in section **??** 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. $\mathsf{genRel}((1^n))$ 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 discirte logarithm in a group $\mathbb{G}$ of a number n as the number m such that for the groups generator g the following holds:

$$g^m = n$$

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

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

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

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

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

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

A valid signature scheme has to fulfill two security properties:

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

- Unforgeability ($\mathsf{EUF} - \mathsf{CMA}$): 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 4.2.3

Definition 3.4 (Cryptographic Hash Function)**.** A cryptographic hash function H is defined as $\mathsf{H}(I) \rightarrow \{0, 1\}^n$ for some fixed number n and some input I [**al2011cryptographic**]. A secure hashing function has to fulfill the following security properties:

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

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

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

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

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

- $PP \leftarrow \mathsf{setup}(1^n)$: The setup procedure is a DPT function, it takes as input a security parameter 1^n and outputs public parameters PP. Depending on PP we define a input space $\mathbb{I}_{PP}$, a randomness space $\mathbb{K}_{PP}$ and a commitment space $\mathbb{C}_{PP}$.

- $C \leftarrow \mathsf{commit}(I, k)$ 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 fullfill 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 probabilty of finding a $I^* \neq I$ and a k^* with $\mathsf{commit}(I, k) = \mathsf{commit}(I^*, k^*)$ 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 $C := \mathsf{commit}(I_b, k,)$ the probabilty of the adversary choosing the correct b out of $\{0, 1\}$ must not be higher then $\frac{1}{2} + \mathsf{negl}(P)$.

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

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

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 $(\mathsf{keyGen}, \mathsf{commit})$ are then instantiated as:

$$\mathsf{keyGen}(1^n) := g, h \leftarrow_{\$} \mathbb{G}$$

$$\mathsf{commit}(I, k) := g^k h^I$$

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 5.3 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 empoloyed. An UTXO is a output constructed in a previous transaction which holds value in the form of an amount expressed in Bitcoin (more precisely in Satoshis, which is the smallest unit of Bitcoin) and a locking condition (referred to as scriptPubKey). Unspent means that this output has not been spent yet in a transaction and its funds are therefore available to be redeemed by a participant capable of unlocking the output. To unlock this value one has to provide a script fulfilling the locking condition, referred to as scriptSig. In the most common case the lock condition will be to provide a valid signature under a public key. This is referred to as a P2PK or P2PKH output which we will see in more detail in section 3.2.1. However, more complex conditions, such we shall see in section 3.2.1 are possible.

Definition 3.8 (Unspent Transaction Output - UTXO)**.** An unspent transaction output is a data structure consisting of a locking condition spk, a value expressed in Bitcoin v and an unlocking script σ which is initially empty and has to be provided by the owner when spending the UTXO in a transaction. In this paper we generally use ψ to refer to a singular UTXO and Ψ to refer to a set of UTXOs.

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

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

createUTXO(v, spk)

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

spendUTXO(ψ, σ)

1 : $\{v, spk\} \leftarrow \psi$

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

verfUTXO(ψ)

1 : $\{v, spk, \sigma\} \leftarrow \psi$

2 : **return** verf$(spk,\ \sigma,\ v)$

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

Definition 3.9 (Bitcoin Transaction)**.** A Bitcoin transaction consists of a series of input UTXOs Ψ_{inp}, a series of output UTXOs Ψ_{out}, a transaction version vs, and an optional locktime t:

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

A transaction is valid if the follwing conditions are fullfilled:

- The total value of inputs is greater or equal the total value of outputs.
- For all $\psi \in \psi$ 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 3.10 (Bitcoin Transaction Scheme)**.** We define a Bitcoin Transaction scheme as a tupel of three DPT functions (buildTransaction, signTransaction, verfTransaction).

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


Figure 3.1: A decoded Bitcoin transaction[1]

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

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

P2PK, P2PKH

P2PK stands for Pay-to-Public-Key and P2PKH for Pay-to-Public-Key-Hash. In this type of output *spk* will be constructed such that its value unlocks if a correct signature is provided in σ for a corresponding public key *pk*. P2PKH is an update to this script in which the *spk* contains a hashed version of the public key *pk*, instead of the public key itself. To spend a P2PKH output one has to provide the unhashed public key in

addition to a valid signature. This type of output, is the most commonly used output in the Bitcoin blockchain to transfer value from one participant to another. Delgado et at. found in their paper Analysis of the Bitcoin UTXO set from 2017 that more then 80% of the UTXO set at that time consisted of P2PKH transactions, whereas about 17% were P2SH and 0.12% P2PK outputs. [**delgado2018analysis**] P2PKH outputs can be encoded into a Bitcoin address using base58 encoding. This addresses can be handed out to request a payment from somebody.

P2SH

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

3.3 Privacy-enhancing Cryptocurrencies

3.3.1 Zero Knowledge Proofs

3.3.2 Range Proofs

Definition 3.11 (Rangeproof System)**.** A Rangeproofs system $\Pi[COM]$ with regards to a homomorphic commitment scheme COM consists of a tupel of functions ($\mathsf{ranPrfSetup}, \mathsf{ranPrf}, \mathsf{vrfRanPrf}$).

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

Definition 3.12 (Multiparty Rangeproof System)**.** A Multiparty Rangeproof Sytem $\Pi_{mp}[COM]$ with regards to a homomorphic commitment scheme COM is an extension of the regular Rangeproof Sytem with the following distributed protocol $\mathsf{dRanPrf}$.

- $\pi \leftarrow \mathsf{dRanPrf}\langle(C, v, r_A), (C, v, r_B)$: 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 particularily 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 transction output (UTXO) in Bitcoin.

Definition 3.13 (Mimblewimble Coin)**.** For two adjacent elliptic curve generators g and h a coin in Mimblewimble is a tuple of the form $(\mathcal{C}, \pi)$, where $\mathcal{C} := g^v \cdot h^k$ 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 pratice $\langle 0, 2^{64} - 1\rangle$ is used in the most prominant implementations.

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

Definition 3.14 (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 3.15 (Transaction validity)**.** A transaction is valid if:

- The transaction is well-balanced as defined in definition 3.14
- $\forall\, (\mathcal{C}_i \pi_i) \in \mathcal{C}_{out}\ \mathsf{vrfRanPrf}(\pi_i, \mathcal{C}_i) = 1$

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

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

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:

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

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

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

Then we can build a single merged transaction:

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

We can easily deduce that if tx_0 and tx_1 are valid, it must follow that tx_m is valid:
If tx_0 and tx_1 are valid as of definition 3.15 that means $\mathcal{C}^0_{inp} \;-\; \mathcal{C}^0_{out} \;-\; h^{s_0} \;=\; \mathcal{E}_0$, $\{\pi_0\}$ contains valid range proofs for the outputs $\mathcal{C}^0_{out}$ and $\{\sigma_0\}$ contains a valid signature to $\mathcal{E}_0 \;-\; h^{s_0}$ as public key, the same must hold for tx_1.

By the rules of arithmetic it then must also hold that

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

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

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

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

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

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

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

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

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

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

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

Essentially this means tx_1 spends a coin created in tx_0. Now lets recall the equation given for transaction well-balancedness in 3.14:

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

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

Figure 3.2 depicts the original transaction flow.

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

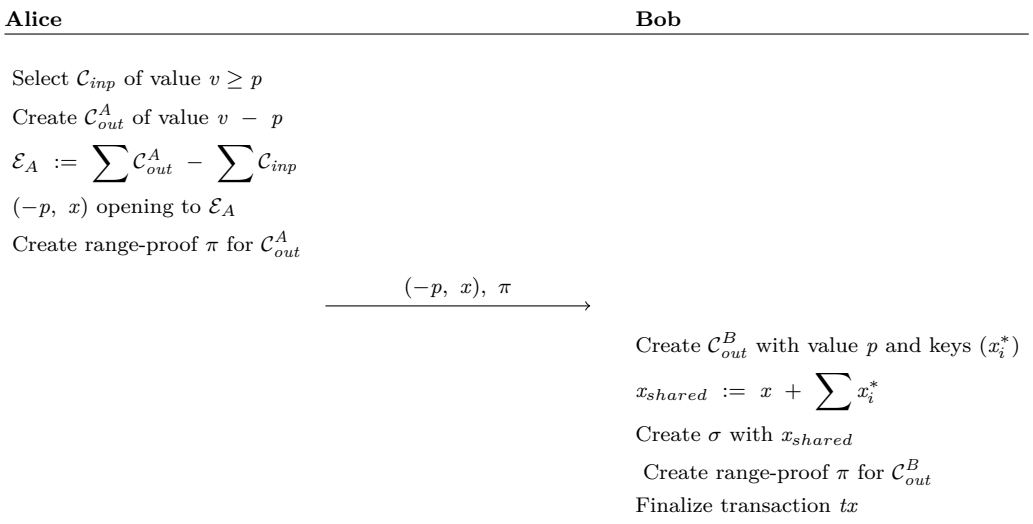

Figure 3.2: Original transaction building process

Pedro: Really nice that you created this protocol :)

In detail this reverting transaction would look like:

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

Again remembering the construction of the Excess value of this construction would look like this:

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

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

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

Pedro: Why range proof is not correct here in the first place?

which is the opening to $\mathcal{E}_{rv}$. Furthermore obtaining a valid range proofs is trivial, as it once was a valid output the ledger will cointain a valid proof for this coin already.

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 Cryptocurrency by making the signing process itself a two-party process which will be explained in more detail in chapter 4.

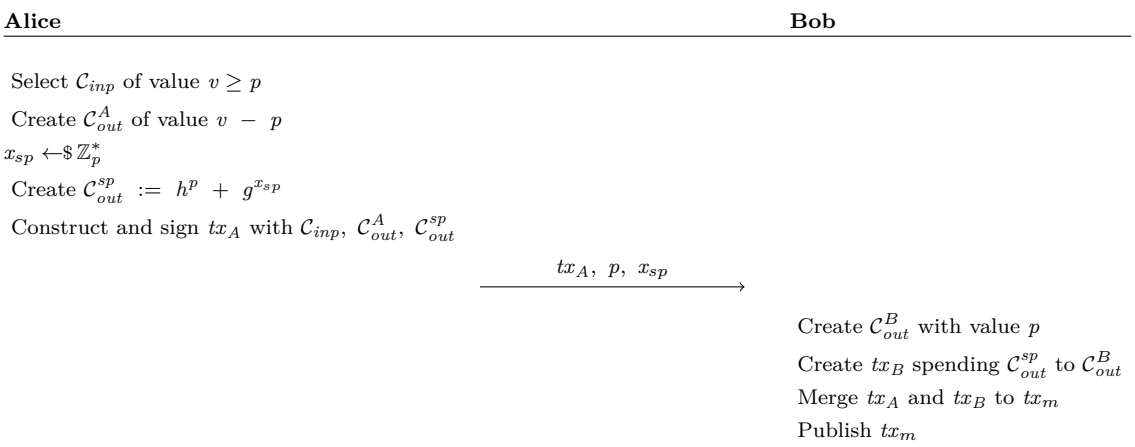

Figure 3.3: Salvaged tranction protocol by Fuchsbauer et al. [**fuchsbauer2019aggregate**]

Pedro: put labels inside captions

Fuchsbauer et al. [**fuchsbauer2019aggregate**] proposed the following alternative way to build transactions which would not be vulnerable to this problem.

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

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

The only 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. A figure showing the protocol flow is depicted in Figure 3.3.

3.5 Scriptless Scripts

3.6 Adaptor Signatures

CHAPTER 4

Two Party Fixed Witness Adaptor Signatures

In this chapter, we will define a variant of the adaptor signature scheme as explained in section **??**. 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 4.2.1.

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 $(\mathsf{dKeyGen}, \mathsf{signPrt}, \mathsf{vrfPt}, \mathsf{finSig})$ defined as follows:

- $((sk_A, pk_A, k_A, \Lambda), (sk_B, pk_B, k_B, \Lambda)) \leftarrow \mathsf{dKeyGen}\langle 1^n, 1^n \rangle$: The distributed key generation protocol takes as input the security parameter from both Alice and Bob and returns the tuple $(sk_A, pk_A, k_A, \Lambda)$ 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.
- $(\tilde{\sigma_A}) \leftarrow \mathsf{signPrt}(m, sk_A, k_A, \Lambda)$: 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 (similiar for Bob) as well as the shared signature context Λ. The procedure outputs $(\tilde{\sigma_A})$, that is, a share of the signature to a participant.
- $\{1, 0\} \leftarrow \mathsf{vrfPt}(\tilde{\sigma_A}, m, pk_A)$: The share verification algorithm is a DPT function that takes as input a signature share $\tilde{\sigma_A}$, a message m, and the other participants public key pk_A (similiar pk_B for Bobs partial signature). The algorithm returns 1 if the verification was successfull or 0 otherwise.
- $\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}, \tilde{\sigma_B})$: The finalize signature algorithm is a DPT function that takes as input two shares of the signatures and combines them into a final signature valid ander the shared public key pk.

We require the two party signature scheme to be correct as well as unforgeable against chosen message attacks (EUF-CMA). EUF-CMA for a two-party signature scheme was defined for instance in [**lindell2017fast**]. For the correctness of the distributed key-generation protcol $\mathsf{dKeyGen}$, special care needs to be taken to gurantee 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:

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

- $\hat{\sigma_B} \leftarrow \mathsf{adaptSig}(\tilde{\sigma_B}, x)$: 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 a adapted partial signature $\hat{\sigma}$ which can be verified to contain x using the $\mathsf{verifyAptSig}$ function, without having to reveal x.

- $\{1,0\} \leftarrow \mathsf{verifyAptSig}(\hat{\sigma_A},\ m,\ pk_A,\ X)$: 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.
- $x \leftarrow \mathsf{extWit}(\sigma_{fin},\ \tilde{\sigma_A},\ \hat{\sigma_B})$: The witness extraction algorithm is a DPT function that lets Alice extract the secret witness x 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 then the definition seen in **??**. 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 layed out in 5.3.

Definition 4.3 (Secure Adaptor Signature Scheme)**.** As defined by Aumayr et al. in [**aumayr2020bitcoinchan**
a secure adaptor signature scheme needs two security properties to be fulfilled:

1. $\mathsf{aEUF-CMA}$
2. Witness Extractability

We proceed by redefining these properties as well as adapted correctness for our adapted two-party fixed witness signature scheme defined in definition 4.2:

Similar to how it is defined in [**aumayr2020bitcoinchannels**] additionally to 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 4.4 (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 $((sk_A, pk_A, k_A, \Lambda), (sk_B, pk_B, k_B, \Lambda)) \leftarrow \mathsf{dKeyGen}\langle 1^n,\ 1^n\rangle$ with their composite public key $\Lambda.pk$ and every statement/witness pair $(X,\ x)\mathsf{genRel}(1^n)$ in a relation R it must hold that:

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

Additionally to the regular definition of *existential unforgeability under chosen message attacks* as defined for example in [**lindell2017fast**] or [**vaudenay2006classical**] we require that it is hard to produce a forged partial signature $\tilde{\sigma}$ if the adversary $\mathcal{A}$ gets to know a valid adapted signature $\hat{\sigma}$ w.r.t. some message m and a statement X.

Definition 4.5 ($\mathsf{aEUF-CMA}$)**.** For the definition of $\mathsf{aEUF-CMA}$ -security we define the experiment $\mathsf{forgeAptSig}_{\mathcal{A}}$ for a PPT adversary $\mathcal{A}$ with a keypair $(sk_A,\ pk_A)$, meaning the attacker plays the role of Alice in the protocol as follows:

$\mathsf{forgeAptSig}_{\mathcal{A}}(n)$

$\mathbb{S} := \emptyset$
$((sk_A, pk_A, k_A, \Lambda), (sk_B, pk_B, k_B, \Lambda)) \leftarrow \mathsf{dKeyGen}\langle 1^n,\ 1^n\rangle$
$m \leftarrow \mathcal{A}^{\mathcal{O}_{ps}(\cdot,\cdot,\cdot,\cdot)}(pk_B)$
$(x,\ X) \leftarrow \mathsf{genRel}(1^n)$
$\tilde{\sigma_A} \leftarrow \mathsf{signPrt}(m,\ sk_A,\ k_A,\ \Lambda)$
$\tilde{\sigma_B} \leftarrow \mathsf{signPrt}(m,\ sk_B,\ k_B,\ \Lambda)$
$\hat{\sigma_B} \leftarrow \mathsf{adaptSig}(\tilde{\sigma_B},\ x)$
$\tilde{\sigma_A}^* \leftarrow \mathcal{A}^{\mathcal{O}_{ps}(\cdot,\cdot,\cdot,\cdot)}(\tilde{\sigma_A}, \hat{\sigma_B})$
$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A}^*,\ \tilde{\sigma_B})$
return $((m) \notin \mathbb{S} \ \wedge\ \tilde{\sigma_A}^* \neq \tilde{\sigma_A} \ \wedge\ \mathsf{verf}(m,\ \sigma_{fin},\ \Lambda.pk))$

$\mathcal{O}_{ps}(m, pk_A, pk_B, \Lambda)$

$(x,\ X) \leftarrow \mathsf{genRel}(1^n)$
$\tilde{\sigma_A} \leftarrow \mathsf{signPrt}(m,\ sk_A,\ k_A,\ \Lambda)$
$\tilde{\sigma_B} \leftarrow \mathsf{signPrt}(m,\ sk_B,\ k_B,\ \Lambda)$
$\hat{\sigma_B} \leftarrow \mathsf{adaptSig}(\tilde{\sigma},\ x)$
$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A},\ \hat{\sigma_B})$
$\mathbb{S} := \mathbb{S} \cup \{m\}$
return $(\sigma_{fin},\ X)$

The adapted signature scheme Φ_{Apt} is called $\mathsf{aEUF-CMA}$ -secure if

$$\Pr[\mathsf{forgeAptSig}_{\mathcal{A}}(n) = 1] \leq \mathsf{negl}(n)$$

Definition 4.6 (Witness Extractability)**.** Informally the Witness Extractability property holds for an adapted signature scheme Φ_{Apt} computed for the statement X when we

can always extract the witness $(x,\ X)$ from the final signature σ_{fin}, given the partial signatures of the participants. To formalize this statement we describe an experiment $\mathsf{aExtrWit}_{\mathcal{A}}$ for a PPT adversary $\mathcal{A}$ with a keypair (sk_B, pk_B), meaning the attacker plays the role of Bob in the protocol.

$\mathsf{aExtrWit}_{\mathcal{A}}(n)$

$\mathbb{S} := \emptyset$
$((sk_A, pk_A, k_A, \Lambda), (sk_B, pk_B, k_B, \Lambda)) \leftarrow \mathsf{dKeyGen}\langle 1^n,\ 1^n\rangle$
$(m,\ (x, X) \in R) \leftarrow \mathcal{A}^{\mathcal{O}_{ps}(\cdot,\cdot,\cdot,\cdot)}(pk_A)$
$\tilde{\sigma_A} \leftarrow \mathsf{signPrt}(m,\ sk_A,\ k_A,\ \Lambda)$
$\tilde{\sigma_B} \leftarrow \mathsf{signPrt}(m,\ sk_B,\ k_B,\ \Lambda)$
$(\hat{\sigma_B},\ \sigma_{fin}) \leftarrow \mathcal{A}^{\mathcal{O}_{ps}(\cdot,\cdot,\cdot,\cdot)}(pk_B)$
$x^* \leftarrow \mathsf{extWit}(\sigma_{fin},\ \tilde{\sigma_A},\ \hat{\sigma_B})$
return $(m \notin \mathbb{S} \ \wedge\ (X,\ x^*) \notin R \ \wedge\ \mathsf{verf}(m,\ \sigma_{fin},\ \Lambda.pk))$

$\mathcal{O}_{ps}(m, pk_A, pk_B, \Lambda)$

$\mathbb{S} := \mathbb{S} \cup m$
$\tilde{\sigma_A} \leftarrow \mathsf{signPrt}(m,\ sk_A,\ k_A,\ \Lambda)$
$\tilde{\sigma_B} \leftarrow \mathsf{signPrt}(m,\ sk_B,\ k_B,\ \Lambda)$
$\sigma_{fin} \leftarrow \mathsf{finSig}(\tilde{\sigma_A},\ \tilde{\sigma_B})$
return σ_{fin}

In order to satisfy witness extractability the following must hold:

$$\Pr[\mathsf{aExtrWit}_{\mathcal{A}}(n) = 1] \leq \mathsf{negl}(n)$$

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 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.

$\mathsf{keyGen}(1^n)$

1 : $x \leftarrow\$ \mathbb{Z}_p^*$
2 : **return** $(sk := x,\ pk := g^x)$

$\mathsf{sign}(m,\ sk)$

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

$\mathsf{verf}(m,\ \sigma,\ pk)$

1 : $(s,\ R) \leftarrow \sigma$
2 : $e := \mathsf{H}(m \mid\mid R \mid\mid pk)$
3 : **return** $g^s \stackrel{?}{=} R \cdot pk^e$

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

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 instantion:

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

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

This function helps the particpents 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.

signPrt and vrfPt 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

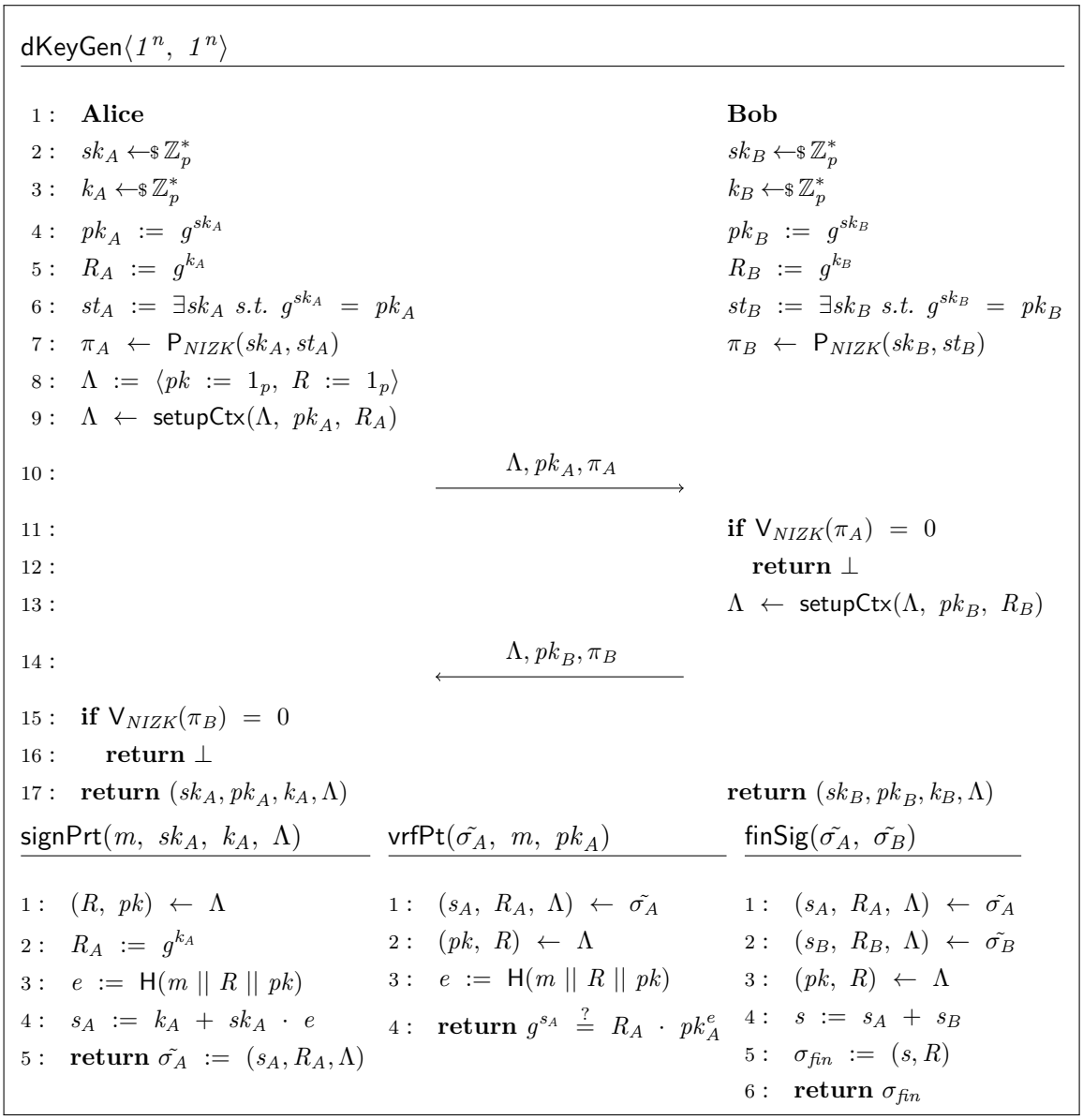

Figure 4.2: Two Party Schnorr Signature Scheme

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 show in figure 4.3 how Alice and Bob can cooperate to produce a final signature which fulfills Correctness as defined in definition 3.3.

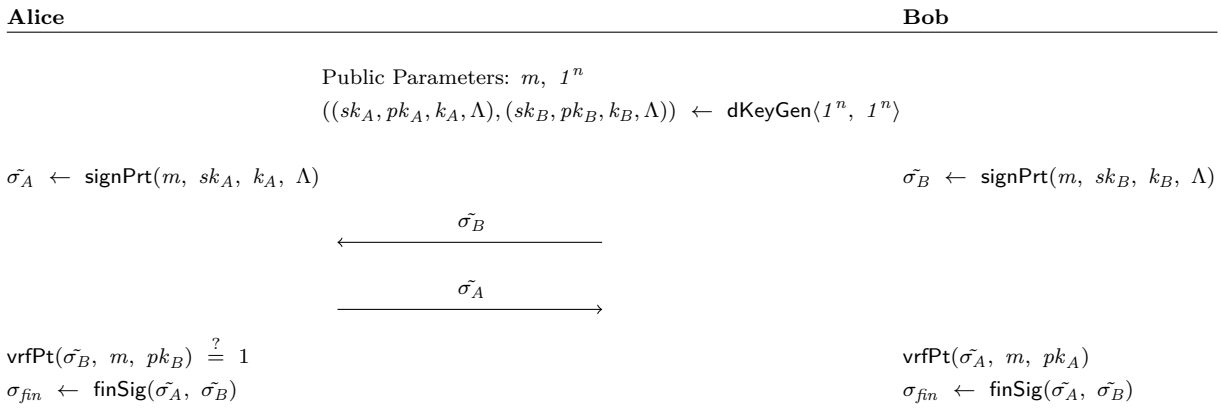

Figure 4.3: Two Party Schnorr Signature Scheme Interaction

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

In figure 4.4 we further provide a schnorr-based instantiation for the fixed witness adapted signature scheme as defined in definition 4.2.

$\mathsf{adaptSig}$ 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 $\mathsf{verifyAptSig}$ 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 portial 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 $\mathsf{aExtrWit}_{\mathcal{A}}$ function.

Again in figure 4.5 we show another example interaction between Alice and Bob creating a signature σ_{fin} for the composite public key $pk := pk_A \cdot pk_B$ while 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. 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.

4.2.1 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 4.3.

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

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

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

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

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

1 : $(s,\ R) \leftarrow \sigma_{fin}$
2 : $(s_A,\ R_A,\ \Lambda) \leftarrow \tilde{\sigma_A}$
3 : $(\hat{s_B},\ R_B,\ \Lambda) \leftarrow \hat{\sigma_B}$
4 : $s_B := s - s_A$
5 : $x := \hat{s_B} - s_B$
6 : **return** (x)

Figure 4.4: Fixed Witness Adaptor Schnorr Signature Scheme

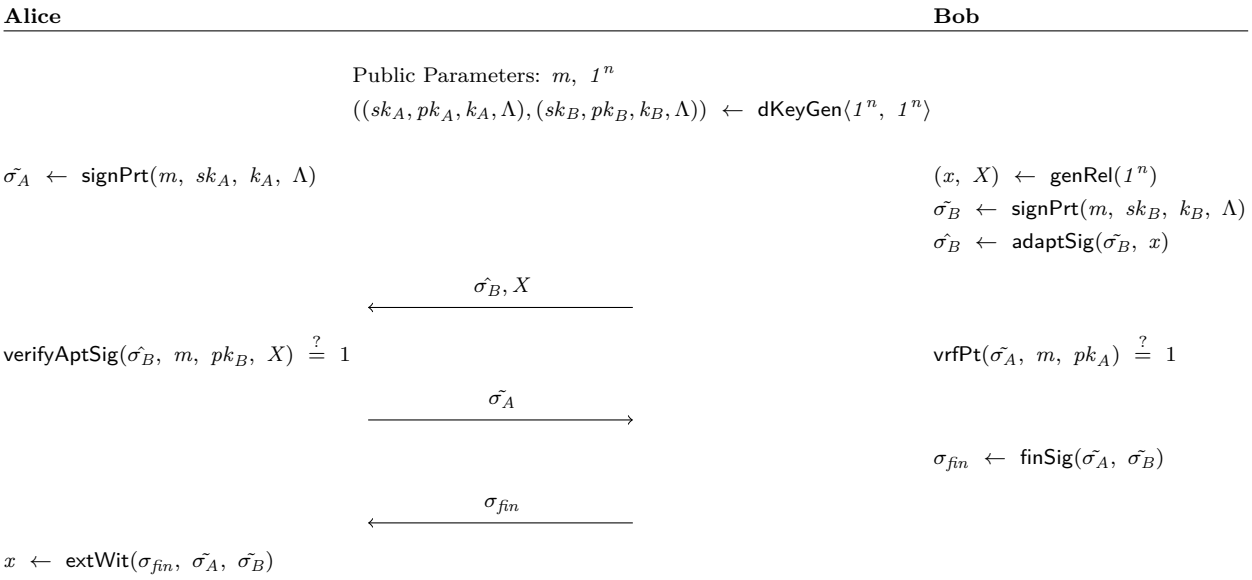

Figure 4.5: Fixed Witness Adaptor Schnorr Signature Interaction

4.2.2 Adaptor Signature Correctness

To prove that Adaptor Signature Correctness holds we have 3 statements to prove, first we prove that $\mathsf{verf}(m,\ \sigma_{fin},\ \Lambda.pk) \stackrel{?}{=} 1$ 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 4.4. 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:

$$g^{s} = R \cdot pk^{e} \tag{4.1}$$
$$g^{s_A} \cdot g^{s_B} \tag{4.2}$$
$$g^{k_A + e \cdot sk_A} \cdot g^{k_B + e \cdot sk_B} \tag{4.3}$$
$$g^{k_A} \cdot pk_A^{e} \cdot g^{k_B} \cdot pk_B^{e} \tag{4.4}$$
$$R_A \cdot pk_A^{e} \cdot R_B \cdot pk_B^{e} \tag{4.5}$$
$$R \cdot pk^{e} = R \cdot pk^{e} \tag{4.6}$$
$$1 = 1 \tag{4.7}$$

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

$$\mathsf{verifyAptSig}(\hat{\sigma_B},\ m,\ pk_B,\ X) \stackrel{?}{=} 1$$

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

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

We now continue to prove the last equation required:

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

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

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

□

4.2.3 Secure Adaptor Signature Scheme

In order to prove the security of the scheme we need to provide a proof that both $\mathsf{aEUF-CMA}$ and Witness Extractability hold in our instantiation. To perform the proof we must first recall the regular definition of $\mathsf{EUF-CMA}$ given for regular schnorr signatures. [**schnorr1989efficient**] For that we define the game $\mathsf{forgeSig}_{\mathcal{A}}$:

$\mathsf{forgeSig}_{\mathcal{A}}(n)$	$\mathcal{O}_s(m, pk)$
$\mathbb{S} \leftarrow \emptyset$	$\sigma \leftarrow \mathsf{sign}(m,\ sk)$
$(sk, pk) \leftarrow \mathsf{keyGen}(1^n)$	$\mathbb{S} := \mathbb{S} \cup \{m\}$
$(m,\ \sigma) \leftarrow \mathcal{A}^{\mathcal{O}_s(\cdot,\cdot)}$	**return** σ
return $((m) \notin \mathbb{S} \ \wedge\ \mathsf{verf}(m,\ \sigma,\ pk))$	

$\mathsf{EUF-CMA}$ holds if $\Pr[\mathsf{forgeSig}_{\mathcal{A}}(n) = 1] \leq \mathsf{negl}(n)$.

Proof. We proof $\mathsf{aEUF-CMA}$ holds by providing a black box reduction from $\mathsf{aEUF-CMA}$ to $\mathsf{EUF-CMA}$ of schnorr signatures. Intuitively if we suppose there exists a PPT adversary $\mathcal{A}$ that wins the $\mathsf{forgeAptSig}_{\mathcal{A}}$ game with probability 1, then $\mathcal{A}$ will also be able to win the $\mathsf{forgeSig}_{\mathcal{A}}$ game with the same probability, which leads us to a contradiction. This can be achieved by splitting up the public key chosen by the challenger in the $\mathsf{forgeSig}_{\mathcal{A}}$ game into pk_1 and pk_2 and then running the $\mathsf{forgeAptSig}_{\mathcal{A}}$ game using the two split keys. Since the signing oracle in $\mathsf{aEUF-CMA}$ and Witness Extractability both provide a signature valid unter the composite of the two public keys, we can simulate the oracle queried by the adversary $\mathcal{A}$ simply by forwarding the query to the $\mathsf{EUF-CMA}$ oracle with the original unsplit version of the public key. The output of the $\mathsf{forgeAptSig}_{\mathcal{A}}$ game will be a forged final signature valid under the combined public key of pk_1 and pk_2 which we can then use to win the $\mathsf{forgeSig}_{\mathcal{A}}$ game. See figure 4.6 for the black-box reduction.

In a very similiar way we can provide a reduction from Witness Extractability to $\mathsf{EUF-CMA}$. Again if we suppose there exists a PPT adversary $\mathcal{A}$ able to win the $\mathsf{aExtrWit}_{\mathcal{A}}$ game with probability 1, then $\mathcal{A}$ will always be able to win the $\mathsf{forgeSig}_{\mathcal{A}}$, leading to a contradiction. Similiar to the previous proof the adversary $\mathcal{A}$ splits up the secret key pk computed during the $\mathsf{forgeSig}_{\mathcal{A}}$ game into pk_1 and pk_2 to use them in the $\mathsf{aExtrWit}_{\mathcal{A}}$. The forged final signature σ_{fin} can then be used to win the $\mathsf{forgeSig}_{\mathcal{A}}$ game. As the black-box reduction is the same as before we again refer to figure 4.6 to see the details.

□

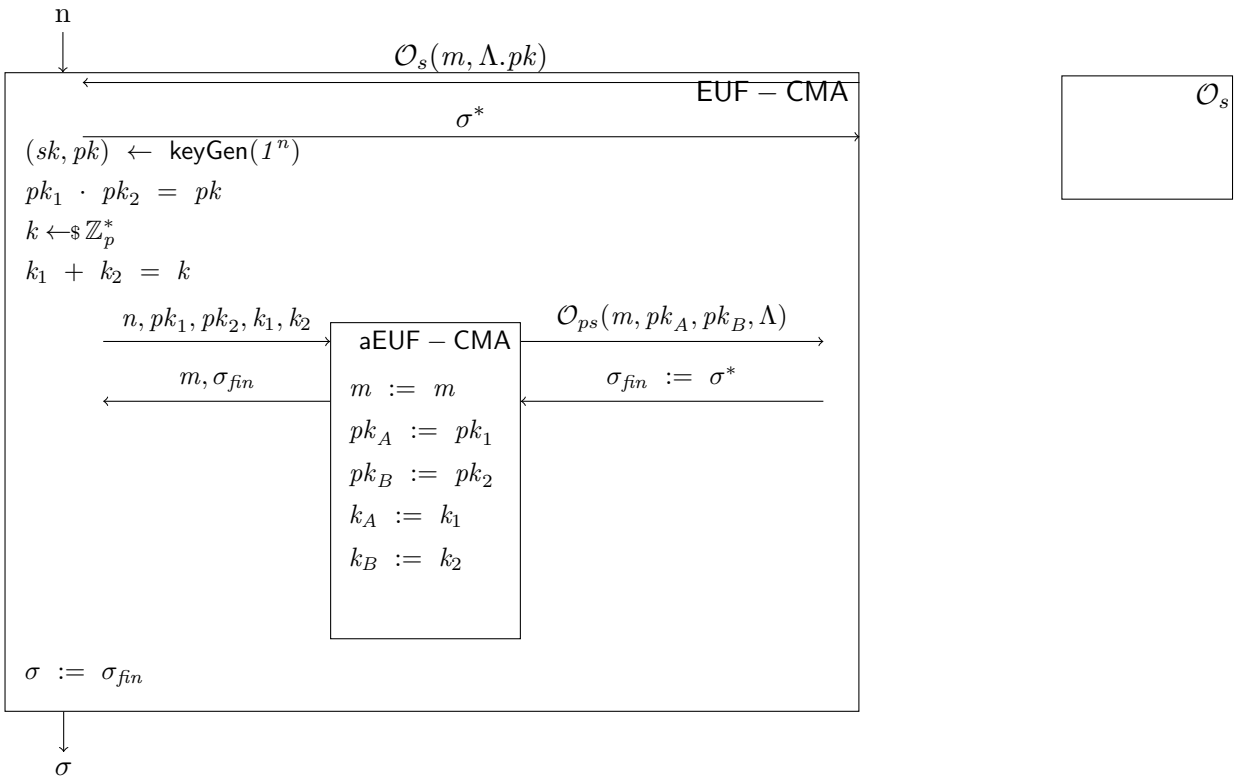

Figure 4.6: Reduction from aEUF − CMA to EUF − CMA

CHAPTER 5

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

In this section, we will define Mimblewimble transactions and security properties which must hold for such a transaction system. The formalizations will be similar to those found by Fuchsbauer et al. in [**fuchsbauer2019aggregate**]. However, 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 also discussed in [**fuchsbauer2019aggregate**] will not be topic of this formalization, as they are not relevant building blocks for the proposed Atomic Swap protocol.

However, 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. The second extended definition is called *Adapted Extended Mimblewimble Transaction Scheme* and will allow the receiver of a coin to hide a secret witness value x in his signature, in a way that the sender (or the senders) can redeem this secret after the protocol has completed.

We will proceed by providing an instantiation of the three transactions schemes which can be implemented and deployed on a Mimblewimble based Cryptocurrency such as Beam or Grin. Furthermore we provide proofs that the schemes are correct and secure with regards to the defined security properties.

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.

5.1 Definitions

As we have already discussed in section **??** 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) 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 functionality of the Ldgr procedure, instead we introduce two functions finTx and verfTx. finTx can be called by the transaction sender to finalize a pre-transaction into final valid transaction, which then could be broadcast with a node connected to the blockchain. The verfTx function is called by nodes (acting as public verifiers) on the blockchain verifying the validity of the transaction, before including them in a block.

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 tupel of procedures:

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

- $(ptx, (sk_A, k_A)) \;\leftarrow\; \mathsf{spendCoins}([\mathcal{C}_{inp}], [r], p, v, t)$: 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 coins $[\mathcal{C}_{inp}]$ which should be spent, the respective blinding factors $[r]$ to the input coins, and a value p which should be transferred to the receiver as well as a value v which is the total value stored in the input coins. Optionally a sender can pass a block height t when this transaction should become valid. It outputs a pre-transaction ptx which can be send to a receiver, as well as the senders signing key and secret nonce later used for signing (sk_A, k_A).
- $(ptx^*, (\mathcal{C}^B_{out}, r_B)) \;\leftarrow\; \mathsf{recvCoins}(ptx, p)$: The receiveCoins algorithm is a DPT routine called by the receiver and takes as input a pre-transaction ptx and a fund value p. It will construct the receivers output coin and output a modified pre-transaction ptx^* together with the new outputcoin $\mathcal{C}^B_{out}$ and its blinding factor r_B.

- $tx \leftarrow \mathsf{finTx}(ptx, sk_A, k_A)$: 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 transaction tx, which can be published to the blockchain.

- $\{1, 0\} \leftarrow \mathsf{verfTx}(tx)$: The transaction verification algorithm is a DPT function which can be called by a public verifier and takes as input a transaction tx. It outputs either 1 on verification success or 0 otherwise.

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

Definition 5.2 (Transaction Scheme Correctness)**.** For any list of input coins $[\mathcal{C}_{inp}]$ with total value v and blinding factors $[r]$ and a transaction fund value p with $p \leq v$ the following must hold:

$$\Pr\left[\begin{array}{l|l} \mathsf{verfTx}(tx) = 1 & \begin{array}{l}(ptx, (sk_A, k_A)) \leftarrow \mathsf{spendCoins}([\mathcal{C}_{inp}], [r], p, v, \perp) \\ (ptx^*, \cdot) \leftarrow \mathsf{recvCoins}(ptx, p) \\ tx \leftarrow \mathsf{finTx}(ptx^*, sk_A, k_A)\end{array}\end{array}\right] = 1.$$

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:

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

- $\langle (ptx, (sk_A, k_A)), (ptx, (sk_C, k_C)) \rangle$
  $\leftarrow \mathsf{dSpendCoins}\langle([\mathcal{C}_{inp}], [r_A], p, v, t), ([\mathcal{C}_{inp}], [r_C], p, v, t)\rangle$: The distributed coin spending algorithm takes as input a list of input coins, as well as a list of blinding factors from each Alice and Carol, and the to be transferred value p and total value of input coins v. Note that for each provided input coin $\mathcal{C}_{inp}$ the blinding factor is composed by combining the shares from Alice and Carol like $r := r_A + r_C$. Again optionally a block height t can be given to time lock the transaction.

- $\langle (ptx^*, (\mathcal{C}^{sh}_{out}, r^*_B)), (ptx^*, (\mathcal{C}^{sh}_{out}, r^*_C)) \rangle \leftarrow \mathsf{dRecvCoins}\langle (ptx, p), (ptx, p)\rangle$: The distributed coin receive procedure takes as input a pre-transaction ptx and a value p which should be transferred during the course of 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 $\mathcal{C}^{sh}_{out}$ and the respective shares of the blinding factor. Note that $\mathcal{C}^{sh}_{out}$ will only be spendable if both owners cooperate running the dSpendCoins protocol.

- $\langle tx, tx \rangle \leftarrow \mathsf{dFinTx}\langle (ptx, sk_A, k_A), (ptx, sk_C, k_C)\rangle$: 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 $\mathsf{dSpendCoins}$ algorithm). In this case we require signing information from both Alice and Carol. Again an optional locktime t can be provided.

Correctness is given very similar to the standard scheme:

Definition 5.4 (Extended Transaction Scheme Correctness)**.** For any list of input coins $[\mathcal{C}_{inp}]$ with total value v and split blinding factors $([r_A], [r_C])$ and a transaction fund value p with $p \leq v$ the following must hold:

$$\Pr\left[\begin{array}{l|l} \mathsf{verfTx}(tx) = 1 & \begin{array}{l} \langle (ptx, (sk_A, k_A)), (ptx, (sk_C, k_C))\rangle \leftarrow \\ \mathsf{dSpendCoins}\langle([\mathcal{C}_{inp}], [r_A], p, v, \bot),\ ([\mathcal{C}_{inp}], [r_C], p, v, \bot)\rangle \\ \langle (ptx^*, \cdot)(ptx^*, \cdot)\rangle \leftarrow \mathsf{dRecvCoins}\langle(ptx, p),\ (ptx, p)\rangle \\ tx \leftarrow \mathsf{dFinTx}\langle(ptx^*, sk_A, k_A), (ptx^*, sk_C, k_C)\rangle \end{array} \end{array}\right] = 1.$$

Definition 5.5 (Adapted Extended Mimblewimble Transaction Scheme)**.** The adapted 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.

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

- $(ptx^*, (\mathcal{C}^B_{out}, r_B), \tilde{\sigma_B}) \leftarrow \mathsf{aptRecvCoins}(ptx, p, x)$: The adapted 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.

- $\langle \tilde{\sigma_{AB}}, tx\rangle \leftarrow \mathsf{dAptFinTx}\langle(ptx^*, sk_A, k_A, X), (ptx^*, sk_B, k_B, \tilde{\sigma_B})\rangle$: The adapted 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 5.3 one of the two senders is also the receiver, we allowed ourselves to model this protocol as being between only two parties to simplify the formalization. In this version of the protocol only Bob will be able to finalize the transaction, which is different to finTx and dFinTx. This has the practical reason that for the atomic swap execution Bob needs to be the one in control of 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 list of input coins $[\mathcal{C}_{inp}]$ with total value v and blinding factors $[r]$ and a transaction fund value p with $p \leq v$ and any witness value x the following must hold:

$$\Pr\left[\mathsf{verfTx}(tx) = 1 \;\middle|\; \begin{array}{l}(ptx, (sk_A, k_A)) \leftarrow \mathsf{spendCoins}([\mathcal{C}_{inp}], [r], p, v, \bot) \\ (ptx^*, (\mathcal{C}^B_{out}, r_B), \tilde{\sigma_B}) \leftarrow \mathsf{aptRecvCoins}(ptx, p, x) \\ \langle \tilde{\sigma_{AC}}, tx\rangle \leftarrow \mathsf{dAptFinTx}\langle (ptx, sk_A, k_A, X), (ptx, sk_C, k_C, \tilde{\sigma_B})\rangle\end{array}\right] = 1.$$

Resistance against inflation: We first define a security property which informally states that a transactions output value can only be less or equal to the value of its input coins (if less then the miner of the transaction can extract fees). In other words a transaction in our definition shall only transfer existing but never generate any new value. We call this security property *Inflation Resistance*. In order to define this property we first have to define a cryptographic game $\mathsf{inflate}$ which takes as input a security parameter n and a value v which the Adversary $\mathcal{A}$ tries to inflate.

We define the game $\mathsf{inflate}$ as follows, whereas a challenger creates a input coin with value v given as a parameter to the game. The adversary then chooses a value v^* with $v^* > v$ and creates a new output coin $\mathcal{C}_{out}$. The adversary wins if he can construct a valid transaction spending the challengers input coin $\mathcal{C}_{inp}$ to $\mathcal{C}_{out}$ and thereby creating new value in the total of $v^* - v$.

$\mathsf{inflate}(n, v)$

1 : $r \leftarrow\!\!\$\ \mathbb{Z}^*_n$
2 : $(\mathcal{C}_{inp}, \pi) \leftarrow \mathsf{createCoin}(v, r)$
3 : $(r^*, v^*) \leftarrow \mathcal{A}(\mathcal{C}_{inp}, v)$
4 : $(\mathcal{C}_{out}, \pi) \leftarrow \mathsf{createCoin}(v^*, r^*)$
5 : $tx \leftarrow \mathcal{A}(\mathcal{C}_{inp}, v, \mathcal{C}_{out})$
6 : **return** $v^* > v \wedge \mathsf{verfTx}(tx) = 1 \wedge tx.out = [\mathcal{C}_{out}] \wedge tx.inp = [\mathcal{C}_{inp}]$

Definition 5.7 (Inflation Resistence)**.** A Mimblewimbe Transaction Scheme is called inflation resistent if for any value v in a valid range (as defined by the public parameters of the ledger), security paramter n and a PPT adversary $\mathcal{A}$ the following holds:

$$\Pr[\mathsf{inflate}(n, v) = 1] \leq \mathsf{negl}(n)$$

Resistance against Coin theft: In a Mimblewimble transaction scheme a coins ownership is given by the knowledge of its blinding factor r. To spend the coin the sender would also have to know the coins value v in addition to the blinding factor, however as the possible values for v in practice is restriced by the blockchains public parameters, it is trivial to guess. Therefore we assume that knowledge of the blinding factor r alone implies ownership of the coin.

We define a game $\mathsf{stealCoin}$ which takes as input a security parameter n and a coin $\mathcal{C}_{inp}$. A PPT adversary $\mathcal{A}$ is given the challenge to spend the input coin given in the parameter. He wins if he can construct a valid transaction tx together with the help of a challenger spending the input coin $\mathcal{C}_{inp}$ and thereby transferring its value.

$\mathsf{stealCoin}(n, \mathcal{C}_{inp})$

1 : $(ptx, sk, p) \leftarrow \mathcal{A}(\mathcal{C}_{inp})$
2 : $ptx^* \leftarrow \mathsf{recvCoins}(ptx, p)$
3 : $tx \leftarrow \mathcal{A}(ptx^*)$
4 : **return** $\mathsf{verfTx}(tx) = 1 \wedge \mathcal{C}_{inp} \in tx.inp$

Definition 5.8 (Theft-resistance)**.** A Mimblewimble Transaction Scheme is called theft-resistent if for any input coin $\mathcal{C}_{inp}$, security parameter n and PPT adversary $\mathcal{A}$ the following holds:

$$\Pr[\mathsf{stealCoin}(n, \mathcal{C}_{inp}) = 1] \leq \mathsf{negl}(n)$$

Transaction indistinguishability: The third security property for a Mimblewimble Transaction System is related to transaction amounts. Due to the use of homomorphic commitments instead of plaintext values, an adversary should be unable to extract plaintext amounts from the final transaction. To formalize the property we define the game $\mathsf{TX} - \mathsf{IND}$: A PPT adversary $\mathcal{A}$ gets access to a transaction oracle $\mathcal{O}_{tx}$ to which he can send two sets of values, one being the input value, the other the output value. We require all values to be in a valid range as defined by the game parameter v_{max}, furthermore we require the transaction to be balanced, so the output values can not be higher then input values. The oracle will select one of the two sets depending on the bit b chosen by the challenger and construct a transaction spending a coin with value v_b to a output coin of value p_b (as well as a change coin with the remainder). From the returned transaction the adversary has to guess the value of b. If the adversary manages to choose the correct value of b he or she has won the game.

$\mathsf{TX}-\mathsf{IND}(n, v_{max})$

1 : $b \leftarrow\$ \{0,1\}$
2 : $b^* \leftarrow \mathcal{A}^{\mathcal{O}_{tx}((\cdot,\cdot),(\cdot,\cdot))}$
3 : **return** $b \stackrel{?}{=} b^*$

$\mathcal{O}_{tx}((v_1, p_1), (v_2, p_2))$

1 : **if** $(v_1, p_1, v_2, p_2) \notin [0, v_{max}]*)$
2 : **return** $\perp$
3 : **if** $p_1 > v_1 \vee p_2 > v_2$
4 : **return** $\perp$
5 : $r \leftarrow\$ \mathbb{Z}^*_{1^n}$
6 : $\mathcal{C}_{inp} \leftarrow \mathsf{createCoin}(v_b, r)$
7 : $(ptx, sk, k) \leftarrow \mathsf{spendCoins}([\mathcal{C}_{inp}], [r], p_b, v_b, \perp)$
8 : $ptx^* \leftarrow \mathsf{recvCoins}(ptx, p_b)$
9 : **return** $\mathsf{finTx}(ptx^*, sk, k)$

Definition 5.9 (Transaction indistinguishability)**.** We say Transaction Indistinguishability holds for a Mimblewimble Transaction System MW if for a PPT adversary $\mathcal{A}$ with access to the transaction oracle $\mathcal{O}_{tx}$ a security parameter n and any max value v_{max} the following must hold:

$$\Pr\left[\mathsf{TX}-\mathsf{IND}(n, v_{max}) \stackrel{?}{=} 1\right] \leq 0.5 + \mathsf{negl}(n)$$

5.2 Mimblewimble 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 Mimblewimble protocol such as Beam and Grin.

5.2.1 Mimblewimble Transaction Scheme

First we provide an instantiation of the simplest form of a transaction in which a sender wants to transfer some value p to a receiver. For the execution of the protocol we assume to have access to a homomorphic commitment scheme such as Pedersen Commitment COM as defined in definition 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.

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

$\mathsf{createCoin}(v, r)$	$\mathsf{createTx}(m, [\mathcal{C}_{inp}], [\mathcal{C}_{out}], [\pi], \Lambda, [C], \sigma, t)$
1 : $\mathcal{C} \leftarrow \mathsf{commit}(v, r)$	1 : **return** (
2 : $\pi \leftarrow \mathsf{ranPrf}(\mathcal{C}, v, r)$	2 : $m := m$,
3 : **return** $(\mathcal{C}, \pi)$	3 : $inp := [\mathcal{C}_{inp}]$,
	4 : $out := [\mathcal{C}_{out}]$,
	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 Transction Scheme using the auxiliary functions provided before.

In the spendCoins function the sender creates his change output coin, which is the difference between the value stored in his input coins and the value which should be transferred to a receiver. He sets up the signature context with his parameters and gets a pre-transaction *ptx*, 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 recievers turn, after the signature context has been completed. In Fuchsbauer et al. 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 a 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 relying upon a two-party signature scheme instead we are closer to what is implemented in practice[1] . Furthermore by starting the signing process at the receivers turn we avoid a potential problem: If an adversary learns the already signed pre-transaction, 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 outputcoin $\mathcal{C}^B_{out}$, add his parameters to the signature context and then create his partial signature $\tilde{\sigma_B}$. The function returns an updated version of the pre-transaction *ptx* which can be sent back to the sender.

Now in finTx the original sender will validate the updated pre-transcation *ptx* sent to him by the receiver. If he finds it as valid, he will only now create his partial signature

[1] **https://medium.com/@brandonarvanaghi/grin-transactions-explained-step-by-step-fdceb905a853**.

and finally finalize the two partial signatures in the final composite one, with which he can then build the final transaction.

In verfTx a public verifier will verify the rangeproofs for the transactions output coins. If they are found valid she will compute the so-called Excess value $\mathcal{E}$ from the difference between output and input coins and use it as the public key for validating the transaction signature. If the signature is now also found to be correct the verifier can deduce that the transaction is well-formed and valid.

5.2.2 Extended Mimblewimble Transaction Scheme

So make the formalization of algorithms easier we define a protocol dSign which is a protocol between two-parties running the partial signature creation outlined in section 4.2. Note that we assume that the secret keys as well as nonces are already given as a paramter (which is the case during the transaction protocol) therefore we don't need an additional call to dKeyGen.

dSign$\langle(m, sk_A, sk_B), (m, k_A, k_B)\rangle$

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

Figure 5.2 shows an instantiation of the dSpendCoins function of the Extended Mimblewimble Transaction Scheme. We have one array of input coins which keys are shared

$\mathsf{spendCoins}([\mathcal{C}_{inp}], [r_A], p, v, t)$

1 : **if** $p > v$
2 : **return** $\perp$
3 : $m := \{0,1\}^*$
4 : $(r_A^*, k_A) \leftarrow\$ \mathbb{Z}_p^*$
5 : $(\mathcal{C}_{out}^A, \pi_A) \leftarrow \mathsf{createCoin}(v - p, r_A^*)$
6 : $sk_A := r_A^* - \sum[r_A]$
7 : $\Lambda := \{pk := 1_p, R := 1_p\}$
8 : $\Lambda \leftarrow \mathsf{setupCtx}(\Lambda, g^{sk_A}, g^{k_A})$
9 : $ptx \leftarrow \mathsf{createTx}(m, [\mathcal{C}_{inp}], [\mathcal{C}_{out}^A], [\pi_A], \Lambda, [g^{sk_A}], \emptyset, t)$
10 : **return** $(ptx, (sk_A, k_A))$

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

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

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

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

$\mathsf{verfTx}(tx)$

1 : $(m, inp, out, \Pi, \Lambda, com, \sigma, t) \leftarrow tx$
2 : **foreach** out as $(i => \mathcal{C}_{out})$
3 : **if** $\mathsf{vrfRanPrf}(\Pi[i], \mathcal{C}_{out}[i]) = 0$
4 : **return** 0
5 : $pk := \sum out - \sum inp$
6 : **return** $\mathsf{verf}(\sigma, m, pk)$
7 :

Figure 5.1: Instantiation of Mimblewimble Transaction Scheme.

between two parties Alice and Carol, which should be spent. We use Carol here to not confuse this party with the receiver, which we previously called Bob. Although Carol and B could be the same person, they not necessarily have to be.

The protocol starts with both Alice and Carol creating her spend 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). Note that when using this protocol to spend coins that finTx also turns into a protocol, using dSign instead of signPrt for signature generation. (Apart from that finTx remains unchanged)

$\mathsf{dSpendCoins}\langle([\mathcal{C}_{inp}], [r_A], p, v_A, t),\ ([\mathcal{C}_{inp}], [r_C], p, v_A, t)\rangle$

1 :	*Alice*		*Carol*
2 :	**if** $p > v$		**if** $p > v$
3 :	**return** $\perp$		**return** $\perp$
4 :	$m := \{0,1\}^*$		
5 :	$(r_A^*, k_A) \leftarrow\$\ \mathbb{Z}_p^*$		$(r_C^*, k_C) \leftarrow\$\ \mathbb{Z}_p^*$
6 :	$(\mathcal{C}_{out}^A, \pi_A) \leftarrow \mathsf{createCoin}(v_A, r_A^*)$		$(\mathcal{C}_{out}^C, \pi_C) \leftarrow \mathsf{createCoin}(v_C, r_C^*)$
7 :	$sk_A := r_A^* - \sum[r_A]$		$sk_C := r_C^* - \sum[r_C]$
8 :	$\Lambda := \{pk := 1_p, R := 1_p\}$		
9 :	$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda,\ g^{sk_A},\ g^{k_A})$		
10 :	$ptx \leftarrow$		
11 :	$\mathsf{createTx}(m, [\mathcal{C}_{inp}], [\mathcal{C}_{out}^A], [\pi_A], \Lambda, [g^{k_A}], \emptyset, t)$		
12 :		$\xrightarrow{\quad ptx \quad}$	
13 :			$(m, inp, out, \Pi, \Lambda, com, t^*) \leftarrow ptx$
14 :			**if** $\mathsf{vrfRanPrf}(\Pi[0], out[0]) = 0 \vee t \neq t^*$
15 :			**return** $\perp$
16 :			$\Lambda \leftarrow \mathsf{setupCtx}(\Lambda,\ g^{sk_C},\ g^{k_C})$
17 :			$ptx \leftarrow \mathsf{createTx}(m, inp, out \mid\mid \mathcal{C}_{out}^C, \pi \mid\mid \pi_C, \Lambda, com \mid\mid g^{k_C}, \emptyset, t)$
18 :		$\xleftarrow{\quad ptx \quad}$	
19 :	**if** $\mathsf{vrfRanPrf}(ptx.\Pi[1], ptx.out[1]) = 0$		
20 :	**return** $\perp$		
21 :	**return** $(ptx, (sk_A, k_A))$		**return** $(ptx, (sk_C, k_C))$
22 :			

Figure 5.2: Extended Mimblewimble Transaction Scheme - dSpendCoins

Figure 5.3 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 verifing 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 engange 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.

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

	Bob		Carol
1 :	Bob		$Carol$
2 :	$(m, inp, out, \Pi, \Lambda, com, \tilde{\sigma_B}, t) \leftarrow ptx$		$(m, inp, out, \Pi, \Lambda, com, \tilde{\sigma_B}, t) \leftarrow ptx$
3 :	**foreach** out *as* $(i => \mathcal{C}_{out})$		**foreach** out *as* $(i => \mathcal{C}_{out})$
4 :	**if** $\mathsf{vrfRanPrf}(\Pi[i], \mathcal{C}_{out}[i]) = 0$		**if** $\mathsf{vrfRanPrf}(\Pi[i], \mathcal{C}_{out}[i]) = 0$
5 :	**return** $\perp$		**return** $\perp$
6 :	$(r_B^*, k_B) \leftarrow\$ \mathbb{Z}_p^*$		
7 :	$(\mathcal{C}_{out}^{sh}, \cdot) \leftarrow \mathsf{createCoin}(p, r_B^*)$		
8 :	$sk_B := r_B^*$		
9 :		$\mathcal{C}_{out}^{sh} \longrightarrow$	
10 :			$(r_C^*, k_C) \leftarrow\$ \mathbb{Z}_p^*$
11 :			$sk_C := r_C^*$
12 :			$\mathcal{C}_{out}^{sh\,\prime} := \mathcal{C}_{out}^{sh} \cdot g^{sk_C}$
13 :		$\mathcal{C}_{out}^{sh\,\prime}, g^{sk_C} \longleftarrow$	
14 :	**if** $\mathcal{C}_{out}^{sh\,\prime} \neq \mathcal{C}_{out}^{sh} \cdot g^{sk_C}$		
15 :	**return** $\perp$		
16 :		$(\sigma_{\tilde{BC}}, pk_{BC}, \Lambda^*) \leftarrow \mathsf{dSign}\langle(m, sk_B, sk_B), (m, k_A, k_B)\rangle$	
17 :		$\Lambda' \leftarrow \mathsf{setupCtx}(\Lambda,\ \Lambda^*.pk,\ \Lambda^*.k)$	
18 :		$\pi_{BC} \leftarrow \mathsf{dRanPrf}\langle(\mathcal{C}_{out}^{sh\,\prime}, p, sk_A), (\mathcal{C}_{out}^{sh\,\prime}, p, sk_B)$	
19 :		$ptx \leftarrow \mathsf{createTx}(m, inp, out \mid\mid \mathcal{C}_{out}^{sh\,\prime}, \Pi \mid\mid \pi_{BC}, \Lambda', com \mid\mid pk_{BC}, \sigma_{\tilde{BC}}, t)$	
20 :	**return** ptx		**return** ptx

Figure 5.3: Extended Mimblewimble Transaction Scheme - dRecvCoins

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

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

Figure 5.4: Extended Mimblewimble Transaction Scheme - dFinTx

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

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

Figure 5.5: Adapted Extended Mimblewimble Transaction Scheme - aptRecvCoins.

5.2.3 Adapted Extended Mimblewimble Transaction Scheme

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

In figure 5.6 we show the updated distributed version of the transaction finalization protocol. Again Alice verifies the pre-transaction ptx received by Bob and then 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.

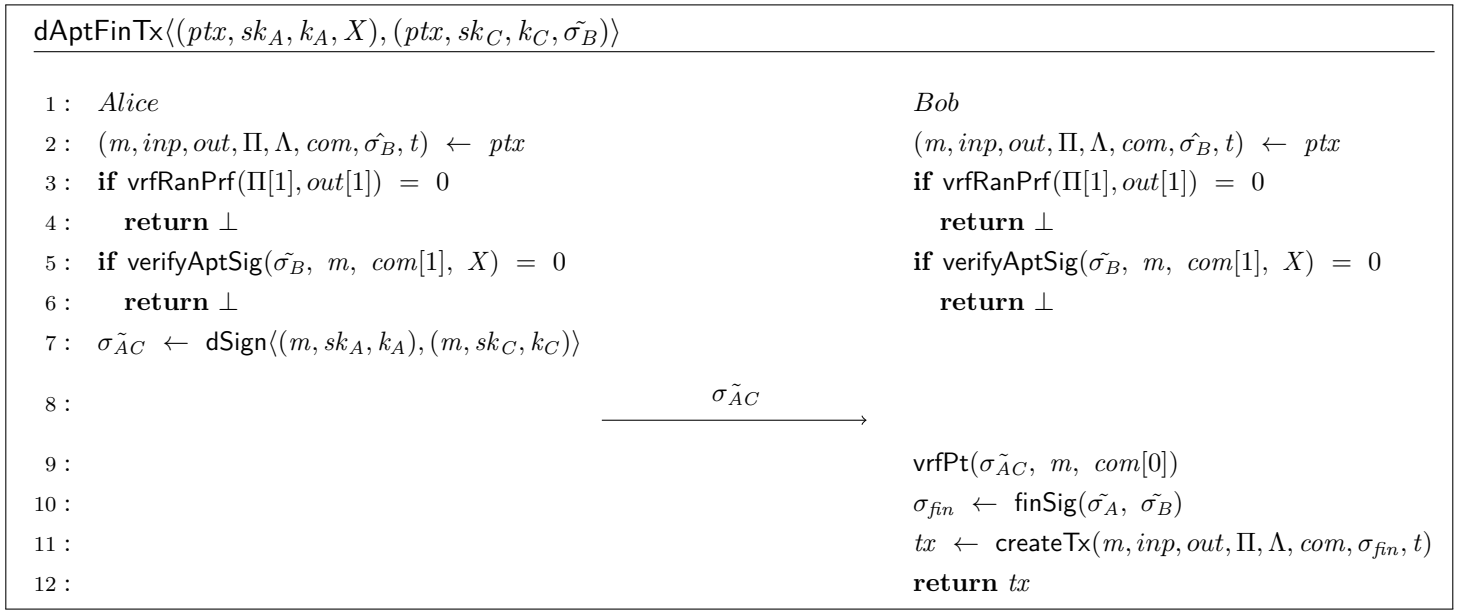

Figure 5.6: Adapted Extended Mimblewimble Transaction Scheme - dAptFinTx.

5.3 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 ($\mathsf{lockBtcScript}, \mathsf{verifyLock}$).

- $(spk) \leftarrow \mathsf{lockBtcScript}(pk_A, pk_B, X, t)$: The locking script function lets Bob construct a Bitcoin script only spendable by Alice if she receives the discrete logarithm x of X with $X = g^x$. Additionally the function requires Bobs public key pk_B and a timelock t (given as a block number) as input which allows Bob to reclaim his funds after some time if the atomic swap was not completed successfully. The function will create and return a Bitcon script spk to which Bob can send funds using a P2SH transaction. 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.

- $\{1, 0\} \leftarrow \mathsf{verifyLock}(pk_A, pk_B, X, v, t)$: 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 $\mathsf{lockBtcScript}$ and then check the Bitcoin blockchain if the value locked under the script equals v. Upon successful verification the function returns 1, otherwise 0.

5.3.1 Setup phase

We assume Alice owns Mimblewimble 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 Mimblewimble coin Alice will swap to Bob.

- t_{btc} The locktime as a blockheight for the Bitcoin side.
- t_{mw} The locktime as a blockheight for the Mimblewimble side.

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 [**tairi2019a2l**], they propose to use a fixed exchange rate for each day and to use a real world timeout like one day and then calculate the specific block numbers by taking the average block time of the blockchain into account. In our setup we can also fix the exchange rate at the beginning of the protocol, which stays unchanged during protocol execution. If the exchange rate fluctuates and one party is negatively impacted he or she could still decide to stop being cooperative which means the coins would be returned to the original owners after the timeout.

We formalize the protocol setupSwp in figure 5.7. The protocol takes as input the shared swap state $\mathcal{A}$ from both parties. From Alice her Mimblewimble 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 Mimblewimble coins into an output shared by both parties. Immediately after, both parties collaborate again to spend this shared coin back to Alice with a timelock of t_{mw}. It is immanent that Alice does not publish the first transaction (A -> AB) before the timelocked refund transaction (AB -> A) was signed, otherwise her funds are locked in the shared output without the possibility of refund. The setup protocol concludes with the funds locked up in both chains and ready to be swapped.

[1] $\mathsf{spendCoins}([\mathcal{C}_{inp}], [r_A], a_{mw}, v_{mw}, \perp)$
[2] $\mathsf{dRecvCoins}\langle(ptx, ptx),\ (ptx, ptx)\rangle$
[3] $\mathsf{dSpendCoins}\langle([\mathcal{C}^{sh}_{out}], [r^*_A], a_{mw}, a_{mw}, t_{mw}),\ ([\mathcal{C}^{sh}_{out}], [r^*_B], a_{mw}, a_{mw}, t_{mw})\rangle$
[4] $\mathsf{dFinTx}\langle(ptx^*_2, sk_A, k_A), (ptx^*_2, sk_B, k_B)\rangle$

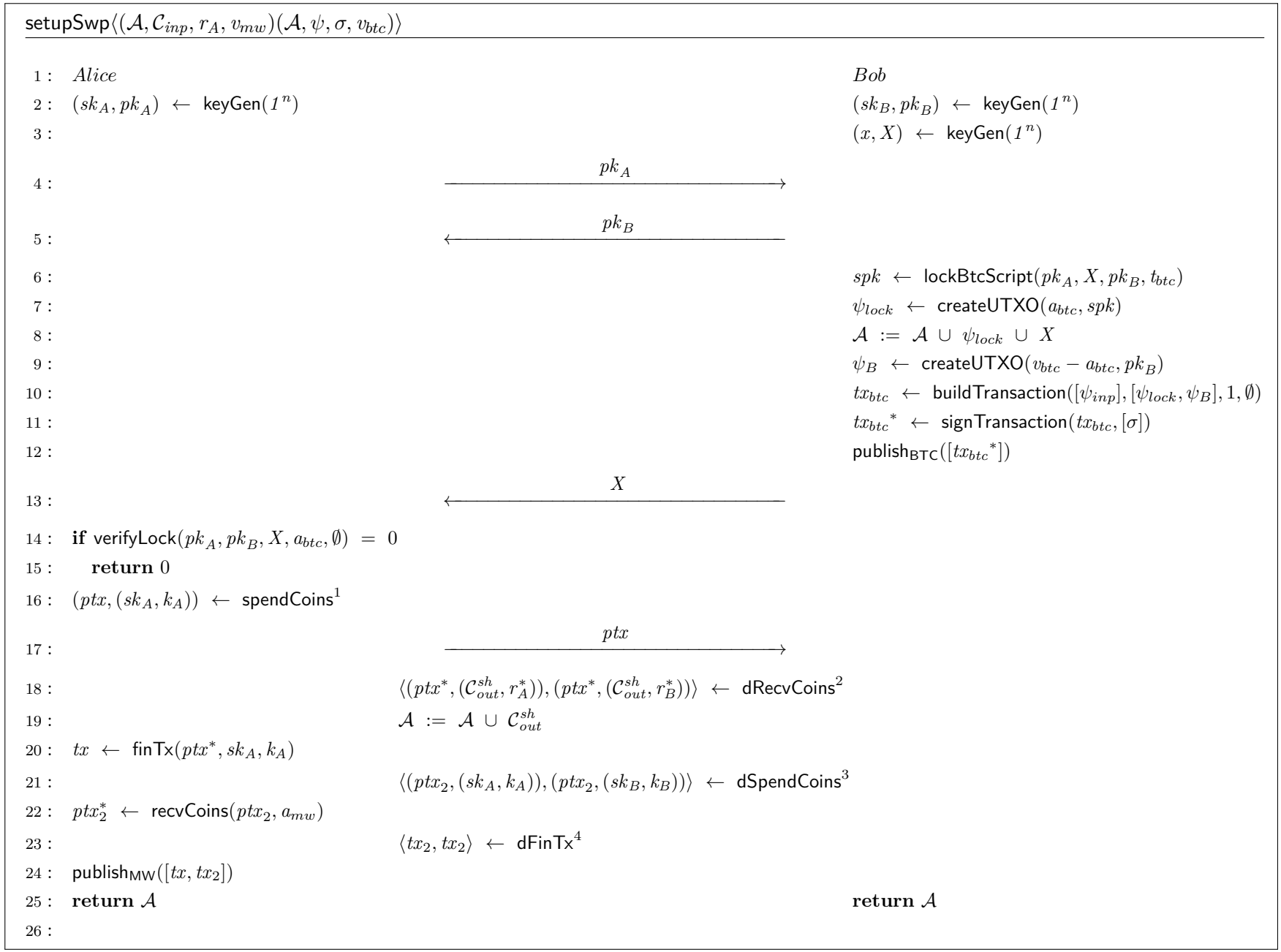

Figure 5.7: Atomic Swap - setupSwp.

5.3.2 Execution Phase

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

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

This function will verify that there is sufficient time to execute the atomic swap protcol. As input it takes a chain 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 Mimblewimble 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 (Mimblewimble side) needs to provide her part of the blinding factor r_A^* of the shared Mimblewimble 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 Mimblewimble 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 Mimblewimble 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 Mimblewimble 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 Mimblewimble 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 $\tilde{\sigma_{AB}}$.

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] $\mathsf{dSpendCoins}\langle([\mathcal{C}_{out}^{sh}],[r_A^*],a_{mw},a_{mw},\perp),\ ([\mathcal{C}_{out}^{sh}],[r_B^*],a_{mw},a_{mw},\perp)\rangle$
[6] $\mathsf{dAptFinTx}\langle(ptx^*,sk_A,k_A,X),(ptx^*,sk_B,k_B,\tilde{\sigma_B})\rangle$

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

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

Figure 5.8: Atomic Swap - setupSwp.

5.3.3 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 the blocknumber hits t_{btc} the transaction will become valid and get mined. Again we stress the importance of using appropriate timeouts, if a timeout is too short the swap might get cancelled if there are some delays, if the timeout is too long the funds might be locked for an unnessary amount of time.

On the Mimblewimble side the second transaction spending the shared output back to Alice guarantees that her funds are returned to her after the timeout t_{mw} hits. 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

List of Figures

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

List of Tables

List of Algorithms