How to Mimblewimble

Jakob Abfalter

September 26, 2019

1 Introduction

This reading explains how Mimblewimble transactions look like, how they are generated, and how the ledger is constructed. It is a summary of parts of a paper by Fuchsbauer et al. [1].

2 Transactions

2.1 Transaction structure

- A coin in Mimblewimble is of the form $C = vH + rG, \pi$ so C is a (Pedersen) commitment to the value v, with randomness r and generators H and G. π is a range proof.
- In contrast to Bitcoin, there is no notion of addresses or scripts. Ownership of a coin is equivalent to the knowledge of its opening. The randomness r of the commitment now acts as the secret key for the coin.
- A transaction consists of $C = (C_1, ..., C_n)$ input coins and $C' = (C'_1, ..., C'_n)$ output coins.

A transaction is valid iff $\sum v'_i - \sum v_i = 0$, so we get the equation

$$\sum C' - \sum C = \sum v'_i * H + r'_i * G - \sum v_i * H + r_i * G$$

If we assume the transaction is valid we are left with the excess

$$E := (\sum r'_i - \sum r_i) * G$$

Knowledge of the opening of all coins and validity of the transaction implies knowledge of E.
However, directly revealing the opening of E would leak too much information. (An adversary knowing the openings for input coins and all but one output coin, could calculate that opening knowing the Excess)
Therefore we prove the knowledge of E by providing a signature which validates

for E as a public key.
Coinbase transactions additionally have a supply added to the excess

$$E := (\sum r'_i - \sum r_i) * G - s * H$$

Finally a Mimblewimble transactions consists of:

$$tx = (s, C, C', K) \quad with \quad K = (\pi, E, \sigma)$$

where s is the supply, C are input coins, C' output coins, K the so called Kernel consisting of π which are range proofs for the output coins, E which is a list of excess values and σ which are signatures.

2.2 Transaction merging

An essential property of the Mimblewimble protocol is that two transactions can easily be merged into one. Assume we have two transactions:

$$tx_0 = (s_0, C_0, C'_0, (\pi_0, E_0, \sigma_0))$$

$$tx_1 = (s_1, C_1, C'_1, (\pi_1, E_1, \sigma_1))$$

Then the merged transaction simply is

$$tx_m = (s_0 + s_1, C_0 \mid\mid C_1, C'_0 \mid\mid C'_1, (\pi_0 \mid\mid \pi_1, (E_0, E_1), (\sigma_0, \sigma_1)))$$

Note that if the signature scheme supports merging of signatures such as the BLS signature scheme, the two signature can be replaced by one reducing the size of the transaction.
The merged transaction is valid iff the transactions it is composed of are valid.

2.2.1 Subset Problem

There is a subtle problem with the way transactions are merged in Mimblewimble. It is possible to reconstruct the constituent transactions in a merged transaction, which is problematic for privacy. Given a set of inputs, outputs and kernels a subset of these will recombine to reconstruct a valid transaction. Given the two transactions:

$$tx_0 = (s_0, (C_1, C_2), (C'_0), (\pi_0, E_0, \sigma_0))$$

$$tx_1 = (s_1, (C_3), (C'_1), (\pi_1, E_1, \sigma_1))$$

As described above we can aggregate the transaction to

$$tx_3 = (s_0 + s_1, (C_1, C_2, C_3), (C'_0, C'_1), (\pi_0 \mid\mid \pi_1, (E_0, E_1), (\sigma_0, \sigma_1)))$$

Upon merging a transaction, the excess values are not combined; otherwise this would invalidate the signatures. Therefore it is trivial to try all possible permutations to recover one of the transactions where it sums successfully to one of the excess values. And once we found this transaction, everything remaining will be the second transaction. This problem is mitigated (in Grin) by including a Kernel Offset with every transaction kernel. This value is a blinding factor that needs to be added back to the kernel excess to verify the sum of the commitments to zero, such that

$$sum(C') - sum(C) = E + offset$$

Every time two transactions are merged, the offsets are combined into one offset, making it impossible to decompose it to the original offsets used in the separate transactions and therefore impossible to recover the original transactions.

2.3 What a miner verifies

After explaining the structure of a Mimblewimble transaction, it is of interest to recap what a mining node needs to verify to conclude that a transaction is indeed valid and can be included in a block, especially in the case of a merged transaction.

- Transaction Balancedness: To verify that a transaction is balanced the miner will need to make sure that the output values subtracted from the input values equal 0. (Not taking transaction fees into account, but those in practice will be explicit) Given our explanation of the excess values and excess offset this property will be fulfilled iff:

  $$\sum C' - \sum C = \sum E + offset$$

  Since the excess offsets are summed into a single value when a transaction is merged, it is impossible to reconstruct original transactions, even though there is one excess value for every historic transaction.

- Rangeproof Correctness: To verify that all of the newly created output coins are in a specific valid range, the Miner verifies the Rangeproof for every output coin.

- Ownership: For every new excess value the miner will validate the signature using the Excess as public key. A valid signature implies ownership of the spent input coins.

2.4 Cut through

Assume C appears as an output in tx_0 and as an input, in tx_1, then we can erase C from the input and output list and the transaction will still be valid. Therefore every time a transaction spends an output, it is virtually forgotten, improving privacy and yielding in space savings.

2.5 Ledger

The Mimblewimble ledger itself is a transaction of the discussed form. Initially, the ledger starts empty, and transactions are added and aggregated recursively. Trivially only transactions which input coins are contained in the output coins of the ledger are valid.

- The supply of the ledger is the sum of the supplies of all transactions added so far. Therefore we can easily read the total circulating supply from the ledger.
- Due to cut through the input coin list of the ledger is always empty, and the output list is what is called UTXO set in Bitcoin.

3 Transaction creation protocol

3.1 Original protocol

1. Sender selects input coins C of total value $v \geq p$ (p is the value he wants to transfer to the receiver), creates change coins C' of total value $v - p$ and then sends C, C', the range proofs for C', plus the opening $(-p, k)$ of $\sum C' - \sum C$ to the receiver.
2. The receiver creates additional output coins C'' plus range proof of total value p with keys (k_i'') and computes a signature with secret key $k + \sum k_i''$ and finalizes the transaction as
$$tx = (0, C, C' \mid\mid C'', (\pi, E = \sum C' + \sum C'' - \sum C, \sigma))$$
and publishes it to the ledger.

Figure 1 shows the transaction creation flow of the original protocol. This protocol, however, turned out to be vulnerable. The receiver can spend the change coin C' by reverting the transaction, as he can compute valid range proofs and excess signatures. Doing this would give the sender his coin back, however as the sender might not have the keys for that coin anymore, the coins could be lost.
In detail, this reverting transaction would look like this:

$$tx_{rv} = (0, (C', C''), C, (\pi_{rv}, E_{rv}, \sigma_{rv}))$$

So the newly created output coins are spent and sent back to the original input coin(s). E_{rv} in this case is defined as follows:

$$E_{rv} = \sum C' + \sum C'' - \sum C$$

The key k originally sent by the sender to the receiver is a valid key for the excess value:

$$E = \sum C - \sum C'$$

If we take the negative of this key we could sign:

$$-k = \sum C' - \sum C$$

Now all we have to do is add the keys k''_i (created and known by the receiver) and get:

$$-k + \sum k''_i = \sum C' + \sum C'' - \sum C$$

which is a valid key for E_{rv}
Furthermore obtaining a valid rangeproof for this coin is trivial, as it once was a valid output the ledger will contain a valid proof for this coin already. This problem was solved (in Grin) by making the signing process a two-party protocol. Now signature computation does not require the sender to send over key k to the receiver. Figure 2 shows the current transaction creation flow in the Grind Cryptocurrency.

3.2 Salvaged protocol

The authors of the paper [1] propose a slightly modified version of the protocol not vulnerable to this problem.

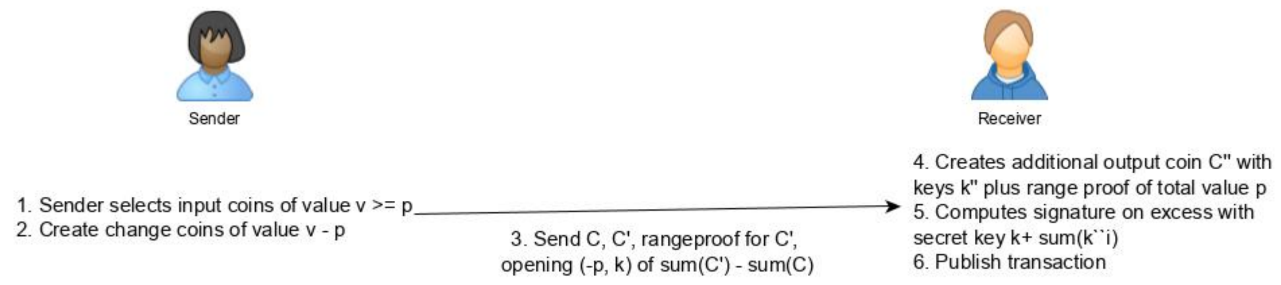
1. The sender constructs a full-fledged transaction spending his input coin C, creating change coins C', as well as a special output coin $C = pH + kG$ and sends the tx and the opening (p, k) of the special coin to the receiver. (Note that, unlike in the previous case, k is now independent of the keys of the coins C and C'.)

2. The receiver now creates a second transaction tx' spending the special coin, creating its output coins C'' and aggregates tx and tx'. The receiver then publishes the merged transaction to the ledger.

The only drawback of this approach is that we have two transaction kernels instead of just one, making the transaction slightly bigger. Figure 3 shows the salvaged transaction creation flow.

References

[1] Georg Fuchsbauer, Michele Orrù, and Yannick Seurin. Aggregate cash systems: a cryptographic investigation of mimblewimble. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 657–689. Springer, 2019.

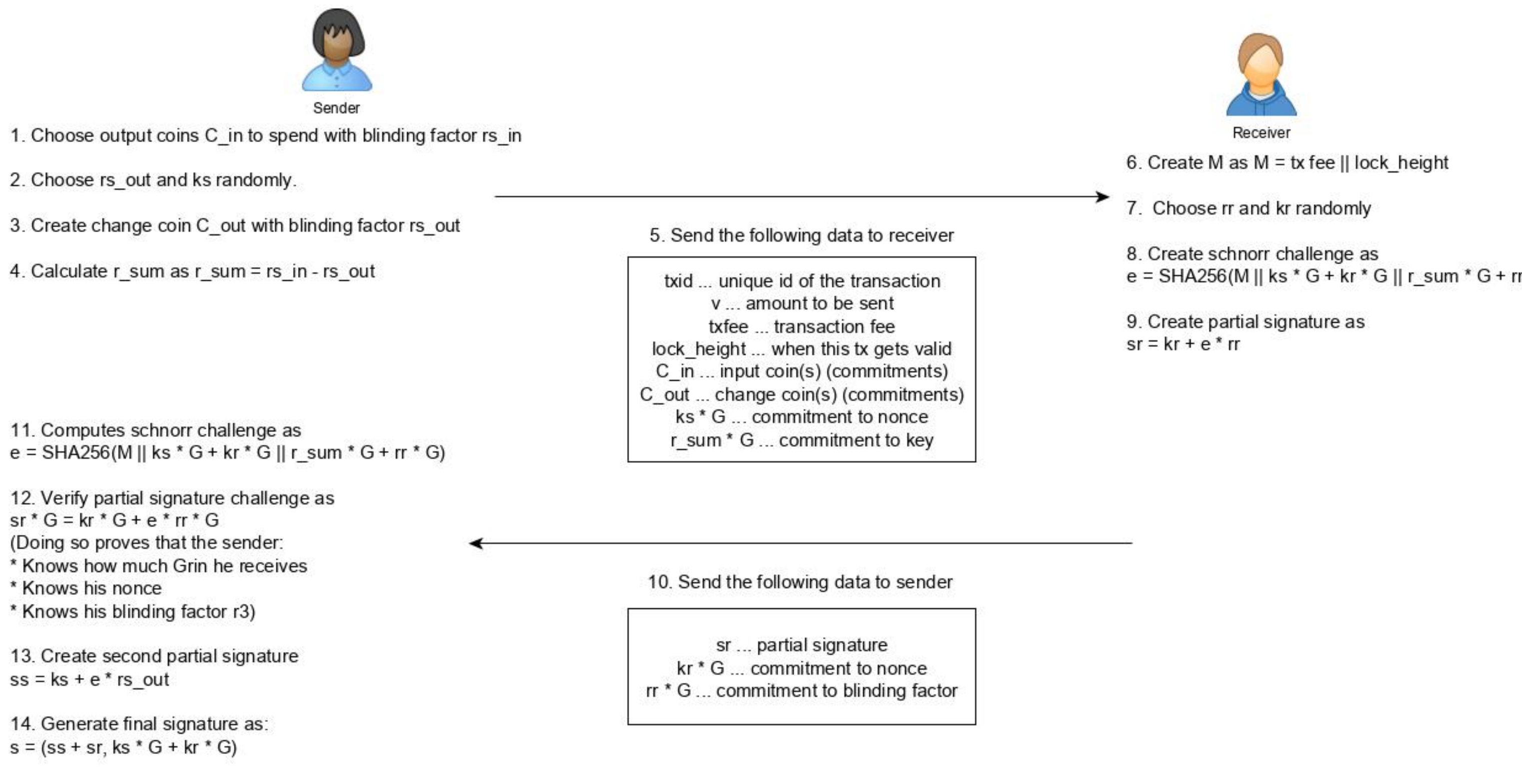


1. Sender selects input coins of value v >= p
2. Create change coins of value v - p

3. Send C, C', rangeproof for C', opening (-p, k) of sum(C') - sum(C)

4. Creates additional output coin C'' with keys k'' plus range proof of total value p
5. Computes signature on excess with secret key k+ sum(k``i)
6. Publish transaction

Figure 1: Original transaction flow



1. Choose output coins C_in to spend with blinding factor rs_in

2. Choose rs_out and ks randomly.

3. Create change coin C_out with blinding factor rs_out

4. Calculate r_sum as r_sum = rs_in - rs_out

5. Send the following data to receiver

txid ... unique id of the transaction
v ... amount to be sent
txfee ... transaction fee
lock_height ... when this tx gets valid
C_in ... input coin(s) (commitments)
C_out ... change coin(s) (commitments)
ks * G ... commitment to nonce
r_sum * G ... commitment to key

6. Create M as M = tx fee || lock_height

7. Choose rr and kr randomly

8. Create schnorr challenge as
e = SHA256(M || ks * G + kr * G || r_sum * G + r

9. Create partial signature as
sr = kr + e * rr

10. Send the following data to sender

sr ... partial signature
kr * G ... commitment to nonce
rr * G ... commitment to blinding factor

11. Computes schnorr challenge as
e = SHA256(M || ks * G + kr * G || r_sum * G + rr * G)

12. Verify partial signature challenge as
sr * G = kr * G + e * rr * G
(Doing so proves that the sender:
* Knows how much Grin he receives
* Knows his nonce
* Knows his blinding factor r3)

13. Create second partial signature
ss = ks + e * rs_out

14. Generate final signature as:
s = (ss + sr, ks * G + kr * G)

Figure 2: Grin transaction flow

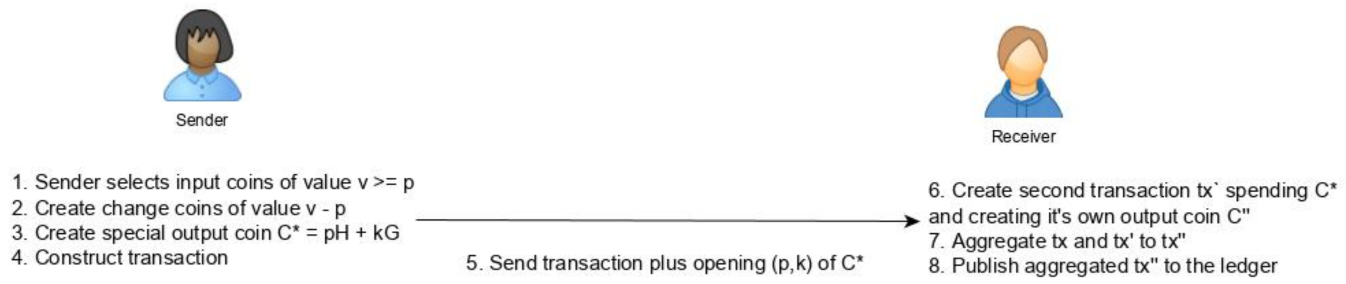

Sender
Receiver
1. Sender selects input coins of value v >= p
2. Create change coins of value v - p
3. Create special output coin C* = pH + kG
4. Construct transaction
5. Send transaction plus opening (p,k) of C*
6. Create second transaction tx` spending C* and creating it's own output coin C''
7. Aggregate tx and tx' to tx''
8. Publish aggregated tx'' to the ledger

Figure 3: Salvaged transaction flow