Introduction to Mimblewimble and Grin

Read this in other languages: [简体中文](), [Español](), [Nederlands](), [Русский](), [日本語](), [Deutsch](), [Português](), [Korean]()

Mimblewimble is a blockchain format and protocol that provides extremely good scalability, privacy and fungibility by relying on strong cryptographic primitives. It addresses gaps existing in almost all current blockchain implementations.

Grin is an open source software project that implements a Mimblewimble blockchain and fills the gaps required for a full blockchain and cryptocurrency deployment.

The main goal and characteristics of the Grin project are:

- Privacy by default. This enables complete fungibility without precluding the ability to selectively disclose information as needed.
- Scales mostly with the number of users and minimally with the number of transactions (<100 byte `kernel`), resulting in a large space saving compared to other blockchains.
- Strong and proven cryptography. Mimblewimble only relies on Elliptic Curve Cryptography which has been tried and tested for decades.
- Design simplicity that makes it easy to audit and maintain over time.
- Community driven, encouraging mining decentralization.

A detailed post on the step-by-step of how Grin transactions work (with graphics) can be found [in this Medium post]().

Tongue-Tying for Everyone

This document is targeted at readers with a good understanding of blockchains and basic cryptography. With that in mind, we attempt to explain the technical buildup of Mimblewimble and how it's applied in Grin. We hope this document is understandable to most technically-minded readers. Our objective is to encourage you to get interested in Grin and contribute in any way possible.

To achieve this objective, we will introduce the main concepts required for a good understanding of Grin as a Mimblewimble implementation. We will start with a brief description of some relevant properties of Elliptic Curve Cryptography (ECC) to lay the foundation on which Grin is based and then describe all the key elements of a Mimblewimble blockchain's transactions and blocks.

Tiny Bits of Elliptic Curves

We start with a brief primer on Elliptic Curve Cryptography, reviewing just the properties necessary to understand how Mimblewimble works and without delving too much into the intricacies of ECC. For readers who would want to dive deeper into those assumptions, there are other opportunities to learn more.

An elliptic curve for the purpose of cryptography is simply a large set of points that we will call *C*. These points can be added and subtracted, and the result is a point on the curve *C*. Furthermore, a point can be successively added to itself, which is represented as point multiplication by integers (also called scalars). Given such a point *H*, an integer *k* and using the scalar multiplication operation we can compute `k*H`, which is also a point on curve *C*. Given another integer *j* we can also calculate `(k+j)*H`, which equals `k*H + j*H`. The addition and scalar multiplication operations on an elliptic curve maintain the distributive property of addition and multiplication:

```
(k+j)*H = k*H + j*H
```

In ECC, if we pick a (very large) number *k* as a private key, `k*H` is considered the corresponding public key. Even if one knows the value of the public key `k*H`, deducing *k* is close to impossible (or said differently, while multiplication is trivial, "division" by curve points is extremely difficult).

The previous formula `(a+j)*H = a*H + j*H`, with *k* and *j* both private keys, demonstrates that a public key obtained from the addition of two private keys (`(a+j)*H`) is identical to the addition of the public keys for each of those two private keys (`a*H + j*H`). In the Bitcoin blockchain, Hierarchical Deterministic wallets heavily rely on this principle. Mimblewimble and the Grin implementation do as well.

Transacting with Mimblewimble

The structure of transactions demonstrates a crucial tenet of Mimblewimble: strong privacy and confidentiality guarantees.

The validation of Mimblewimble transactions relies on two basic properties:

- **Verification of zero sums.** The sum of outputs minus inputs always equals zero, proving that the transaction did not create new funds, without revealing the actual amounts.
- **Possession of private keys.** Like with most other cryptocurrencies, ownership of transaction outputs is guaranteed by the possession of ECC private keys. However, the proof that an entity owns those private keys is not achieved by directly signing the transaction.

The next sections on balance, ownership, change and proofs details how those two fundamental properties are achieved.

Balance

Building upon the properties of ECC we described above, one can obscure the values in a transaction.

If *v* is the value of a transaction input or output and *H* a point on the elliptic curve *C*, we can simply embed `v*H`, instead of *v* in a transaction. This works because using the ECC operations, we can still validate that the sum of the outputs of a transaction equals the sum of inputs:

```
v1 + v2 = v3  <=>  v1*H + v2*H = v3*H
```

Verifying this property on every transaction allows the protocol to verify that a transaction doesn't create money out of thin air, without knowing what the actual values are. Note that knowing *v1* (from a previous transaction for example) and multiplying `v1*H` reveals the outputs with value *v1* across the blockchain. We introduce a second point *G* on the same elliptic curve (practically *G* is just another generator point on the same curve group as *H*) and a private key *r* used as a blinding factor.

An input or output value in a transaction can then be expressed as:

```
r*G + v*H
```

Where:

- *r* is a private key used as a blinding factor, *G* is a point on the elliptic curve *C* and the point `r*G` is the public key for *r* (using *G* as generator point).
- *v* is the value of an input or output and *H* is another point on the elliptic curve *C*, together producing another public key `v*H` (using *H* as generator point).

Neither *v* nor *r* can be deduced, leveraging the fundamental properties of Elliptic Curve Cryptography. `r*G + v*H` is called a *Pedersen Commitment*.

As an example, let's assume we want to build a transaction with two inputs and one output. We have (ignoring fees):

- *v1* and *v2* as input values.
- *v3* as output value.

Such that:

```
v1 + v2 = v3
```

Generating a private key as a blinding factor for each input value and replacing each value with their respective Pedersen Commitments in the previous equation, we obtain:

```
(r1*G + v1*H) + (r2*G + v2*H) = (r3*G + v3*H)
```

Which as a consequence requires that:

```
r1 + r2 = r3
```

This is the first pillar of Mimblewimble: the arithmetic required to validate a transaction can be done without knowing any of the values.

As a final note, this idea is actually derived from Greg Maxwell's Confidential Transactions, which is itself derived from an Adam Back proposal for homomorphic values applied to Bitcoin.

Ownership

In the previous section we introduced a private key as a blinding factor to obscure the transaction's values. The second insight of Mimblewimble is that this private key can be leveraged to prove ownership of the value.

Alice sends you 3 coins and to obscure that amount, you chose 28 as your blinding factor (note that in practice the blinding factor, being a private key, is an extremely large number). Somewhere on the blockchain, the following output appears and should only be spendable by you:

```
X = 28*G + 3*H
```

X, the addition result, is visible to everyone. The value 3 is only known to you and Alice, and 28 is only known to you.

To transfer those 3 coins again, the protocol requires 29 to be known somehow. To demonstrate how this works, let's say you want to transfer those 3 same coins to Carol. You need to build a simple transaction such that:

```
X1 == Y
```

Where *X1* is an input that spends your *X* output and *Y* is Carol's output. There is no way to build such a transaction and balance it without knowing your private key of 28. Indeed, if Carol is to balance this transaction, she needs to know both the value sent and your private key so that:

```
Y - X1 = (28*G + 3*H) - (28*G + 3*H) = 0*G + 0*H
```

By checking that everything has been zeroed out, we can again make sure that no new money has been created.

Wait! Stop! Now you know the private keys in Carol's output (which, in this case, must be the same as yours to balance out) and so you could steal the money back from Carol!

To solve this, Carol uses a private key of her choosing. She picks 113 say, and what ends up on the blockchain is:

```
Y - X1 = (113*G + 3*H) - (28*G + 3*H) = 85*G + 0*H
```

Now the transaction no longer sums to zero and we have an excess value (85), which is the result of the summation (and correspondingly subtraction) of all blinding factors. Note that `85*G` is a valid public key for the generator point *G*.

Therefore, the protocol needs to verify that the transacting parties collectively can produce the private key (85 in the above example) for the resulting point `Y - X1` (this should be the corresponding public key for generator point *G*, `85*G` in the above example). A simple way of doing so is by using the public key `Y - X1` (`85*G`) to verify a signature, that was signed using the excess value (85). This ensures that:

- The transacting parties collectively can produce the private key (the excess value) for the public key `Y - X1`.
- The sum of values in the outputs minus the sum of values in the inputs is zero (otherwise there would be no correspondence between private and public keys, which is exactly the reason for having a signature).

This signature, attached to every transaction, together with some additional data (like mining fees), is called a *transaction kernel* and is checked by all validators.

Some Finer Points

This section elaborates on the building of transactions by discussing how change is introduced and the requirement for range proofs so all values are proven to be non-negative. Neither of these are absolutely required to understand Mimblewimble and Grin, so if you're in a hurry, feel free to jump straight to Putting It All Together.

Change

Let's say you only want to send 2 coins to Carol from the 3 you received from Alice. To do this you would send the remaining 1 coin back to yourself as change. You generate another private key (say 12) as a blinding factor to protect your change output. Carol uses her own private key as before.

```
Change output:    12*G + 1*H
Carol's output:   113*G + 2*H
```

What ends up on the blockchain is something very similar to before. And the signature is again built with the excess value, 97 in this example.

```
(12*G + 1*H) + (113*G + 2*H) - (28*G + 3*H) = 97*G + 0*H
```

Range Proofs

In all the above calculations, we rely on the transaction values to always be positive. The introduction of negative amounts would be extremely problematic as one could create new funds in every transaction.

For example, one could create a transaction with an input of 2 and outputs of 5 and -3 and still obtain a well-balanced transaction. This can't be easily detected because even if *v* is negative, the corresponding point `v*H` on the curve looks like any other.

To solve this problem, Mimblewimble leverages another cryptographic concept (also coming from Confidential Transactions) called *range proofs*. A proof that a number falls within a given range, without revealing the number. We won't elaborate on the range proof, but you just need to know that for any `r*G + v*H` we can build a proof that shows that *v* is non-negative and does not overflow.

It's also important to note that range proofs for both the blinding factor and the values are needed. The reason for this is that it prevents a censoring attack where a third party would be able to lock UTXOs without knowing their private keys by creating a transaction such as the following:

```
Carol's UTXO:     113*G + 2*H
Attacker's output: (113 + 99)*G + 2*H
```

which can be signed by the attacker because Carol's blinding factor cancels out in the equation `Y - X1`:

```
Y - X1 = ((113 + 99)*G + 2*H) - (113*G + 2*H) = 99*G
```

The output (`(113 + 99)*G + 2*H`) requires that both the numbers 113 and 99 are known in order to be spent; the attacker would thus have successfully locked Carol's UTXO. The requirement for a range proof for the blinding factor prevents this because the attacker doesn't know the number 113 and thus neither (113 + 99). A more detailed description of range proofs is further detailed in the range proof paper.

Putting It All Together

A Mimblewimble transaction includes the following:

- A set of inputs, that reference and spend a set of previous outputs.
- A set of new outputs that include:
  - A blinding factor *r* and a value *v* used for scalar multiplication for the curve points *G*,*H* correspondingly, and subsequently summed to the `r*G + v*H`.
  - A range proof that among other things shows that *v* is non-negative.
- A transaction fee in clearand.
- A signature signed with the excess value (the sum of all output values and the fee minus the input values) as the private key.

Blocks and Chain State

We explained above how Mimblewimble transactions can provide strong anonymity guarantees while maintaining the properties required for a valid blockchain, i.e., a transaction does not create money and proof of ownership is established through private keys.

The Mimblewimble block format builds on this by introducing one additional concept: cut-through. With this addition, a Mimblewimble chain gains:

- Extremely good scalability, as the great majority of transaction data can be eliminated over time, without compromising security.
- Further anonymity by mixing and removing transaction data.

Transaction Aggregation

Recall that a transaction consists of the following:

- a set of inputs that reference and spend a set of previous outputs
- a set of new outputs
- a transaction kernel consisting of:
  - kernel excess (the public key corresponding to the excess value)
  - transaction signature signed by the excess value (and verifies with the kernel excess)

(The presentation above did not explicitly include the kernel excess in the transaction, because it can be computed from the inputs and outputs. This paragraph shows the benefit in including it, for aggregation within block construction.)

We can say the following is true for any valid transaction (ignoring fees for simplicity):

```
sum(outputs) - sum(inputs) = kernel_excess
```

The same holds true for blocks themselves once we realize a block is simply a set of aggregated inputs, outputs and transaction kernels. We can sum the outputs, subtract the inputs from it and equalizing the resulting Pedersen commitment to the sum of the kernel excesses:

```
sum(outputs) - sum(inputs) = sum(kernel_excess)
```

Simplifying slightly (again ignoring transaction fees), we can say that Mimblewimble blocks can be treated exactly as Mimblewimble transactions.

Kernel Offsets

There is a subtle problem with Mimblewimble blocks and transactions as described above. It is possible (and in some cases trivial) to reconstruct the constituent transactions in a block. This is clearly bad for privacy. This is the "subset" problem: given a set of inputs, outputs and transaction kernels, a subset of these will recombine to reconstruct a valid transaction.

Consider the following two transactions:

```
(in1, in2) -> (out1), (kern1)
(in3) -> (out2), (kern2)
```

We can aggregate them into the following block (or aggregate transaction):

```
(in1, in2, in3) -> (out1, out2), (kern1, kern2)
```

It is trivially easy to try all possible permutations to recover one of the transactions (where it successfully sums to zero):

```
(in1, in3) -> (out1), (kern1)
```

We also know that everything remaining can be used to reconstruct the other valid transaction:

```
(in2) -> (out2), (kern2)
```

Remember that the kernel excess `r*G` simply is the public key of the excess value *r*. To mitigate this we redefine the kernel excess from `r*G` to `(r-kernel_offset)*G` and distribute the *kernel offset* to be included with every transaction kernel. The kernel offset is thus a blinding factor that needs to be added to the excess value to ensure the commitments sum to zero:

```
sum(outputs) - sum(inputs) = r*G + (r-kernel_offset)*G = kernel_offset*G
```

or alternatively

```
sum(outputs) - sum(inputs) = kernel_excess + kernel_offset*G
```

For a commitment `r*G + v*H` with the offset `a`, the transaction is signed with `(r-a)` and is published so that `r*G` could be computed in order to verify the validity of the transaction: given the kernel excess (recall that it is given as part of the transaction kernel), `(r-a)*G` and the offset `a`, one computes `a*G` and obtains `(r-a)*G + a*G = r*G`. During block construction all kernel offsets are summed to generate a single aggregate kernel offset to cover the whole block. The kernel offset for any individual transaction is then unrecoverable and the subset problem is solved.

Cut-through

Blocks let miners assemble multiple transactions into a single set that's added to the chain. In the following block representations, containing 3 transactions, we only show inputs and outputs of transactions. Inputs reference outputs they spend. An output included in a previous block is marked with a lower-case `o`.

```
I1(o1) --- O1
        |- O2

I2(o2) --- O3
I3(M2) -|

I4(M3) --- O4
        |- O5
```

We notice the two following properties:

- Within this block, some outputs are directly spent by following inputs (*O3* spends *O2* and *M* spends *O3*).
- The structure of each transaction does not actually matter. Since all transactions individually sum to zero, the sum of all transaction inputs and outputs must be zero.

Similarly to a transaction, all that needs to be checked in a block is that *ownership* has been proven (which comes from the transaction kernels) and that the whole block did not create any coins (other than what's allowed as the mining reward). Therefore, matching inputs and outputs can be eliminated, as their contribution to the overall sum cancels out. Which leads to the following, much more compact block:

```
I1(o1) | O1
I2(o2) | O4
       | O5
```

Note that all transaction structure has been eliminated and the order of inputs and outputs does not matter anymore. However, the sum of all inputs and outputs is still guaranteed to be zero.

A block is simply built from:

- A block header.
- The list of inputs remaining after cut-through.
- The list of outputs remaining after cut-through.
- A single kernel offset (sum of all kernel offsets) to cover the full block.
- The transaction kernels containing, for each transaction:
  - The public key `(r-a)*G`, which is the (modified) kernel excess.
  - The signatures generated using the (modified) excess value, `(r-a)`, as the private (signing) key.
  - The mining fee.

The block contents satisfy:

sum(outputs) - sum(inputs) = sum(kernel_excess) + kernel_offset*G

When structured this way, a Mimblewimble block offers extremely good privacy guarantees:

- Intermediate (cut-through) transactions will be impossible to verify by their transaction kernels.
- All outputs look the same: very large numbers that are impossible to meaningfully differentiate from one another. If someone wants to exclude a specific output, they'd have to exclude all.
- All transaction structure has been removed, making it impossible to tell which inputs and outputs match.

And yet, it all still validates!

Cut-through All The Way

Going back to the previous example block, outputs *x1* and *x2*, spent by *i1* and *i2*, must have appeared previously in the blockchain. So after the addition of this block, those outputs as well as *i1* and *i2* can also be removed from the blockchain as they now are intermediate transactions.

We conclude that the chain state (excluding headers) at any point in time can be summarized by just these pieces of information:

1. The total amount of coins created by mining in the chain.
2. The complete set of unspent outputs.
3. The transactions kernels for each transaction.

The first piece of information can be deduced just using the block height.

Both the UTXOs and the transaction kernels are extremely compact. This has two important consequences:

- The blockchain a node needs to maintain is very small (on the order of a few gigabytes for a bitcoin-sized blockchain, and potentially optimizable to a few hundreds of megabytes).
- When a new node joins the network the amount of information that needs to be transferred is very small.

In addition, the UTXO set cannot be tampered with. Adding or removing even one input or output would change the sum of the transactions to be something other than zero.

Conclusion

In this document we covered the basic properties that underlie a Mimblewimble blockchain. By using addition of elliptic curve points, we're able to build transactions that are completely opaque but can still be properly validated. And by generalizing those properties to blocks, we can eliminate a large amount of blockchain data, allowing for great scaling and fast sync of new peers.