Introduction to MimbleWimble and Grin

Read this in other languages: English, 简体中文, Español, Nederlands, Русский, 日本語, Deutsch, Portuguese, 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, subtracted, or multiplied 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 commutative and associative properties 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 `(k+j)*H = k*H + j*H`, with *k* and *j* both private keys, demonstrates that a public key obtained from the addition of two private keys (`(k+j)*H`) is identical to the addition of the public keys for each of those two private keys (`k*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. However, there are a finite number of usable values (transaction amounts) and one could try every single one of them to guess the value of your transaction. In addition, knowing v1 (from a previous transaction for example) and the resulting `v1*H` reveals all outputs with value v1 across the blockchain. For these reasons, 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 their product `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):

- `vi1` and `vi2` as input values.
- `vo3` as output value.

Such that:

```
vi1 + vi2 = vo3
```

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:

```
(ri1*G + vi1*H) + (ri2*G + vi2*H) = (ro3*G + vo3*H)
```

Which as a consequence requires that:

```
ri1 + ri2 = ro3
```

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, it's 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 result of the addition, is visible by 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 28 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:

```
Xi => Y
```

Where *Xi* 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 - Xi = (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 key 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 - Xi = (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* on *G* (85), which is the result of the summation of all blinding factors. But because `85*G` is a valid public key on the elliptic curve *C*, with private key 85, for any x and y, only if `y = 0` is `x*G + y*H` a valid public key on the elliptic curve using generator point *G*.

So all the protocol needs to verify is that (`Y - Xi`) is a valid public key on the curve and that the transacting parties collectively know the private key `x` (85 in our transaction with Carol) of this public key. If they can prove that they know the private key to `x*G + y*H` using generator point *G* then this proves that `y` must be `0` (meaning above that the sum of all inputs and outputs equals `0`).

The simplest way to do so is to require a signature built with the excess value (85), which then validates that:

- The transacting parties collectively know the private key (the excess value 85), and
- The sum of the transaction outputs, minus the inputs, sum to a zero value (because only a valid public key, matching the private key, will check against the 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, following the definition in the previous sections. This can't be easily detected because even if *x* is negative, the corresponding point `x*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 will show that *v* is greater than zero and does not overflow.

It's also important to note that in order to create a valid range proof from the example above, both of the values 113 and 28 used in creating and signing for the excess value must be known. The reason for this, as well as a more detailed description of range proofs are further detailed in the range proof paper. The requirement to know both values to generate valid rangeproofs is an important feature since it prevents a censoring attack where a third party could lock up UTXOs without knowing their private key by creating a transaction from

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

which can be signed by the attacker since Carols private key of 113 cancels due to the adverserial choice of keys. The new output could only be spent by both the attacker and Carol together. However, while the attacker can provide a valid signature for the transaction, it is impossible to create a valid rangeproof for the new output invalidating this attack.

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 value and a blinding factor (which is just a new private key) multiplied on a curve and summed to be `r*G + v*H`.
  - A range proof that shows that v is non-negative.
- An explicit transaction fee, in clear.
- A signature, computed by taking the excess blinding value (the sum of all outputs plus the fee, minus the inputs) and using it as a private key.

Blocks and Chain State

We've 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.
- And the ability for new nodes to sync up with the rest of the network very efficiently.

Transaction Aggregation

Recall that a transaction consists of the following -

- a set of inputs that reference and spent a set of previous outputs
- a set of new outputs (Pedersen commitments)
- a transaction kernel, consisting of
  - kernel excess (Pedersen commitment to zero)
  - transaction signature (using kernel excess as public key)

A tx is signed and the signature included in a *transaction kernel*. The signature is generated using the *kernel excess* as a public key proving that the transaction sums to 0.

```
(42*G + 1*H) + (99*G + 2*H) - (113*G + 3*H) = 28*G + 0*H
```

The public key in this example being `28*G` .

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 tx outputs, subtract the sum of the tx inputs and compare 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.

For example, given 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 sums successfully to zero) -

```
(in1, in2) -> (out1), (kern1)
```

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

```
(in3) -> (out2), (kern2)
```

To mitigate this we include a *kernel offset* with every transaction kernel. This is a blinding factor (private key) that needs to be added back to the kernel excess to verify the commitments sum to zero -

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

When we aggregate transactions in a block we store a *single* aggregate offset in the block header. And now we have a single offset that cannot be decomposed into the individual transaction kernel offsets and the transactions can no longer be reconstructed -

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

We "split" the key `k` into `k1+k2` during transaction construction. For a transaction kernel `(k1+k2)*G` we publish `k1*G` (the excess) and `k2` (the offset) and sign the transaction with `k1*G` as before. During block construction we can simply sum the `k2` offsets to generate a single aggregate `k2` offset to cover all transactions in the block. The `k2` offset for any individual transaction is unrecoverable.

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 x.

```
I1(x1) --- O1
        |- O2

I2(x2) --- O3
I3(O2) -|

I4(O3) --- O4
        |- O5
```

We notice the two following properties:

- Within this block, some outputs are directly spent by included inputs (I3 spends O2 and I4 spends O3).
- The structure of each transaction does not actually matter. As 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 *transaction kernels*) and that the whole block did not add any money supply (other than what's allowed by the coinbase). 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(x1) | O1
I2(x2) | 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 outputs in this block, minus the inputs, 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 to cover the full block.
- The transaction kernels containing, for each transaction:
  - The public key `r*G` obtained from the summation of all the commitments.
  - The signatures generated using the excess value.
  - The mining fee.

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

- Intermediate (cut-through) transactions will be represented only by their transaction kernels.
- All outputs look the same: just very large numbers that are impossible to differentiate from one another. If one wanted to exclude some outputs, they'd have to exclude all.
- All transaction structure has been removed, making it impossible to tell which output was matched with each input.

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 overall chain, as they do not contribute to the overall sum.

Generalizing, 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 (its distance from the genesis block). And both the unspent outputs and the transaction kernels are extremely compact. This has 2 important consequences:

- The state a given node in a MimbleWimble blockchain 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 a network building up a MimbleWimble chain, the amount of information that needs to be transferred is also very small.

In addition, the complete set of unspent outputs cannot be tampered with, even only by adding or removing an output. Doing so would cause the summation of all blinding factors in the transaction kernels to differ from the summation of blinding factors in the outputs.

Conclusion

In this document we covered the basic principles that underlie a MimbleWimble blockchain. By using the addition properties of Elliptic Curve Cryptography, 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.