Mimblewimble Non-Interactive Transaction Scheme

Gary Yu
gary.yu@gotts.tech

Revised, Oct. 10, 2020

Abstract. I describe a non-interactive transaction scheme for Mimblewimble protocol, so as to overcome the usability issue of the Mimblewimble wallet. With the *Diffie–Hellman*, we can use an *Ephemeral Key* shared between the sender and the receiver, a public nonce R is added to the output for that, removing the interactive cooperation procedure. And an additional *one-time public key* P' is used to lock the output to make it only spendable for the receiver, i.e. the owner of P'. The new data R and P' can be committed into the bulletproof to avoid the miner's modification. Furtherly, to keep Mimblewimble privacy character, the *Stealth Address* is used in this new transaction scheme. All the cost of these new features is 66-bytes additional data (the public nonce R and the *one-time public key* P') in each output, and 64-bytes additional signature data in each input. That is about 12% payload size increasing in a typical single input double outputs Mimblewimble transaction.

Keywords: Mimblewimble, Stealth address, Bitcoin, Grin, Confidential transaction, Privacy

License. This work is released into the public domain.

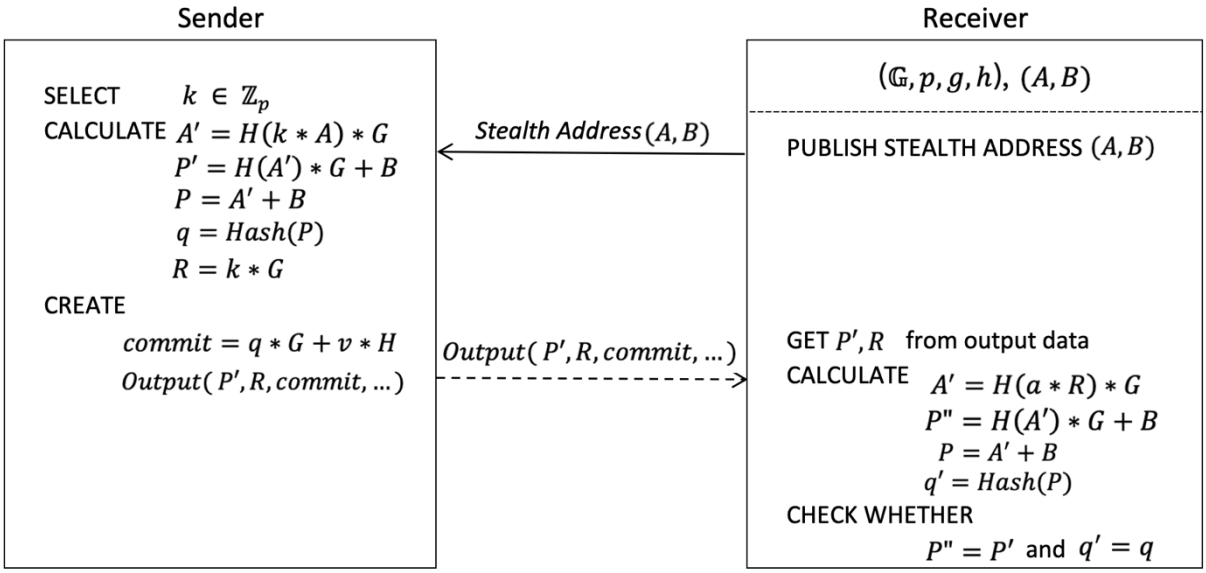

Fig.1 Mimblewimble non-interactive transaction scheme design.

1 Introduction

Mimblewimble. In July 2016, someone called Tom Elvis Jedusor (Voldemort's French name in J.K. Rowling's Harry Potter book series) placed the original Mimblewimble white paper[MW16] on a bitcoin research channel, and then disappeared. Tom's white paper "Mimblewimble" (a tongue-tying curse used in "The Deathly Hallows") was a blockchain proposal that could theoretically increase privacy, scalability and fungibility. In January 2017, Andrew Poelstra, a mathematician at Blockstream, presented on this work at Stanford University's Blockchain Protocol Analysis and Security Engineering 2017 conference. And he

wrote a paper[Poe16] to make precise Tom's original idea, and added further scaling improvements on it. Mimblewimble is a blockchain protocol with confidential transaction and obscured transaction graph, also it has the ability to merge transactions in transaction pool, or even merge them across blocks.

Because only UTXOs are kept, Mimblewimble blockchain data is much smaller than other chain types. For example, Bitcoin[Bit08] today there are about 646,300 blocks, total 300GB or so of data on the hard drive to validate everything. These data are about 560 million transactions and 68 million unspent nonconfidential outputs. Estimate how much space the number of transactions take on a Mimblewimble chain. Each unspent output is around 0.7KB for bulletproof[BBB16]. Each transaction kernel also adds about 100 bytes. The block headers are negligible. Add this together and get 104GB -- with a confidential transaction and obscured transaction graph!

Grin. At the end of 2016, Ignotus Peverell (name also comes from "Harry Potter", the original owner of the invisibility cloak, if you know the Harry Potter characters) started a GitHub project called Grin[Pev16]. Grin is the first project that implements a Mimblewimble blockchain to provide extremely good scalability, privacy and fungibility, by relying on strong elliptic curve cryptographic primitives. And it is a purely community driven project, just like Bitcoin.

Interactive Transaction. In Mimblewimble and Grin, a typical transaction with 1 input and 2 outputs is defined as:

$$(x_i * G + a_i * H) + (excess' + offset * G) = (x_c * G + a_c * H) + (x_r * G + a_r * H) + fee * H$$

Where,

- $(x_i * G + a_i * H)$ is the input coin owned and selected by the sender.
- $(x_r * G + a_r * H)$ is the output coin created by the **receiver**.
- $(x_c * G + a_c * H)$ is the change coin created by the sender.
- x_i, x_c, x_r are the private keys.
- a_i, a_c, a_r are the transaction values, which is hidden in the bulletproof attached on each output commitment.
- fee is the transaction fee, which is an open value in the transaction kernel.
- $offset$ is a random number selected by the sender.

The $excess'$ is called as "*public excess*" which is the signature public key of the transaction kernel and consists of:

$$excess' = (x_c - x_i - offset) * G + x_r * G$$

Where,

- $(x_c - x_i - offset) * G$ is a public key which only sender knows the private key.
- $x_r * G$ is a public key which only receiver knows the private key.

To sign this transaction with $excess'$ as the public key, the *Simpler Variants of MuSig*[DCC19] interactive signature scheme is used, meaning both the sender and the receiver exchanges the public key and public nonce info, then executes a *MuSig* partial signature in both side, then either the sender or the receiver finally aggregate these two partial signatures to get a final joint Schnorr signature, which can be verified exactly as a standard Schnorr signature with respect to a single public key: $excess'$.

The pros of this transaction scheme are impressively on the simplicity and the minimum size, which only needs **one** 2-of-2 Schnorr signature to authorize this spending, i.e. a 64-bytes signature info. But the cons are also extremely impressed at:

- The bad usability on the wallet implementation, mainly because of the interactive process.
- Slow, because of the cooperation time between payer and payee.
- The wallet security concern, because the receiver wallet must listen online to the payments and the private key must be used to receive.

Grin should have gotten much more adoption and be much more popular than today if it does not need an interactive transaction.

My Contribution. In this paper, I propose a new transaction scheme for Mimblewimble protocol, which is non-interactive so as to overcome above major weakness. With the Diffie–Hellman, we can use an *Ephemeral Key* shared between the sender and the receiver, a public nonce R is added to the output for that, so as to remove the interactive cooperation process. And an additional *one-time public key* P' is used to lock the output to make it only spendable for the owner of P'. The new data R and P' can be committed into the bulletproof to avoid the miner's modification. Furtherly, to keep Mimblewimble privacy character, the *Stealth Address*[Byt11, Sab13, Tod14, CM17, Yu20] is used in this new transaction scheme. All the cost of these new features is 66-bytes additional data (a public nonce R and the *one-time public key* P') in each output, and 64-bytes additional signature data in each input. That is about 12% payload size increasing in a typical single input double outputs Mimblewimble transaction, which is supposed to be about 1.6KB with the original interactive transaction scheme.

2 Mimblewimble Non-Interactive Transaction Scheme

2.1 Non-Interactive Transaction Scheme Design

A typical transaction with 1 input and 2 outputs is defined as:

$$(x_i * G + a_i * H) + (excess' + offset * G) = (x_c * G + a_c * H) + (q * G + a_r * H) + fee * H$$

Where,

- $(x_i * G + a_i * H)$ is the input coin owned and selected by the sender.
- $(q * G + a_r * H)$ is the output coin created by the **sender**.
- $(x_c * G + a_c * H)$ is the change coin created by the sender.
- x_i, x_c are the private keys of the sender.
- q is the *Ephemeral Key* shared between the sender and the receiver, which will be explained later.
- a_i, a_c, a_r are the transaction values, which is the hidden info in the bulletproof attached on each output commitment.
- fee is the transaction fee, which is an open value in the transaction kernel.
- $offset$ is a random number selected by the sender.

The $excess'$ is called as "*public excess*" which is the signature public key of the transaction kernel and consists of:

$$excess' = (x_c - x_i - offset + q) * G$$

Where,

- $(x_c - x_i - offset + q)$ is a private key which can be calculated by the sender.

To sign this transaction with $excess'$ as the public key, the standard Schnorr signature scheme [WNR18] is used.

Now, look at the *Ephemeral Key* q, which is the core part of this non-interactive transaction scheme.

Definitions.

$$A' = H(k * A) * G \equiv H(a * R) * G$$
$$P' = H(A') * G + B$$
$$P = A' + B$$
$$q = H(P)$$

Where H is a hash function, and (A, B) is the concatenation of the *public view key* and the *public spend key* of the recipient's *Stealth Address*, which is designed to protect recipient privacy. k is a secret nonce (a random number) selected by the sender and a related public nonce $R = k * G$ is attached to the transaction output.

Thanks to the Diffie–Hellman key exchange, i.e. the truth that $a * R \equiv k * A$, the recipient can also calculate this *Ephemeral Key* q by a, where a is the recipient's *private view key* of A.

The receiver checks every passing transaction (UTXO actually) with his/her private key (a, B), picks the R and P' from the UTXO, computes $A' = H(a * R) * G$ and then $q' = H(A' + B)$ and $P" = H(A') * G + B$, collects the payments if $q' = q$ by bulletproof rewinding and if $P" = P'$.

With the sharing private key of A, an auditor for example can also computes this q' and $P"$ therefore is capable to view every incoming transaction for that recipient's *Stealth Address*.

As the sender, he/she must attach both R and P' to the payment output, i.e. together with $(q * G + a_r * H)$ commitment in above example. That is 66-bytes additional cost for each output, compared to the native interactive Mimblewimble transaction scheme. The new data R and P' need be committed into the bulletproof to lock the output data. For example, $H(commit \| R \| P')$ is committed.

For the receiver, when spending this received coin $(q * G + a_r * H)$ in the future, he/she must attach a Schnorr signature of P' to provide the additional ownership proof, in the input structure, which is 64-bytes additional cost for each input, compared to the native interactive transaction scheme. The private key of P':

$$p' = H(A') + b$$

Where $A' = H(a * R) * G$ and (a, b) is the private keys of the recipient's *Stealth Address* and R is the public nonce in the output data. The signature in transaction kernel is still needed as before to prove he/she knows the secret q.

2.2 Payment Confirmation and Insecure Zero-Confirmation Transaction

A common concept in blockchain is the transaction confirmations, which presents the truth that as blocks are buried deeper and deeper into the blockchain the transactions become harder and harder to change or remove, this gives rise of blockchain's *Irreversible Transactions*. And because of the possible forks of the chain, a popular best practice for a recipient is to wait

enough block confirmations before he/she confirms the payment and deliver the products or service, for example waiting 6 block confirmations in Bitcoin or waiting 10 block confirmations in Grin.

The *0-confirmation transaction* is defined as an exchange that has not yet been recorded and verified on the blockchain. Instead the seller immediately assumes he received his money and delivers what was sold.

Normally, in blockchain world, the *0-confirmation transaction* is insecure, mainly because of the possible forks on the chain. But in this Mimblewimble non-interactive transaction scheme, the *0-confirmation transaction* is insecure by the design, precisely by the *CoinJoin* feature of Mimblewimble, even there is no forks happening.

For example, we have two transactions T_1 and T_2:

$$T_1\colon\ I_1 + E_1 = C_1 + O_1$$
$$T_2\colon\ I_2 + E_2 = C_2 + O_2$$

Where $I_2 = O_1$, meaning the transaction T_2 is spending the output O_1 which is just created in the transaction T_1, and both T_1 and T_2 are created by same people. When transaction T_1 has 0-confirmation, both T_1 and T_2 exist in the transaction pool. So, a CoinJoin for T_1 and T_2 will happen in the transaction pool, as one of the outstanding features of Mimblewimble protocol. The merged transaction T_{12} becomes:

$$T_{12}\colon\ I_1 + E_1 + E_2 = C_1 + C_2 + O_2$$

Where both I_2 from the inputs and O_1 from the outputs disappear, because of Mimblewimble cut-through. It seems no problem when both T_1 and T_2 are honest transaction, means their transaction validation is ok. But in case a dishonest people is spending O_1 which is just created by self, he/she can create a fake T_2 which cannot pass the transaction validation by itself, because the wrong signature in I_2. Unfortunately, the merged transaction T_{12} cuts that fake I_2, and the remaining parts of T_{12} will pass the validation well. Then, after the merged transaction T_{12} is packed by a block and get enough confirmations, the creator of transaction T_1 is capable to provide a valid payment proof about O_1, which belongs to other people but he/she already spent it.

Therefore, there are two aspects to solve this problem. One is the payment proof, which need provide the output MMR proof, so as to make the above T_1 become an invalid payment proof. This will be explained in §2.5 for detail.

Another one is the transaction confirmation, an important security tip here, the Mimblewimble non-interactive transaction confirmation must stick to the UTXO confirmations, meaning the payment **output** must be unspent on the chain and have **enough** confirmations before someone confirms he/she receives the payment. This is same as other blockchains but differentiate itself from the original Mimblewimble protocol which instead use the *transaction kernel* for the confirmations.

2.2.1 Dishonest Receiver

In above example of T_1 and T_2, we described the case that a dishonest sender is spending O_1. Here we will look into the case of a dishonest receiver.

For the dishonest receiver, he/she can scan the transaction pool instead of the block chain for his/her outputs, which someone is paying for him/her, once a payment is found, in a transaction T_1 for example, he/she can collect it immediately by creating a transaction T_2 to spend that O_1, then the merged transaction T_{12}: $I_1 + E_1 + E_2 = C_1 + C_2 + O_2$ could be packed into next block, instead of T_1 and T_2. The consequence is the payer will not be able to provide a valid payment proof even the dishonest receiver got his/her payment.

In this case, even we should not call the receiver as "dishonest" since the protocol allows this cut-through behaviour.

To fix this problem, we can use a consensus to forbid the cut-through on the payment output, O_1 in above example, but obviously a dishonest node will still be able to do that cut-through and this broken transaction can't be detected by other nodes.

In Mimblewimble framework, with this non-interactive transaction scheme, a feasible solution is freezing all cut-through behaviour in the transaction pool, by adding a field n into the transaction kernel, to indicate how many *Inputs* a transaction has. In block validation, we sum all n of kernels and validate the total *Inputs* number.

The cut-through between the blocks is still feasible, which is great to reduce weight for non-archive nodes. Normally the cut-through between blocks is launched for the old blocks under the *Horizon*, we will discuss more about that in §3.1.

The cost of this fixing solution is a little losing of the privacy, since the cut-through in transaction pool is helpful for the further transaction graph obscuring. In original Mimblewimble protocol, the O_1 in above example will never appear into the block chain, that is a nice way to avoid tracking. But considering the block chain analysers is able to track the transaction pool also, instead of tracking the block chain, this so-called losing is really trivial.

Another concern in this solution is the new field n increases the linkable risk. But actually it is not the truth. For example, still in above example, T_{12}: $I_1 + I_2 + E_1 + E_2 = C_1 + C_2 + O_1 + O_2$ with E_1 contains a $n = 1$ and E_2 contains a $n = 1$, this obviously does not increase a bit of the linkability.

The last concern is the payload size increment. But since we can use the variable size transaction kernel, we can define a missing field n means $n = 1$, so as to cover over 90% use cases with zero size increment. For other cases with $n > 1$, one octet should be enough to encode this n.

2.3 Zero-Confirmation Transaction for Change Spending

In a special situation when spending the change output/s, the Mimblewimble 0-confirmation transaction is secure, both in native interactive transaction scheme, and in this non-interactive transaction scheme.

For example, we have two transactions T_1 and T_2:

$$T_1: I_1 + E_1 = C_1 + O_1$$
$$T_2: I_2 + E_2 = C_2 + O_2$$

Where $I_2 = C_1$, meaning the transaction T_2 is spending the change output C_1 which is just created in the transaction T_1, obviously both T_1 and T_2 are created by same people in this case.

A *CoinJoin* for T_1 and T_2 will happen in the transaction pool. The merged transaction T_{12} becomes:

$$T_{12}: I_1 + E_1 + E_2 = C_2 + O_1 + O_2$$

Obviously, even we lose the signature info in I_2 after merging, the merged transaction T_{12} is still good and is doing the job as we wanted. With this character, a fast continuous payments feature is feasible for Mimblewimble blockchain, meaning a Mimblewimble wallet will never be locked for waiting the transaction confirmation unless the wallet balance is over, when this wallet is only doing the payments.

Nevertheless, after we freeze the cut-through for solving the insecurity in §2.2, this *CoinJoin* for T_1 and T_2 will not cut-through, so the merged transaction will look like this even $I_2 = C_1$:

$$T_{12}\colon\ I_1 + I_2 + E_1 + E_2 = C_1 + C_2 + O_1 + O_2$$

We can see this is not helpful for getting a smaller size of the transaction, and the transaction fee is also not attractive for user since the fee is same as the sum of T_1 and T_2. Therefore, this continuous spending on the change output is not proposed. Instead, for payments to multiple receivers in this non-interactive transaction scheme, we propose to create a single transaction for that:

$$T\colon\ I_1 + E_1 = C_1 + O_1 + O_2$$

This single transaction with multiple payment outputs for multiple receivers can save 25% transaction fee for above example. For m receivers, the fee of the continuous spending scheme needs $0.004 * (2m)$ coins, but the fee of the single transaction scheme only needs $0.004 * (m + 1)$ coins. In addition, the latter is much more elegant.

2.4 The Migration

The mixing of the native interactive transaction and the new non-interactive transaction scheme is possible but strongly not proposed, not only because of the complexity of the mixing, but also the privacy concern. All outputs data should have same data structure and they should looks no difference between interactive transaction output and non-interactive transaction output.

Therefore, for those existing Mimblewimble blockchains, a hard fork and a migration is proposed, to obsolete the interactive transaction and adopt the new non-interactive transaction scheme. All existing UTXOs can be kept as same as before, but all the new transaction outputs will use the new format, meaning with the additional 66-bytes for storing the public nonce *R* and the one-time public key P'.

For the universal output format, all the change output should also use the same structure as the payment output.

2.5 The Mixing

Even not proposed, the mixing of the native interactive transaction and the new non-interactive transaction scheme is possible, but it needs a very careful design to consider all kinds of security concerns.

For the easiness of description, we will use the following abbreviations for the remaining parts:

IT - *Interactive Transaction*
NIT - *Non-Interactive Transaction*
ITO - *Interactive Transaction Output, meaning the output without that* P' *and R*
NIO - *Non-Interactive Transaction Output, meaning the output with that* P' *and R*

First of all, to support the mixing, an indicator has to be added to differentiate the *ITO* and *NIO*. And the dedicated MMR for unspent *ITO* and *NIO* could be applied because of different output data size.

This will leave a linkability issue that an analyser is able to differentiate the change output and the payment output, i.e. the *Input* and the *ITO* belongs to same people, that's why this mixing is not proposed. But considering the flexibility benefit, this cost could be deserved. It's up to the designer to decide using this mixing or not.

This linkability issue does not have a fix solution yet, but a workaround exists. The user has to use an *IT* to spend a *NIO*, so as to get 2 obscured *ITOs*, then spending this obscured *ITO* either with *IT* or with *NIT*, the aforementioned linkability issue is eliminated.

2.5.1 Spending *NIO*

A general rule for the transaction which is spending a *NIO* is that the signature of P' is mandatory in *Input*, no matter what type the transaction is, either *IT* or *NIT*. This guarantee that only the coin's true owner can spend the coin. The principle is quite similar to Bitcoin's *P2PKH*(Pay-to-Public-Key-Hash).

2.5.2 Spending *ITO*

An *ITO* commit with $(x * G + v * H)$ has the blinding factor x which is only known to the owner, and *ITO* does not have the fields of P' and R. Therefore, the transaction spending an *ITO* only need one signature into the transaction kernel, no matter it's an *IT* or *NIT*.

2.5.3 Outputs of *NIT*

At least one *NIO* must be there in a NIT, and normally that *NIO* is for the receiver. For the *Change* output/s, there is an option to use *NIO* or *ITO* in the mixing scheme.

Change output/s format in a *NIT*	Pros	Cons
NIO	Obscured *Change* and *Payment* for same format	67 bytes size increment
ITO	Smaller size	Linkability between *Input* and *Change*

2.6 Payment Proof

Payment proof means a proof to the third party (normally an arbiter) to prove the payment was made, when someone sends money to a party who then disputes the payment was made. The payment proof in Bitcoin is simple since the recipient address is recorded in the chain and open to anyone, but for a blockchain which uses *stealth address*, the payment proof is not so straight.

A simple method is to use the secret nonce k since only the sender knows this secret. Just provide a signature on a given message from the third party with this k as the secret key.

In a payment proof with signature, the following info will be provided as the payment proof:

1. The transaction output, which can be used to get that corresponding public nonce R;
2. The transaction output MMR[Tod12] proof;
3. The receiver's address but please note the third party arbiter will also need to know this address to assert it all ties together;

4. A message from the third party and the corresponding signature from the sender. The signature can be verified with above R as the public key.

The pros of this method are obviously the simplicity of proof construction. The cons are mainly on the reliability, meaning the sender is incapable to create the proof once the secret k is lost, since this secret nonce k is only stored in local wallet.

2.7 Special Application with Missing P'

With P' missing in the output, an interesting feature is feasible with this *NIT* scheme, which makes the *NIO/s* spendable both for the sender and for the receiver, since both of them know that *Ephemeral Key* q. With this feature, recovering funds sent to the wrong receiver is also easy. The developers can facilitate this feature to design some kinds of interesting applications.

For example, for the transaction among the trusted people such as family members, it will be able to recover funds if the receiver lose his/her key or forget wallet password. Another example is the airdrop in an early stage, if many years later some of those airdropped coins are still there unspent, it will be a very high probability that those airdropped receivers lost their keys, then the one who did that airdrop can recover those unspent airdrop coins. The 3rd example is the gift application, which enable the receiver to "reject" receiving, considering a Bitcoin wallet is unable to refuse receiving any payment.

The common ground of all these special applications is that the receiver will never require a payment proof, which is not available in this feature.

To spend a *NIO* with missing P', there is no need (actually impossible) to provide the signature of P'. And to finalize such kind of payments, the receiver must complete a final receiving step to transfer the funds into an output commitment which only he/she knows the blinding factor, i.e. creating a new transaction to send these payments to him/her self.

2.8 Security of the Mimblewimble Non-Interactive Transaction Scheme

2.8.1 Horizon Attack

I call it *Horizon Attack* here, to differentiate it from *Long Range Attack* which only makes sense in PoS (*Proof of Stake*).

In Mimblewimble system, only UTXOs and transaction kernels are stored in MMR which will be downloaded for every fresh node to get blockchain state synced. The *Horizon* here means some recent blocks will be transferred one by one via p2p protocol, instead of downloading by MMR data package. For example, in Grin, this *Horizon* consensus is 48 hours, or 2,880 blocks. That means for all fresh installed nodes, only recent 2,880 blocks will be validated for all inputs signature, and all inputs signature in older blocks will be ignored. This has no any problem when a fork happened but the fork depth less than the *Horizon* height. However, if the fork depth is bigger than the *Horizon* height, the *Horizon Attack* could happen in all fresh installed nodes, the payments which has at least *Horizon* confirms could be stolen by the original payer. But for all existing nodes in Mimblewimble network, there is another consensus called *Cut-through Horizon* which is 7x24 hours or 10,080 blocks for example in Grin, means the *Horizon Attack* fork depth has to be much longer.

In another side, in case somebody really has the amazing hash power and go to execute the *Horizon Attack*, he/she is able to do any double-spending as he/she want, no matter the

double-spending is executed in this way or that way. Therefore, this *Horizon Attack* only has the meaning in theory, it has almost same effect as 51% attack. Anyway it is possible to simply increase this *Horizon* consensus to improve this *Horizon Attack* depth.

An easy workaround to avoid this *Horizon Attack*, just remember to collect all payments by a new transaction which is created by self, so as to avoid sharing that *Ephemeral Key* with the payer, before the payment output reaching the *Horizon* confirmations.

2.8.2 Leakage of the One-Time Key Difference

In case the sender makes two payments to the same receiver, the one-time public key P' difference between both payment outputs is known to the sender.

O_1: $P_1' = H(H(k_1 * A) * G) * G + B$
O_2: $P_2' = H(H(k_2 * A) * G) * G + B$
=> $P_1' - P_2'$ and the related secret $p_1' - p_2'$ is known for the sender

This is not an issue at all for us, since p_1'/p_2' must be known for spending O_1/O_2, considering the similar *Stealth Address* scheme has been used in Monero[Xmr13] for years. But it must be clear that any future application must not rely on this difference.

Reference

DCC19	Gregory Maxwell, Andrew Poelstra, Yannick Seurin, Pieter Wuille. Simple Schnorr multi-signatures with applications to Bitcoin. Designs, Codes and Cryptography volume 87, pages2139–2164(2019). https://doi.org/10.1007/s10623-019-00608-x
BBB16	Benedikt Bunz , Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, Greg Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. https://eprint.iacr.org/2017/1066.pdf
Byt11	user 'bytecoin'. Untraceable transactions which can contain a secure message are inevitable. 2011. https://bitcointalk.org/index.php?topic=5965.0
Sab13	Nicolas van Saberhagen. CrypoNote v 2.0. 2013. https://cryptonote.org/whitepaper.pdf
Tod14	Peter Todd. [Bitcoin-development] Stealth addresses. 2014. http://www.mail-archive.com/bitcoin-development@lists.sourceforge.net/msg03613.html
CM17	Nicolas T. Courtois, Rebekah Mercer. Stealth Address and Key Management Techniques in Blockchain Systems. In Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017), pages 559-566.
Yu20	Gary Yu. Blockchain Stealth Address Schemes. https://eprint.iacr.org/2020/548.pdf
Tod12	Peter Todd. Merkle Mountain Range. 2012. https://github.com/mimblewimble/grin/blob/master/doc/mmr.md
Bit08	Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. http://bitcoin.org/bitcoin.pdf
WNR18	Pieter Wuille, Jonas Nick, Tim Ruffing. Schnorr signatures for secp256k1, 2018. https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki
MW16	Tom Elvis Jedusor. Mimblewimble. 2016. https://github.com/mimblewimble/docs/wiki/Mimblewimble-origin

Poe16 Andrew Poelstra. Mimblewimble.
https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf
Pev16 Ignotus Peverell. Introduction to Mimblewimble and Grin.
https://github.com/mimblewimble/grin/blob/master/doc/intro.md
Xmr13 https://web.getmonero.org/resources/moneropedia/stealthaddress.html