BLS signatures in Mimblewimble

February 26, 2019

Abstract

We discuss the properties of BLS signatures when applied to MimbleWimble. We find that these are likely incompatibility with script-less scripts unless major modifications are made to the signature scheme. Instead we propose an alternative option to recover the properties of Schnorr signatures. Our idea has some interesting further applications such as offline transactions, publicly verifiable transactions and user issued assets.

Contents

1 Introduction

The MW (MimbleWimble) protocol is based on Pedersen commits

$$C = vL + kG\,,$$

where we will use small letters to refer to numbers (modulo a large prime) and capital letters for elliptic curve-points.

The Pedersen commit combines the value of the output v to the basepoint L with the private key k to the basepoint G. It is the key concept that allows aggregation of most chain data. When a transaction is sent a verifier does not need to care who paid whom, the only relevant information is that all outputs have a well defined set of owners and that no additional money was created. When an output $C = vL + kG$ is spent to create a new output $C' = v'L + k'G$, a signature is required to the difference of both commits and nothing else (apart form rangeproofs for v').

If the value in this transaction is conserved: $v = v'$, this requires a signature of $k - k'$ to the base point G since the contribution proportional to L drops out. In collaboration this is easily created because the sender knows their key k and the receiver k'. Signatures could be using Schnorr or BLS schemes. But when the amount does not add up, the offset is given by a sum of the curve-points G and L and as long as the relation between the two points is unknown they cannot create a valid signature.

This does not only hold for a single transaction, but for the entire transaction history. By signing all kernel differences it can be proven that no new money was created and that all transactions were signed by the owners of the commitments. All kernels can be aggregated to create a very lightweight chain, but using Schnorr signatures this requires an interactive protocol which makes full aggregation unrealistic in practice. As a result most of the individual kernel offsets must be stored indefinitely. However using BLS signatures kernels may be combined non-interactively resulting in a scaling of the chain with the UTXO size and not the sum of all historical transactions.

2 Main Differences between Schnorr and BLS Signatures

A signature ideally is a zero-knowledge proof of knowledge where the prover convinces a verifier that they know some secret information without revealing anything that could be used by the verifier (or anyone else) to gain knowledge of that secret.

In our case we are interested in signatures to a known public key $P = xG$ and the prover wants to convince the verifier that they know x. One method to do so is following the Schnorr signature scheme where the prover commits to a random nonce $R = rG$ and then displays the combination $s = r - ex$, where e is a hash created from the public key P, R and the message to be signed. The signature s does not reveal any information since r is unknown and the addition hides the secret key perfectly. Yet the verifier can compute $sG = R - eP$ and verify that the prover must know x using only publicly available information.

Schnorr signatures are particularly interesting because they are linear in the secret s. This allows for signature aggregation where $P = P_1 + P_2 = x_1G + x_2G$ and two provers can demonstrate that they know x_1 and x_2 respectively in one single proof. The individual signatures $s_1 = r_1 + ex_1$ and $s_2 = r_2 + ex_2$ can be added to provide a valid signature to the combined public key P.

But in this case a well-known weakness of the Schnorr (and in fact also BLS) signatures is relevant. A prover may make an adversarial choice of public key to forge such a signature. After the public key of the first prover is known they can choose $P_2 = x_2G - P_1$ and now $P = P_1 + P_2 = x_2G$ such that $s = r_2 - ex_2$ appears to be a valid multi-signature even though the first prover did not participate in the proof at all. As a consequence the simple scheme presented above needs to be modified. In essence, instead of taking a sum of the public keys, a different combination derived from the public keys of all provers in a random oracle model needs to be signed, eliminating the possibility to pick a tailored public key. However, it remains necessary to commit to a value of $R = R_1 + R_2$ in the signature, making the Schnorr multi-signature interactive.

Another weakness of the Schnorr signature is exploited to construct scriptless scripts. A Schnorr signature is only secure as long as the nonce is random. In essence the Schnorr signature is just the plain private key hidden behind that random number and the zero-knowledge property of the scheme depends on a secure nonce. A prover that, for example, reuses a nonce is therefore at risk of leaking their private key. But this weakness can also be exploited in multi-signatures to leak a secret when signing a transaction and thereby enable atomic transactions and scriptless scripts.

While Schnorr signatures are mathematically simple, BLS signatures are significantly more complex. They also require stronger security assumptions and should generally be less favourable for these two reasons.

BLS signatures rely on a bilinear map $e(,)$ that combines two elliptic curve-points into one point on a separate elliptic curve of the same order. Assuming that we know such a function we can use the bilinear properties to create a proof of knowledge. Let M be a hash of the message multiplied with the base point G and $\Sigma = xM$, where x is the private key of a prover. Then it follows that

$$e(\Sigma, G) = e(P, M)\,,$$

and Σ is a signature that can be verified using public information.

BLS signatures are not numbers, but elliptic curve-points and as such do not leak any information as long as the discrete logarithm problem is hard. This explains why there is no nonce needed in this scheme; instead the ecliptic curve multiplication already hides the secret sufficiently. Just as in the case of Schnorr signatures the linearity allows simple signature aggregation and again the simplest implementation is not secure against rouge key attacks. The solution to this problem is identical to the Schnorr case; a combination of keys is aggregated that depends on a hash of the provers public keys. But since there is no nonce, the signatures can be aggregates without the provers involvement. All that is needed is a valid single signature from each prover and then anyone can aggregate these individually valid signatures into a valid and secure multi-signature.

While non-interactive signature aggregation is possible in a BLS scheme, this comes at a necessary cost of complicating scriptless scripts. The BLS signatures are by design zero-knowledge and no information is leaked when signing. It is also not easy to introduce a nonce in the scheme to leak some information on purpose as the ecliptic curve multiplication would also hide it. The advantage that makes non-interactive signature aggregation possible is now an obstacle for the construction of scripts.

Possibly there could be some method to make BLS signatures weaker and leak some information, but that will likely change BLS signatures so fundamentally that it would be a different signature scheme altogether. Furthermore any option to leak information likely will also destroy the non-interactive signature aggregation.

We therefore conclude that BLS signatures are not compatible with scriptless scripts. In the following we will discuss a potential modification of the MimbleWimble protocol instead that does allow for full kernel aggregation while keeping the possibility of scriptless scripts.

3 Extending MimbleWimble

We propose the following modification of MW for the use of BLS signatures. Currently a UTXO is a commitment plus a range-proof. In addition we introduce an address to go along those. This makes the UTXO set larger (not too much since we do not need to add an additional range proof), but the fundamental scaling properties of the protocol are left unchanged. The commit takes then form

$$C = vL + kG - sG = vL + \tilde{k}G\,,$$

and the corresponding public key is

$$P = sG\,.$$

When summing the commit and the public key we recover the usual Pedersen commit that is currently used in MimbleWimble. What we propose is essentially isolating a part of the Pedersen commit and separately committing to it as a 'public key'.

When transacting from a commitment (C, P) to another output (C', P') the kernel offset $\Delta C = C + P - C' - P'$ is signed and this results in the usual MW verification that no new coins were generated and that all transfers were authorised by their respective owners. The address in fact does not appear in the verification where only the private key k is needed.

However, $k = \tilde{k} + s$ and we can write

$$\Delta C = \tilde{k}G + sG - k'G\,,$$

and that involves an explicit signature to s that could be individually verified against the known public key P via checking that $e(\Sigma_s, G) = e(P, M)$.

This still does not help us with restoring scriptless scripts and so far we have just artificially increased the size of a UTXO's. But as a part of the consensus rules we can enforce that for spending an output in addition to signing the kernel excess, we also require a signature to to the corresponding public key. And this signature does not need to be provided in form of a BLS signature, but it could be required as a Schnorr signature.

After a transaction is finalised what remains is an updated kernel excess and a database of recently spent UTXO's and corresponding Schnorr signatures. Furthermore, these signatures can eventually be pruned as they are not needed for the fundamental security of the protocol. They are only added to provide some level of added security as Schnorr signatures have weaker security assumptions and to enable scriptless scripts. The only possible attack to spend coins without a signature to their public key is a deep reorg that extends beyond the time of pruning Schnorr signatures. But spending UTXO's still requires knowledge of the private key k for each output.

Now we have achieved the fundamental scaling of BLS signatures, while keeping Schnorr signatures to leak some information when needed. The main question is wether it is reasonable to introduce an additional Schnorr signatures just to leak some information. But the simplicity of Schnorr signatures suggests that this is in fact an almost minimal scheme to do that. To leak information there needs to be a possibility to verify the commitment, here realised via the public key, plus the output of the information to be leaked, here the Schnorr signature. And since the signature is just the secret hidden behind an addition this also seems to be minimal. In case the native signature algorithm does not allow leaking any information (it is an elliptic curve-point), then this seems to be the minimal possible addition that enables scriptless scripts.

3.1 Recovering MimbleWimble

There are a few simple choices of s that are particularly interesting. First the choice of $s = 0$ means that the public address is zero and this corresponds to a case where no public address at all is associated to an output. This trivially reduces to the standard MW protocol which is included as a limit of this extension. Most transactions can therefore still have no public address and no additional signatures attached to them, but this comes at the disadvantage of making outputs that do distinguishable from ordinary MW outputs.

3.2 Public Verifiable Transactions

Another option is $k = s$. In that case the commit $C = vL$ is just a commit to the value. The possible range for v is of course much smaller than the one for k and this commit does no longer hide the transferred value reliably. It therefore is a 'public' transaction to a public address $P = kG$.

While this jeopardises a lot of the privacy features of MW, it allows for simple verifiable transactions. In any form of verifiable payment the receiver necessarily needs to verify their identity (at least to a third party). In this case the authentication is realised via publishing a public key $P = kG$. We believe that this has advantages compared to identifying via IP or other measures, since new public addresses can be generated by everyone at zero cost.

After the receiver has publicly committed to the public key, anyone can send coins to this public address by creating an output (C, P) where $C = vL$. In doubt of payment the sender can then, also publicly, show the UTXO, or when pruned a Merkle proof thereof. By then revealing v, or signing C to the base point L, it is proven that the receiver can spent that output and the transaction was successful.

This idea can in fact be realised also in normal MW. When the receiver pre-commits to $k'G$, the sender can later verify to a third party the existence and value of the transaction. But this cannot be done in retrospect as it would require the interaction of the receiver and when wanting to prove a transaction to a third party we cannot rely on the participation of all participants. By allowing for an extra field in the transaction this commitment can be implemented directly and immutably on the chain level.

In this way MW obtains a simple on-chain solution for publicly verifiable transactions. And while Bitcoin does store the entire transaction history, this solution still has all of the advantageous scaling properties of MW. All data of this public transaction can eventually be pruned since the only information that needs to be kept is proof that no coins were created in the transaction.

Furthermore public and private transactions can be linked to each other without the need for any additional infrastructure or conversion. A private commit without public address can pay to a public address, or vice versa.

3.3 Verifiable Private and Scripting Addresses

The last option is choosing s independent of k and this provides an output that is associated to a potentially verifiable public address, while the amounts are still hidden. Alternatively the public address might in fact be used to force a Schnorr signature for the use with scriptless scripts as detailed above.

4 Building Non-Interactive Transactions

After studying the basic properties of our extension to MW, we will now look at a few interesting options that are enabled by the use of BLS signatures (independent of scriptless scripts).

It is well known that MimbleWimble transactions are interactive, opposed to most other currencies where the sender can send coins to other parties without their direct involvement. One reason for this is the interactive nature of Schnorr multi-signatures, and is removed by the use of BLS signatures. But this alone is not sufficient to create a non-interactive transaction. For any transaction to be considered successful, the sender must transfer control of the coins and no longer be able to access the funds afterwards.

This is in-fact a non-trivial problem in MimbleWimble, because of the symmetry in signatures between sender and receiver. Any signature that the sender can do on their own naturally points to an output that the sender can still spend, i.e. there needs to be some mechanism to 'delink' the sender from the UTXO. The best that can be done without interaction is creating a 1-2 multi-sig that both parties can spend. Otherwise the receiver's involvement in transaction building is necessary to make sure the sender can no longer spent the output.

In order to change this we need to break the symmetry between sender and receiver and the public address that we have introduced provides this option. Just as in most cryptos, the sender can commit to the public key of the receiver and that process delinks them from the output. But since the signature of the kernel excess does not refer to this public key, we need make a modification of the MW protocol.

Let us assume that Alice wants to pay Bob and she knows one of his public keys $P_B = s_B G$. Now she can construct (and sign) a transaction from (C_A, P_A) to (C_B, P_B), where $C_B = vL$ is just a commit to the value. She computes the kernel offset (assuming that she did not create any new coins) $k_A G - s_B G$ and signs partially with k_A. This is a valid signature to the kernel excess once Bob adds his signature to s_B and aggregates them. And this can be verified by anyone by showing that Alice has in fact signed the kernel offset minus Bob's public key $k_A G$ by verifying that

$$e(\Sigma_A, G) = e(k_A G, M)\,,$$

with $\Sigma_A = k_A M$.

A third party observer can still not verify that no coins were created in this transaction. Alices signature proves that she did not cheat, but Bobs public key might contain some term proportional to L. But in that case Bob cannot sign the kernel and the transaction can never be claimed by him. Only when Bob signs, this becomes a fully verifiable and valid MW transaction.

After receiving the signature from Alice, Bob knows that he can at any moment spent the coins, using a signature to his public key. From this point of view there is no need for Bob to sign immediately, he could wait until later and sign just when he wants to spent that output again, which will anyways require a signature to P_B.

But there is one flaw. Alice cannot spent the output of her transaction, but Alice may spent one of the inputs, since the transaction to Bob is not jet finalised on the chain and in that case the transaction to Bob would be invalidated. But since we have public keys, we can enforce the same concept used in most other chains. We can consider an input as spent, as soon as only signatures to the public keys of outputs are missing and record this on the chain.

We introduce the concept of a 'hanging' transaction, a transaction that is validated up to public keys of the outputs. In fact this concept is not something new, all Bitcoin outputs are hanging transactions in this notion. Such a transaction can already be committed to the chain, marking all inputs as spent. These are removed from the UTXO set and instead stored as inputs to the hanging part. The outputs are not yet finalised outputs, but 'hanging' outputs.

The general chain verification cannot run over the hanging outputs since the proof that no money was created is still missing. Instead it runs over all confirmed outputs and the inputs to hanging segments. Once the missing signatures are added, the inputs are finally removed and the outputs are finalised. Now, when Alice sends coins to Bobs address, this creates a hanging output, but Bob knows that only he can spent these coins. He may claim them immediately or later when he wants to spent the coins.

Let us look at this again more formally. The chain consists of a UTXO set and an accumulated kernel excess (assuming BLS aggregation). And the sum over all UTXO commitments $C + P$ can be verified against the signed kernel excess. Now assume that a hanging transaction is created using some of these UTXO's as input. To do this the inputs are signed and the only outstanding signatures are to the new outputs public keys. We can therefore collapse the inputs and values into one aggregated intermediate excess κ_I plus a valid signature

$$\kappa_I = \sum_{\text{inputs i}} P_i + C_i - \sum_{\text{outputs j}} C_j \ ,$$

and a set of hanging outputs (C_j, P_j). All that needs to be stored for such a hanging transaction is the set of outputs, the intermediate offset κ_I belonging to this hanging section plus a valid signature to it and the sum over the input commits $I = \sum_{\text{inputs i}} C_i + P_i$ to commit to the number of coins in the hanging part. Any additional information related to the inputs can be discarded. A node can still verify the entire chain by checking that the sum over all confirmed outputs plus I validates against the total signed kernel offset.

When the public key of an output is signed, it can be added to the intermediate offset κ_I. Once the hanging section internally verifies, i.e. the sum over all outputs minus the input I matches κ_I and is signed, the hanging outputs become confirmed and the internal excess κ_I is added to the overall kernel excess. All inputs are removed and the outputs become fully validated outputs.

In essence, once

$$I - \sum_{\text{outputs j}} C_j + P_j = \kappa_I \ ,$$

plus a signature to κ_I is present, the hanging part becomes a normal MW transaction and the signature to κ_I is aggregated into the basic kernel excess.

It is not even necessary to store the hanging UTXO's differently from normal UTXO's. Given a signed kernel excess, a number of signed internal excesses plus input commitments I there is only one possible combination that will fully validate.

4.1 Locking Up Commits and Security

There is one weakness in hanging transactions. An attacker can lock up the outputs of other participants without their involvement. Let us look at a commit $C = vL + kG$ without a public key. The attacker can construct a payment to the output $(0L, vL + kG + k'G)$, or many other combinations where kG appears as a part of the public key of the attacker. The signature of the sender to their private key k is no longer required as it is promised later as a part of the signature to the attackers public key. The attacker may never claim the outputs, but is able to lock up commits.

There exists a simple solution to this problem, analogous to other currencies such as Bitcoin. In order to create a hanging transaction, the sender needs to verify themselves. And this can be done via the public key that we have introduced. Security is restored by allowing hanging outputs only from addresses that do have a public key attached to them, requiring a separate signature that an attacker cannot provide.

This means that non-interactive transactions can only be performed from a public to another public addresses, while the amounts sent could still be hidden. Security requires storing all chain data and signatures as done in bitcoin, creating a verifiable past. While in bitcoin this past has to reach back until genesis, we can eventually prune it. The proof that no coins were created is contained within the aggregated kernel excess and the validity of the UTXO set (including hanging transactions) is validated against this excess. The history is only needed to prevent attacks avoiding signatures to public keys. We can prune it, forcing attackers into a deep reorg. And by choosing a sufficiently deep history this can be made practically impossible (and if it happens there are bigger problems).

New nodes that join the network at a later stage need to trust other nodes that there was no cheating done in the past history where they only have the compressed chain. But that trust is in fact very limited since they can still verify the total supply of coins. And in case somebody did spent without a public key, that does not really concern the new participant. And from that point onwards, the new node can verify that all transactions are following the consensus rules and can for fully trust the chain for all transactions it is involved in.

5 Conditional Payments

We have already seen how hanging payments can be used in combination with the possibility to aggregate BLS signatures. But the idea of hanging transactions in fact allows for much more flexibility. One such option are conditional payments.

Suppose Alice wants to pay Bob on the condition that Carol approves. Alice can now create a transaction with two outputs. One to Bob (public or private) that she (co-)signs together with Bob. The second payment is to Carols public address and contains a value of zero. The outputs of this transaction are now hanging and depend on Carols signature. We have created a conditional transaction.

5.1 Transactions between Hanging Outputs

In fact, we can even have further transaction within the hanging parts of the chain. Let us assume a hanging subpart was created with an input of I and a signed internal offset of κ_I. Even though the hanging outputs are not fully verified and depend on additional signatures before they can be merged into the main branch, we can make transactions within the hanging parts.

Let us assume we have a hanging output (C, P) and want to transfer it to create a second hanging output (C', P') locked behind the same outstanding signatures. We compute the offset $C + P - C' - P'$ and add the corresponding signatures. Now we remove (C, P) from the hanging UTXO's and replace it by (C', P') and aggregate the signature to the transaction offset into the offset of the hanging part κ_I. In this way we have created a new set of hanging outputs that becomes valid when the missing signatures to the open public keys are committed, i.e. we have successfully transacted within a hanging subpart.

While verifying the main chain is checking that the signed offset validates against the UTXO's, we find the same verification rule internally within the hanging part. Internal transactions are validated by checking that

$$\kappa_I = I - \sum_{\text{outputs j}} C_j - \sum_{\text{signed keys j}} P_j \, ,$$

up to the remaining unsigned public keys and that there exists a valid signature to κ_I.

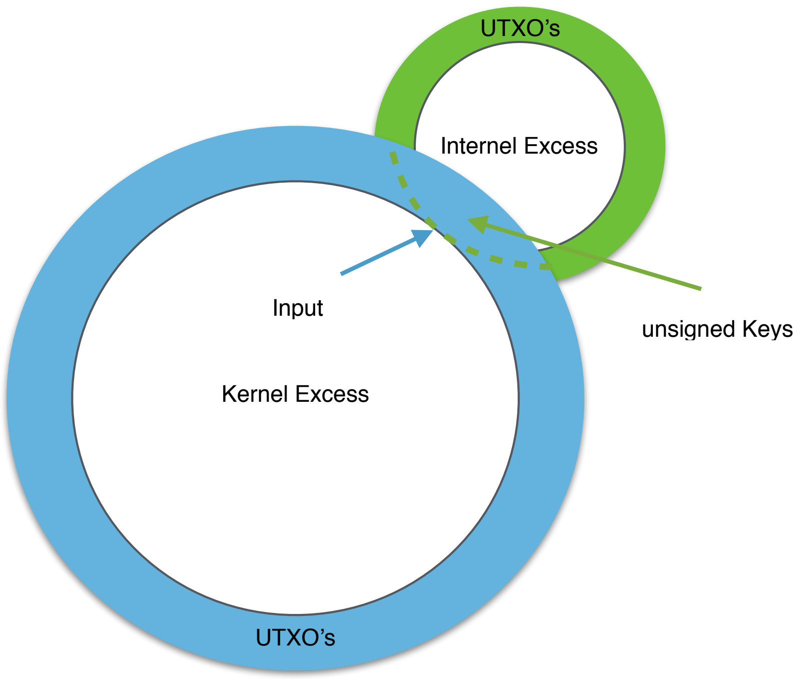

Figure 1: Visualisation of a hanging child-chain. The blue part is the parent chain, where the verification criterium is that all UTXO's (along the blue contour) add up to the kernel excess. In case of a child-chain the parent verification does not include the child's UTXO's in green, but the input to the child chain. The child chain is itself verified to its internal kernel apart from unsigned public keys. Once all keys are signed the input to the child chain can be dropped, the excesses aggregated and child UTXO's promoted to fully validated UTXO's. The two chains combine into a single larger one.

The hanging part itself is a fully verifiable MimbleWimble chain, only that the number of coins is not derived from coinbase transactions, but encoded in the initial input to the hanging part I. Hanging parts are essentially child-chains to the main chain.

It is even possible to add another conditional payment behind an already hanging output, creating a child-chain that itself is based on another child-chain. And once the missing signatures are provided it can be merged into its parent-chain, or on the other hand once the parent chain validates, the hanging child-chain becomes a child-chain to the main chain instead.

Potentially many more complex structures could be created, such as payments between different child-chains, but these become conceptually much more difficult. Instead allowing a few layers of hanging transactions remains relatively simple. It should be noted that this comes at no additional scaling cost to the main chain. All UTXO's have the same size independent of being connected to the main or child-chain. Each child-chain just adds one initial input I and one aggregated offset κ_I plus a signature.

6 User Issued Assets

As can be seen above what happens in a hanging part is described by a small autonomous MimbleWimble universe, that individually verifies that no new money was created within it and can have internally valid transactions of hanging coins.

We can take this to the next level by creating a commit that can never be merged to the main chain because it creates a certain amount of coins. This can be proven publicly and verifies that these coins can never be aggregated to the native MW chain and stay within their hanging sub-chain forever. These coins can now be transacted and used just like regular coins. They are in fact a user-issued token living on top of the MW chain.

Let us assume that Alice wants to launch her private version of 'AliceGrin'. First she creates a public output with a value of zero $(0L, sG)$. Then from there she creates a hanging transaction $(0L, sG) -> (100L + k_A G, 0) + (0L, -100L)$ and signs the transaction to everything but her pretend public key $-100L$, requiring only s and k_A. This creates a valid hanging transaction with the first output being of value 100 and private key k_A, as a private transaction without an address (but an address could be added if desired). The second output should be understood as value zero commit with private keys $k' = 0$ and $s' = -100L$. She pretends to not have created fake coins, and promises to sign $100 * L$ to the base point G in the future to prove this.

Of course she can never sign to this public key, unless she knows a relation between L and G. This means that this hanging child-chain can never be merged into the parent chain. And she can also prove to anyone that this is the case. The input to the hanging part is $I = s * G$ and by signing it with her key s she proves that no native coins were committed to the child-chain. Then she can prove that the conditional output is in fact $-100 * L$, by simply revealing the number of created coins. This confirms to everyone that there are 100 new AliceGrin issued at the genesis of this child-chain. It also implies that her hanging output $(100L + k_A G, 0)$ has a value of exactly 100 AliceGrin.

Furthermore anyone can see that the public key that needs to be signed to merge the child-chain cannot be signed.

From this point onwards all transactions within her child-chain are internally verified and the amount of coins within the child-chain will remain at 100 forever. She has successfully created a user-issued token that can be transferred both publicly and privately using MW.

7 Outlook

We have presented a simple extension of the MW protocol that enables verifiable public and non-interactive transactions, while allowing for full compression of the chain-data into a residual kernel excess. The protocol includes MW as a subset and keeps all privacy features enabled by MimbleWimble.

We further provide a simple method to realise conditional payments at the native chain level. It is possible to attach an additional key to any payment, that needs to be signed by a Schnorr signature, allowing a leak of information and scriptless scripts. At the same time the advantageous scaling properties of BLS are kept as all Schnorr signatures can eventually be pruned and only a single aggregated BLS kernel offset remains. This is secure as long as the Schnorr signatures are kept for a sufficiently long time, such that a deep reorg is unlikely. Note that Pedersen commits are secure assuming the discrete logarithm assumption, only the signatures to the kernel excess in BLS are weaker.

We further find that conditional payments enable simple user-created assets that only need a minimal overhead. Each asset lives on a child-chain that apart from the UTXO's has one signed additional kernel offset and one initial commit. The creation of a new asset thus takes a similar amount of space as a normal transaction.

These features are enabled by two changes to the MW protocol, first the use of BLS signatures and second the introduction of public keys. Not all of the changes rely on both and a lot of the ideas can be realised having only one of these modifications.