Mimblewimble

Originally published on 2016-07-19 by Tom Elvis Jedusor

Incredibly powerful protocol

- Built-in anonymity
  - Transactions are confidential
  - No addresses, public identities, or etc.
  - Obfuscated transaction graph
    - Several challenges however yet to be solved to guarantee this
- Great scalability
  - No high CPU/memory demand, mobile/embedded-friendly
  - Modest transaction size
  - Transaction cut-through
- Despite its versatility, it's formulated entirely in terms of "elementary" ECC
  - No complex constructs like bilinear pairing, zk-SNARK, or etc.
  - No trusted setup needed
  - Relies solely on the hardness of the discrete logarithm problem

What’s different in MW

- No addresses
  - Each UTXO has a secret key, and it belongs to whoever knows it
- Transaction
  - No scripts in the blockchain
  - To build a valid transaction entities must collaborate.
    - i.e. it’s an interactive process
    - This is where complex schemes may be negotiated (a.k.a. scriptless scripts)
  - Once built, the transaction is obscured, and basically only proves that:
    - No illegal inflation, i.e. money created from nothing
    - Authorization verification. In order to build a valid transaction the secret keys of all the input UTXOs must have been used.
      - This is the meaning of ownership – ability to spend it.

UTXO encoding

- Two generator points: G,H (for starters).
  - Must be “nothing-up-my-sleeve“ – i.e. their relation must not be known. A brief generating scheme must be specified (such as hashing strings).
- $C = \alpha \cdot G + v \cdot H$
  - α- blinding factor, a uniform (pseudo)random.
  - v - Value
- Pedersen Commitment (linear combination of those generators).
  - Hiding: the value of v is blinded
  - Binding: impossible to substitute other values for α, v.
  - Homomorphic: $C(\alpha_1, v_1) + C(\alpha_2, v_2) = C(\alpha_1 + \alpha_2, v_1 + v_2)$

Naïve transaction

Alice owns an UTXO containing v_A, wants to send Bob v_B, and receive a change v_A-v_B. This is their transaction:

- $C(\alpha_A, v_A) \rightarrow C(\alpha_A - \alpha_{A`}, v_B) + C(\alpha_{A`}, v_A - v_B)$

The verifier checks:

- $\sum(Input\ UTXOs) = \sum(Output\ UTXOs)$

Is it a good scheme? Of course no.

- Illegal inflation verification – FAILED.
  - no verification that $v_A \geq v_B$, output UTXO may contain "negative" (overflown) value.
- Authorization verification – FAILED.
  - Anyone can spend UTXO without the knowledge of its opening (the blinding factor and the value):
  - $C(?\,,?) \rightarrow C(\alpha, v) + C(? - \alpha, ? - v)$

  (The second transaction output is a "fake" UTXO, its opening is unknown.)

Rangeproof

- A zero-knowledge non-interactive proof that proves that the value of the UTXO is within a limited range.
- Practically for a 256-bit ECC the value of the UTXO is restricted to 64 bits, which is both a fairly large number to encode the value, and far enough from the overflow risk when large number of UTXOs are summed.
- In addition to restricting the value of the UTXO, it can also be seen as a cryptographic signature, which is impossible to create (with non-negligible probability) unless the opening of the UTXO is known.
  - Prevents “tampering” with existing UTXO (adding/removing value or blinding factor).
  - Prevents creation of “fake” UTXOs with unknown opening.
- MW relies on Bulletproofs
  - Pretty sophisticated, yet implemented in terms of “elementary” ECC.
  - Dramatically smaller than other similar schemes (but not on par with zk-SNARK of course).
    - 64-bit rangeproof in terms of 256-bit ECC is encoded with 674 bytes.
  - Supports multi-signature (would require 3 iteration cycles).
  - Modest CPU load
  - Seems to be feasible to implement on embedded devices (HW wallets)
  - Verification is faster than signing
  - Multiple verification (like verifying a block) is speeded-up.

Another attempt

- $C(\alpha_A, v_A) \rightarrow C(\alpha_A - \alpha_{A`}, v_B) + C(\alpha_{A`}, v_A - v_B)$
- Rangeproofs are attached to all the outputs

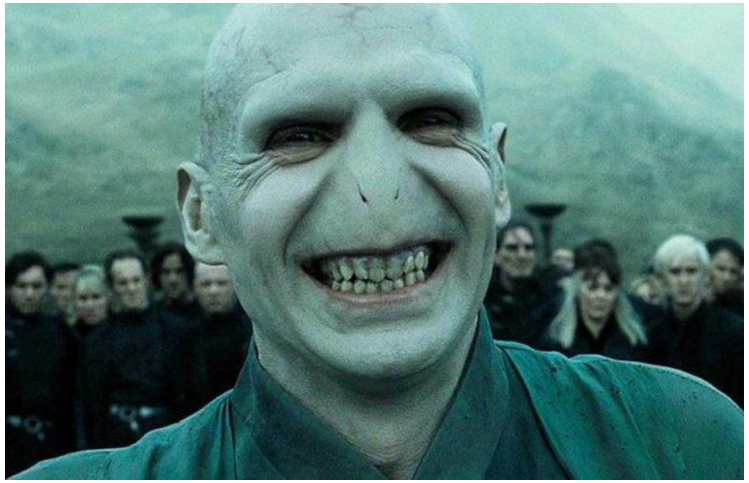
The verifier checks:

- $\sum(Input\ UTXOs) = \sum(Output\ UTXOs)$
- Rangeproofs for all the outputs are valid

Still not good enough:

- Authorization verification – FAILED.
  - Such a transaction is reversible
  - Alice knows the opening of Bob's UTXO, hence she can spend it any moment without Bob's permission.
    - No ownership transfer
  - This is inherent property of transactions which sum to zero, regardless to how many outputs there are.
    - Means, if Bob creates several outputs to receive v_2, Alice still knows their overall value and the total blinding factor, and can spend them all at-once.

Transactions with excess



- $C(\alpha_A, v_A) \to C(\alpha_B, v_B) + C(\alpha_{A`}, v_A - v_B) + \Delta\alpha \cdot G$
- Bob picks a random α_B, and it's unknown to Alice
- $\Delta\alpha = (\alpha_A - \alpha_{A`}) + (0 - \alpha_B)$

The $\Delta\alpha \cdot G$ is the transaction excess. It must be signed (Schnorr's signature), which proves that:

- It only contains the blinding factor, no Value is hidden
- The creator(s) of the excess must know the transaction excess ($\Delta\alpha$).

How the transaction is negotiated

- In a simple scenario Alice reveals $\alpha_A - \alpha_{A`}$ to Bob, and he completes the transaction
- In an advanced scenario – no one reveals blinding factors. Instead Alice and Bob co-sing the transacton excess (Schnorr's multi-signature)

The verifier checks:

- $\sum(Input\ UTXOs) = \sum(Output\ UTXOs) + \sum(Excesses)$
- Rangeproofs for all the outputs are valid
- Excess(es) are properly signed

Is this is a robust system? Are there unnoticed pitfalls?

- Illegal inflation verification.
  - Based on the homomorphic property of Pedersen Commitments
  - Rangeproofs prevent overflow attacks
  - Excesses are signed to prove (in particular) no money is hidden in the excess.
- Authorization verification.
  - All the transaction elements (UTXOs and excesses) are signed, to prevent tampering and creation of unknown objects.
  - Outputs are known – means inputs must be known as well.
  - Irreversibility of a transaction is due to the fact that excess may only be created in a transaction, and never spent.

Transaction kernel

- Contains the excess and additional validation rules and parameters
  - Public excess $\Delta\alpha \cdot G$
  - Optional fields (timelock parameters, transaction fee, hashlock preimage, etc.)
  - Schnorr's signature.
    - Signs all the kernel contents (to prevent tampering)
    - The public key is assumed to be $\Delta\alpha \cdot G$.
    - The private key is naturally $\Delta\alpha$.
- Unlike UTXOs kernels can only be created, and never spent
  - This has an impact on the system scalability (dead weight accumulated).
  - Since they're guaranteed to stay - they may be used in various ways:
    - Prove the fact of the transaction: It's built collectively by all the transaction parties, and cannot be crafted/modified otherwise.
    - Implicitly reveal secret data to the transaction parties upon successful payment (private keys, hash preimages)
    - Flag transactions for 3rd party (more about this later)

Block

- Merged transactions is also a valid transaction
- Block is essentially one big transaction with many inputs and outputs.
- All the transaction elements (inputs, outputs, kernels) are sorted to obscure the original transaction graph

Is the transaction graph truly obscured? Well, No.

- Transactions are mixed, but not “dissolved”
- All the elements are blinded and signed – means it’s impossible to combine them non-interactively
- Trying different combinations it’s still feasible to puzzle out the original transactions.

Transaction Offset

- $C(\alpha_A, v_A) \rightarrow C(\alpha_B, v_B) + C(\alpha_{A`}, v_A - v_B) + \text{Kernel}(\Delta\alpha` \cdot G) + \beta$
  - Whereas $\alpha_A = \alpha_{A`} + \alpha_B + \Delta\alpha` + \beta$
  - Means – the transaction excess $\Delta\alpha$ is split into 2 parts
    - $\Delta\alpha`$ - goes into kernel (as before)
    - β - just revealed unencoded (scalar).

The verifier checks:

- $\sum(Input\ UTXOs) = \sum(Output\ UTXOs) + \sum(Excesses) + \beta \cdot G$

There is finally a transaction element, which can be merged (simply summed)!

- Doesn't break the robustness of the system, since offset – is a preimage. It can't conceal money or compensate for unknown blinding factors.
- Once the transactions are combined, their offsets are merged, and this is irreversible.
  - It's not possible anymore to split a combined transaction into independent components.
- Block contains multiple inputs, outputs, and kernels (sorted in an unambiguous way), and a single offset
- Transaction graph is now truly obscured (almost…)

Transaction cut-through

- Block is a big transaction
- Multiple blocks can be merged as well, to create one big transaction
- Output UTXOs that are created and then spent can be removed completely
  - Means – combined blocks tend to be smaller
- The whole blockchain can be combined into a single huge block, with only outputs that are unspent yet.
  - Dramatic scalability improvement
  - Some information is lost (obviously). But it’s still possible to verify that the combined block describes a valid system transformation according to the rules.
- Each original block header contains a commitment to the kernels (only).
  - It’s sufficient to prove the authenticity of the compressed blockchain, after redundant inputs/outputs were removed
  - Means – all the original transactions were included

Transaction broadcast

- Obscured Transaction graph is of <u>critical</u> importance
- Naïve broadcast scheme immediate reveals the transaction graph!
  - A single malicious node immediately gets all the original transactions
- Known solutions
  - CoinSchuffle, ValueSchuffle
    - Seems promising, but maybe cumbersome in practice
    - Requires large group of unrelated users to collaborate
    - DoS attack is easy
    - Attacker may create many malicious users “for free”
  - Dandelion(++)
    - Was designed to conceal the identity, regardless to the transaction graph
- A simple solution, which *may* be practically good enough
  - Modified Dandelion, with partial transaction merge during the stem phase.
  - No hassles for the users (actually transparent, may complement CoinSchuffle and etc.).
  - No guarantee of expected behavior, but non-conforming Nodes can be identified
  - Disadvantage: easy to abuse the transaction fee.

Transaction Negotiation

- Secure channel with authentication is a must
- P2P – ok, but
  - Requires users to be online simultaneously
  - Cumbersome in some networks (NAT & friends)
  - Identities can be traced by traffic analyzers
- Secure BBS system
  - Separate from the blockchain, but may use the same network addresses
  - Solves network configuration hassles
  - Asynchronous communication
  - Messaging via “addresses”
    - May be (and usually are) temporary for one-time usage
    - Have nothing to do with the blockchain
  - E2E encryption, To/From addresses are not leaked
  - For obfuscation: many unrelated negotiators exchange messages over the same channel
    - Every user receives all the channel messages, but is able decrypt only the intended ones

Extensions

- Non-interactive payments
  - Allows to receive payments non-interactively (without the need for negotiation)
  - Supports fixed values only (and their combinations)
  - Needs 2 kernels for such a transaction
  - Requires ***Kernel Fusion***, to prevent separation of the transaction into its donor and acceptor parts.
  - Requires proper handling of multiple identical UTXOs (implemented)
- Auditable Wallet
  - Applicable for business, obliged to operate w.r.t. regulations
  - As transparent as possible to appropriate authorities
  - Preserve the anonymity to others
  - Disclose only the required information, without compromising other parties
  - Allow the auditor to fully reconstruct the transaction graph of this wallet
- Confidential assets
  - Very straightforward to implement in the context of MW
    - Just add more H-generators and tweak the bulletproofs!
  - What's unclear yet:
    - How the emission of other assets should be regulated?

Thank you

- For more information please visit our project sites:
  - https://github.com/BeamMW/beam/wiki
  - https://www.beam-mw.com/