BEAM description. Comparison with classical MW

MW at a glance, brief description

- There are no inherent addresses in MW. All the values are encoded via UTXOs, and transactions essentially are transformations of existing UTXOs into new ones.
- UTXO is encoded via Pedersen commitment (an opaque point on the elliptic curve) + rangeproof, which proves the following:
    - The encoded amount is within the allowed range.
    - The creator of the signature knows its opening, i.e. the *blinding factor* and the amount.
    - UTXO was not modified (i.e. tampered with) after the rangeproof was created. So that the rangeproof acts also as a signature.
- Knowledge of the UTXO opening (*blinding factor* and the amount) is necessary and sufficient to authorize the transaction that consumes it. In other words it means ownership.
    - This is also correct for arbitrary set of UTXOs, whereas knowledge of opening of each individual UTXO is not required, strictly speaking.
- Because of the above during the transaction the overall *blinding factor* of inputs/outputs should be different among different participants, to ensure proper ownership transfer. Because of this transactions don't sum to zero, there should be an *excess*.
- The excess is equal to the input/output difference, and is represented by a *transaction kernel*, signed by the Schnorr's signature, which proves the following:
    - The excess consists purely of *blinding factor*, it doesn't carry any amount.
    - The *blinding factor* is known collectively (but not necessarily individually) by the participants that signed it.
    - The excess was not modified (i.e. tampered with) after it was signed.

The robustness of the MW, besides of the general cryptography, is based on the following essential details:

- All the transaction elements (UTXOs and kernels) are signed to prevent tampering and creation of unknown objects.
- Irreversibility of a transaction. Once entered the blockchain the transaction can't be reversed by just swapping inputs/outputs. This is due to the restriction that transaction kernels may only be produced in a transaction, and never consumed.

What's different in BEAM

Note about UTXO uniqueness

BEAM supports duplicate UTXOs.

What are the implications if the user creates several identical UTXOs (identical amounts and *blinding factors*)?

Speaking technically, there is no problem with this w.r.t. transaction verification pipeline as long as it's implemented carefully, keeping in mind such a possibility. But this will have implications when a user would like to spend such an UTXO. Suppose **A** owns several identical UTXOs and consumes it in a transaction that transfers funds to **B**. If **B** notices that there are several such UTXOs (and this is a public info, visible in blockchain), then he can repeat the same exact transaction again, and get more funds from **A**.

In order to prevent such an unauthorized spending, **A** should only use such an UTXO in conjunction with at least one unique UTXO. So that the set of all the transaction inputs is unique.

Obviously this puts restrictions on transactions, and there's no point in doing this in normal circumstances, but, as we'll see later, this has important use-cases.

Confidential and public UTXOs

BEAM supports both confidential and non-confidential UTXO signatures.

According to MW UTXOs are signed with *Bulletproofs*, which are (relatively) compact zero-knowledge rangeproofs. In BEAM the UTXO can either be signed by a *Bulletproof*, or by a compact non-confidential signature, such that the amount is revealed, yet the *blinding factor* is not. So that everyone sees the encoded amount, and can verify that it's within the allowed range, but the ownership of the UTXO is preserved.

In most of the transactions *Bulletproofs* should be used, however there is no restriction, both signatures are permitted. On the othe hand, the coinbase UTXO (miner reward for block closure) must be signed with non-confidential signature, because it has extra restriction applied, so that everyone can see that the coinbase amount is indeed encapsulated in this UTXO, and not hidden within another confidential one (thus bypassing the restrictions).

Explicit UTXO incubation period

In BEAM there are both implicit and explicit incubation periods, whereas the term *incubation period* is in terms of blockchain height, and denotes

the minimum number of blocks created after this UTXO entered the blockchain, before it becomes liquid, i.e. can be spent in a transaction.

Implicit incubation period in determined by the system rules (such as for coinbase UTXO), whereas explicit can optionally be specified in addition. It's accounted in the UTXO signature, to prevent tampering. Note: this is different from the timelocked transactions that can't be put in a block until the specified blockchain height is reached.

One-side transfers

BEAM supports one-side transfers (though with somewhat limited functionality).

Suppose **A** wants to transfer funds to **B** without **B** being directly involved in the transaction negotiation. It's not a trivial task, because MW essentially needs mutual cooperation.

For simplicity let's assume that **B** expects to receive a known amount from **A**, and he prepared the UTXO in advance, that is supposed to collect the transferred funds. The challenge here is naturally the *blinding factor* of this UTXO. If **A** knows it, then he can easily construct such a transaction, but, as we know, once **A** knows the opening of this UTXO, he essentially owns it, and can anonymously spend it later at any moment.

BEAM uses the following scheme: **B** expects to receive a known amount from **A**. He creates an appropriate UTXO (with the signature). Instead of revealing the *blinding factor*, he creates a *transaction kernel* that compensates for it. Both the UTXO and the compensatory kernel are known to **A** (or whoever wishes to transfer the funds), so that he can construct the transaction. But this doesn't end here, let's see why.

Naive scheme

A wishes to transfer funds to **B**, according to the UTXO and compensatory kernel published by him. He picks his input UTXOs, creates the appropriate change output UTXOs, creates a transaction kernel that compensates for his input/output *blinding factor* difference, and appends **B**'s UTXO and kernel. So far there is a perfectly valid transaction with 2 kernels.

Why this isn't good enough?

Let's recall the knowledge of the *blinding factor* essentially means ownership. The above constructed transaction may look ok, but it actually consists of 2 parts, each of which has zero (or other agreed known value) overall *blinding factor*. Whereas the amount of each part is non-zero: there are donor and acceptor parts.

Suppose such a transaction is sent to a malicious node. It can notice that the above transaction can be split into donor and acceptor parts: either he has the info that **B** expects to receive funds and recognizes its UTXO and kernel, or it may just try different split combinations with different transferred amounts (brute-forcing the amount is easy, since it's limited to just 64 bits, and not a uniformly-distributed random).

Once the transaction is split into donor and acceptor, the malicious node is free to substitute his own acceptor (UTXO + appropriate kernel) instead of **B**'s.

Improved scheme

BEAM solves this issue by introducing a *kernel fusion*. A kernel can optionally contain an explicit reference to another kernel, so that it's considered valid only of both are present. This prevents substituting different acceptor part in the above scheme.

Technically instead of inter-kernel references, kernels in BEAM can optionally contain nested kernels. So that a kernel can explicitly contain several nested ones, which are accounted in the outer kernel signature (removing or replacing any of the nested kernels would break the signature of the outer one). There is a restriction on the maximum kernel recursion depth (to protect against malicious kernels causing stack overflow or etc.).

Naturally during the verification all the kernels are verified, and the overall excess is summed.

Multiple one-side transfers

Since there are no restrictions on the number of one-side transfers with the same donor UTXO + kernel, this potentially leads to duplicate UTXOs in the blockchain. As we've said already this is allowed in BEAM, however this applies restrictions on how they can be spent.

Arbitrary amount in one-side transfers

Basically unsupported, since it's impossible to create a signed UTXO with arbitrary amount, without knowing its *blinding factor*. However the receiver may create and publish several pairs of UTXO + kernel for common amounts, to approximate to some extent any amount with reasonable UTXO count.

Kernel features

Timelocked transactions

BEAM transactions can be timelocked from "both sides". This is achieved by kernels with optionally specified lower and upper blockchain height thresholds. Whereas the use of the lower threshold is described well in GRIN documentation, the use-case of the upper threshold is equivalent to a "Cheque validity period", and is actually very useful in practice.

In practice when a transaction is sent to a node it may take time until it becomes visible in the blockchain. And if it doesn't show up after reasonable period - this creates an uncertainty for the participants: is it lost or just delayed, would it eventually appear in the blockchain after indefinite amount of time?

By defining the upper timelock threshold this uncertainty is resolved: if the transaction didn't appear in the blockchain up to the specified threshold, it will definitely not appear later, and may safely be considered aborted.

~~Embedded contract~~

~~BEAM kernels may optionally contain a *signed contract*, which consists of a hash of an arbitrary message (not stored in the blockchain), the signer public key (or sum of multiple public keys, if it was co-signed by multiple participants), and the Schnorr's signature. The contract message and signer(s) public key are accounted in kernel signature (to prevent tampering), however the contract signature is constructed last, after the kernel excess is signed, and accounts for it. Naturally the kernel is considered valid only if it has both correct excess and contract signatures. This proves the following:~~

- ~~All the transaction participants signed the transaction kernel, while they were aware of the contract.~~
- ~~The contract is co-signed after it's embedded within the specified kernel. Means - the contract was signed for a specific transaction, and it's impossible to substitute it to another transaction.~~

~~This scheme allows to bind a contract (arbitrary message) and signer(s) public key to a specific transaction. Existence of such a kernel in the blockchain proves that the payment took place while participants agreed on the specific message. This can be used in later investigations or etc.~~

~~The signer(s) public key is explicitly specified in the contract. If participants don't want to reveal it (until they actually want to reveal all the details during future investigation) - it's possible to hide it. For this they may create a *virtual blinding co-signer* with arbitrary private key, known for all the participants, but not to the others.~~

Correction: It was decided recently not to keep the signed contracts on-chain. Instead the contract (or whatever it's called) is kept off-chain, but it contains a *reference* to the transaction kernel (whereas the reference is the unique kernel identifier in the system). During the negotiation the party interested in the contract will not co-sign the transaction kernel until the contract, containing the reference to the being-constructed kernel, is properly signed.

In this scheme the signing of the contract and the kernel is **not** atomic, but there's no problem with it, because the signed contract contains the reference to the specific kernel, which **must** be present in the blockchain to confirm the fact of the transaction. So that the final proof for whatever investigation would consist of a signed contract, and the merkle proof to the referenced kernel in the current blockchain state.

Hashlocked transaction

BEAM transactions can be hashlocked. The kernel may optionally contain a "Hash preimage" field (constrained to 256 bits). If present - it's hashed, and then accounted for in the kernel signature.

The primary use of this feature is "transaction in exchange for the preimage of a known hash". This can be used to implement atomic swaps (as described in HTLC).

However there are more uses for this. Since it's just an arbitrary value, which can be anything, it may serve as a placeholder for important information for 3rd party. More about this here.

Eliminating transaction kernels

Unlike classical MW, in BEAM it's possible to consume kernels in a transaction, yet without compromising the *transaction irreversibility* principle. This is valuable, because unlike UTXOs kernels play no role in future transactions and are actually a "dead weight" in the system. In order to accomplish this we apply additional restrictions on the input kernels, such that the verifier can:

1. Verify that for each input kernel there's a corresponding output kernel, that can only be signed if its creator(s) know the opening of the input kernel.
2. Such a transaction is irreversible "on its own". Means - input and output kernels can't be exchanged.
3. The above rules must also hold when transactions are merged and intermediate outputs are fully deleted.
4. In addition the output kernel should remain compact, and not inflate with every such a transaction. There's no benefit of keeping one large kernel vs many small ones.

Scheme

Kernels in BEAM can optionally contain an explicit *multiplier*, a 64-bit integer, considered 1 by default unless specified. It's accounted in the kernel signature (to prevent tampering).

During the verification stage, the *effective* kernel excess is considered to be the standard kernel excess, multiplied by this factor. In addition the verifier must ensure that for every input kernel there is a corresponding output kernel, which:

- Has the same unmultiplied excess
- Has a multiplier which is **greater** than that of the input kernel.

It's easy to see that such a scheme conforms to the rules 1-4.

1. In order to co-sign the new kernel, the participants **must** know the multiplied blinding factor, hence they know the unmultiplied one. Means - those are the same participants that signed the input kernel.
2. Such a transaction is irreversible, since the output multiplier must be (strictly) greater than the input.
3. When such a procedure repeated many times, and intermediate outputs are deleted - we have the same unmultiplied excess correspond ence, and the same inequality for multipliers.
4. The kernel remains compact, its size is independent of the reuse count (up to order of 2^{64}).

Remarks

How to encourage this scheme

Naturally users are not obliged to consume their old kernels, and are free to create new ones (even several kernels) with every transaction. However we'd like to encourage this scheme to keep the blockchain compact. Hence we may define an implicit refundable kernel fee. So that consuming old kernels will be beneficial to the users. Optionally this fee may depend on the kernel size (the size of a single kernel is limited, but it may contain arbitrary number of nested kernels).

How to use it in practice

This scheme can be used by the same set of users (2 in the most common scenario), which perform several transactions. On the first transaction each peaks a random `k` used to construct the first transaction. On the consequent transactions they decide to reuse the older kernel and increment the multiplier `m` for the new kernel.

Degrees of freedom for newly-created UTXOs

The total blinding factor of each consequent transaction is pre-defined (initial excess multiplied by the multiplier difference). It may seem that due to this fact the users are restricted to pick specific blinding factors for their UTXOs (hence - making them less confident and easier to track), but this is not so. The transaction, in addition to the UTXOs and kernels, contains also an arbitrary *offset* - the arbitrary non-encoded blinding factor, which is summed when transactions are combined. Using this degree of freedom users can pick arbitrary blinding factors for the newly-created UTXOs.

Confidentiality consideration

One obvious consequence is that anyone can see that the same set of users performs several transactions. However it's impossible to identify and track the appropriate UTXOs without actually seeing all the appropriate transactions. I can't see other issues.

What if the user wants the kernel to remain forever?

In fact kernels may encode some information useful for users. For instance, it may be a co-signed contract, which users may use later to prove the payment. In this specific case users should wait until the transaction becomes visible in the block, and obtain the Merkle proof for this kernel. Once they have it - they can safely reuse it. Whereas the kernel itself will eventually be deleted from the system, they still will be able to prove it was there. The proof consists of two parts:

1. Proof that a specific kernel was in the specific system state (blockchain height + hash).
2. Proof that this older state is a part of the present blockchain.

This is useful if non-unique UTXO should be spent

As we've said, spending a non-unique UTXO is tricky, and possible only given the overall set of transaction inputs is unique, i.e. contains at least 1 unique object. But if the kernel is consumed in the transaction, then we get the uniqueness automatically, since this kernel is actually a transaction input, and is unique!

Block headers in BEAM

Block header in BEAM consists of the following:

- Reference (Hash) to the previous block
- Actual system definition Hash
- Other parameters: blockchain height, timestamp, etc.
- Proof-of-work

The overall block header Hash is computed from those parameters (excluding Proof-of-work).

The *system definition Hash* is a hash root of a tree constructed from the following distinct hash trees:

- Radix tree containing UTXOs and their maturity (their liquidity height), with duplicate count
    - Means - an entry for every UTXO + maturity pair, with the count how many such UTXOs with exactly this maturity.
- Radix tree of Transaction kernels (duplicates are not allowed)
- MMR tree of the inherited previous block header hashes.

To minimize the header size it only contains one Hash value for the system definition (although it's constructed internally from distinct trees). Still it's possible to get the Merkle proof for everything. So that the user can get a Merkle proof for a specific UTXO (in case of duplicity - a full report with maturities and counts), and for a specific transaction kernel. In addition it may get a proof for the previous system state, which is useful for SPV-clients: in order to verify that the blockchain didn't revert beyond the point previously seen, the SPV client just receives the most recent block header, and the Merkle proof for the last block header hash it knows about.

Radix-Hash tree

BEAM node uses a Radix-Hash tree (a variation of the Patricia tree) with lazy-evaluation principle. It's a binary search tree, whereas the contained objects are represented by the tree leafs, and its internal state is fully determined by the contained objects only, and independent of the insertion/deletion history, i.e. order in which objects entered/left the tree. Tree junction (non-leaf tree node) contains a cached hash of the children with the "dirty" flag, so that whenever tree is modified, the appropriate hashes are invalidated, whereas others are not. This allows the following:

- Search, insert and delete in logarithmic time
- Lazy hash recalculation: for tree of size `N`, with `M` recent modifications, `M << N`, all the operations, including all hashes recalculation, takes `O(M * log(N))`.

Note about potential weakness and its prevention

First, worth to note, that, unlike in Bitcoin, in BEAM the Radix-Hash trees contain object hashes (computed from objects with asserted parameters), rather than raw objects themselves. This makes it harder for an attacker to get a proof to a non-existing object by inserting specifically forged objects. More details here: https://bitslog.wordpress.com/2018/06/09/leaf-node-weakness-in-bitcoin-merkle-tree-design/

Since 3 different kinds of objects (UTXOs, kernels, and previous block hashes) are combined into a single hash, it may be possible for a malicious node to forge proofs to non-existing objects by mixing different object kinds.

In particular there is a hypothetical possibility (if not doing it carefully) to construct a proof to an UTXO that existed in an earlier block, but was spent later. Such a malicious proof would consist of 2 parts:

1. Proof to this UTXO in an earlier block, in which it still existed, i.e. Merkle proof that, after interpretation, produces the state definition of an earlier block.
2. Proof that this earlier block is included in the current one, i.e. Merkle proof that, after interpretation, produces the state definition of the current block.

By such it may seem that such a malicious proof could work. The proof actually consists of 2 parts appended, but the SPV may not notice this.

In BEAM this is prevented by the distinction between *system definition hash*, and the *block header hash*. They are different, hence those 2 parts can't be just appended. Moreover, the header hash depends on the system state definition hash, in such a way that it may NOT be easily constructed via Merkle path. Means, it's not of the form:

- `Block_Hash = Hash(System_definition_hash | <some data of the length of the hash>)`

Advantages of BEAM (wrt blockchain organization)

Smaller Compressed history

BEAM fully supports the excellent MW feature of compressed blockchain history. The compressed history is essentially a single transaction (or several ones, if the cascade aggregation strategy is used) with a small modification: for every specified UTXO there's also an appropriate blockchain height, in which it appeared in the original blockchain. This is needed to reconstruct proper system state, whereas each UTXO has right maturity.

Note however that unlike what's stated in the MW whitepaper, and (AFAIK) current GRIN implementation, in BEAM there is no need to include the Merkle proofs in the compressed history, because as we've said, the internal structure of Radix trees depend only on the contained objects, and not on the history. Once the node interprets all the compressed history, it computes its current system definition hash, and ensures that it matches what should be according to the blockchain headers (in addition to other checks, such as continuity of the headers, validation of PoW, overall blockchain subsidy and etc.).

Less motivation to produce empty blocks

One of the problems in Bitcoin is empty block mining. When mining pools receive a block from a rival pool they naturally verify it, but to save time even before the verification is completed they immediately start mining of an empty block, i.e. block that contains only the coinbase transaction. Constructing an empty block is easy in bitcoin (putting mining aside), because it only contains the root hash of the UTXOs produced by this specific block.

In BEAM, however, in order to create a block, even an empty one, it's essential to realize the future system state after this block is interpreted, whereas the coinbase UTXO and the previous block hash are added to the appropriate data structures. This in turn demands the interpretation the rival block first. And performing all this without adding available transactions doesn't make sense.

Simpler protocol for SPV clients

Unlike GRIN, BEAM node doesn't require clients to send any proofs to it (since it has all the information anyway). Same applies to consuming coinbase UTXOs: clients don't need to prove their maturity. The input UTXO is referenced just by its commitment. Client may get a proof for the previous system state in order to verify that the blockchain didn't revert beyond the point previously seen.