Litecoin — Mimble Wimble audit

Technical Audit Report

Reference	21-08-872-REP
Version	1.0
Date	2021/10/20

Quarkslab

Quarkslab SAS
13 rue Saint Ambroise
75011 Paris
France

Table of Contents

1. Project Information

Document history			
Version	**Date**	**Details**	**Authors**
1.0	2021/10/20	Initial Version	Robin David & Laurent Grémy

Quarkslab		
Contact	**Role**	**Contact Address**
Frédéric Raynal	CEO	`fraynal@quarkslab.com`
Matthieu Duez	Services Manager	`mduez@quarkslab.com`
Camille Spokojny	Project Manager	`cspokojny@quarkslab.com`
Robin David	R&D Engineer	`rdavid@quarkslab.com`
Laurent Grémy	R&D Engineer	`lgremy@quarkslab.com`

Litecoin		
Contact	**Role**	**Contact Address**
David Burkett	R&D Engineer	`davidburkett38@gmail.com`
Charlie Lee	CEO	`coblee@litecoin.org`

2. Executive Summary

This report describes the results of the security evaluation made by Quarkslab of a Mimble Wimble (MW) implementation in Litecoin. The MW protocol [MW], [MWU] enables performing confidential transactions where amounts are hidden. It provides new privacy features, such as fungibility of the coins and coins aggregation from multiple transactions.

Litecoin is based on Bitcoin and is thus an UTXO-based blockchain where implementing such a protocol is impossible on the main chain. As such, MW is implemented as a side-chain tighted to the canonical chain. The audit aims at verifying the correctness of the implementation and the additional consensus rules that it implies.

The two main components reviewed are both the MW implementation written as a library and its integration within the Litecoin codebase. The main focus relies on ensuring that both MW blocks and transactions are valid with regards to the new added consensus rules. Also, we focused on checking that no regressions are introduced in the current implementation.

The evaluation was carried by two auditors for a duration of 45 days. A major vulnerability has been found which can have implications on the consistency of the chain. Findings are described in Section *Findings summary*.

2.1 Disclaimer

This report reflects the work and the results obtained within the duration of the audit on the specified scope (see *Scope*). Tests are not guaranteed to be exhaustive and the report does not prove the code to be bug free.

2.2 Findings summary

ID	Description	Recommendation	Impact
HIGH01	MWEB Blocks are not properly validated which enables rogue miner to submit invalid blocks with regards to policy that will be accepted.	Adding the call to `MWEB::CheckBlock` function from the canonical side.	High
MED01	Risky usage of `assert` in `GetHogEx` that if triggered causes a denial-of-service	Checking vout emptiness and return `nullptr` in that case.	Medium
LOW01	Risky vector iteration which could result in integer-underflow and out-of-bound access if `GetMWEBHash()` was not performed *(untriggerable)*.	Strengthening the code by adding a supplementary check on size before the for loop.	Low
LOW02	Risky usage of `assert` in `ContextualCheckBlock`	Returning false instead if assertion does not hold.	Low
INF01	Duplicate transaction validation *(Schnorr signatures, bulletproofs)* for a same block.	Refactoring code to avoid unecessary double checks.	Info
INF02	The keystream generation may be strengthen in the building of an output coin.	Using a domain specific tag or a KDF function for example.	Info

3. Context and Scope

3.1 Context

The Litecoin foundation is planning on integrating MW in its blockchain. MW is a privacy oriented cryptographic algorithm. It enables not disclosing transactions content, recipient, and amount. Implemented as a library it is integrated in the main Litecoin code as a side-chain. The broad specification has been defined as Litecoin Improvement Proposals (LIP). Four LIPs are of interest for this audit (see. Section *LIPS*).

3.2 Audit Settings

Quarkslab performed a security evaluation of the MW integration as developed on a specific repository *(not the official Litecoin repository)*. A dedicated branch named *quarkslab* has been provided to Quarkslab. As the project is still in development, we also reviewed some modifications introduced in commits pushed during the audit. Audit setting is summarized in the following table.

Name	Litecoin
Repository	https://github.com/ltc-mweb/litecoin
Branch	quarkslab
Commit	`84fc661e821a2aa6d1bc6d7e63077291bab0aae4`
Network	Testnet

A dedicated testnet has been deployed as a prerequisite of the audit. Blocks were mined automatically either by the running node or by Quarkslab depending on tests to perform.

The MWEB feature has been developed to be put in production as a soft-fork through the BIP009[1] signaling mechanism. In the chain parameters MWEB was meant to switch in `ACTIVE` state at height 432.

Timeline In order to communicate more easily, we setup a Telegram channel with the developers. Throughout the audit version commits have been performed to patch parts of the code or to expose some data through RPC. The timeline of the audit is the following:

- mid-august: audit kick-off;
- August 26th, commit: `84fc661e821a2aa6d1bc6d7e63077291bab0aae4` to fix testnet parameters;
- August 30th, commit: `b35c7d9c25e553ba2b7f8b4366ea2aa81cb8d166` to fix some wallet code;
- September 28th, commit: `a4bfa6783a73e35b676f5fc3788412c3b285d0e0` to expose MWEB data through RPC;
- September 29th intermediate call for mid-term discussion;
- October 20th first version of the report sent to the Litecoin foundation;
- October 25th final version of the report sent to the Litecoin foundation;
- November 5th, final restitution.

[1] https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki

3.3 Safety and Security Properties

All the properties of the Litecoin protocol, integrity and availability should be preserved through the implementation of MW in Litecoin. The considered threat model is thus the same as traditional blockchains. An attacker shall not be able to corrupt the ledger, bypass the consensus policy or prevent the other users from using the blockchain.

But the considered threat model also encompasses side-chain related properties (consistency, finality) and also privacy-related properties linked to MW. To sum up, high-level properties that have to be preserved or added are the following:

- all rules already defined in Litecoin must be preserved (consensus etc.);
- new consensus rules need to be more restrictive on accepted blocks to be accepted through a soft-fork;
- at all times, the sum of pegged-in coins should equal the amount of coins on the side-chain;
- transaction amounts on MW side-chain should remain confidential between emitter and recipient.

LIPs formalize how MW is meant to be implemented in Litecoin. They describe how the protocol and these improvements are implemented: an example is [LIP0004]. Indeed, the traditional way to explain how MW creates outputs assumes that there exist communications between the senders and the receivers; this undesirable feature was removed from the Litecoin implementation of MW, in favor of a non-interactive way to create outputs.

Given the security properties that MW have to provide, the audit aims at answering the following security concerns:

- can an adversarial miner craft corrupted integrating transactions, or corrupted MW block?
- can LTC UTXO be spent on MW chain without having been pegged-in?
- are the security goals of MW achieved with the Litecoin modifications?

Note: Mimble Wimble is implemented as a synchronized side-chain. That means a MW block is produced for each LTC block. They thus evolve at the same speed. That excludes a whole range of security issues (trustwhorthy, finality) present on commit-chains or any cross-chain communications.

3.4 Scope

The scope of the audit is narrowed on two aspects. First, the MWEB library that provides all primitives to craft MW transaction and to craft MW blocks. Very low-level cryptographic primitives are left out of scope (secp256k1, bulletproof) and considered secure. That cryptography implementation review aims at ensuring it is correctly implemented and satisfies LIPs and the associated security properties. Secondly, the focus is given on changes made in the Litecoin codebase to bind the side-chain. It implies changes on the consensus policy, addresses, storage etc. The following concerns are left out of scope for the audit:

- low-level cryptographic primitives secp256k1, bulletproof *(external implementation already*

audited);

- cut-through features of the MW algorithm;
- Initial Block Download (IBD) and especially pruned one of MW;
- any probabilistic attacks to deanonymize transactions, through correlations etc. *(only cryptographic aspects are considered)*;
- wallet and UI related issues in the Qt implementation;
- serialization / deserialization of blocks transactions *(use litecoin mechanism)*;
- multi-signature considerations.

All other unmodified components of Litecoin are considered safe and left out-of-scope.

3.5 Methodology

To cover the different modifications performed, and the security properties to check, the audit has roughly been divided in the following steps:

1. code and implementation discovery;
2. static code review of the MW cryptographic primitives;
3. static code review of the whole MW implementation;
4. static review of changes performed in the Litecoin main codebase;
5. wallet changes;
6. dynamic testing of the blockchain;
7. report writing.

Thanks to the decentralized aspect of blockchains, all dynamic tests have been performed as any full node on the network.

Report outline This report is organized as follows, first Section *Litecoin Integration* describes the MWEB library implementation and integration into the Litecoin codebase and the found issues. Secondly, Section *Mimble Wimble Cryptography* dives into the cryptography related to MW. Lastly, Section *Dynamic Tests* discusses dynamic tests performed as part of the audit.

4. Litecoin Integration

4.1 Side-chain: Key concepts

4.1.1 LIPS

The audit aims at reviewing the implementation of various LIPS introducing a general framework for implementing side-chains in Litecoin and especially MW. Draft LIPS considered are hosted on David Burkett's Github.

LIP0002 Extension Block

This improvement introduces the extension blocks (EB) mechanism enabling the interconnection of a side-chain to Litecoin without altering consensus rules. The LIP describes the inner working such that EB can be soft-forked into LTC. If that improvement is accepted, an EB block is generated for each canonical block and communications are performed through an **integrating transaction** (ExtTxn) that have to be embedded in every canonical block *(after activation)*. The integrating transaction is completely valid with regards to Litecoin consensus rules and can be transparently accepted by legacy nodes (not having forked). Specific UTXO transfer to and from the side-chain will be performed respectively by Peg-In and Peg-Out transactions. Peg-In LTC transaction to a specific extension address (ExtAddr) are derived from a new witness program (ExtVer). Peg-outs are specific outputs created in the ExtTxn transaction to a LTC address.

More details: LIP0002

LIP0003 Mimble Wimble Extension Block (MWEB)

This improvement describes an implementation of LIP0002 to introduce the MW protocol as a side-chain. In this context the integrating transaction is called `HogEx` (Hogwart Express) and ensures the 2-way peg from and to the MW side-chain. The LIP then describes MW extension addresses HogAddr using version 9 of witness program. It then describes coin creations, transactions and internal workings of the side-chain. This LIP also describes the fee issue that should be spent either on the canonical or the MW side.

More details: LIP0003

LIP0004 One-Sided Transaction in MimbleWimble

In the original MW protocol, both emitter and recipient need to collaboratively create a MW transaction, as the receiver should know the blinding factor of the outputs he will own. This LIP describes how to create outputs without interaction between parties and how to achieve a sufficient security with the non-interactive process. It implies that both the emitter and the recipient know the blinding factor of an output.

More details: LIP004

LIP0005 Peer Services

This LIP describes low-level serialization and deserialization formats of MWEB transactions and blocks between peers. MWEB relies on existing Litecoin mechanisms to declare the support of MWEB features. The flag byte value 0x08 is used to indicate inclusion of MWEB data in the transaction. The LIP then describes the transaction format. For now the LIP is not completely specified.

More details: LIP005

4.1.2 Naming Convention

To better clarify types of transactions for the remaining of the report we use the following naming convention:

- `LTC-LTC` for standard Litecoin transactions *(canonical transactions)*;
- `LTC-MW` or also `PEGIN` for transactions transferring coins to the MWEB side-chain;
- `MW-LTC` or also `PEGOUT` for transaction transferring coins from MWEB to the canonical chain;
- `MW-MW` MWEB transactions manipulating commitments *(and not scripts and UTXOs)*.

Note: The report references a `TxBody` class that regardless of its name references the content of both a block or a transaction. It is a container for a sequence of inputs, outputs and kernels. It will be used for both block validations and transaction validations.

4.2 Consensus: Block Validation

Once the MWEB is accepted by all miners through BIP9 soft-fork, each Litecoin block should contain an MWEB. In the implementation, such block is held as an attribute (`mweb_block`) of the `CBlock` canonical chain. The extension block *LIP0003 Mimble Wimble Extension Block (MWEB)* adds additional consensus rules on both the canonical block and the MWEB block. Figure 4.1 shows the overall function call workflow to validate a block in the source code *(with both functions in canonical and mweb part)*.

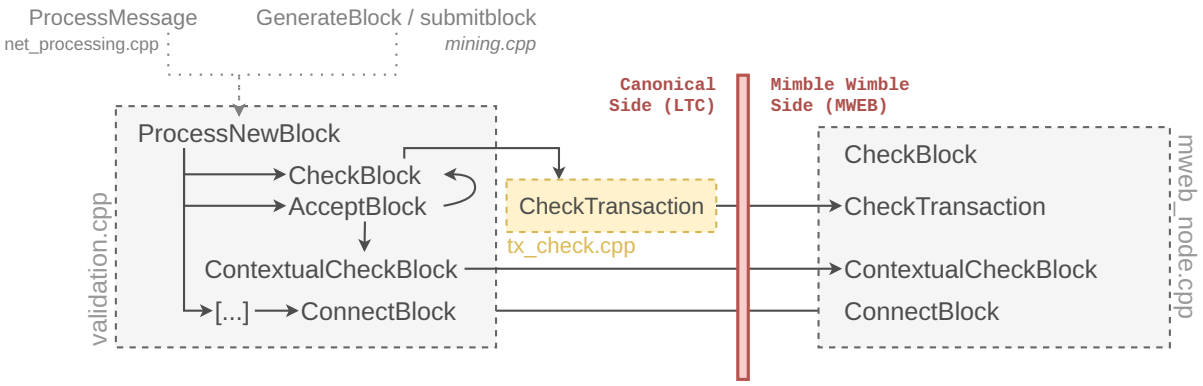

Figure 4.1: Block validation call graph overview

These checks aim at verifying that peg-in and peg-out transactions are valid and all transactions within the MW world as well. All consensus rules are described in the documentation.

4.2.1 Block Checks

These structural checks ensure that the content of the block is valid regardless of its ancestors and its location in the chain. The main function performing these checks is `CheckBlock` of *mweb_node.cpp*. For ease of understanding we identified each check and discussed whether or not they have been correctly implemented in the code. The table below shows the enforced rules. Some issues have been found as part of these checks.

id	Component	Description	Ok
Chk1	Node::CheckBlock()	The unique HogEx transaction is the final transaction in the block	✓
Chk2	Node::CheckBlock()	All MWEB data has been stripped from canonical transactions	✓
Chk3	BV::Validate()	MWEB header hash matches hash committed by HogEx transaction	✓
Chk4	BV::ValidatePegInCoins()	Peg-In kernels match canonical peg-in outputs (amounts and commitments)	✓
Chk5	BV::ValidatePegOutCoins()	Peg-Out kernels match HogEx peg-out outputs (amounts and scriptPubKeys)	✓
Chk6	Block::Validate()	Kernel MMR size and root match the MWEB header	✓
Chk7	OwnerSumValidator::Validate()	Owner sums balance, proving the sender(s) knew the input receiver keys	✓
Chk8-12 performed on `TxBody` (see Table 4.4)			✓

BV:BlockValidator

Table 4.1: CheckBlock verifications (context-independent).

4.2.2 `HIGH01`: Missing call to `MWEB::CheckBlock`

HIGH01	Missing call to `MWEB::CheckBlock()`		
File	`validation.cpp`		
Category	Consensus		
Status	Vulnerable		
Rating	Severity: Critical	Impact: High	Likelihood: Critical

Description: As shown on Figure 4.1 there is no call to `MWEB::CheckBlock()` from the canonical side and more specifically the `CheckBlock()` function in *validation.cpp*. All the other functions `CheckTransaction`, `ConnectBlock` etc, are correctly bound to theirs Litecoin counterpart.

As shown on the above table the function performs many checks which are consequently not enforced. Fortunately, some of them are also performed in `ContextualCheckBlock` and `CheckTransaction`. As such, it does not appear to be possible to induce discrepancies of coin amounts between the two chains. **Chk1** and **Chk4** are also implicitly being performed by

`ContextualCheckBlock` and **Chk7-18** by `CheckTransaction` (defined in Table 4.2). That leaves the following verification unchecked:

- **Chk2**: all MWEB data has been stripped from canonical transactions.
- **Chk3**: MWEB header hash matches hash committed to by HogEx.
- **Chk5**: peg-out kernels match HogEx peg-out outputs.
- **Chk6**: kernel MMR size and root match the MWEB header.

Not enforcing these checks has nonetheless a strong impact on chain integrity and trustworthiness.

4.2.3 MEDIUM01: Assert in GetHogEx() triggerable by malformed block

MEDIUM01	Assert in `GetHogEx()` triggerable by malformed block		
File	`block.cpp`		
Category	Denial-of-Service		
Status	Defect present		
Rating	Severity: Moderate	Impact: Moderate	Likelihood: Unlikely

Description: The function `GetHogEx` returns the HogEx transaction in a block. It properly checks that it contains at least two transactions *(the coinbase and the extension block)*. However, if a miner generates an HogEx transaction without outputs in `vout` the assert is triggered.

```
CTransactionRef CBlock::GetHogEx() const noexcept
{
    if (vtx.size() >= 2 && vtx.back()->IsHogEx()) {
        assert(!vtx.back()->vout.empty());
        return vtx.back();
    }

    return nullptr;
}
```

Under normal compilation settings, asserts are preserved and abort the execution of the program. Thus, the assert could result in a denial-of-service of any node receiving such malformed blocks. As the mined transaction has to contain outputs, it should be enforced by returning `nullptr` which would result in the block being rejected.

4.2.4 LOW01: Unsafe iteration on transaction vector leading to underflow

LOW01	Weak iteration on transaction vector leading to underflow		
File	`mweb_node.cpp`		
Category	Memory Access Violation		
Status	**Defect present**		
Rating	Severity: Low	Impact: None	Likelihood: None

Description: The **Chk1** ensures no transaction but the last one is flagged has HogEx. The for loop performing the check is rather unsafe because a malformed block without transaction results in an integer-underflow generating a 2^{64} loop iteration up until performing an out-of-bound access out of the vector (segfault).

Fortunately, the scenario cannot happen thanks to a prior call to `block.GetMWEBHash()` which itself calls `GetHogEx()` that make sure the vector size is at least 2. Nonetheless, as the check is performed in a secluded location compared to the loop, a size sanity check would harden the code *(in case of refactoring)*.

```
for (size_t i = 0; i < block.vtx.size() - 1; i++) {
    if (block.vtx[i]->IsHogEx()) {
        return state.Invalid(BlockValidationResult::BLOCK_CONSENSUS, "bad-
↪hogex-position", "hogex in wrong position");
    }
}
```

4.2.5 INFO01: Duplicate transaction validation of signatures and range proofs

INFO01	Duplicate transaction validation *Schnorr signatures and bulletproof*	
File	`TxBody.cpp`	
Category	Duplicate checks	
Status	**Defect present**	
Rating	Severity: Info	Impact: Computation overhead

Description: Upon block reception, both the function `MWEB::CheckBlock()` and `MWEB::CheckTransaction()` perform a call to `TxBody::Validate()`. The function performs all the verifications on the transactions within a block and thus perform the Schnorr signatures and Bulletproof range proofs. These verifications are performed using `BatchVerify` and not individually for each transactions. As such, the computation overhead is rather limited. Some code refactoring would could avoid performing these duplicate checks even though the impact is negligible.

4.2.6 Contextual Checks

These checks ensure the block is valid with regard to its location in the chain and its ancestors. The function `ContextualCheckBlock` performs all the checks ensuring a consistent chain. Table 4.2 lists all the consensus checks that are performed for the MWEB blocks in order to be accepted at the top of the chain.

id	Component	Description	Ok
Chk13	Node::ContextualCheckBlock()	MWEB is included when the feature is active, or not included when MWEB has not yet been activated	✓
Chk14	Node::ContextualCheckBlock()	MWEB header block height matches the expected height	✓
Chk15	Node::ContextualCheckBlock()	The first input in the HogEx points to the HogAddr output of the previous HogEx *(except for HogEx in first block after MWEB activated)*	✓
Chk16	Node::ContextualCheckBlock()	The remaining inputs in the HogEx exactly match the pegin outputs from the block's transactions	✓
Chk17	Node::ContextualCheckBlock()	HogEx fee matches the total fee of the extension block	✓
Chk18	Node::ContextualCheckBlock()	HogAddr amount differs from previous HogAddr amount by the exact amount expected *(previous + pegins - pegouts - fees)*	✓

Table 4.2: ContextualCheckBlock verifications (context-dependent).

4.2.7 LOW02: Unsafe usage of assert to check HogEx presence

LOW02	Unsafe usage of assert to check HogEx presence		
File	`mweb_node.cpp`		
Category	Denial-of-Service		
Status	**Vulnerable**		
Rating	Severity: Low	Impact: None	Likelihood: None

In the function `ContextualCheckBlock` at line 88, the assert `assert(pHogEx != nullptr);` checks that the block contains a HogEx transaction. In the current state of the code because `Node::CheckBlock` is not called, the assert can be triggered by a malicious miner generating a block without HogEx. Once a `Node::CheckBlock` call will be added in `CheckBlock` the code will theoretically be safe, as it will have been checked before. Still, using assert in such a code portion puts the availability of the node at risk. Instead of an assert, one can simply check that `block.GetHogEx()` does not return a `nullptr`, and returning immediately false in this case.

4.3 Consensus: Transaction Validation

4.3.1 Transaction Reception

The MWEB introduces a new kind of transactions (`MW-MW`). All other transactions `PEGIN`, `PEGOUT` and the `HogEx` are standard Litecoin transactions using the scripting engine.

To encompass the new `MW-MW` type, the class `CTxInput` has been created to wrap either a MWEB `Commitment` object or a canonical `CTxIn` transaction. It is the same for `CTxOutput`. Figure 4.2 shows the basic call graph workflow upon transaction receptions. Added consensus checks are described in the following tables.

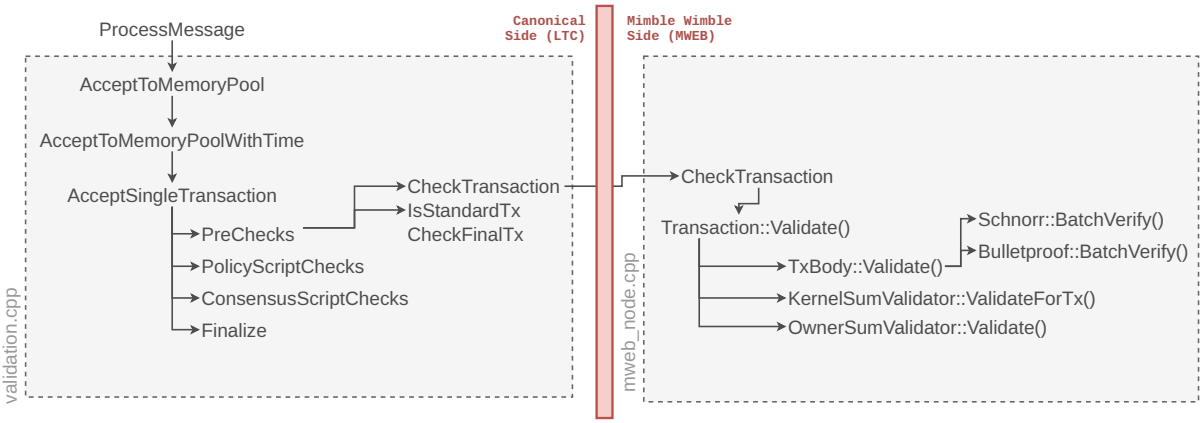

Figure 4.2: Transaction reception acceptance call graph

id	Component	Description	Ok
Chk19	Node::CheckTransaction()	No pegin witness programs are included in Coinbase and HogEx outputs	✓
Chk20	KernelSumValidator::Validate()	Kernel sums balance, proving no inflation occurred, and the sender(s) knew the input blinding factors	✓
Chk7	OwnerSumValidator::Validate()	Owner sums balance *(also in CheckBlock)*	✓
Chk8-12 performed on TxBody (see Table 4.4)			✓

Table 4.3: CheckTransaction verifications.

id	Component	Description	Ok
Chk8	TxBody::Validate()	MWEB block does not exceed max weight	✓
Chk9	TxBody::Validate()	Inputs, outputs, and kernels are properly sorted	✓
Chk10	TxBody::Validate()	No invalid duplicate inputs, outputs, or kernels	✓
Chk11	TxBody::Validate()	All signatures and rangeproofs are valid	✓
Chk12	TxBody::Validate()	Kernel features are valid	✓

Table 4.4: TxBody Validate verifications.

As shown on Figure 4.2, prior to acceptance to the mempool a MWEB transaction is completely verified. No specific issues have been found in the processing of incoming transactions.

4.3.2 Transaction crafting & coin selection

MWEB also complexifies transactions creation in the wallet. Indeed, both LTC and MWEB coins cohabit together. Upon transaction creation, depending on the output address type, the implementation has to properly select either LTC or MWEB coins. In Litecoin Core, the wallet will by default automatically select the appropriate coin to spend. Usual optimizations apply. The algorithm will first intent to spend a single UTXO that fits the amount to spend and fees, or will try aggregating small UTXOs together. An additional change output address will be created if needed.

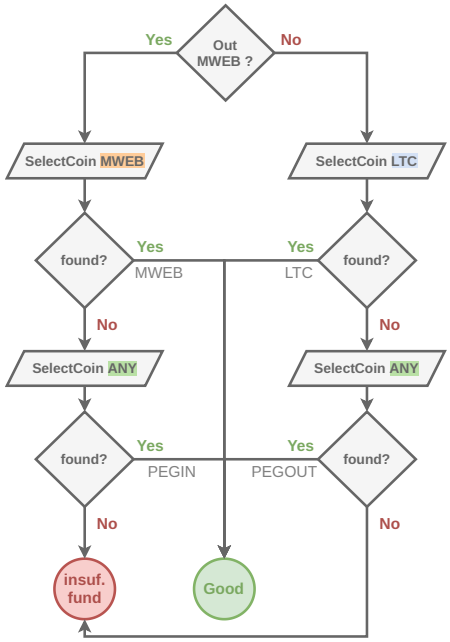

Figure 4.3: Automatic coin selection logic of `AttemptCoinSelection`

As shown on Figure 4.3, depending on the destination address and the availability of coins, different transaction types will be created. The algorithm is implemented in the function `AttemptCoinSelection`. If the destination address is an MW one (StealthAddress), the algorithm first tries to create a MWEB transaction. If no coin is available on the MW side-chain a LTC coin is retrieved and in that case a `PEGIN` transaction is created. Similarly, if the destination is a standard bech32 address, a standard LTC transaction is first created. Otherwise the function tries to spend an MW coin and thus creates a `PEGOUT` transaction.

While such behavior might induce spending coins from an unwanted chain, that behavior is only triggered when no input coin is given. While reviewing the implementation, no unexpected behavior has been identified.

4.4 Block Mining

With the MWEB extension the miner is in charge of creating the integrating transaction (HogEx) and also producing the MW block (a `mw::Block`). The corresponding code is located in *mweb_miner.cpp* and exposes the following functions:

- `NewBlock()`: initialize local Miner object fields and especially a BlockBuilder instance;
- `AddMWEBTransaction()`: add MWEB transactions from the mempool to the block;
- `AddHogExTransaction()`: create the HogEx transaction, by recovering the previous block to chain previous HogAddr as HogEx input etc. This is the function that will also build the MW block and add it in the attribute `mweb_block` of a canonical `CBlock`.

Static review of the source code did not revealed any structural issue. The behavior of the code should not be considered as granted from a consensus and block reception perspective as anyone can implement its own block mining algorithm to submit corrupted blocks.

4.5 Miscellaneous Changes

Implementation of MWEB induces many minor changes throughout the Litecoin source code. Manual review of these changes did not reveal any issues. Here is a non-exhaustive list of some changes:

- change the transaction weight computation *consensus/validation.h* in `GetTransactionWeight()` to take into account peg-in and peg-out instructions;
- exclude the MWEB block from block weight computation in `GetBlockWeight()`;
- adapt the feerate, taking into account the weight of MWEB data. The computation fee is: $\text{num\_kernels} \times 2 + \text{num\_owner\_sigs} \times 1 + \text{num\_outputs} \times 18$. Changes are made in `GetFee()`, `GetTotalFee()` or else `MeetsFeePerK()`;
- change the serialization function of CBlock *block.h* to embedded or not the associated MWEB block depending on whether `nVersion` enables it;
- add the `MWEB_PEGIN` `TxoutType` in the the script engine;
- create a DestinationAddr class to wrap either a standard CScript or StealthAddress to represent a destination address;
- change the address decoding function `Decode` in *bech32.h* to enable larger addresses for MW.

The whole code source has been been analyzed with `clang-static-analyzer`. Most serious alerts mentioned null pointer dereference which manual review also revealed not to be triggerable.

5. Mimble Wimble Cryptography

The MW protocol was first introduced by an anonymous researcher in [MW]. The name comes from a Harry Potter curse. Some other names of the MW protocol and its implementation also come from this universe. The protocol was better formalized and described in [MWU] and began to be analyzed by the cryptographic community, especially in [ACS] and [FAM]. This protocol was really well-received in the crytocurrency community, where a lot of blog posts and articles were published to describe how MW works. Finally, two major implementations were proposed to make MW practicable: [GRiN] and [Beam]. The goal of this introduction is not to be exhaustive about the MW literature, but the interested reader may find on the internet a lot of resources with different levels of description. As the current MimbleWimble implementation is derived from GRiN, we mostly base our explanations on GRiN's documentation[1], the different Litecoin LIPs (see. *LIPS*) and the project documentation[2].

5.1 Cryptographic libraries

All the cryptographic primitives are written in *src/secp256k1-zkp*. There is however another directory where cryptographic primitives may be found, which is *src/secp256k1*. Note that this code is not used during the compilation process, and does not provide the implementation of useful MW primitives. In addition, the source code of `libsecp256k1` seems to evolve independently from the Litecoin code, making it more difficult to track the modification of the Litecoin core code. If it is possible, having `libsecp256k1-zkp` as a submodule will help to maintain the library upstream in case of security issues or algorithmic improvements.

5.2 Randomness

Warning: The following statements on number generators have been changed during the audit in commit `07f666d2ea6ea5bfd94e683fa02effeba01709f2` in favour of the use of `GetStrongRandBytes`, the default random source for Litecoin Core. The following observations are not valid anymore at the time of delivery.

In most of the cryptographic protocols, randomness is used, to generate various secret or non-secret elements. At the beginning of the audit, two different sources were used, implemented in *src/libmw/include/mw/crypto/Random.h*. These two sources are:

1. `FastRandom`, which is never called;
2. `CSPRNG`, which consecutively reads `n` bytes of `/dev/urandom`.

We can guess that `FastRandom` is implemented if a non-secure random number generator is needed. The implementation using a non-secure PRNG confirms this hypothesis. In the `FastRandom`, if `min > max`, the code will enter in a `segfault` mode.

With `CSPRNG`, it is possible to add a `static_assert` to verify that `NUM_BYTES` does not exceed `SSIZE_MAX`, defined in `limits.h`.

[1] https://docs.grin.mw/wiki/table-of-contents/
[2] https://github.com/ltc-mweb/litecoin/tree/0.21/doc

5.3 Pedersen commitment and blinding factor

In different parts of the MW protocol, the use of blinding factors for Pedersen commitment is needed to ensure some of the fundamental security properties of MW. To briefly recall, a Pedersen commitment to a value v is an element $P = rG + vH$, where r is a random element and G and H are elements of a group $(\mathbb{G}, +)$. Revealing v and r after revealing P allows to verify that the committed value P was a valid commitment for the value v.

One of the obvious properties needed to trust the committer is that the relation between the public elements G and H is not known. First, note that the group used in MW is the additive group of rational points on the curve *secp256k1* [SEC2]; in this group, the discrete logarithm problem is considered as hard. Since this group has a prime order, G and H are both generators of the group. Then, the discrete logarithm of H in base G (or conversely) must be unknown. A common choice for G is the standardized generator. To allow to reasonably consider that a built element H is chosen with no known dependency with G, a nothing-up-my-sleeve process is used to compute H and is documented in *src/secp256k1-zkp/src/modules/generator/main_impl.h*.

5.4 Schnorr signature

The Schnorr signature is a digital signature scheme. There are numerous advantages in using Schnorr signatures instead of the classical ECDSA deployed in today's Litecoin blockchain, summarized in [BIP340]. At the first iteration of the code we reviewed, the signature scheme was not aligned on [BIP340], for example the implementation does not use domain separation tag for hashing and does not follow how the element *rand* is generated. Note that the last version we audited is now aligned on [BIP340].

In the MW protocol, there are many signatures to verify, then a batch verification procedure may be used, in order to provide some speed-up, in comparison with verifying all the signatures individually. The code called by `Schnorr::BatchVerify` uses the C function in backend which, as far as we saw, respects [BIP340] to mitigate the risk of fake signatures.

Note: Some signatures in the MW protocol are used to only prove possession of a private key. Note that a replay attack on the knowledge of a pair of a public key and a signing message is not an issue, since a key pair is used only once.

5.5 Bulletproofs

Bulletproofs [BP] are a widespread type of zero-knowledge rangeproofs, which has the advantage of proving that an amount blinded in a Pedersen commitment contains a value in a range of the type $[0, 2^n)$, where n is often equal to 64. Since the amount of a transaction is blinded in a Pedersen commitment, the use of Bulletproofs is particularly interesting. Since it will be necessary to verify a lot of Bulletproofs at the same time, a batch verification is used.

5.6 Overview of the protocol

Let consider that Alice wants to send some LTC to Bob. Let us suppose that Alice is in possession of a_0 LTC, wants to send b_1 LTC to Bob and wants him to send back to her the amount a_1. The fees of the transaction are f_1. We then have the relationship: $a_0 = b_1 + a_1 + f_1$, in order not to create or destroy money.

However, what is publicly visible as being in Alice's possession is something like:

- a commitment $C_{[a,0]} = r_{[a,0]}G + a_0H$;
- a rangeproof proving a_0 is positive and less than 2^{64};
- a *receiver's one-time* public key $K_{[a,0]}$;

In addition to this public information Alice knows:

- the amount a_0 and the blinding factor $r_{[a,0]}$;
- the receiver's one-time private key $k_{[a,0]}$.

To spend her coin, Alice will sign the message `MWEB` with the private key $k_{[a,0]}$, proving that she knows the private key associated to the public key attached to the coin she wants to spend.

To send a coin to Bob, Alice needs to know Bob's address. In the MW world, an address is not just a public key, but two. Indeed, MW uses a *stealth address*, as described in [SA]. The process described in the section "Output Construction" allows to produce the elements needed for the receiver to fully possess its coins. Indeed, in the classical way to describe MW, Alice and Bob need to communicate in order for Bob to control the blinding factor of a coin and to know which amount he will receive. [LIP0004] describes how to non-interactively produce a coin, in a way that only Bob is able to possess the coin.

The idea of the creation of an output is to allow a receiver to be sure that the coin is legitimately built and does not introduce some way to reduce the privacy introduced by MW. The basic idea is to perform a non-interactive Diffie–Hellman to share a secret which is used to derive a key itself used to encrypt the coin value, the blinding factor, and elements needed to find the private key corresponding to the public key attached to the coin. All the public elements are signed with an ephemeral private key, called *sender key*, for which the associated public key is used in the computation of the rangeproof on the value of the coin.

Note: Contrary to what is done in GRiN's [GRiN] implementation[3], there is no rangeproof associated to the blinding factor. In addition, as opposed to the classical MW description, the blinding factor of a coin is known by both the emitter and the receiver. This implies a modification of the MW design and then maybe a loss for some features MW may offer.

Now, all the inputs and outputs are blinded, we have namely:

- in inputs: $r_{[a,0]}G + a_0H$;
- in outputs: $r_{[a,1]}G + a_1H$, $r_{[b,1]}G + b_1H$, f_1H.

The sum of the outputs minus the inputs is equal to the sum of the blindings with their signs $K_1 = (r_{[a,1]} + r_{[b,1]} - r_{[a,0]})\,G$: this is the kernel of the transaction. In order to blind the kernel K_1,

[3] https://github.com/mimblewimble/grin/blob/master/doc/intro.md#range-proofs

an offset $K_1^o = k_1^o G$ is in addition removed from K_1, leading to a blinded kernel $K_1^b = K_1 - K_1^o$. This blinded kernel may be seen as a public key from which Alice knows the private key $k_1^b = r_{[a,1]} + r_{[b,1]} - r_{[a,0]} - k_1^o$. She will sign a message containing some public information (fee, peg-in, peg-out, etc.) with the private key and add the signature of the message and the public key (called `excess_commit` in the code) to the transaction.

Finally, a last step is needed to build a coin. To sign the kernel and prove its authenticity, an owner signature key is drawn at random to sign the kernel. The so-called *owner offset* is equal to the difference of the sum of the public sender key of all the emitted coins with the sum of public keys used to prove the ownership of the input and the public owner signature key. Are added to the coin:

- the signature of the kernel;
- the public owner signature key;
- the owner offset.

Note: In the file *src/libmw/src/wallet/TxBuilder.cpp*, we note that there is a "`MW: TODO - I'm no longer convinced this is even necessary`" about the owner signature usage. In [LIP0004], the design of the protocol and the use of the owner signature is described as solving issues which come from the non-interactive way to build outputs. The rational of the design to remove or add a feature needs to be documented in order to be justified against different attacker modelisations. At this step of the audit, it is not possible for us to have a strong opinion about the impact of removing or not a feature.

Once a coin is available, only the legitimate user is able to compute all the secret data thanks to the public data. However, since the coin is not sent to a precise address, all the players of the MW protocol will verify if the new output coin (which will become the input coin of one of them) belongs to their wallet or not. The section "Output Identification" allows to check that, with some tests allowing to abort earlier than the final check. The first check (step 2) allows around $1/256 \simeq 0.4\%$ of the users to continue for the second step, and the second check (step 7) around $1/2^{256}$ to continue for the third and last check (step 10).

Note: At the time of report writing, there is a typo in the documentation about the computation of the private spend key (you must read $\mathbf{b_i}$ instead of $\mathbf{a_i}$). The corresponding implementation is nonetheless correct.

5.6.1 INFO02: Keystream generation may be strengthened in the building of an output coin

INFO02	Keystream generation may be strengthened in the building of an output coin	
Category	Hardening	
Status	**Defect present**	
Rating	Severity: Info	Impact: None

Description: The element `m = HASH64(e)` is used as a mask for some of the elements. It may be a good idea to use a tag as for the other elements derived from a hash, for example `m =`

`HASH64('E', m)`, or a KDF function as specified in Section 4.1 of [KDM], but since `HASH64` is instantiated as the BLAKE3 function, a tag must be sufficient, since BLAKE3 is not vulnerable to a length attack. Note that BLAKE3 is not a standard, contrary to SHA3 [SHA3], but offers, according to [BLAKE3], fastest performances.

5.7 Node verification

Node verification is used to verify that a MW transaction is valid. At this step, there are lot of things to be verified, some of them being specifically about the cryptographic part of the MW protocol. Indeed, a node receiving orders to be included in the MW must verify the legitimacy of the transaction:

- the validity of the rangeproofs;
- the validity of the signatures;
- the ability for a user to spend inputs;
- the validity of the balance.

By checking (at least) all of these elements, the node may be sure that no money is created or destroyed, which is one of the fundamental properties of the blockchain. We report to Section 5.3 for a description in a large perspective of the mechanisms used to verify a MW transaction. These verifications are done in `TxBody::Validate` and `OwnerSumValidator::Validate` with the others. With **Chk8**, **Chk9** and **Chk10**, the node verifies that the data have the proper raw formats in order to be validated through the cryptographic checks **Chk11**. The following signatures are verified, thanks to the batch verification algorithm:

1. the signature of the kernel with the owner signature key;
2. the signature of the message “MWEB” with the key to prove possession of the input coin;
3. the signature of some public elements [LIP0004] of a coin with the sender key.

Then, rangeproofs are also verified. If they are valid, then it proves that the public sender keys are authentic. If not, the rangeproofs will not be verified since the public sender keys are elements used in the rangeproofs to initiate some seeds. And consequently, it proves that the signature verification of the elements with the sender keys are correct. The verification of the 3rd item allows to prove the authenticity of the receiver’s one-time public keys, allowing the coins to be spent in the future; there is no need to verify at this time the authenticity of the receiver’s one-time public key, since it was done in the past.

Then, the algorithm has to prove the authenticity of the owner signature public key. The verification is made by **Chk7**. It first needs to verify the correctness of the kernel which is signed by the owner signature key. Since the inputs and outputs are now validated, the computation of the kernel with the kernel offset may be verified, through **Chk20**.

Given the allocated time to review the implementation, the implementation does not suffer from any issue and seems to correctly implement its MW protocol specification.

6. Dynamic Tests

6.1 Wallet

Most dynamic tests have been performed using the built-in Litecoin core wallet supporting the MWEB side-chain. The support is still experimental but both `PEGIN`, `PEGOUT`, and MW side-chain transactions have been tested and proven to be functional.

Only a few glitches induced by the current state of development hindered the testing. Among them, we can denote the need to specify the coin to spend for MW transactions, the need to specify the fee explicitly as in both cases the automatic computation did not appear to be fully functional.

6.2 RPC endpoints

Multiple RPC endpoints have been enhanced with MW data enabling the manipulation and verification of transactions and blocks as they were produced and validated. Main endpoints modified for reading data are `getblockheader`, `getblock` and `getmempoolentry`. To submit transactions, the API does not change significantly. To simplify interacting with the RPC API, a tiny Python wrapper partially based on `bitcoinlib`[1] has been written. The snippet below shows how to retrieve the side-chain block for a given canonical block and listing the content.

```python
from mwbitcoinlib import Blockchain

chain = Blockchain.from_litecoind()

block = chain.latest_block

print(block.__repr__)
"""<Block 878 txs:[4] time:2021-10-06T14:35:19+00:00>"""

mwblock = block.mweb
print(f"{len(mwblock.inputs)} {len(mwblock.outputs)} {len(mwblock.kernels)}")
"""1 5 3"""

commitment = mwblock.outputs[0]
print(f"commit:  {commitment.commit:.20}[..]\n"
      f"proof:   {commitment.range_proof:.20}[..]\n"
      f"message: {commitment.message:.20}[..]"
)
"""
commit:  087118956d5f866c5aa5[..]
proof:   7d58b9aa4cf86594b459[..]
message: 033ee6c283626a64bf6b[..]
"""
```

We can also manipulate the mempool content and track pending MWEB transactions.

[1] https://bitcoinlib.readthedocs.io/en/latest/

```
for tx in chain.mempool:
    if tx.is_mweb():
        print(tx.mweb.dict())

"""
{'weight': {'base': 39, 'ancestor': 39, 'descendant': 39},
 'fee': 0.039,
 'lock_height': 0,
 'pegins': [],
 'pegouts': [],
 'inputs': ['08de3d0405d9bc4768d1d472081ec2d4ba4c176f25f9d129ab57fed4f9b80b8a0f
↪'],
 'outputs': ['08255a4b9c3b83179e6f7b19d255bd934cc372d9ef2f252a17db34223f2e54ff45
↪',

↪'0840fe0d5c5144d3634d974f10f3bc4bf4ba8eaec83c628010155095d53f62ca27']}
"""
```

Thanks to that tiny wrapper we have been able to manipulate data and perform multiple checks but it has not been possible to trick a node in a bad or unwanted state by submitting corrupted data.

Note: Due to lack of time, no fuzzing campaign has been intended on the source code. At low-level MW uses the serialization and deserialization mechanism of Litecoin which was left out-of-scope. Yet, some functions do manipulate deserialized data throughoutly and would deserve a dedicated fuzzing campaign. Among them we can denote `PreCheck` or `IsStandardTx`.

---

7. Conclusion

This report summarizes the audit of the implementation of the MimbleWimble protocol in the Litecoin blockchain. The protocol provides valuable new security enhancements about the privacy of transactions on the blockchain. As described in the original whitepapers [MW], [MWU], the protocol suffers from some user-friendliness issues. The Litecoin implementation proposes some improvement in order to use MimbleWimble without these limitations (one-sided transaction).

The code audited is written in a defensive manner and fits the code style of Litecoin (Bitcoin). Usage of the crypto API is good and even if documentation is lackluster, the implementation seems to satisfy the specification of MimbleWimble. Thanks to the responsiveness of the developer, this lack of documentation has not been an issue. Still, it may be difficult for an external auditor to understand the rational of some features or design.

Multiple low impact defects have been identified but a more serious security issue has been identified in the block validation process which impacts the consensus. Even if the code to perform the verification is implemented (`CheckBlock`), this function has not been properly called from canonical chain side. It theoretically enables invalid blocks to be accepted on the MimbleWimble side-chain. This is the only issue that must be fixed before MimbleWimble integration into Litecoin.

8. Glossary

MW Mimble Wimble is a privacy protocol introduced by an anonymous researcher. It enables transacting without revealing amounts. Some bases of the algorithm are Schnorr signatures and zero knowledge proofs based on bulletproof algorithm.

EB Extension Block is a term introduced by Johnson Lau as a proposal to create side-chain on Bitcoin through soft-fork via peg-in and peg-out transactions.

MWEB Mimble Wimble Extension Block is the implementation of MW protocol using the Extension Block proposal within Litecoin.

LIP Litecoin Improvement Proposal is a document with an unique identifier used in the litecoin community to identify all improvement proposals.

IBD Initial Block Download is the process of downloading the whole blockchain when a node synchronizes with the other peers.

UTXO Unspent Transaction Output is an entity representing a given amount of coins (here litoshi) and some data indicating how the UTXO can be unlocked and spent.

RBF Replace-by-Fee (BIP-125) enables replacing a pending transaction by a more recent one (by signaling it)

PSBT Partially Signed Bitcoin Transactions

MMR Merkle Mountain Range is an alternative to Merkle tree.

PMMR Pruned Merkle Mountain Range is a pruned version of the MMR.

BIP009 Bitcoin proposal is a Bitcoin Improvement Proposal standardizing the introduction of new features through soft-fork. It introduces a signaling mechanism between miners and a predefined state machine transition leading either to acceptance or refusal of the feature.

9. Bibliography

[ACS] Georg Fuchsbauer, Michele Orrù and Yannick Seurin, Aggregate Cash Systems: A Cryptographic Investigation of Mimblewimble, EUROCRYPT 2019, LNCS, vol. 11476. https://doi.org/10.1007/978-3-030-17653-2_22

[Beam] https://beam.mw/, https://github.com/BeamMW/beam

[BIP340] Pieter Wuille, Jonas Nick and Tim Ruffing, Schnorr Signatures for secp256k1, 17 May 2021. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki

[BLAKE3] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves and Zooko Wilcox-O'Hearn, BLAKE3 one function, fast everywhere, 21 February 2020. https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf

[BP] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille and Greg Maxwell, Bulletproofs: Short Proofs for Confidential Transactions and More, IEEE Symposium on Security and Privacy 2018, pp 315-334. https://doi.org/10.1109/SP.2018.00020

[FAM] Adrián Silveira, Gustavo Betarte, Maximiliano Cristiá and Carlos Luna, A Formal Analysis of the MimbleWimble Cryptocurrency Protocol, Sensors 2021, 21(17):5951. https://doi.org/10.3390/s21175951

[GRiN] https://grin.mw/, https://github.com/mimblewimble/grin

[KDM] Elaine Barker, Lily Chen and Richard Davis, Recommendation for Key-Derivation Methods in Key-Establishment Schemes, NIST Special Publication 800-56C Revision 2, August 2020. https://doi.org/10.6028/NIST.SP.800-56Cr2

[LIP0004] David Burkett, One-Sided Transactions in Mimblewimble, 29 December 2020. https://github.com/DavidBurkett/lips/blob/master/lip-0004.mediawiki

[MW] Tom Elvis Jedusor, MIMBLEWIMBLE, 19 July 2016. https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt

[MWU] Andrew Poelstra, Mimblewimble, 6 November 2016, https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf

[SA] https://github.com/ltc-mweb/litecoin/blob/master/doc/mweb/stealth_addresses.md

[SEC2] SEC 2: Recommended Elliptic Curve Domain Parameters, Certicom Research, version 2, 27 Junuary 2010. https://www.secg.org/sec2-v2.pdf

[SHA3] SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, FIPS PUB 202, August 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf