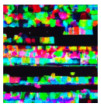


ZKSECURITY

Audit of Lighter's zkLighter Circuits

January 22nd, 2024

Introduction

On January 22nd, 2024, Lighter tasked zkSecurity with auditing its zkLighter circuits. The specific code to review was extracted and shared on a private repository. The audit lasted 3 weeks with 2 consultants.

During the engagement the team was given access to the relevant parts of the zkLighter codebase through a custom repository. In addition, a whitepaper detailing many aspects of the protocol was also shared.

Scope

The scope of the audit included the following components:

The zkLighter main circuit. This consisted of a large number of branches and files that implemented the different flows of the zkLighter protocol. This included transactions between L1 and L2, and pure L2 transactions like token transfer and order book management.

The zkLighter exit hatch circuit. This was a separate circuit that was used to allow users to exit the L2 network and withdraw their funds to the L1.

Implementation of the MIMC hash function. This made use of the GKR feature of gnark. We specifically focused on the implementation of the MIMC hash function and the way it made use of the GKR feature of gnark.

Recommendations

In addition of the findings discussed later in the report, we have the following strategic recommendations:

Audit integration. Ensure that the integration with external systems is secure. While this audit specifically focused on the zkLighter circuits, the security of the entire system is only as strong as its weakest link. This includes the security of the L1 logic that verifies zkLighter proofs, enforces the correctness of the public inputs, and safely processes the public outputs produced by the proofs.

Fix MIMC issues. Ensure that the findings related to the use of the MIMC hash functions are properly fixed, as both of these findings led to critical issues in the system. See **Prover Can Forge Merkle Proofs Due To Bad Fiat-Shamir Initial Randomness** and **Prover Can Forge Merkle Proofs Due To Lack Of Second Preimage Resistance**. Consider re-auditing the MIMC part of the system and having a more thorough audit of the non-standard GKR implementation once fixed.

Specify the protocol. Consider writing a specification of the flow implemented to verify transactions. Most of the complexity of zkLighter comes from the `VerifyTransaction` logic which needs to handle all possible state transitions. A specification (in english and/or pseudo-code) of the trading logic and how optimal trades are enforced

in the protocol would greatly help in ensuring that the flow implemented matches the expected behavior, and would potentially help simplifying a lot of the code.

Improve testing capabilities. During the engagements zkSecurity had access to fixture-based tests that exercised some of the flows. However, it would be useful to have a solid framework to easily test edge-case scenarios and quickly be able to create new markets and new orders.

Overview of the Lighter circuit

Lighter is an exchange implemented as a Layer 2 (L2), as its state transitions are verified and finalized on a Layer 1 (L1). The L1 ensures that each state transition is valid by verifying zero-knowledge proofs, making Lighter a ZK rollup. Zero-knowledge proofs not only prove that a series of state transitions were executed correctly, but also expose compressed summaries of sub state transitions to the L1.

Exposing sub state transitions to the L1 is important for two reasons: not only some of the transitions require additional execution of logic and enforcement of policy on the L1, but issues arising in the operation of the service must not lead to a complete stop of the system. Thanks to data being published on the L1, in case of emergency any L1 user is able to recompute the latest state of the L2 by replaying all of the exposed diffs.

There are two main circuits supporting the Lighter L2 application, comprising what we call the zkLighter circuits:

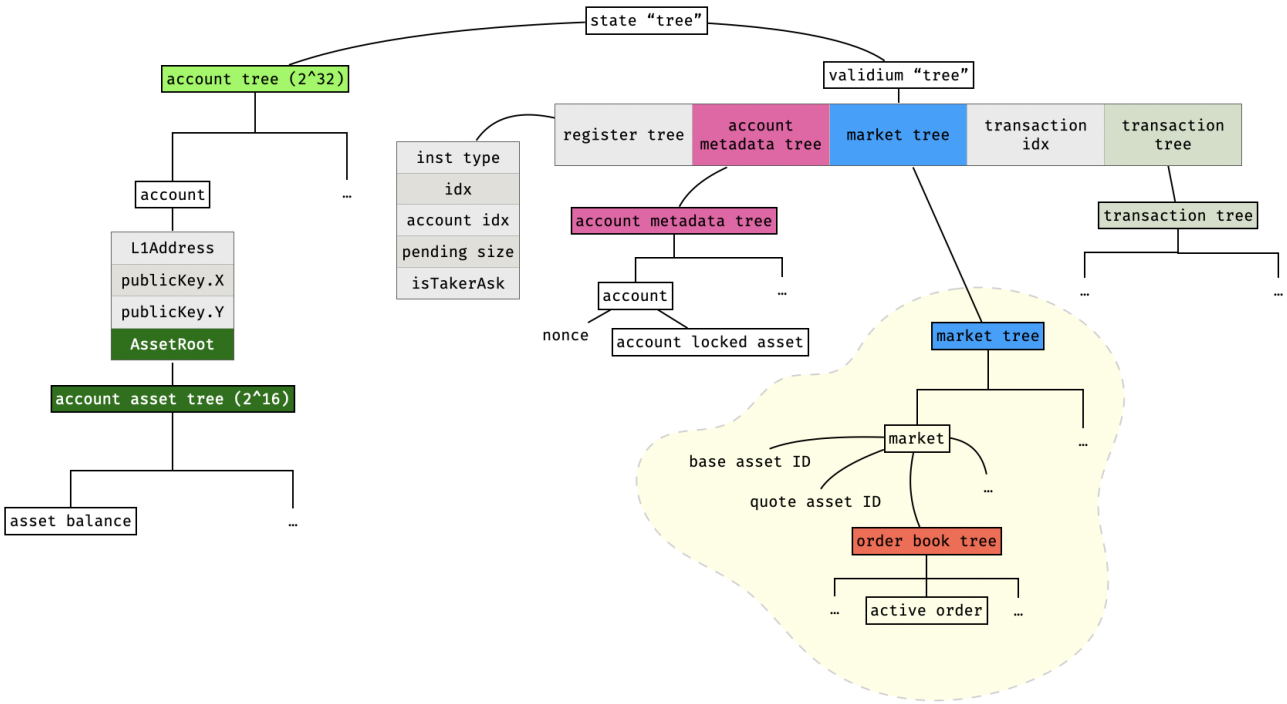
1. **The main operation circuit**, which is supposed to be the main circuit, for as long as the system operates normally.
2. **The exit hatch circuit**, so-called "desert" circuit in the code, which takes over in case of an emergency and allows users to exit the system. See the end of this overview for more information.

In the rest of the overview, we will abuse the term *zkLighter circuits* to refer to the first circuit.

The state of the application

In order to provide updates on a persistent storage, the storage of Lighter is constructed so that it can be authenticated by Merkle trees. An update can take a Merkle tree root which authenticates the state of the application fully, and produce an updated Merkle tree root which can be saved as an authenticated snapshot of the new state.

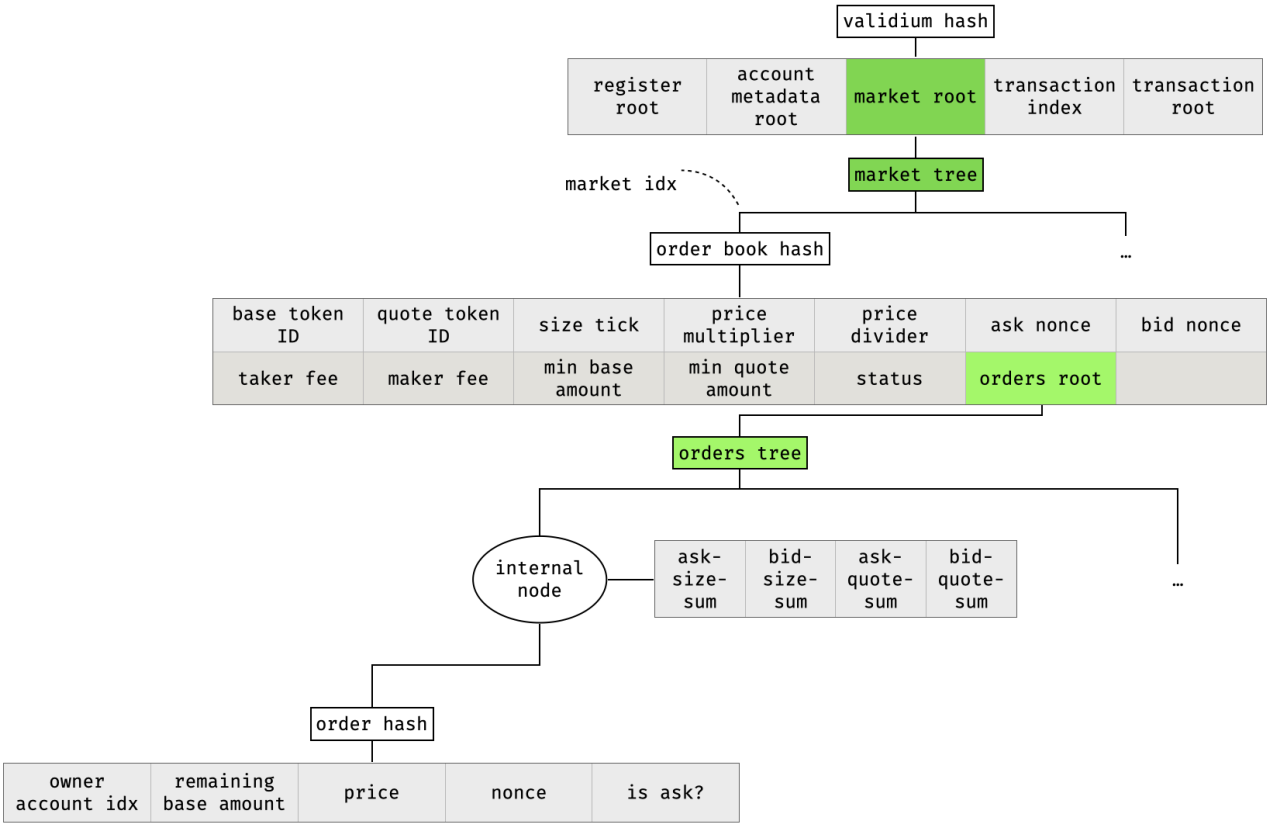
We can summarize the whole state of Lighter as a hypertree (tree of trees) as follows:



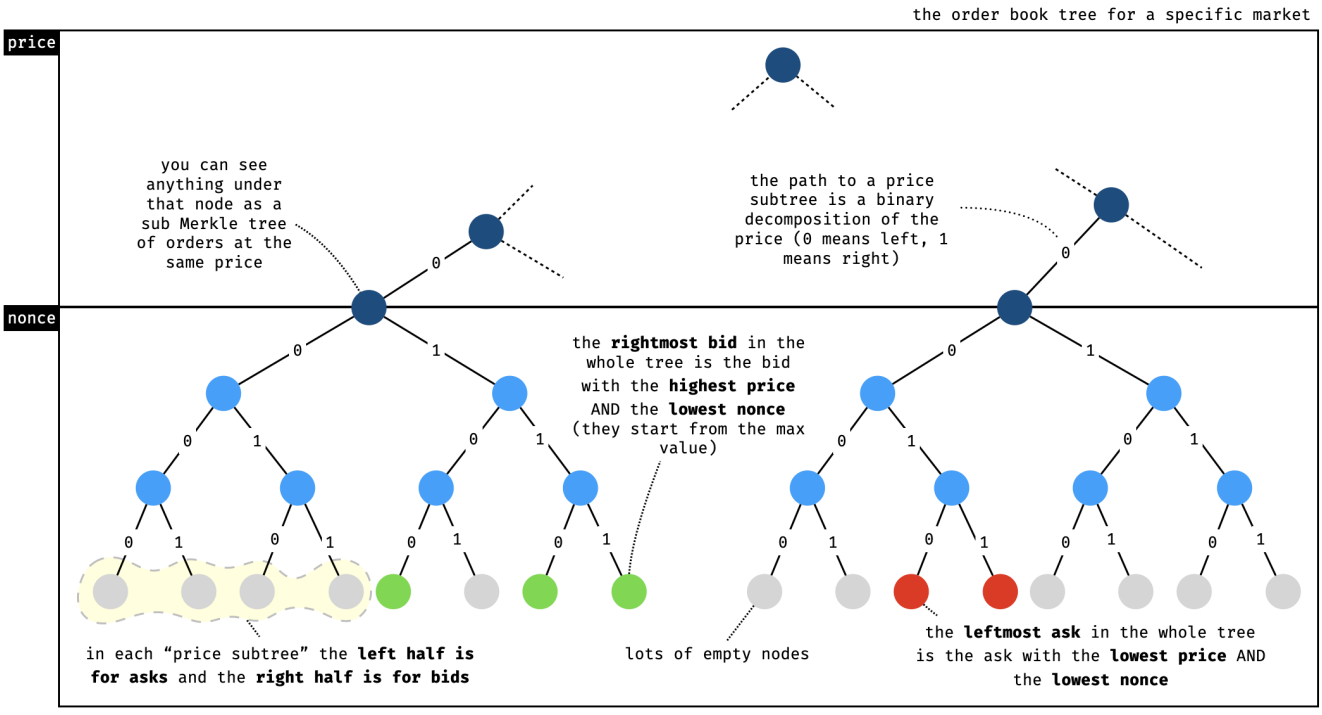
There are 6 real Merkle trees in the above diagram, which are assigned to a different color. The other "trees" are just hashes of their inputs, and do not have internal nodes.

Notice that an account's state is split in two different trees: the one on the left contains all account information except for the amount of assets locked in orders that have been inserted in order books. This extra information is contained in a "account metadata" tree for optimization purposes.

The order book tree of a market is quite special, as it is quite a large tree (of height 72) and contains extra information in all of its internal nodes. So it is worth taking a look at it in isolation:



Orders in the order book are stored under a special path, calculated based on the price and a nonce. As the price is the first part of the path, the leaves are ordered from lowest to highest price, and will naturally paint the tree with bid orders on the left side and ask orders on the right side (and a visible spread in the middle).



There should **always** be a gap somewhere in the tree, with bids on the left side and asks on the right side of the gap, making up the spread.

Note that while asks start being indexed at zero, bids start from the maximum nonce value. This design choice leads to the previous observation on higher priority orders, but also to a special case when asks and bids are crossing in the same "price subtree": if an ask is ever inserted in the same subtree as other bids, then these other bids can be seen as crossing orders to the matched part if the incoming order and will be found on the right side of the ask (and vice versa for a bid inserted in a price subtree with several asks). This observation will be useful later when we discuss the matching engine.

zkLighter blocks

The main object handled by the zkLighter circuit is a block. As in your typical blockchain, a block contains a list of transactions and produces a state transition by applying these transactions to a known previous state (as attested to by the previous block), and attests to the updated state.

Transactions are "executed" one by one, in a way that ensures no cheating in the matching algorithm. That is, that trades are optimal, as long as the ordering of the transactions is done in a fair manner.

Note that some transactions are not real user transactions, but inserted "fake" transactions that help finalize the execution of some types of user transactions (specifically, transactions that might require several "cycles" to complete). We explain this in more detail in later sections.

The logic of a block goes like this:

1. Ensure that the first transaction builds on the block's old state root.
2. Verify each transaction, ensuring on the way that each transaction builds on the previous transaction's output state.
3. Expose as public input the block number, a timestamp, the root state before and after the state transition (as the L1 will need to verify that the state transition built on the latest recorded state), other useful roots of Merkle trees (validium and transaction trees), and finally a vector of public data which describes some of the transactions that made up the state transition.

We summarize the same logic in pseudo-code as follows:

```
def verify_block(public, private):
    # init
    pub_data = [0] * BLOCK_TXS * 24
    pos = 0

    # check first transaction
    assert(private.old_state_root == private.txs[0].state_root_before)
    pending_pub_data, tx_pub_data_size, roots = verify_transaction(private.txs[0],
private.created_at, roots)
    copy(pub_data[pos:], pending_pub_data) # via multiplexer
    pos += tx_pub_data_size

    # check the rest of transactions
    for i in range(1, BLOCK_TXS):
```

```
        assert(private.txs[i-1].state_root_after == private.txs[i].state_root_before)
        pending_pub_data, tx_pub_data_size, roots = verify_transaction(private.txs[i],
private.created_at, roots)
        copy(pub_data[pos:], pending_pub_data) # via multiplexer
        pos += tx_pub_data_size

    # compute new state root (unnecessary as compute)
    new_state_root = private.txs[i-1].state_root_after
    [new_account_root, new_validium_root, new_transaction_root] = roots

    # create commitment as public output
    tx_pub_data_hash = sha256(pub_data)
    commitment_data = [ # 176 Bytes in total:
        private.block_number,         # 8
        private.created_at,           # 8
        private.old_state_root,       # 32
        new_state_root,               # 32
        new_validium_root,            # 32
        new_transaction_root,         # 32
        tx_pub_data_hash,             # 32
    ]
    commitment = sha256(commitment_data)
    assert(commitment == public.block_commitment)
```

Types of transactions

Most of the logic in zkLighter comes from the execution of a cycle, where a cycle contains the logic that can execute a transaction. As previously mentioned, a transaction can be a user transaction or a "fake" transaction that helps drive the execution of a user transaction in multiple cycles.

One way to categorize transactions is to split them in different categories. Some of the transactions are L2 transactions as they are authenticated by the public key contained in the L2 account state:

- **Transfer**. This is a direct transfer of assets between accounts in the L2.
- **Withdraw**. This is a withdraw transaction from the L2 to the L1.
- **CreateOrder**. This creates an order in some market.
- **CancelOrder**. This cancels an order in some market.
- **ChangePubKey**. This allows a user to update their account's public key. Note that this transaction is not "purely" an L2 transaction as it is authenticated mostly on the L1: the L1 contract's logic is in charge of ensuring that the account's associated L1 address is the `msg.sender`.

Some of the other transactions are deemed L1 transactions and are submitted to the L1 contract. We presume that when a block exposes state transitions coming from such transactions, the L1 logic will ensure that they were submitted on the L1 too. These transaction types are:

- **CreateOrderBook**. An admin on the L1 should be able to create a new market to trade a pair of assets.
- **UpdateOrderBook**. An admin on the L1 should be able to update information for an existing market.

- **Deposit**. This allows an L1 user to deposit some specific asset in any account on the L2.
- **FullExit**. This allows a user to empty their account for a specific asset. This means that not only the account of the user will be drained, but all orders for that asset (in any market where that asset could be a pair) will be canceled. Note that this transaction exists to ensure that users can still exit the system even when the system is censoring them (so unlike withdraw transactions, these are posted and authenticated on the L1).

Finally, there are two types of "fake" transactions that are used to finalize the execution of some of the previous user transactions:

- **ClaimOrder**. This transaction is used to continue processing a CreateOrder transaction, as a taker order might remove and update several maker orders in a market's order book.
- **ExitOrder**. This transaction is used to continue processing a FullExit transaction, as the prover requires as many cycles as there are orders associated with that exit transaction.

To help keep track of the status of the execution of a user transaction across cycles, an additional variable named "register" is used. Since there is no theoretical limit to the number of fake transactions that need to be inserted as part of the execution of a single user transaction, the register is also a persistent variable across blocks (as can be seen in the first state diagram).

A note on unrolled logic

The logic of zkLighter is encoded in a zero-knowledge circuit, due to this loops have to be unrolled and real branching cannot occur. This adds quite a lot of complexity to the code which has to ensure that every branch can always be computed and produce dummy results. We mention some of the techniques in this section.

Most of the techniques in this section can be reduced to using **multiplexers**. For example, booleans are often computed and used to ensure that only one path is taken at a time (and one transaction is set at a time):

```
// compute tx type
isEmptyTx := types.IsEqual(api, tx.TxType, types.TxTypeEmpty)
isChangePubKey := types.IsEqual(api, tx.TxType, types.TxTypeChangePubKey)
// TRUNCATED...

isSupportedTx := api.Add(
    isEmptyTx,
    isChangePubKey,
    // TRUNCATED...
)
api.AssertIsEqual(isSupportedTx, 1)

// verify nonce, expired at and signature
isLayer2Tx := api.Add(
    isChangePubKey,
    isTransferTx,
    // TRUNCATED...
)
```

Branches that are not taken often take such flags to avoid asserting on dummy values. To do that, asserting functions are in turn augmented to assert conditionally. For example:

```
func AssertIsVariableEqual(api API, isEnabled, i1, i2 Variable) {
    zero := 0
    i1 = api.Select(isEnabled, i1, zero)
    i2 = api.Select(isEnabled, i2, zero)
    api.AssertIsEqual(i1, i2)
}
```

In addition, seeking and writing to arrays must be done in a way that cover all possible cases:

```
pos := 0
inc := 1

// use incremental arrays to avoid high constraint cmp operation
// pos is used to indicate the current position to write in pubData
// inc is used to indicate whether current position needs to be incremented
// Let's say pubDataSize is 4, the values in iteration will be:
// IsEqual : 0 0 0 0 1 0 0 0 0
// Inc     : 1 1 1 1 0 0 0 0 0
// Pos     : 1 2 3 4 4 4 4 4 4
for i := 0; i < types.PubDataDWordSizePerTx; i++ {
    // for the first tx, no need to iterate for position.
    pubData[i] = pendingPubData[i]
    inc = api.Sub(inc, types.IsEqual(api, i, txPubDataSize))
    pos = api.Add(pos, inc)
}
```

Verifying transactions

A transaction in zkLighter contains all possible transactions for the reason stated above. In addition, it contains enough data to verify a signature (in case of the transaction being a user transaction), as well as witness data provided by the sequencer for the prover to apply the transaction on the state. The witness data mostly consist of the data stored in the latest state that will be mutated by the transaction, as well as Merkle proofs to authenticate the integrity of the data.

```
type TxConstraints struct {
    // tx type
    TxType Variable
    // different transactions
    ChangePubKeyTxInfo    ChangePubKeyTxConstraints
    DepositTxInfo         DepositTxConstraints
    TransferTxInfo        TransferTxConstraints
    CreateOrderBookTxInfo CreateOrderBookTxConstraints
    UpdateOrderBookTxInfo UpdateOrderBookTxConstraints
```

```
    CreateOrderTxInfo     CreateOrderTxConstraints
    CancelOrderTxInfo     CancelOrderTxConstraints
    ClaimOrderTxInfo      ClaimOrderTxConstraints
    ExitOrderTxInfo       ExitOrderTxConstraints
    WithdrawTxInfo        WithdrawTxConstraints
    FullExitTxInfo        FullExitTxConstraints

    // nonce
    Nonce Variable
    // expired at
    ExpiredAt Variable
    // signature
    Signature SignatureConstraints

    // everything else after this point is storage data that is provided by the sequencer and
that can be inserted in the circuit to help execution
}
```

We can summarize the logic of the transaction circuit as follows:

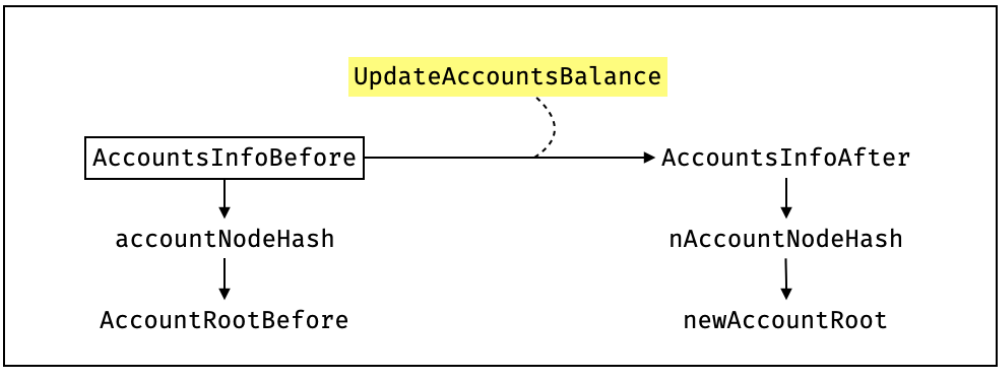
1. Figure out what transaction we are looking at.
2. Hash the transaction data and verify a signature over the transaction in case of an L2 transaction.
3. Compute public data from the transaction (which we will expose publicly eventually).
4. Verify the fields of the transaction (the checks are different depending on the transaction).
5. At this point we also verify that, if the register is non-empty, the correct transaction is associated with it.
6. Calculate "deltas" -- changes based on the transaction type and witnessed leaves of the state.
7. Verify that the current witnessed leaves are in the current state. This means verifying a number of Merkle membership proofs. Up until the roots that are exposed in the public input.
8. Update the now-verified leaves with the previously computed "deltas".
9. Recompute the Merkle roots (using the same proofs used to verify Merkle memberships), up to the roots that are exposed in the public input.
10. Ensure that every transaction is set to zero except the transaction we're looking at.
11. Return the public data, its size (as the vector is padded with zeros), and the updated roots.

The three-step concept of a witnessed leaf being authenticated in a Merkle tree, being updated, and new roots being computed, a huge part of the logic in the protocol described above. We give an example of that three-step process in the diagram below.

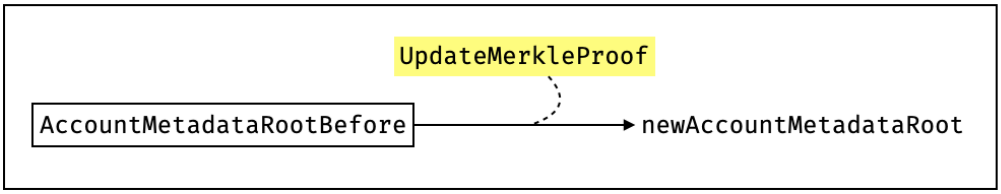
witness data

extended witness

function



The leaves that are being udpated are provided as witness. Two things happened: 1) we recompute the roots from these leaves up to the state root and ensure that they match the previous state and 2) after we perform updates of the leaves in-circuit we recompute the new roots to ultimately provide the updated state root.



We are in a hypertree setting (trees of trees) so updates of leaves trigger updates of roots, that trigger updates of roots, and so on. Note: It is important that the updates use the same neighbor nodes that were used during step 1 of the diagram above to verify Merkle tree memberships.

Most of the complexity within this, itself a complex part of the codebase, comes from updating the state and more specifically, the matching engine responsible for performing user trades. We go over this in more detail in the next section.

Order book transactions

As stated previously, there are two types of order transactions that can be submitted by users: *create order* and *cancel order*. As its name indicates, a cancel order is merely responsible for removing a leaf in an order book, and as such this is the last time we'll mention it.

There are four different types of create order transactions, and they all combine different techniques in order to convince users that there was no cheating on the part of the matchmaker, and that each processed trade was optimal (as long as transactions are ordered fairly). Before reviewing them we introduce a few terms (that are not part of Lighter's documentation):

- **Target leaf**: this is the leaf, that either represents the worst (from the taker's point of view) and latest maker order we need to match — in the case of a full match, or that we might need to insert in the order book — in the case of a partial match. If we end up inserting, the target leaf represents the same type of order as the user's order transaction.
- **Pending leaves**: these are all the leaves that an order could fully consume to partially or fully complete a trade, and which are either higher-priority than the target leaf in case they are of the same type, or crossing orders in case they are of different types (due to the way bids and asks are arranged in a price subtree).

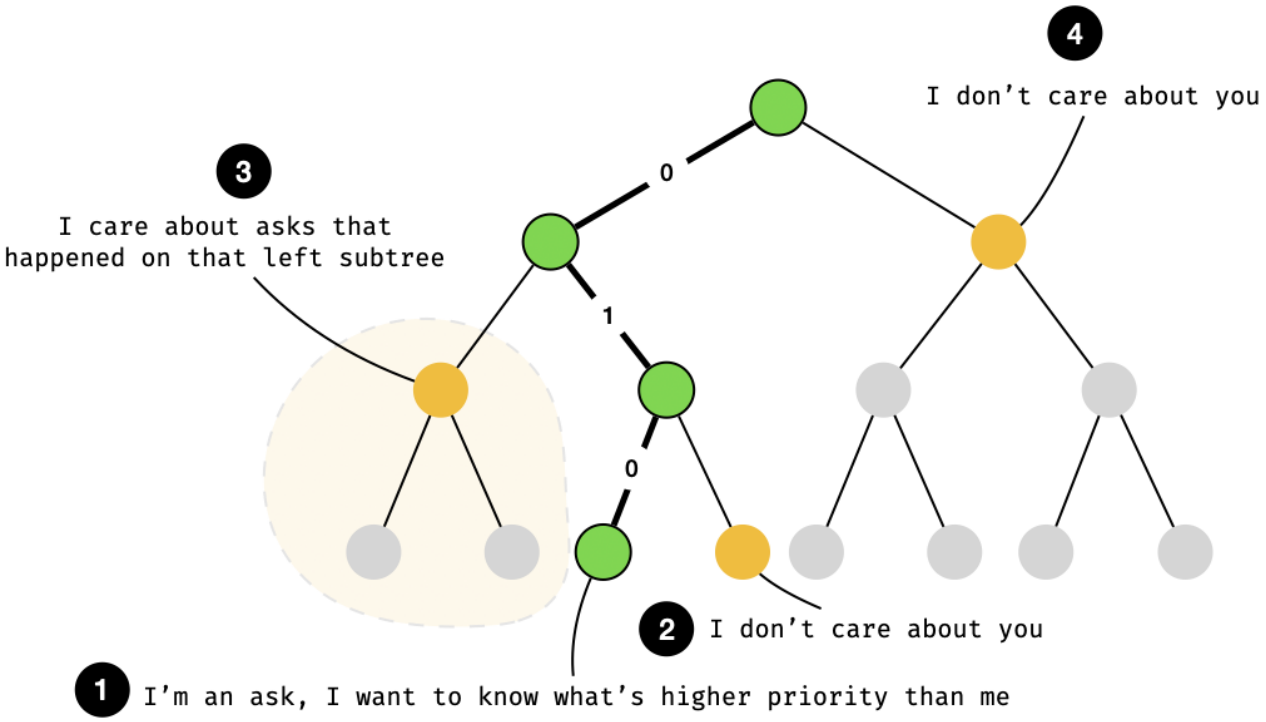
Note that **pending leaves are always constrained to be less than the size of the order transaction**. This is an important property as it is used to prove optimality of the matching for different order types. Intuitively, this property allows us to position ourselves at the right spot in the order tree: right in-between fully matching and not fully matching.

We now review the different types of order transactions.

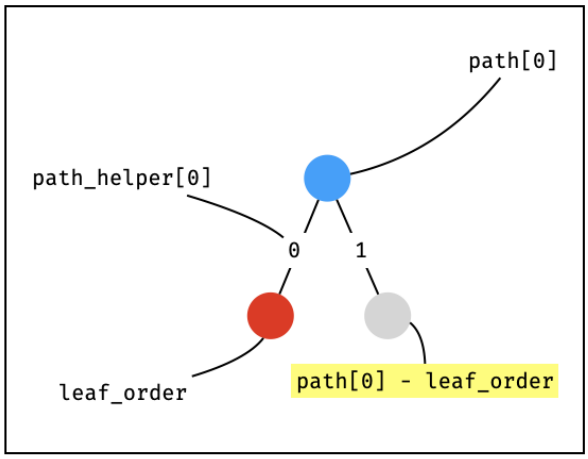
Limit order. This order sets a maximum (resp. minimum) price for a bid (resp. ask). If it cannot be fully matched with existing maker orders in a market, then the remaining amount to match will be inserted in the order book (after being partially matched against as many marker orders as possible).

Thanks to the special composition of the internal node of the order book tree (which contains the aggregated bids and quotes of all children nodes), one can use a target leaf and climb up the tree to compute the sum of all pending orders. In the case of a full match, the target leaf and the pending orders should represent the full amount to trade with (and is easy to verify). In the case of a partial match, the crossing orders in between the leaf to insert and the rest of the tree can be verified to be less than the matching amount (and the price will be right because we are in the right price subtree).

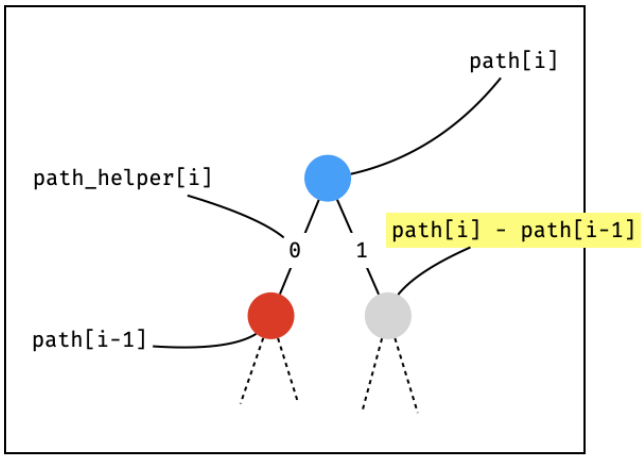
We illustrate how one can climb the tree to compute the sum of all pending orders in the diagram below:



Interestingly, one can guess the aggregated sums contained in a neighboring subtree without witnessing a preimage of the neighboring node (avoiding having to compute a hash): it can be derived from the parent and the current child, as illustrated in the next diagram:



at initialization



at some point i

In both situations, the register is set to `ExecutePendingTrade` and given the remaining amount as "pending size" to claim from the order tree. Enough claim transactions are inserted by the sequencer to reduce that pending size to 0, where each claim transaction consumes the current highest priority order (we can check via the internal nodes that no other higher-priority transactions exist). Once the pending size is decreased to 0, the register can be reset to accept the next user transaction.

Market order. This order is similar to a limit order, but does not end up in the order book. As such it is a taker order (as it does not "make" the market).

For full matching, which is what is expected of a market order, the same kind of target leaf as a full matching with a limit order is expected. If everything goes right the rest of the protocol follows the same logic as a successful limit order.

Market orders have two valid scenarios where they can fail. In such situations, the prover must still be able to process them and prove that there was no possible way to process the orders, as some orders are priority orders which MUST be fulfiled otherwise the protocol's exit hatch is triggered (see later).

The first edge case is that there is not enough liquidity in the order book to match the trade, this is easy to prove as the total liquidity is contained in the root node of the order tree.

The second edge case is when the trade is matched but at a non-desirable price: each market order is explicitly associated with a *slippage* value which dictates the worst price at which to accept a trade. If such a case is met, the prover provides a similar target leaf as in the limit order case. As the sum of pending leaves is always constrained to be lower than the order's size, the circuit can check that the market order is right in-between the target leaf and the pending leaves.

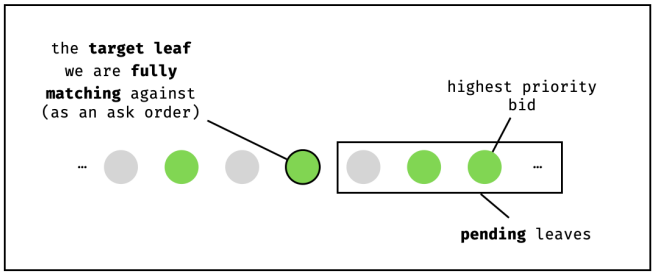
Fill-or-kill (FoK) order. This is a taker order, that is a no-op if it cannot be fully matched against a set of existing maker orders.

FoK orders follow the same strategy as limit orders, except that the process of claim is not started unless there's a full match.

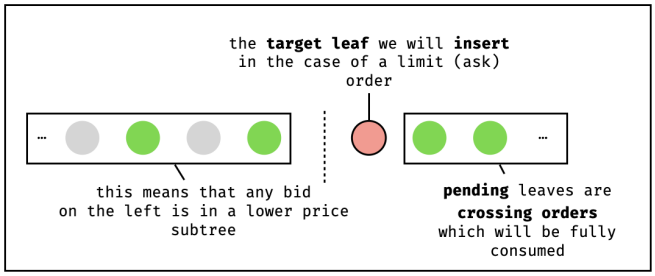
Immediate-or-Cancel order. This is another taker order that can match against several maker orders and does not end up in the order book if only partially matched.

They follow the same strategy as the previous FoK orders, except that the process of claim is still started if there isn't a full match.

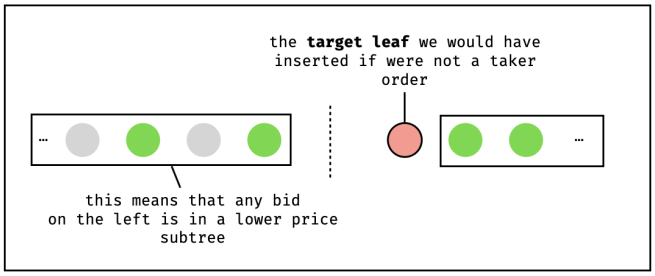
We recapitulate some of the detail in the following diagram:



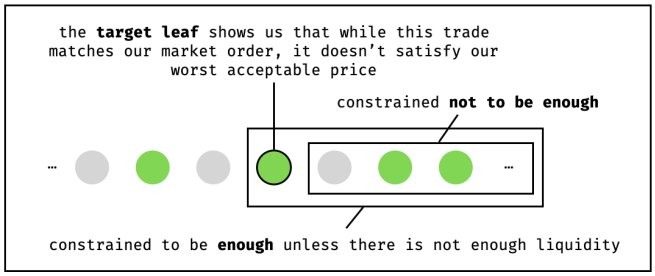
A **full match** is easy to check: a target leaf gives us what we want (or more) when combined with higher-priority leaves (pending leaves). We know it is the optimal leaf because **pending leaves are constrained to contain strictly less than what we want to trade.**



A **partial match** can be proven by pointing to a new leaf (which we would insert in case of a limit order) and looking at the the pending leaves.



Like a limit order, FoK and IoC transactions can be shown to be partial by using an **hypothetical leaf** we would have inserted, and looking at the pending leaves.



In the case of a **bad market order**, we also want to prove that the (**smallest**) fully matched amount is not acceptable.

Exit Hatch Mode

The Lighter smart contract supposedly (as we only looked at the circuit-side of things) has two modes of operation: a normal mode of operation, and a fallback mode (called "desert" in the code) that becomes the persistent default mode in event something bad happens. "Something bad" is defined in the paper to include, for example, not being able to process priority transactions in a timely manner.

Once the exit hatch is activated, the only proofs accepted change to a so-called "desert" proof which only allows users to exit the application and withdraw any funds locked in the application. These proofs can be generated by users as long as they know the state, which can be reconstructed (as previously discussed) by using the data exposed publicly by all state transitions that the smart contract processed.

The proof is on the execution of a simple circuit:

- Check that the state tree correctly authenticates an asset in an account (by doing a bunch of merkle proof verifications).

- As public inputs, expose:

- The account index: so we can save that event and enforce a single exit per account.
- The account L1 address: so we know how to route the unlocked funds.
- The asset ID and asset balance: to know what kind of token and how much to unlock from the lighter app.
- The state root: so we can verify that the execution was done against the correct state saved in the smart contract.

For completion, in pseudo-code:

```
def verify_desert(public, private):
    # 1. verify state root based on account and validium root
    new_state_root = mimc(private.account, private.validium_root)
    assert(private.state_root == new_state_root)

    # construct public data
    pub_data = [
        to_bytes(private.tx.account_index, 32/8),
        to_bytes(private.tx.asset_id, 16/8),
        to_bytes(private.tx.asset_amount, 128/8),
        to_bytes(private.tx.L1_address, 160/8),
    ]

    # verify provided exit tx
    assert(private.tx.L1_address == private.account_info.L1_address)
    assert(private.tx.account_idx == private.account_info.account_idx)
    assert(private.tx.asset_id == private.account_info.assets_info.asset_id)
    assert(private.tx.asset_amount == private.account_info.assets_info.asset_amount)

    # sanitize assets info
    assert(private.account_info.assets_info.asset_id < 2^16)

    # verify account asset root
    asset_merkle_helper = to_binary(private.account_info.assets_info.asset_id, 16)
    asset_node_hash = mimc(private.account_info.assets_info.balance)
    Merkle.verify(private.account_info.asset_root, asset_node_hash,
private.merkle_proofs_account_asset, asset_merkle_helper)

    # sanitize account info
    assert(private.account_info.account_index < 2^32)

    # verify account root
    account_index_merkle_helper = to_binary(private.account_info.account_index, 32)
    account_node_hash = mimc(private.account_info)
    Merkle.verify(private.account_root, account_node_hash, private.merkle_proofs_account,
account_index_merkle_helper)

    # compress public input into public output
    pending_commitment_data = append(private.state_root, pub_data)
    commitment = sha256(pending_commitment_data)
    assert(commitment == public.commitment)
```

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

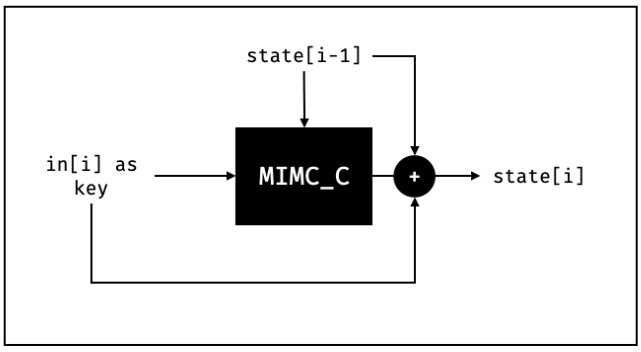
ID	COMPONENT	NAME	RISK
00	mimc_gkr.go	Prover Can Forge Merkle Proofs Due To Lack Of Second Preimage Resistance	High
01	block_constraints.go	Prover Can Forge Merkle Proofs Due To Weak Fiat-Shamir	High
02	matching_engine.go	Prover Can Wrongly Match Limit Orders	Medium
03	*	Manually Hardcoded Constants Is Error Prone	Informational
04	*	Fields of Reference Types in Structs Passed by Value	Informational
05	*	Repetitive Code is Error Prone	Informational
06	*	Redundant Witness Values Can Lead To Underconstraint Issues	Informational
07	*	Witness Constraints Are Not Explicitly Codified	Informational

00 - Prover Can Forge Merkle Proofs Due To Lack Of Second Preimage Resistance

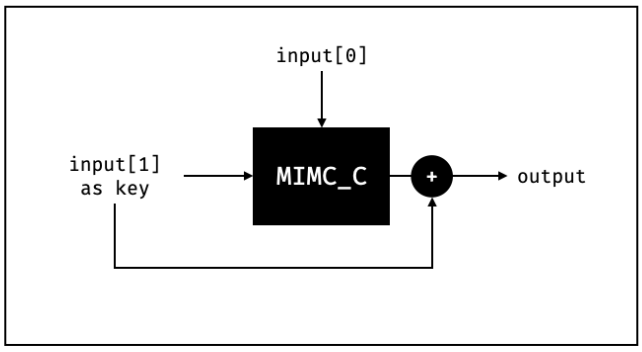
- mimc_gkr.go

High

Description. The MIMC hash function is used throughout the zkLighter circuit logic to verify accesses to the application state, as well as verify updates to the application state. Unfortunately, the way the MIMC hash function is implemented uses an insecure variant of the **Miyaguchi–Preneel construction**, in which two inputs are hashed by using one of the inputs as key to a cipher, and the other output as the previous state.



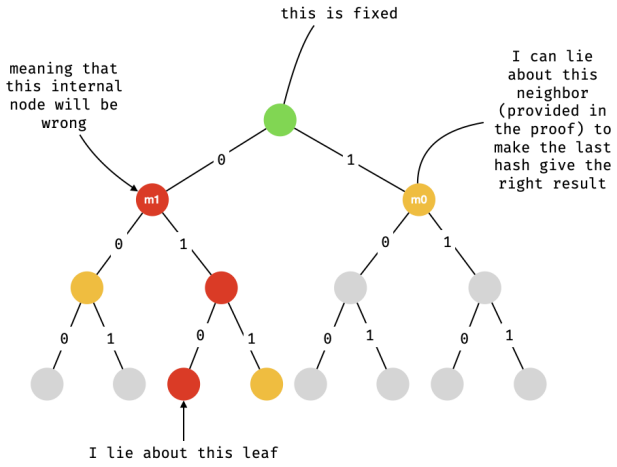
Miyaguchi–Preneel construction used by the native MIMC implementation in gnark

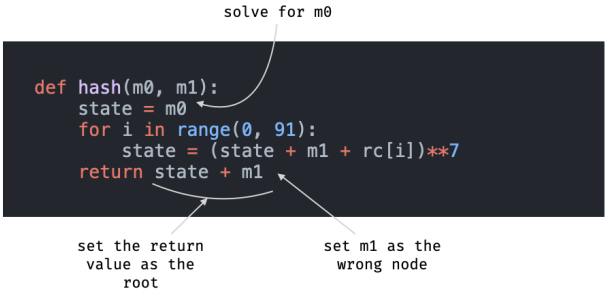


construction used by the MIMC implementation in zklighter

While a correct implementation of Miyaguchi-Preneel is present in the gnark standard library, it is only used in zkLighter to verify signatures on user transactions.

The problem with the construction used in zkLighter is that one can find a second preimage with an arbitrary half of the input (which is of size 2 in zkLighter), as long as they know the other half of the input being used as key (which is always the case in zkLighter). The attack works by modifying the key part of the input, then reversing the MIMC cipher (which at this point is just a permutation fixed by our new key) to obtain a new second part of the input. By doing this, a malicious prover can easily make up a fake application state and falsely prove that it belongs to the Merkle tree. We illustrate this attack in the following diagram:





Reproduction steps. One can use this example to see that we have two tuples `(Left, Right)` and `(FakeLeft, FakeRight)` that lead to a collision using the current MIMC implementation:

```
type Circuit struct {
    Left      frontend.Variable `gnark:",public"`
    Right     frontend.Variable `gnark:",public"`
    FakeLeft  frontend.Variable `gnark:",public"`
    FakeRight frontend.Variable `gnark:",public"`
}

func (circuit *Circuit) Define(api frontend.API) error {
    bN := 1
    gkrMimc := mimc_gkr.NewMimcGKR(api, bN)

    // hash and check expected result for sanity check
    out := gkrMimc.NewMimcWithGKR(circuit.Left, circuit.Right)
    var expectedResult big.Int

expectedResult.SetString("155093269104738063326893757925810573550517767655831606853715588697035
9135725", 10)
    api.AssertIsEqual(out, expectedResult)
    api.Println(out)

    // second hash that results in the same root
    api.AssertIsDifferent(circuit.Left, circuit.FakeLeft)
    api.AssertIsDifferent(circuit.Right, circuit.FakeRight)
    out2 := gkrMimc.NewMimcWithGKR(circuit.FakeLeft, circuit.FakeRight)
    api.AssertIsEqual(out, out2)
    api.Println(out2)

    // (this requires Groth16 to run, IsSolved won't work)
    initial_challenge := 1
    err := gkrMimc.VerifyGKRMimc(initial_challenge)
    if err != nil {
        panic(err)
```

```
    }
    return nil
}

func TestMIMCWithGroth(t *testing.T) {
    circuit := Circuit{Left: 43242304923, Right: 49308420398432, FakeLeft:
"85466375412188628555171888069704528372910204014129792970172931210517590020 63", FakeRight:
403584930850349}
    witness, err := frontend.NewWitness(&circuit, ecc.BN254.ScalarField())
    if err != nil {
        panic(err)
    }
    oR1cs, err := frontend.Compile(ecc.BN254.ScalarField(), r1cs.NewBuilder, &circuit)
    if err != nil {
        panic(err)
    }
    pk, err := groth16.DummySetup(oR1cs)
    if err != nil {
        panic(err)
    }
    _, err = groth16.Prove(
        oR1cs, pk, witness, backend.WithSolverOptions(solver.WithHints(types.IntegerDivision,
mimc_gkr.MIMC2Elements)),
    )
    if err != nil {
        panic(err)
    }
}
```

To obtain similar results, or perform different types of attacks, one can use the following **SageMath** code:

```
# the zkLighter MIMC implementation
def mimc_hash(left, right):
    state = left
    for i in range(0, 91):
        state = (state + right + F(rc[i]))**7
    return state + right

# function to find a left input, given a fixed root and an arbitrary right input
p = 21888242871839275222246405745257275088548364400416034343698204186575808495617
F = GF(p)
inv7 = inverse_mod(7, p-1)
def find_left(right, root):
    state = F(root - right) # reverse whitening
    for i in range(90, -1, -1):
        state = state^inv7
        state = state - right - F(rc[i])
    left = state
    assert(mimc_hash(left, right) == root)
    return left
```

```
# round constants
rc = [
    "12136087830675299266258954793902014139747133950228959214677232437877732505267",
    "19492629157426165099246390879950520574395336886394435284199020501015112532 19",
    "19696026105199390416727112585766461108620822978182620644600554326664686143928",
    "48372029280867185761938806382954314614987645555984302211572830922387763420 56",
    "20604733757835564048563775050992717669420271708242272549953811572144277524421",
    "32114757189773765658803873131106064505251425917079612263037408259565638669 38",
    "20322324153453907734144901224240556024026610989461831427966532469618423079473",
    "79349733197609764850217910305375018656566197192647596602778234466490821473 12",
    "93041548695073791401376927980165147551185850265626366943934211183225472685 0",
    "13233069796564124818867608145207094060962111073577067140167532768756987412088",
    "21056848409984369169004352317081356901925342815742628419179104554944602838181",
    "76099650490602515516911280764522001681936746280229734044752567392313962950 27",
    "28755699896075890803237840516374028764673074023382535860461461594741383885 18",
    "16384054153715373365524617689856265664643515017145151628498643996405801025 78",
    "15419971351110340210204119021937390512827818431981795058254849419538982779889",
    "62665208979082970230425701168882469005671395315900207709526024883485582650 61",
    "14039893748423238973883972164169603996654684831868979824941451015257316714495",
    "17495914808944773938291362208338085117997720817217450529979495804567647637318",
    "55600433679412966632969088829271023187098036931675545715583171387751654575 66",
    "67951636862023291737677541693726941166060673922567760993116053167379170915 9",
    "20771173458695616083113300195745636859853909352816078680615698162064064254487",
    "90619499457323494975540376714873094080195951758882535636390037408463451732 68",
    "65892830897560496271665771322641711234812246893609694703708116041001567642 33",
    "35335275162027560963890603567773952693089818394034766520289176460882 4581131",
    "56169422278506176780468402509033043583332983068072207663477468092670324552 99",
    "19134688161961603498559262912818080142324701420702686735061929518115443109100",
    "44550121386300754862543075336068581252672142651315018165980588111726923281 01",
    "10793166202851599893237663367817743308639336679992856426902640679097285197834",
    "15056866545271068254544312503685146788561860865190761206515636207319868585312",
    "18588820303015761108497689698317977183405401884497470414262723108370171102023",
    "19833892328086915832797048699984794667331247199325415348986995942708708405059",
    "89895339673044582594000348846525198348687685968888118035638669308589942721 54",
    "11340240789075205057343229997968129213431697722131695573513514123825351727265",
    "19892667690111598338150747633561348627404155092311695371528902260306500224478",
    "56752655667522648790353740326672206981342171734644621062435511633730508476 32",
    "20083467967117235977304805384179142748526655108920556941636271719496304583696",
    "17241462814856629393310447955833139605117065616411368427339901837278291194631",
    "45952300585670766828334890208187307967576529119196746082375600354066237650 3",
    "75361041112463978074286110272674704670600287624375265446518791773916299029 52",
    "10590470262497492482063013216166955143786637478931633085736787678537413766247",
    "15043612058042949906959587136396856430320834576029342208816587940851697052752",
    "73300560668983401393476786893859001070772599491111824223296271418312218638 55",
    "19916604609861621491722130626309103960340872015429661158412002662197451448107",
    "30830652999707021353313913387586244328639801957291438053801564518750582331 8",
    "16861578445042558674239888228729122953078668310622552358464602254565597746836",
    "23663597559390990316695740807706689413122802406629858152539339006289486451 91",
    "87889144015742234246327817183582284684598441755153830314405523062657120714 50",
    "97799875140997040072790270211667466303181489451767980631518893268958891744 58",
```

```
"131356093038262001400652206698313030271682273758514832141614333893869376141 36",
"118824159826171237109037040931740712608016100041085016675048261032646445018 8",
"469498662215718397357268216550077761373913731468901939977678820430437663499 2",
"280889811426289863581848013851749424156779773586847475545537819491758407653 7",
"594881556321213784966972913980427943353177799337203673725876248341280093652 4",
"849683814107726263662301346978028376180701801182901389606674194904391130348 2",
"137027028000861187733148226291464578864213846185090276535110221004036087359 78",
"128688574990165958444408798980207546660746573848805400153153989929390707086",
"127158953615751104534375914831218030136556029378657835805041310505197796815 04",
"171799781201809570503305238492841679363363913179378616381416134314070202956 29",
"115884590028413365871021290610650701543475596480889060777599497169147378792 89",
"168729015953654988689927854892774175085256078299955360517770595983890795563",
"171658306321293553575062661489525572681714448712932074816352640370521950451 34",
"122851384228111757805584265519047159897737127182243124283955752354669527139 40",
"198715459324780727086834750671705847042178296015108449147524715847558618448 6",
"936832494647391757736430222708683185711203089414475274201868885273866499292",
"276390375400048028770868893043339394227746425002804095835086821971106698321 2",
"145575242459525978936366749843760610341402097150563076010994981050108118179 76",
"862108354652939878467134653496741037664236685176575195514167035293726226296 4",
"166271336979508762235205710908227972845299503155492434432132685077325898096 68",
"303723762305683995828190932857714327434925916324390000789863988642703554571 5",
"129953224448982261091500404880922969182875011696813134864509280802658596784 31",
"273317513961346011833109147570522958733589934253998259932757862822598729669 3",
"633090402485079915424146825243739126836390686026807726967963510593091049369 8",
"673729388333305357435458182733079795628693734408546321192938845510515338184 0",
"516983374825367886164645961044000760681248086390214515334191868046019974493 7",
"566334215202987672533710579761545770464975508873027804231788316899729732365 8",
"782328933854362285928106347130632764069779533315203123527092257176814439044 2",
"134062001076271865392959774695110253334561695109759633185224065203785416025 8",
"148977288698021409615982049119956312031574475694046013256745682269823099541 50",
"392018124866990209108785883333829486897323525593091953436038191276969589879",
"435898044557960665437580260079284983588895103844486797156208731954139457048",
"999349789670302598986704864604317341834553635571594976929945423779738628683 3",
"154509309335161865521237774056590144114207291035350318264054405547669016840 12",
"209030342685823754776732196012163748440765053922631955738196382539636289876 77",
"348224202227067409523015034594760540724155416788447046272749651496558771702 0",
"732769172997982430216649945191915096929481167749060464037156114774422339607 7",
"203203514612199027342796649365513322853094204915328382288831164301440434702 24",
"100801720658345824319139010682789600336445209935650226005277981462273897062 43",
"758548485765564331487492743943021782712638295532038894145928628138183930261 2",
"702048357029231369272975870438326776182962776748659786521535277002437836371 3",
"941291532104334424641305059501533291522696777175241048173250246652890953591 5",
"971727933437166027795268157074272047241234442688299912329947560792851916 57",
"989936738509869696303461259961180416799336996278839726298549392279135131892 0",
"204931020783300644621340683366669244273236010253836179836641211483574213871 85",
"376104193236800645784598601442980462029717214514241805420328690904096811824 1",
"153873969800204452597241760614080960134751820491582072838485466710914733351 1",
"138022438756179908100772290531596394031898616262108989187960145753830627904 41",
"148024161691010272482362353563845284988673887800499572979161999596945759897 98",
"128557447266518500233115642526147814225048145397610236184087231130574408865 58",
"301736504303808632364878020852862142039403228600798977539162715578424897876 6",
```

```
    "6315674106586331242761612192226077184856834393892197849679034296526217823177",
]

# try it with some arbitrary values
left = 43242304923
right = 49308420398432
root = mimc_hash(left, right)
fake_right = 403584930850349 # modify the right input
fake_left = find_left(fake_right, root) # solve for a new left input
assert(mimc_hash(fake_left, fake_right) == root)
```

Recommendation. Implement a sponge construction, such as the one used in the **Poseidon hash function**, which is a more secure variant of the Miyaguchi-Preneel construction (as it protects against **length-extension attacks**). This will ensure that the MIMC hash function is resistant to second preimage attacks, and that the zkLighter circuit is secure against forged Merkle proofs.

Although, note that in practice the Miyaguchi-Preneel construction is simpler to implement than a sponge construction, and that length-extension attacks should not impact zkLighter as there are no secrets being hashed.

01 - Prover Can Forge Merkle Proofs Due To Weak Fiat-Shamir

● block_constraints.go

High

Description. The zkLighter circuits make use of recursion to delegate the computation of a large number of **MIMC hashes** to a more efficient proof system (**GKR**). The way it works is that MIMC digests are directly inserted (so-called "hints") in the zkLighter circuits, and later a GKR verifier (in-circuit) verifies a proof that these values were computed correctly.

A naive implementation of this approach leads to computational issues as the number of in-circuit hashes required to transform the (interactive) GKR scheme into a non-interactive one would be larger than the number of hashes delegated to the GKR proof. This is because transforming an interactive proof system into a non-interactive one requires a Fiat-Shamir (F-S) transformation, which starts by hashing the entire statement. In other words, imagine implementing the F-S part of the in-circuit GKR verifier by starting to hash 100 inputs in your circuit, so that you can verify a GKR proof that 100 (other) inputs were hashed correctly. Nonsensical!

A solution is documented in **Recursion Over Public-Coin Interactive Proof Systems; Faster Hash Verification**, and implemented in gnark, which is what zkligther uses. The idea is to provide a public input of a hash of the initial transcript, and prove that this hash is computed correctly outside of the circuit.

Unfortunately, the zkLighter circuit does not make use of this feature, and provides its own initial randomness in the F-S transcript using the block commitment:

```
api.AssertIsEqual(commitment, block.BlockCommitment)
err = gkrMimc.VerifyGKRMimc(commitment)
if err != nil {
    return err
}
```

This means that in practice, a malicious prover can produce a GKR proof for any input/output pairs, as the initial randomness is not strongly linked to the statement being proven. For example, a prover could perform this attack to obtain a wrong root hash (which would then store false data) of one of the state Merkle trees. This issue is called a weak F-S transformation as per **Weak Fiat-Shamir Attacks on Modern Proof Systems**:

> *"In weak F-S, only the prover's first message A is hashed. In strong F-S, the hash additionally includes the public parameters and public input"*

We give a short description of how this attack could work. But first, let's give a short recap of an honest verification of a GKR proof under the weak F-S construction. Steps unnecessary for the explanations have been omitted. Note also

that because GKR expects values to be encoded on the boolean hypercube, most values in this description are supposed to be seen as vectors of their binary decomposition.

1. The GKR verifier produces $\tilde{V}_0$ by interpolating the outputs.
2. The GKR verifier produces a random point g with some **initial randomness**. This is used to reduce the problem of checking $\tilde{V}_0(z) = \sum_{x,y} f(z,x,y)$ to checking it at a random point g (S-Z).
3. The GKR verifier begins the first sumcheck with $\tilde{V}_0(g) = cst = \sum_{x,y} \cdots$ (we omit the rhs)
4. The rest of the protocol goes on...
5. At the end, the GKR verifier interpolates the inputs into the polynomial representing the last layer $\tilde{V}_d(z)$.
6. Then they perform the final check by computing $\tilde{V}_d(r_1, r_2, \cdots)$ on a number of challenges obtained during the last sumcheck.

There are two possible attacks here, we could attack the inputs, or the outputs.

Let's explain an attack on the outputs. To do that, let's first notice that since the initial randomness does not include the inputs and the outputs (by definition of the weak Fiat-Shamir construct), then we can reorder the beginning of the previous list:

1. The GKR verifier produces a random point g with some **initial randomness**.
2. The GKR verifier produces $\tilde{V}_0$ by interpolating the outputs.
3. The GKR verifier begins the first sumcheck with $\tilde{V}_0(g) = cst = \sum_{x,y} \cdots$ (we omit the rhs)
4. The rest of the protocol goes on...

We perform the reordering of step 1 and 2 to emphasize that a malicious prover can at this point produce outputs that interpolate to $\bar{V}$ to conform with $\bar{V}(g) = cst$ (where cst is the same constant used in the honest run of the protocol).

A malicious prover can do that by arbitrarily choosing all $output[i]$ values except for one $output[j]$ (such that $i \neq j$) which we'll call the **scrambled output**. Then the malicious prover can find a scrambled $output[j]$ that satisfies the following equation (where everything but the scrambled output is fixed):

$$cst = \bar{V}(g) = \sum_{i \neq j} (eq(g,i) \cdot output[i]) + eq(g,j) \cdot output[j]$$

Where $eq(x,i)$ can be seen as a Lagrange Basis on the boolean hypercube.

Note that in an attack, a malicious prover would use a scrambled output that is likely not to cause issues right away (for example, an account's asset root).

Recommendations. Make use of the commit feature of gnark that was designed specifically for this use case. To do this, pass every input being hashed as well as outputs obtained to the `Committer.Commit` function of the gnark frontend:

```
cmt, err := api.(frontend.Committer).Commit(whatToCommit)
if err != nil {
    return fmt.Errorf("commit: %w", err)
```

```
}
```

Alternatively, consider implementing hashing in-circuit. We measured that 2^{13} hashes were being delegated to the GKR prover, and that verifying a GKR proof consumed $\sim 2^{20}$ constraints in-circuit. It is possible that verifying hashes in-circuit would be cheaper due to the "low" number of hashes.

02 - Prover Can Wrongly Match Limit Orders

● matching_engine.go

Medium

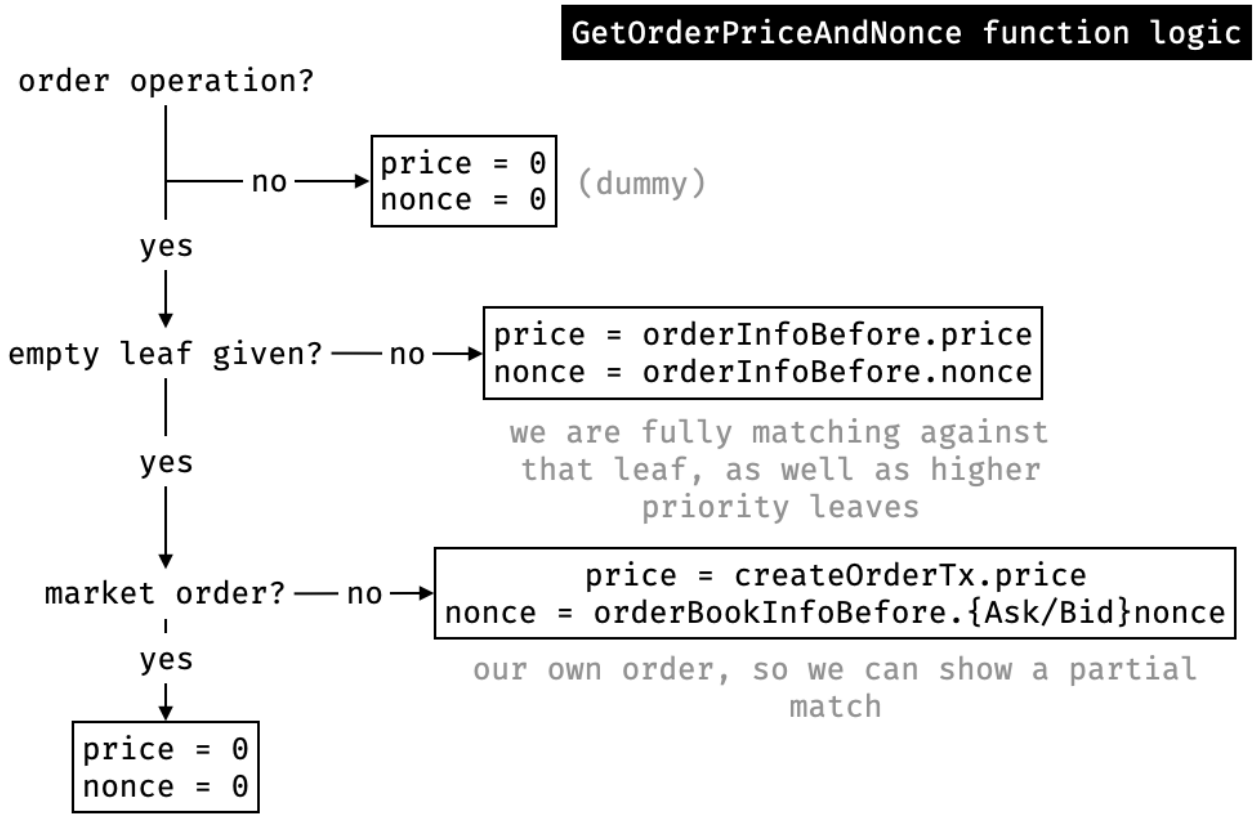
Description. As described in the overview of the report, limit orders are order transactions that can be fully matched or partially matched. In both cases, the constraints must ensure that the match executed by the prover is the correct one.

In the case of a **full match**, the prover is given (as `tx.OrderInfoBefore`) the lowest priority leaf that, when combined with higher priority leaves from the same types, fully matches the limit order. In the case of a **partial match**, on the other hand, the prover must use the leaf at which it will insert the limit order eventually (after processing all the crossing orders in the tree).

The function in charge of producing the correct path to the leaf is `GetOrderPriceAndNonce`:

```
// Calculates and returns the order price and nonce for the given order leaf
// If given order is not empty, returns the order price and nonce from the order leaf
// If given order is empty:
//   - If cancel, claim  or exit order, VerifyCancelOrderTx, VerifyClaimOrderTx or
VerifyExitOrderTx should fail
//   - If market order, returns LastOrderBookOrderPrice and LastOrderBookOrderNonce
//   - If limit, FoK, or IoC order creation, returns the given order price and nonce
//   - If not order operation, returns LastOrderBookOrderPrice and LastOrderBookOrderNonce
func GetOrderPriceAndNonce(
```

Unlike what the function's documentation seems to indicate, it only behaves correctly in the case of a limit order when given an empty leaf. The problem is: `tx.OrderBookInfoBefore` is not guaranteed to be the correct matching maker order. We illustrate the logic as a diagram below:



Note that immediate-or-cancel (IoC) and fill-or-kill (FoK) orders behave in similar ways (but without inserting orders in the order book as explained in the overview of this report). Luckily, they both check that if the match is partial, then the given leaf was empty (which would have force`GetOrderPriceAndNonce` to produce the correct result):

```
// verifyImmediateOrCancelOrder executed on all available liquidity
func verifyImmediateOrCancelOrder(
    api API,
    isImmediateOrCancelOrder Variable,
    isFilled Variable,
    isLeafEmpty Variable,
) {
    // Immediate and cancel orders will always execute, so it cannot fail like the FoK one.
    //
    // If the order was not fully filled, we must check that all possible liquidity was
accounted for by the `GetQuote` method.
    // This is done by ensuring that the provided `orderInfoBefore` references the current
order.
    // If the provided leaf is empty, the behaviour of `GetOrderPriceAndNonce` ensures that the
order path will be
    // generated using tx.Nonce and tx.Price, thus pointing to our current order.
    types.AssertIsVariableEqual(api, api.And(isImmediateOrCancelOrder, api.IsZero(isFilled)),
isLeafEmpty, 1)
}
```

Recommendation. Add verification for limit orders making sure that what is passed in `tx.OrderInfoBefore` is the correct matching maker order.

03 - Manually Hardcoded Constants Is Error Prone

● *

Informational

Description. A lot of the values used in the zkLighter circuits are hardcoded (for example, in files like `circuit/constants.go` and `types/typesize.go`).

For example, many of the witness variables are range-constrained and exposed as public inputs using hardcoded values in bits:

```
// --- circuit/types/pubdata_helper.go ---
func CollectPubDataFromDeposit(api API, txInfo DepositTxConstraints) (pubData
[PubDataBitsSizePerTx]Variable) {
    // TRUNCATED...
    copyLittleEndianSliceAndShiftOffset(api, TxTypeDeposit, TxTypeBitsSize, &currentOffset,
pubData[:])
    // TRUNCATED...
}
// --- circuit/types/utils.go ---
func copyLittleEndianSliceAndShiftOffset(api API, txField Variable, txFiledBitsSize int,
currentOffset *int, pubData []Variable) {
    txFiledBits := api.ToBinary(txField, txFiledBitsSize)
    // TRUNCATED...
}
```

Some of the `_BitsSize` hardcoded constants are meant to represent the different sizes of the types associated with the in-circuit variables. For example, `AccountIndex` is a `uint32` and so `AccountIndexBitsSize` is hardcoded to 32 bits.

Some other constants are meant to handle the largest possible values for enum types. For example, a `TxTypeBitsSize` (hardcoded to 8 bits) should be large enough to handle the largest `TxType` value (11).

While hardcoding values is good practice, hardcoding a large number of them can be error prone. An update of the protocol or a refactoring of the code might change assumptions about the logic, impact several of these values, and lead to undefined behavior.

Recommendations. One way to improve auditability of constants is to generate them dynamically from a single source of truth. Listing the different types and enums and generating the corresponding bit sizes and other constants from them would ensure that the values are always correctly hardcoded in the circuit. Golang's `go generate` could be used for this purpose.

In addition, the generating code could ensure that some sizes have proper limits, in order to ensure that no overflow can occur when operating on values within the zkLighter logic. For example, multiplications of large values could overflow the scalar field size used to instantiate the zkLighter circuits (currently **BN254** is used, which has a scalar field of 254 bits) and lead to undefined behavior.

04 - Fields of Reference Types in Structs Passed by Value

● *

Informational

Description. Passing structs by value creates an illusion of fully copying the struct data, when in fact that may not be happening.

For example, fields of `AccountConstraints` struct are either `Variable` or struct with `Variable` fields:

```
type AccountConstraints struct {
    AccountIndex Variable
    L1Address    Variable
    AccountPk    eddsa.PublicKey
    AssetRoot    Variable
    // at most 4 assets changed in one transaction
    AssetsInfo [NbAccountAssetsPerAccount]AccountAssetConstraints
}
```

Passing a struct by value creates an illusion that it is safe to copy or mutate. This is only true, when no fields in the struct are of "reference types". By "reference type" we mean that, when assigned to a variable or passed to a function, the data is not copied.

In the code below, `accountInfos` is a fixed size array of `AccountConstraints` values which are structs with `Variable` fields. Fixed size arrays in Go are not reference types, but fields of `AccountConstraints` potentially are. `Variable` is an alias to `interface{}` so it can store anything including pointers.

```
func UpdateAccountsBalance(
    api API,
    accountInfos [NbAccountsPerTx]types.AccountConstraints,
    accountDeltas [NbAccountsPerTx][NbAccountAssetsPerAccount]AccountAssetDeltaConstraints,
) (AccountsInfoAfter [NbAccountsPerTx]types.AccountConstraints) {
    AccountsInfoAfter = accountInfos
    for i := 0; i < NbAccountsPerTx; i++ {
        for j := 0; j < NbAccountAssetsPerAccount; j++ {
            AccountsInfoAfter[i].AssetsInfo[j].Balance = api.Add(
                accountInfos[i].AssetsInfo[j].Balance,
                accountDeltas[i][j].BalanceDelta)
        }
    }
    return AccountsInfoAfter
}
```

Bad things that can happen when passing a struct like `AccountConstraints` by value:

- A copy of the struct is created, giving an impression that all data is copied, when in fact it's not.
- Data passed in the `accountInfos` arg might be mutated *in* the function (not the pointer, but the data itself).
- Data returned from the function (which was "copied" from the argument) might be mutated *long after* being returned, if the client code mutates the original argument.

Happily, we couldn't find any evidence of the above happening, since **most** Gnark API methods do not mutate passed `Variable` in-place, **except** for `api.MulAcc()`, and it is not used in zkLighter circuits we reviewed.

(Side note: in the code sample above, **copying** `AccountsInfoAfter = accountInfos` is **unnecessary** since we're immediately overwriting all the data in `AccountsInfoAfter`.)

Recommendations. The following includes an actionable item on the current code, and what to look out for during future development of the protocol:

1. Pass and return structs with `Variable` fields **by pointer only** — this helps recognize that the data in them is mutable and avoids passing large amounts of data via the call stack. Replace unnecessary copying of structs (like in the code sample above).
2. Future development: implement `Clone()` method for structs that **require** copying (most don't, and none of the currently implemented in zkLighter do). A `Variable` could be cloned with something like `b := api.Add(a, 0)`. Use `Clone()` every time a copy of a struct is needed. Be aware of in-place `Variable` mutation when/if using `api.MulAcc()` — this is a case when cloning **is** necessary.

05 - Repetitive Code is Error Prone

● *

Informational

Description. The zkLighter circuits are written to handle many different branches, and due to the nature of zero-knowledge circuits, all branches must be encoded. Even unused branches and their unused objects. As such, many objects are also sometimes declared as `dummy` objects.

A common pattern in the codebase is thus to check if a struct is "empty". The following example is a function that returns 1 if a struct is empty, and 0 otherwise. The struct is defined as being empty if all of its fields are defined as being empty.

```
func IsEmptyOrderNode(api API, order OrderConstraints) (isEmpty Variable) {
    isEmpty = 1
    isEmpty = api.And(isEmpty, IsEqual(api, order.OwnerAccountIndex, ZeroInt))
    isEmpty = api.And(isEmpty, IsEqual(api, order.RemainingBaseAmount, ZeroInt))
    isEmpty = api.And(isEmpty, IsEqual(api, order.Price, ZeroInt))
    isEmpty = api.And(isEmpty, IsEqual(api, order.Nonce, ZeroInt))
    isEmpty = api.And(isEmpty, IsEqual(api, order.IsAsk, ZeroInt))
    return
}
```

The problem is that it is not certain that **all fields of the struct have been checked**. Manual checking is currently the only way to ensure that this has been the case. For example, the snippet above might have forgotten to check some `order.Something` field. While this is not the case, it might in the future due to a refactor or code change.

Recommendations. Note that in Rust we can use a destructuring operator and rely on the compiler to throw warnings if we don't use a variable:

```
let OrderConstraints { a, b, c } = order;
// the compiler will warn us if we didn't use one of the fields
```

A proposal for such a syntax in Golang was **shut down** with the argument that Golang can still do this using **reflection** (runtime functions to analyze actual types being used, as this can't be done at compile time).

For example, the following function could perform the same logic while being resistant to refactoring mistakes:

```
import "reflect"

func IsEmpty(api API, v reflect.Value) (isEmpty Variable) {
    isEmpty = 1
```

```
    for i := 0; i < v.NumField(); i++ { // will panic if v is not a struct
        field := v.Field(i) // TODO: perhaps panic if the field is not a Variable
        isEmpty = api.And(isEmpty, IsEqual(api, field, ZeroInt))
    }
    return
}
```

Another way could be to use `` `go generate` `` which we have already recommended in **Manually Hardcoded Constants Is Error Prone**.

06 - Redundant Witness Values Can Lead To Underconstraint Issues

● *

Informational

Description. It is important that in a circuit, each witness value that needs to be constrained is properly constrained. This is an error-prone process, especially when the language or framework at play does not provide strong typing or other mechanisms to create safe-to-use objects.

Having redundant variables for the same witness is a common source of underconstraint issues, as one of the variables could be properly constrained while the other one is not. In these cases, if no code constrains that the two values are identical, a bug could be introduced.

One example is in `desert/` where `AccountInfo` repeats a number of witness variables already provided in the user transaction (`L1Address, AccountIndex, AssetId, AssetAmount`):

```
type ExitTxConstraints struct {
    AccountIndex circuit.Variable
    L1Address    circuit.Variable
    AssetId      circuit.Variable
    AssetAmount  circuit.Variable
}

type AccountAssetConstraints struct {
    AssetId Variable
    Balance Variable
}

type AccountConstraints struct {
    AccountIndex frontend.Variable
    L1Address    frontend.Variable
    AccountPk    eddsa.PublicKey
    AssetRoot    frontend.Variable
    AssetsInfo   types.AccountAssetConstraints
}
```

Another example exists in `VerifyBlock()` where `NewStateRoot, NewValidiumRoot, NewTransactionRoot` are reintroduced unnecessarily as they are already contained (and constrained by `VerifyTransaction()`) in the last `roots` as well as `block.Txs[block.TxsCount-1].StateRootAfter`:

```
// Verify new state root
newStateRoot := types.MimcWithGkr(gkrMimc, roots[0], roots[1])
blockNewStateRoot := block.NewStateRoot
api.AssertIsEqual(newStateRoot, blockNewStateRoot)
```

```
// Verify validium root
blockNewValidiumRoot := roots[1]
api.AssertIsEqual(blockNewValidiumRoot, block.NewValidiumRoot)

// Verify transaction root
blockNewTransactionRoot := roots[2]
api.AssertIsEqual(blockNewTransactionRoot, block.NewTransactionRoot)
```

Recommendations. Remove redundant witness variables whenever possible.

07 - Witness Constraints Are Not Explicitly Codified

● *

Informational

Description. Throughout the zkLighter circuits, witnesses are often introduced or used with implied assumptions that are not explicitly codified. For example, a witness value representing a `uint32` must be range-constrained to $[0, 2^{32} - 1]$ and a witness value representing a `bool` must be range-constrained to be 0 or 1.

During the audit, we found that most private inputs were constrained far away from their definition. For example, most transaction types are range-constrained by being exposed as public data in `circuit/types/pubdata_helper.go` (as explained in **Manually Hardcoded Constants Is Error Prone**).

While we have manually audited all witnessed variables and could not find any constraint issue, nothing prevents a future developer mistake from using a witness value without constraining it, which could be devastating if it happens in practice.

Another example is when a gnark `Variable` is used by a function that implicitly assumes some precondition on that variable. For example, such a function assumes that `b` is a boolean:

```
func SelectAssetDeltas(
    api API,
    b Variable,
    i1, i2 [NbAccountsPerTx][NbAccountAssetsPerAccount]AccountAssetDeltaConstraints,
) (deltasRes [NbAccountsPerTx][NbAccountAssetsPerAccount]AccountAssetDeltaConstraints) {
    // TRUNCATED...
}
```

Recommendations. For booleans, the `AssertIsBoolean` API should be enough to ensure that it has the right type. For other types, one could take the code generation approach outlined in **Manually Hardcoded Constants Is Error Prone** to create wrapper types that would carry information on how to properly constrain a variable and enforce that a witness is properly constrained if used in a circuit (either by forcing the constraint to use the variable, or by checking that all wrapper types have some flag set to true at the end of a circuit).