Making Web3 Space Safer for Everyone

Polybase Labs

Security Assessment

Published on : 23 Feb. 2024
Version v1.0

Security Report Published by KALOS

v1.0 23 Feb. 2024

Auditor: Allen Roh (rkm0959)

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	3	3	-	-
High	3	3	-	-
Medium	1	1	-	-
Low	-	-	-	-
Informational	1	1	-	-

Contents

1 ABOUT US

Pioneering a safer Web3 space since 2018, KALOS proudly won 2nd place in the Paradigm CTF 2023 with external collaborators. As a leader in the global blockchain industry, we unite the finest in Web3 security expertise. Our team consists of top security researchers with expertise in blockchain/smart contracts and experience in bounty hunting. Specializing in the audit of mainnets, DeFi protocols, bridges, and the ZKP circuits, KALOS has successfully safeguarded billions in crypto assets.

Supported by grants from the Ethereum Foundation and the Community Fund, we are dedicated to innovating and enhancing Web3 security, ensuring that our clients' digital assets are securely protected in the highly volatile and ever-evolving Web3 landscape.

Inquiries: `audit@kalos.xyz`

Website: `https://kalos.xyz`

2 Executive Summary

2.1 Purpose of this Report

This report was prepared to audit the overall security of the ZKP circuits and rollup contracts developed by the Polybase team.

In detail, we have focused on the following.

- Completeness of the ZKP Circuits
- Soundness of the ZKP Circuits
- Correctness of the Rollup contract implementation

2.2 Codebase Submitted for the Audit

The commit hash for the audit is fc96e6fdd6580e4ec858cecf4aff4007b41be3d5.

2.2.1 Full audit

Circuits

- pkg/zk-circuits/src/aggregate_agg
- pkg/zk-circuits/src/aggregate_utxo
- pkg/zk-circuits/src/burn
- pkg/zk-circuits/src/insert
- pkg/zk-circuits/src/mint
- pkg/zk-circuits/src/utxo
- pkg/zk-circuits/src/chips/add.rs
- pkg/zk-circuits/src/chips/is_constant.rs
- pkg/zk-circuits/src/chips/is_zero.rs
- pkg/zk-circuits/src/evm_verifier.rs

Contract

- eth/contracts/Rollup.sol
- eth/contracts/MintVerifierV1.sol

- eth/contracts/BurnVerifierV1.sol
- eth/contracts/AggregateVerifierV1.sol

2.2.2 Partial audit (diff from original)

- pkg/zk-circuits/src/chips/aggregation/aggregate.rs
  - original: aggregation - `https://github.com/scroll-tech/snark-verifier/blob/develop/snark-verifier/examples/evm-verifier-with-accumulator.rs`
- pkg/zk-circuits/src/chips/binary_decomposition.rs
  - original: binary_decomposition - `https://github.com/scroll-tech/halo2/blob/develop/halo2_gadgets/src/utilities/decompose_running_sum.rs`
- pkg/zk-circuits/src/chips/merkle_path.rs
  - original: modified to use poseidon from `https://github.com/scroll-tech/halo2/tree/develop/halo2_gadgets/src/sinsemilla`
- pkg/zk-circuits/src/chips/poseidon.rs
  - original: `https://github.com/scroll-tech/halo2/tree/develop/halo2_gadgets/src/poseidon`
- pkg/zk-circuits/src/chips/swap.rs
  - original: `https://github.com/scroll-tech/halo2/blob/develop/halo2_gadgets/src/utilities/cond_swap.rs`

2.3 Audit Timeline

- 2024/01/29 ~ 2024/02/23

This report was initially sent as a markdown, but on March 21st KALOS decided to compile it into a PDF form for the Polybase team. This compilation was done with Spearbit's template with some modifications.

3 Codebase Overview

3.1 ZKP Chips

3.1.1 add

There is a selector `s` and two advice columns `add, culm`. The constraint is `selector * (culm_prev + add - culm) == 0`. Given an array `cells` of `AssignedCell<F. F>`, the `assign` function works as follows to compute the sum of the cells with constraints.

- uses a copy constraint to take the first cell to offset `0` of `culm`
- uses a copy constraint to take the remaining cells to offset `1` to `cells.len() - 1` of `add`. In these rows, `selector` is enabled to add the constraints.
- assigns the `culm` column, and returns value at offset `cells.len() - 1`.

3.1.2 is_zero

This is a classical chip for zero check - using the constraints

- `is_zero_expr = 1 - value * value_inv`
- `selector * value * is_zero_expr = 0`

which forces (when the selector is on) `is_zero_expr = 1` when `value = 0` and `is_zero_expr = 0` when `value != 0`. The `assign` function correctly assigns `value_inv`.

3.1.3 is_constant

There is a selector, and two advice columns `zero_advice` and `output_advice`, and a field element `constant`. A `IsZeroChip` is configured to test whether `zero_advice - constant` is zero. The constraint here is `selector * (output_advice - is_zero_expr) == 0`, which shows that, when the selector is on, we are guaranteed that

- `output_advice = 1` when `zero_advice == constant`
- `output_advice = 0` when `zero_advice != constant`

The `assign` function handles a `AssignedCell<F, F>`, called `comparison`. It

- enables the selector at offset `0`

- copy constrains the `comparison` to `zero_advice` at offset `0`
- correctly assigns the `IsZeroChip` and `output_advice`

It then returns the `AssignedCell<F, F>` corresponding to `output_advice`.

3.1.4 swap

`swap` takes a pair of `AssignedCell<F, F>` and `Value<F>`, and a value `swap` which is a `Value<F>`. It copies the `AssignedCell` and simply assigns it to the `CondSwapChip`.

On the other hand, `swap_assigned` takes the pair and `swap` all as `AssignedCell<F, F>` and uses copy constraints to send them over to the `CondSwapChip`. Both calls `swap_in_region`. The `swap_in_region` function

- enables the selector `q_swap`
- assignes the correct swap result to `a_swapped` and `b_swapped`

The constraint system for `CondSwapChip` is, when the selector `q_swap` is on,

- `a_swapped = b * swap + a * (1 - swap)`
- `b_swapped = a * swap + b * (1 - swap)`
- `swap * (swap - 1) = 0`, i.e. `swap` is boolean

3.1.5 binary decomposition

Modified from halo2_gadget's decompose_running_sum.

Given constant `WINDOW_NUM_BITS`, a selector `q_range_check`, two advice columns `z, b`, the constraint is that, when `q_range_check` is turned on, we have

```
b_cur = z_cur - z_next * 2^{WINDOW_NUM_BITS} \in [0, 2^{WINDOW_NUM_BITS})
```

A `witness_decompose` or `copy_decompose` can be used to decompose a value - the former simply assigns a `Value<F>` while the latter copy constrains a `AssignedCell<F, F>`. A `decompose` function, given an offset as well as `word_num_bits` and `num_windows`, does the following:

- turns on `q_range_check` for offset `[offset, offset + num_windows)`
- assigns `b` values for offset `[offset, offset + num_windows)`
- assigns `z` values for offset `[offset + 1, offset + num_windows]`
- when `strict` is turned on, constrains that the **last** `b` **value is zero**

    – this is incorrect - should be last `z` value is zero
    – see vulnerability #2 for the impact of this issue

We note that even when `strict` is turned on and vulnerability #2 is patched, the `decompose` function doesn't guarantee that the input is within `word_num_bits` bits. Indeed, it only shows that the input is within `num_windows * WINDOW_NUM_BITS` bits. Also, when `strict` is off, there is no guarantee of `bs` being the "intended" limbs from the binary decomposition.

After all the fixes, the `strict` flag is removed and all invocations of `decompose` is regarded as the strict decomposition. The check is fixed to checking that the last `z` value being zero, and the entire `b` values are returned. The `b` values will be the little-endian bit decomposition of the input, as `WINDOW_NUM_BITS = 1`.

3.1.6 is_less_than

This is added as a part of a fix to vulnerability #2. This chip verifies that the two columns in `max` and `alpha` satisfy `alpha < max`, where the bits are in big-endian order. The two advice columns `is_less` and `can_set` are constrained

- `is_less_cur = is_less_prev - max_cur * (1 - alpha_cur) * can_set_prev`
- `can_set_cur = can_set_prev * (max_cur * alpha_cur + (1 - max_cur) * (1 - alpha_cur))`

Given an array of `AssignedCell<F, F>` named `max_bits` and `alpha_bits`, already constrained to be bits and of equal array `N`, the `assign` function does

- assigning `1` as a constant to offset `0` of `is_less` and `can_set`
- enabling the selector at offset `[1, N]`
- copying the advice to `max` and `alpha` on `[1, N]`
- assigning `is_less` and `can_set` on `[1, N]` as the formula
- constraining that the final `is_less` value is equal to zero

Indeed, this set of constraints are sufficient to show that

- `can_set` checks that `max` and `alpha` are equal so far
- `is_less` checks that `alpha >= max` so far

  - indeed, if `is_less_prev = 0` then `alpha < max` already, so `can_set_prev = 0` so `is_less = 0`, which is correct as the bits are in big-endian order
  - if `is_less_prev = 1` then `alpha >= max` - and if `alpha > max` then `can_set_prev = 0` so `is_less_cur = 1` which is correct. if `alpha = max`, then `can_set_prev = 1`, and the only case where `is_less_cur = 0` is the case where `alpha_cur = 0` and `max_cur = 1`, the case where `alpha < max` becomes true

3.1.7 merkle path

`merkle_root_value` and `merkle_root` computes the merkle root.

`merkle_root_value` takes a `AssignedCell<Fr, Fr>` as leaf and a `[Value<Fr>, Value<Fr>]` as siblings - here, the first value represents the actual value of the sibling and the latter represents whether the sibling is right or left. If it's `0`, the sibling is on the right, and if it's `1`, the sibling is on the left. Then, the `poseidon_hash_gadget` is used to hash the pair.

Here, `swap` is used instead of `swap_assigned` - so the siblings are only assigned.

`merkle_root` on the other hand takes all siblings as pairs of `AssignedCell<Fr, Fr>` - the logic is the same, but `swap_assigned` is used, so all cells are appropriately copy constrained. The `enforce_inclusion_constraints` function takes the siblings and the path values simply as `Value<Fr>`, then uses `merkle_root_value` to compute the merkle root, and returns it.

3.1.8 poseidon

This code directly uses Scroll's poseidon circuit.

3.1.9 aggregate

This code is a straightforward application of Scroll's accumulator. It now returns

- all cells for the instances
- all cells for the limbs of the `KzgAccumulator`

3.2 ZKP Circuits

3.2.1 utxo - Note

A `Note` is composed of `address`, `psi`, `value`, `token`, `source`. The corresponding commitment is calculated as `poseidon(value, address, psi, source, 1, 1)`.

The `enforce_constraints` for a `Note` simply

- assigns `zero, value, address, psi, source, version`
  - note that this is assigned, not constrained - for example, `zero` is not actually constrained to be zero within the actual circuit
  - **this was fixed as a patch for vulnerability #5**
- runs `cm = poseidon([value, address, psi, source, version, version])`
- uses a `is_zero_chip` to have `is_value_zero`, checking `value == 0`
- uses `swap_assigned` to swap `cm` and `zero` if `is_value_zero` is true
- returns `NoteConstraintCells`
  - this consists of `value, address, cm, is_value_zero, source`

3.2.2 utxo - InputNote

An `InputNote` consists of a `note`, `secret_key`, and a `merkle_path`.

The nullifier for an input note is

- if the `note.value` is zero, it's simply zero
- otherwise, it's `poseidon(note.commitment, secret_key)`

The new nullifier is `poseidon(commitment, secret key, psi, zero)`.

The `enforce_constraints` for an `InputNote` does

- calls `enforce_constraints` on the `note` to get the `NoteConstraintCells`
- calls `enforce_inclusion_constraints` on the merkle_path with the commitment as leaf, getting merkle root `root` as an assigned cell.
- witnesses `secret_key, padding (0)`
  - after the fix, the padding is assigned as a constant
- computes `verified_address = poseidon(secret_key, padding)` in-circuit

- constrains that `verified_address` is equal to `note.address`
- computes `nullifier` in-circuit
- returns `InputNoteConstraintCells`
  - this consists of `note.commitment, nullifier, root`.
  - after the fixes, the `secret_key` and `padding` are added

3.2.3 utxo - Utxo

The `Utxo` is composed of two `InputNote`s, two `Note`s (which are `outputs`), a root, and a `UtxoKind` (which is either `Null`, `Transfer`, `Mint`, or `Burn`).

The `enforce_constraints` for a `Utxo` is

- witnesses `root` as `unverified_root`, `zero`, `utxo_kind`
  - after the fix, `zero` is assigned as a constant
- assigns `is_mint_chip` and `is_burn_chip`
  - these check if the `utxo_kind` is `Mint` or `Burn`
- for each input note
  - calls `enforce_constraints` to get `cells: InputNoteConstraintCells`
  - sets `root` to either `unverified_root` or `cells.root`
  - sets `nullifier` to either `zero` or `cells.nullifier`
  - this depends on whether the `cells.commitment` is a padding
  - collects the `root, nullifier, cells.commitment.value`
- for each output note
  - calls `enforce_constraints` to get `cells: NoteConstraintCells`
  - collects the `cells.cm, cells.value`
  - after the fix, the `cells.value` is checked to be less than `2^240`
- computes `total_in` as the sum of input values with `add_chip`
- computes `total_out` as the sum of output values with `add_chip`
- sets `mb_hash = is_mint ? output_hashes[0] : zero`

- sets `mb_hash = is_burn ? input_hashes[0] : mb_hash`
- sets `value = is_mint ? total_out : 0`
- sets `value = is_burn ? total_in : value`
- sets `total_out = is_mint ? zero : total_out`
- sets `total_in = is_burn ? zero : total_in`
- asserts that `total_in == total_out`
- asserts that `root, mb_hash, value, hashes` match public input

The public inputs are

- the merkle root
- the input/output hash and the value (for mint/burn)
- input note's nullifier
- output note's commitment

3.2.4 utxo - UTXOProof

The proof consists of

- `recent_root`, the root hash
- `mb_hash` and `mb_value`, the mint/burn hash and value
- `input_leaves`, two input note leaves
- `output_leaves`, two output note leaves
- `proof`, the corresponding proof, a vector of bytes

3.2.5 burn

The burn circuit simply calls `enforce_constraints` for the input notes, and constrains the the note's nullifier, value, source, recent root equals the provided public instance values. To fix vulnerability #8, some logic was added.

It now assigns `to_address` and checks it against the public instance. It loads `secret_key` as a witness, then after calling `enforce_constraints` for the notes (note that this is for note, not input note as before) computes `nullifier = poseidon(commitment, secret key, psi, zero)` and compares it against the public instance. The `value` and `source` are also compared against the public instance. The signature is computed as `poseidon(nullifier, secret key, to address, zero)` and checked against the public instance.

3.2.6 mint

The mint circuit calls `enforce_constraints` for the notes, and constrains that the note's commitment, value, source equals the public instance values.

3.2.7 insert

The circuit aims to add new nodes to the merkle tree, and do it in a batch.

The `Insert` struct consists of a `leaf` element and a `path`.

The `enforce_constraints` for a `Insert` does

- assigns a `new_leaf` as the `leaf` value as a witness
- assigns a `null_leaf` as zero, as a witness
  - after the fixes, this is assigned as a constant
- decomposes `new_leaf` with `copy_decompose` with to `MERKLE_D` bits
  - this was previously done with `strict` turned off
  - after the fixes, the decomposed result is checked to be less than `p`
- assigns all siblings for the merkle path as witnesses
- compute `old_root` and `new_root` based on the leaf and siblings
- compute `is_padding` as `new_leaf == padding note's commitment`
- the public inputs are the leaf, old root, and the new root
- returns `InsertConstraintCells`
  - this consists of `old_root, new_root, new_leaf, is_padding`

The `Batch` struct consists of multiple `Insert`, and handle them at once.

The `enforce_constraints` for a `Batch` does

- witnesses `old_root` and sets `last_new_root` as `old_root`
- for each `Insert`, run `enforce_constraints` to get `InsertConstraintCells`

  - this includes `old_root, new_root, new_leaf, is_padding`
  - runs `old_root = is_padding ? last_new_root : old_root`
  - runs `new_root = is_padding ? old_root : new_root`
  - asserts `last_new_root == old_root`
  - assigns `last_new_root = new_root`
- returns `BatchConstraintCells`,
  - this consists of `old_root, new_root (last_new_root), leafs`

The `enforce_instances` for a `Batch` does

- asserts `old_root, new_root, leafs` match the public inputs
- the public inputs are `old_root, new_root, leafs`

3.2.8 aggregate_uxto

The `AggregateUtxo` struct contains

- `UTXO_N` number of `Snark`, which are individual `UTXO` proofs
- `insert`, which is a `BatchInsert`
- `agg_instances` and `proof`, used for batch SNARK

The `enforce_constraints` for a `AggregateUtxo` does

- uses the `aggregate` function from the `aggregation_chip`
  - this returns `agg_cells` (aggregated result) and `utxo_snarks` (instances)
- the `agg_cells` are compared against public inputs
- the `old_roots` and `new_roots` are compared against public inputs
- compares the `utxo`'s roots, mint/burn hash and value against public inputs
- puts all remaining public inputs (leafs) into `utxo_leafs`
- asserts that these leafs are the ones in `insert` via `constrain_equal`

The public inputs are

- `agg_instances`
- `old_root`, `new_root`

- all UTXO's public inputs (root, hash, value)

3.2.9 aggregate_agg

The circuit aims to aggregate the aggregated SNARKs

The `AggregateAgg` struct contains

- `AGG_N` number of `Snark`, which are individual aggregated proofs
- `agg_instances` and `proof`, used for batch SNARK

The `enforce_constraint` for a `AggregateAgg` does

- uses the `aggregate` function from the `aggregation_chip`
  - this returns `agg_cells` (aggregated result) and `aggregates` (instances)
- the `agg_cells` are compared against public inputs
- constrains that the `old_root` and `new_root` of the aggregates are connected: the previous `new_root` is the current `old_root`.
- constrains that the first `old_root` and final `new_root` match the public inputs and that all UTXO inputs match the public inputs

3.3 Rollup Contract

3.3.1 Rollup.sol

The rollup contract deals with verifying proofs and handling states.

The `mint` function and `mintWithAuthorization` function allows anyone to mint a `commitment` with a given `value` by proving via ZKP that the `commitment` is a valid commitment using the mint circuit and by transferring USDC, either by approving directly or by providing two signatures. The two signatures check

- `from` did sign the commitment, value, source, nonce
- `from` did allow USDC transfers via `receiveWithAuthorization`

After the mint is successful, it gets stored as `mints[commitment] = value`. After the fixes, the mint function reverts if the commitment was already minted.

The `burn` function can be called by anyone - by providing a `root` that belongs in the recent roots alongside `nullifier, value, source`, the function verifies the burn circuit. After the burn is successful, it gets stored as `burns[nullifier] =`

`Burn(to, value)`. After the fixes, the burn also requires the caller to submit a signature as well to prevent frontrunning on the L1.

The contract stores 64 recent root hashes, and they can be used in the ZKP.

There are provers and validators. A prover can add an address as a prover, and a validator can add an address as validator. The number of validators is stored as `validatorsLength`. After the updates and fixes for vulnerability #6, only the owner can add a prover and can set validators.

The `verifyBlock` function, which only the approved prover can call, takes

- the final ZKP proof
- the aggregated instances
- old root and the new root
- the utxo hashes (6 utxo, 3 hashes per utxo)
- the other hash from block hash
  - used to compute the block hash alongside with the root
- (after the update) the height and skips
- and a list of signatures

The function

- checks that the `old root` is the current root hash stored in the L1
- for each utxo instance
  - check that said instance's root is within recent roots
  - if it's a mint, check value is correct, and set `mint[commitment]` to zero
  - if it's a burn, check value is correct, and set `mint[commitment]` to zero
    - also, transfers the USDC to the desired recipient
- checks that over 2/3 of the validators have signed the new root, height, otherHashFromBlockHash, skips, and network
- aggregate verifier verifies the ZKP proof
- set new root to the list of recent roots, updates block hash and block height

There is a `setRoot` function, which allows the owner to add arbitrary merkle root. As the owner has this capability, as well as the power to upgrade

contracts and set validators, there is a built-in centralization issue. In this audit report, we assume that the owner is not malicious and does not make mistakes in contract calls. The implementation of the actual owner, which was communicated to the auditor to be a Gnosis Safe, is not audited by KALOS. All issues that arise from incorrect management of the owner role and relevant private keys are out of scope. For example, we note that in the case where owner becomes malicious via private key leak, the malicious owner can drain the bridge.

We also note that the prover is added by the owner, so we can assume that the prover is not malicious. A malicious prover could act such in many ways, such as ignoring a user's transfer request or handling 64 proofs before dealing with the user's UTXO proof, to make the "recent root" of the UTXO proof not sufficiently recent.

3.3.2 Verifiers

These contracts have the `verify` functions, which verifies the zkSNARK. The key part here is the correct format of sending over the public inputs and the proof to the actual SNARK verifier contract. After the updates, these verifiers check that all public inputs are less than `p`.

The node operation and the client side that deals with proof generation is out of scope. Also, note that the SNARK verifier itself is out of scope.

4 Findings

4.1 [Medium] Fixed `rng` in `gen_proof` leads to predictable blinding factors, losing theoretical zero-knowledgeness

```
    // src/evm_verifier.rs
    #[allow(dead_code)]
    pub fn gen_proof<C: Circuit<bn256::Fr>>(
        params: &ParamsKZG<Bn256>,
        pk: &ProvingKey<bn256::G1Affine>,
        circuit: C,
        instances: &[&[bn256::Fr]],
    ) -> Result<Vec<u8>, halo2_base::halo2_proofs::plonk::Error> {
        let mut transcript: EvmTranscript<_, _, _, _> =
            halo2_base::halo2_proofs::transcript::TranscriptWriterBuffer::<_,
↪   G1Affine, _>::init(
                Vec::new(),
            );
        create_proof::<KZGCommitmentScheme<Bn256>, ProverSHPLONK<Bn256>, _, _,
↪   _, _>(
            params,
            pk,
            &[circuit],
            &[instances],
            StdRng::seed_from_u64(0),
            &mut transcript,
        )?;

        Ok(transcript.finalize())
    }
```

The code uses a fixed seed `0` as the `rng` in `create_proof`, so it is predictable. The `rng` is used to create blinding values. For example, see the following code.

```
    // halo2_proofs/src/plonk/prover.rs
    let blinds: Vec<_> = advice_values
                        .iter()
                        .map(|_| Blind(Scheme::Scalar::random(&mut rng)))
                        .collect();
```

Since the blinding values are fixed, the zero-knowledge property of the zk-SNARK would be lost. This doesn't necessarily mean that all the witness values can be extracted, but in a theoretical sense, the current proof system does not satisfy the zero knowledge property.

We compare this to the `evm_verifier` of Scroll, which uses `OsRng` instead.

```
    fn gen_proof<C: Circuit<Fr>>(
        params: &ParamsKZG<Bn256>,
        pk: &ProvingKey<G1Affine>,
        circuit: C,
        instances: Vec<Vec<Fr>>,
    ) -> Vec<u8> {
        MockProver::run(params.k(), &circuit,
↪   instances.clone()).unwrap().assert_satisfied_par();

        let instances = instances.iter().map(|instances|
↪   instances.as_slice()).collect_vec();
        let proof = {
            let mut transcript = TranscriptWriterBuffer::<_, G1Affine,
↪   _>::init(Vec::new());
            create_proof::<
                KZGCommitmentScheme<Bn256>,
                ProverSHPLONK<_>,
                _,
                _,
                EvmTranscript<_, _, _, _>,
                _,
            >(params, pk, &[circuit], &[instances.as_slice()], OsRng, &mut
↪   transcript)
            .unwrap();
            transcript.finalize()
        };
        // ....
        proof
    }
```

We recommend changing the `rng` to a unpredictable randomness source.

4.1.1 Fix Notes

The `rng` is changed to use `OsRng` in this pull request.

4.2 [Critical] Incorrectness and misuse of `binary_decomposition` leads to unlimited double spending

We first make some observations about the `binary_decomposition` function.

First, the `zs` is a vector of `b_assigned`, which is the assigned limbs of `WINDOW_NUM_BITS` bits. This is different from the original code in `halo2_gadgets` code, where `zs` is a vector of `z`, which is the running sum of the remaining limbs. When `strict` is turned on, the last element of `zs` is constrained to be the constant zero.

```
    // binary_decomposition in Polybase
    {
        // ....
        let b_assigned =
            region.assign_advice(|| format!("b_{i:?}"), self.b, offset + i, ||
↪   word)?;
        // Update `z`.
        z = z_next;
        zs.push(b_assigned);
    }
    assert_eq!(zs.len(), num_windows);

    if strict {
        // Constrain the final running sum output to be zero.
        region.constrain_constant(zs.last().unwrap().cell(), F::zero())?;
    }
```

```
    // decompose running sum in halo2-gadgets
    {
        // ......
        // Update `z`.
        z = z_next;
        zs.push(z.clone());
    }
    assert_eq!(zs.len(), num_windows + 1);

    if strict {
        // Constrain the final running sum output to be zero.
        region.constrain_constant(zs.last().unwrap().cell(), F::ZERO)?;
    }
```

When `strict` is turned on, the intention is that the code securely constrains that `z` can be decomposed into `num_windows` limbs of `WINDOW_NUM_BITS` bits each. However, due to the differences above, the code now simply constrains that the final limb is zero. In particular, now there is no check that the limbs are a correct decomposition of `alpha` at all.

It also leads to a completeness issue, since not all correct decompositions have zero as its final limb. We demonstrate this with a test case below.

```
    // this test fails
    #[test]
    fn test_binary_decomp() {
        let k = 14;

        let circuit = BinaryDecompCircuit::<Fr, 1> {
            alpha: Value::known(Fr::from(7)),
            strict: true,
            num_windows: 3,
            word_num_bits: 3,
        };

        // 111 -> 1,1,1
        let bits = vec![Fr::one(), Fr::one(), Fr::one()];

        let prover = MockProver::<Fr>::run(k, &circuit, vec![bits]).unwrap();
        prover.assert_satisfied();
    }
```

We add that with `strict = false`, there is practically no check on the limbs other than the fact that the limbs are within `[0, 2^WINDOW_NUM_BITS)`, as their last `z` value can be anything. Overall, this chip is underconstrained.

The underconstrains in binary decomposition leads to a double spend. Indeed, the way the circuits prevent double spend is that they check that the leaf corresponding to the `nullifier` of the input note is filled with zero, and then they add the `nullifier` value at that leaf. If one wants to spend the input note commitment again, they would either have to

- find a different nullifier that corresponds to the same commitment, or
- provide a merkle proof that the leaf at nullifier has value zero, when it has nullifier value: which can be done by either forging a merkle proof or finding an instance with `nullifier = 0`

which is not computationally feasible. It is very important that `nullifier` is added to the leaf that corresponds to the index `nullifier`. To do so, `copy_decompose` is used, which is, as we mentioned, underconstrained.

```
    // insert.rs
    let decomposed_bits = layouter.assign_region(
        || "decompose",
        |mut region| {
            // We use non-struct because the merkle tree is not as big as the
↪   hash (i.e. we're only
            // interested in the last n bits)
            decompose.copy_decompose(
                &mut region,
                0,
                new_leaf.clone(),
                false,
                MERKLE_D,
                MERKLE_D,
            )
        },
    )?;
```

Therefore, we can perform a double spend by

- use a input note commitment
- insert the nullifier at the correct nullifier location
- use the input note commmitment
- insert the nullifier at an incorrect nullifier location which has zero filled in

We recommend doing the following using a bitwise decomposition method that has soundness and completeness. One should also consider that the bitwise decomposition of `x + p` cannot be used as a bitwise decomposition of `x` - which makes the bitwise decomposition nontrivial.

4.2.1 Fix Notes

The fix is done in this pull request. We explain it below.

The `strict` variable is removed, as all usage of the chip is now strict. The last `z` value is constrained to be zero. The value is decomposed into 256 bits, and the first `MERKLE_D - 1` bits are taken as the merkle path for the insertion. The check that path bits are less than p is implemented in this pull request.

4.3 [Critical] Overflow in addition chip leads to asset drain

In the usual transfer conditions, one of the main checks is that the sum of the input values and sum of the output values is equal. However, there are no check for overflows in $\mathbb{F}_p$, which leads to an attacker possibly draining the entire L1. For example, consider an attacker has a valid input commitment of value 10. The attacker can remove a value of 50 by

- split the value 10 commitment into two outputs of value $p - 40$ and 50
- calling burn on the new commitment of value 50.

We recommend to add an overflow check for the `add_chip`. One can also simply range check the `value`s into a reasonable size to prevent such overflows as well.

4.3.1 Fix Notes

The fix is done by range checking that all the output values are within 240 bits. This enforces that the sum of output values does not overflow, as desired. Note that sum of input values overflowing only hurts the user using the commitment, and that the sum of input values actually cannot overflow as one needs to either mint a new commitment by transferring actual USDC (which is very difficult to be over 240 bits) or get a new commitment as a output commitment of a transfer (which is already constrained to be within 240 bits).

This range check is implemented in this pull request.

4.4 [High] Low merkle tree depth leads to low security

As the merkle tree depth is only `33`, there is a non-negligible probability of commitments and nullifiers colliding in their merkle path. For example, there could be an two commitments A, B such that A's nullifier is equal to B's commitment. In that case, if B is added to the merkle tree, A will never be successfully used due to the nullifier leaf being already filled.

This vulnerability was found independently by both Polybase and the auditor.

We recommend to increase the merkle tree depth to avoid such collisions.

4.4.1 Fix Notes

The fix is added in this pull request, increasing the depth to `161`.

4.5 [Critical] Attacker can mint new commitments when `is_burn` is true using constants that are only witnessed

A common pattern in the codebase is assigning constants as witnesses.

```
    // utxo/note.rs/enforce_constraints()
    let zero = assign_private_input(
                || "zero witness",
                layouter.namespace(|| "zero witness"),
                advice,
                Value::known(Fr::zero()),
    )?;
```

This assigns zero to the AssignedCell `zero`, but it doesn't constrain it to be zero. We highly recommend to fix all such patterns with additional constraints that the cells are indeed equal to the constants. Here, we showcase an example of a critical severity attack, where attacker mints arbitrary output commitments when `is_burn` is true in the Utxo proof.

Consider the usual Utxo proof, when `is_burn` is true (`kind = 2`),

- `mb_hash = input_hashes[0]` is verified against the public input
- `value = total_in` is verified against the public input
- `total_out` remains as the sum of values of the output commitments
- `total_in` is swapped out to be `zero`, which is assigned but not constrained

The intention here is that `total_in = 0` at the end, and as there is a constraint that `total_in = total_out`, the output commitments are all forced to be of value zero, as we expect for burning. However, as `zero` is not constrained to be zero, we can set `zero` as any value then make output commitments have a nonzero value. This technique will lead to a full drain of the L1 contract.

We highly recommend changing all assignments of fixed constants to be done with additional constraints. These do not add significant cost in proving.

4.5.1 Fix Notes

Fixed in this pull request. The following are changed to constants

- `null_leaf` in `insert.rs`, `padding` in `input_note.rs`
- `zero` in `note.rs` and `utxo.rs`

4.6 [High] Any validator can add a new validator, leading to a single malicious validator sufficient for attack

```
    function addValidator(address validator) public onlyValidator {
        require(validators[validator] == 0, "Validator already exists");

        validators[validator] = 1;
        validatorsLength += 1;
    }
```

The above code shows that any validator can add a new validator. Therefore, a single malicious validator can add many new addresses as a validator, making more than 2/3 of the validator set malicious. This allows the malicious validator to successfully submit proofs that exploit the vulnerabilities in the ZKP circuits that are described above, making the 2/3 rule practically useless.

We recommend to add a stronger restriction on the addition of validators.

4.6.1 Fix Notes

The fix is in this pull request. The addition of validators will be an `onlyOwner` functionality, which is controlled by a Gnosis safe with a multi-party lock.

We note that the audit process did not go through the relevant Gnosis safe code. We assume that for any owner-approved validator set, at most 1/3 of them are malicious. We also note that the added code has a potential vector for gas running out via iterating dynamic arrays (such as functions like `updateValidatorSetIndex`) but as the addition of validator set is done by the owner, we assume that the owner takes sufficient care into such possible vectors by themselves.

4.7 [Informational] Public inputs in solidity's verifier should be checked to be less than p for additional safety

The public inputs, especially `nullifier`, `commitment`, `value`, `source`, `root` should be checked to be less than p, to avoid different uint256 values corresponding to the same element in $\mathbb{F}_p$. Indeed, in the current implementation, a transfer of value `1` and transfer of value `p + 1` is the same. Of course, such is not possible as a transfer of value `p + 1` requires unreasonably large amount of USDC in the L1 network. However, for additional safety we recommend adding this check.

4.7.1 Fix Notes

Fixed in this pull request as recommended.

4.8 [High] `burn()` is front-runnable, leading to asset theft

```
    // Anyone can call burn, although this is likely to be performed on behalf
↪   of the user
    // as they may not have gas to pay for the txn
    function burn(
        // to address is not verified, we don't care who they send it to
        address to,
        bytes calldata proof,
        bytes32 nullifer,
        bytes32 value,
        bytes32 source,
        bytes32 root
    ) public {
        // Check recent roots
        bool recent_root_check = false;
        for (uint i = 0; i < rootHashes.length; i++) {
            if (root == rootHashes[i]) {
                recent_root_check = true;
                break;
            }
        }
        require(recent_root_check, "Root used in proof is invalid");

        burnVerifier.verify(proof, [nullifer, value, source, root]);

        // Add burn to pending burns, this still needs to be verifier with the
↪   verifyBlock,
        // but Solid validators will check that this commitment exists in the
↪   burn map before
        // accepting the burn txn into a block
        burns[nullifer] = Burn(to, uint256(value));
    }
```

The burn function allows one to set the `to` address, the recipient of the burned value in the L1 blockchain. While this function correctly checks that the caller knows the corresponding secret key for the commitment, it doesn't consider possible front-running attacks.

For example, assume that a user `user1` wants to burn their commitment `C`, sending the value to `address1`. To do so, `user1` would call `burn` with the appropriate `proof, nullifier, value, source, root`, and with `to = address1`. Looking at these inputs from the mempool, an attacker `user2` can call `burn` with the same `proof, nullifier, value, source, root`, but with `to = address2`. This will lead to the burned commitment's value being sent to `address2` instead of the intended `address1`, after `verifyBlock` is called.

A similar attack can be designed for products like Tornado Cash, but they prevent it by incorporating the recipient address inside the ZKP circuits. We also recommend such methods - for example, requiring the prover to submit `poseidon(secret key, commitment, to_address)` could be a possible patch, as one must know the secret key to submit it.

4.8.1 Fix Notes

Fixed in this pull request, by enforcing the caller to submit `poseidon(nullifier, secret key, to address, zero)` as a public input `signature`. This forces the user deciding `to` must know the `secret key` corresponding to the commitment.

5 Further Discussion

We explain why we believe the current updated version of the circuits and contracts are safe. We first explain the circuit's main ideas, then discuss possible attack ideas and why they fail. The proofs in this section are informal.

We occasionally use "hard" as a short term for computational infeasibility.

5.1 Facts on Commitment and Nullifier

In the final version, the commitment and nullifier are computed as

- `poseidon(value, address, psi, source, version, version)`
- `poseidon(commitment, secret key, psi, padding)`
  - where `poseidon(secret key, padding) = commitment's address`

Therefore, due to the Poseidon hash's properties, we obtain the following facts.

- **Fact 1: It is hard to find a value** `x` **that is both a nullifier of a known commitment and a commitment of a known parameter set.**
- **Fact 2: One must know the** `secret key` **to compute the valid nullifier or signature, but having many** `(commitment, nullifier, signature)` **does not allow an attacker to recover** `secret key`**.**
- **Fact 3: It is hard to find multiple** `nullifier`**s for a single** `commitment`**.**
- **Fact 4: It is hard to compute** `commitment` **from the** `nullifier`**.**
- **Fact 5: It is hard to find a non-padding zero commitment or nullifier.**

5.2 Facts on the Merkle Tree and the UTXO proofs

We prove some various facts on the merkle tree and the UTXO proofs.

Fact 6: A non-padding commitment cannot be used twice

Proof: Once a commitment is used, the UTXO proof forces that the relevant nullifier is inserted to the merkle tree - and not only that, but the nullifier must be inserted to the leaf with the index `nullifier`. The check in the insert circuit forces that the previous value at the leaf is zero, and the new value at the leaf is `nullifier`. Not only that, the circuit also checks that the `nullifier`'s bitwise decomposition for the merkle path is unique by checking that the decomposition is for an integer within `[0, p)`. To double spend a commitment, one has to either find multiple nullifiers for the same commitment or find a nullifier that is equal to zero. By **Fact 3** and **Fact 5**, both of these ideas are computationally infeasible, finishing the proof of **Fact 6**. This proves that double spending is impossible.

Fact 7: All notes on the merkle tree have a value of at most `2^240`

Proof: All notes are output commitments of a UTXO proof which check this. This also proves that no overflow attacks discussed in vulnerability #3 are possible.

Fact 8: The cases in `verifyBlock` correspond to `UTXOKind` in the ZKP.

To explain the meaning of this, basically

```
    // Check mints/burns
    for (uint i = 0; i < 18; i += 3) {
        bytes32 mb = utxoHashes[i + 1];
        bytes32 value = utxoHashes[i + 2];

        if (value == 0) {
            continue; // ### CASE 1
        }

        if (mints[mb] != 0) {
            // ### CASE 2
            // ....
        }

        if (burns[mb].amount != 0) {
            // ### CASE 3
            // ....
        }

        revert("Invalid mint/burn");
    }
```

- In Case 1, the UTXOKind must be transfer
- In Case 2, the UTXOKind must be mint
- In Case 3, the UTXOKind must be burn

Here, we disregard the padding commitments and nullifiers for convenience.

Proof: The UTXO proof forces that

- `mb_hash = 0` if UTXOKind is transfer
- `mb_hash = output_hashes[0]` (a commitment) if UTXOKind is a mint
- `mb_hash = input_hashes[0]` (a nullifier) if UTXOKind is a burn

Therefore, if we are in Case 2, as `mb` is a valid commitment (as proved by the mint circuit) by **Fact 1** and **Fact 5** it's true that the UTXOKind must be a mint. Similarly, if we are in Case 3, as `mb` is a nullifier (as proved by the burn circuit) it's true that UTXOKind must be a burn.

Note that in the burn circuit itself, there is no check that the `secret key` actually corresponds to the `commitment`'s address. However, it's still true that the `nullifier` is a hash of four inputs. If this `nullifier` was equal to a `commitment`, then a hash collision would still occur for Poseidon.

We now prove that in Case 1, the UTXOKind must be a transfer. As the `value` is zero, if UTXOKind was not a transfer, then `total_in` and `total_out` must be both zero. As all values within commitments are less than `2^240` by **Fact 7**, this implies that all input/output commitments have value zero, i.e. every input/output is a padding. We end with an overview of transfer/mint/burn.

transfer

- `mb_hash = 0`, `value = 0` are public inputs
- `total_in == total_out` is checked

mint

- `mb_hash = output_hashes[0]`, `value = total_out` are public inputs
- `total_in == 0` is checked

burn

- `mb_hash = input_hashes[0]`, `value = total_in` are public inputs
- `total_out == 0` is checked

Fact 9: The values in the merkle tree match the transferred USDC values.

To mint a new commitment, the UTXOKind must be mint - so we have to be in Case 2 by **Fact 8**. The check inside the ZKP of `verifyBlock` is that the `value` is equal to the `total_out`, and the check from the ZKP in mint function guarantees that the `mb_hash` is a commitment with the value `value`. Since the mint function in the contract takes `value` USDC from the minter, **Fact 9** holds.

Similarly, for a burn, the following values are all guaranteed to equal to `value`

- the transferred USDC amount
- the `burns[mb].amount`, the value of the commitment inside the nullifier
  - this is guaranteed by the burn circuit when a user calls `burn()`
- `total_in` in the UTXO proof, checked by the UTXO circuit

which also shows that the **Fact 9** is true. We also note that all public inputs are range checked to be less than `p`, so equality over the finite field is sufficient.

Fact 10: To use a commitment, it must be inside the merkle tree as a commitment, and one must know the secret key corresponding to the commitment's address to do so. No attacker can recover the secret key.

The UTXO proof checks that the commitment is inside a `root`, and the smart contract checks that the `root` is indeed one of the recent roots. Therefore, the commitment is indeed in the merkle tree. The input note constraints also checks the knowledge of `secret_key`. Note that the `secret_key` cannot be recovered from previous sets of nullifiers and signatures by **Fact 2.**

We note that the burn circuit itself doesn't check the knowledge of the correct `secret_key`, but to actually use the commitment in a `verifyBlock` proof one needs to know `secret_key` for to compute the valid nullifier.

We also note that merkle inclusion proof doesn't check that the `commitment` is at the leaf with the index `commitment` - but this is okay, as checking inclusion is sufficient. Also, there is the case where the recent root used for the merkle proof is just `0` (when there are less than 64 recent roots) but utilizing this requires a preimage of `0` for the poseidon hash, which is clearly hard to find.

Also, we note that a nullifier cannot be used as a commitment, by **Fact 1.**

5.3 Review on Replay, Malleability, MEV Attacks

We first review signature related attacks. Each validator signs the set of `newRoot`, `height`, `otherHashFromBlockHash`, `skips`, `NETWORK_LEN`, `NETWORK`. Since all signatures' signers are checked to be different in the smart contract, we can simply consider the case where the same set is used twice.

If a single `newRoot` value is used twice, then that immediately implies that the final merkle root itself is the same between the two instances. Therefore, no additions (both commitment and nullifier) on the merkle tree was done. Therefore, an attack scenario described below is impossible.

- a set of validators signs the `newRoot` value `A`
- then, another `verifyBlock` is called, updating the `newRoot` value to `B`
- an attacker reuses the signatures for `newRoot` value `A` to make the root `A`

Of course, this is impossible as the ZKP must show that one can add new commitments and nullifiers to make the merkle tree with root `B` to a merkle tree with root `A`, which is computationally infeasible.

Therefore, a replay attack allows the attacker to only do

- a set of validator signs the required dataset
- a replay attacker submits the signature for the exact same set

However, in this case the attacker's work is meaningless, as the L1 state stay the same. Of course, we note that the prover is added by the owner, so we can assume it's not malicious anyway.

There's also the `signer = address(0)` type issues - but this is naturally handled in `verifyBlock`. In `mintWithAuthorization`, the `usdc.receiveWithAuthorization()` call checks that `from != address(0)`, showing that `signer != address(0)`.

In `mintWithAuthorization`, as both structs that are signed includes `from` and `nonce`, the USDC contract enforces that signature replay is impossible. This is as USDC checks that the same nonce isn't used twice for a single `from`.

We now review possible front-running attack vectors. For `mint()`, if an honest user submits a `mint()` call, a frontrunner can indeed mint the commitment beforehand by submitting the same ZKP proof. This *would* make the honest user's call revert. However, only the honest user can actually use the commitment as they are the only one who know the relevant secret key. Also, as the USDC is transferred by the frontrunner, this attack fails to even hurt the user.

If a frontrunner *could* use the commitment, an idea would be

- frontrunner submits the mint proof beforehand, minting the commitment
- the minted commitment is used by the frontrunner
  - so far, it's not a problem as the frontrunner paid for the commitment
- the honest user's `mint()` call goes through - as `mint[commitment] = 0` now, the call doesn't revert - however, as the commitment is already used, it cannot be used again by the user, leading to loss of funds.

This would lead to a loss for the honest user. However, as mentioned in **Fact 10**, one needs to know the secret key to actually use the commitment. If the honest user also submits the UTXO proof, the frontrunner can submit that as well - but all it does is that the honest user gets what they wanted in the first place anyway, while the frontrunner pays the USDC for nothing.

For the `mintWithAuthorization` call, a similar frontrunning attack (such as minting the same commitment via `mint()` or `mintWithAuthorization()`) doesn't hurt the honest user. An alternative, interesting idea is shown by Trust Security, but this still fails in the context of this codebase. The idea is to submit the signature for USDC authorization directly to USDC, so that the signature and nonce is already used by the USDC contract. This would make the honest user's call revert, so while the honest user doesn't lose USDC, they would lose access to the `mintWithAuthorization()` functionality. However, this is not possible for EIP3009. The `receiveWithAuthorization` function checks that the `to` address the address that called the function in the first place, and the signature includes `from, to, value, nonce` among other things. Therefore, to front-run the `receiveWithAuthorization` function, one needs to make the `RollupV1.sol` contract call the `receiveWithAuthorization` function. To front-run the honest user's signature, one needs to use the same `from, to, value, validAfter, validBefore, nonce`. However, under usual circumstances, the signature for `structHash` that the frontrunner can get and contains these values at the same time is the signature for the intended `commitment` itself. Therefore, in this case also, the frontrunner's attempt does not hurt the honest user, as we desired.

For the `burn` call, as the `nullifier` must be an actually correct one (with the correct secret key) to be used in a UTXO proof and the contract and the circuits require a `sig = poseidon(nullifier, secret key, to address, 0)` (where the `secret key` is the same one used to compute the `nullifier`), it's clear that an attacker cannot change the `to` address. Indeed, to give a signature the attacker must know the valid `secret key`, which is infeasible as we discussed before.

End of Document