Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Rate Limiting Nullifier

Veridise Inc.

September 1, 2023

► **Prepared For:**

Privacy and Scaling Exploration
https://appliedzkp.org

► **Prepared By:**

Benjamin Sepanski
Kostas Ferles
Hanzhi Liu
Jacob Van Geffen

► **Contact Us:** contact@veridise.com

► **Version History:**

May 29, 2023	V1
May 17, 2023	Initial Draft

Contents

1 Executive Summary

From May 1, 2023 to May 12, 2023, Privacy and Scaling Exploration engaged Veridise to review the security of the circuits for Rate Limiting Nullifier. The review covered their Zero Knowledge Circuit (ZK-circuits), written in Circom. Veridise conducted the assessment over 8 person-weeks, with 4 engineers reviewing code over 2 weeks on commits `0x022b690b-0xb40dfa63`. In response to issues raised by Veridise auditors (RLN-003), the circuits were modified before formal verification. As a result, formal verification was performed on commit `0x10437bc2`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing. In parallel, the auditors wrote specifications of several key correctness properties. Following the manual audit, the Veridise engineers formally verified these properties.

Code assessment. The Privacy and Scaling Exploration developers provided the source code of the Rate Limiting Nullifier circuits for review. Rate Limiting Nullifier is a set of ZK-circuits which can be used by anonymous systems which need to prevent spam, perform votes, or otherwise limit the number of actions performed by anonymous participants. This is implemented using Shamir's Secret Sharing scheme, and verified using zero knowledge proofs.

To facilitate the Veridise auditors' understanding of the code, the Privacy and Scaling Exploration developers shared two RFCs*† detailing the protocol, extensive documentation‡, along with the circuits themselves§. The source code also contained some documentation in the form of READMEs.

The source code contained a test suite, which the Veridise auditors noted covered most of the basic correctness criteria. Their tests verify correct output and verification success on random inputs, as well as verification failure for incorrect input-output pairs.

Veridise auditors feel that the circuits are well thought-out and carefully designed. Their design enables many checks to be performed outside of the ZK-circuits, minimizing the chance of issues such as under-constrained bugs. The code is written in an accessible fashion, and its intent and implementation are clear.

RFC V1 notes that there are some algebraic attacks which use polynomial GCDs to lower the security from 256 bits to 160 bits. We investigated these attacks, and did not find any improvements beyond the 160 bit margin. While this is still a sufficient margin of security for most applications, users of RLN should take note of this fact when performing their own security assessments. We have also recommended a line of inquiry which may address this issue.

* RFC-V1: `https://rfc.vac.dev/spec/32/`
† RFC-V2: `https://rfc.vac.dev/spec/58/`
‡ `https://rate-limiting-nullifier.github.io/rln-docs/`
§ `https://github.com/Rate-Limiting-Nullifier/circom-rln/`

Summary of issues detected. The audit uncovered 4 issues, 0 of which are assessed to be of high or critical severity by the Veridise auditors. 1 issue was determined to be of medium severity (RLN-001), which describes the possible consequences of self-slashing. The Veridise auditors also identified additional minor issues. The Privacy and Scaling Exploration developers identified two of these issues as intended, both due to an upcoming change in documentation (RLN-002) and (RLN-004).

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve Rate Limiting Nullifier. Primarily, this involves adding additional documentation (RLN-003) and supplying suggestions on how to handle slasher fees (RLN-001).

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Rate Limiting Nullifier	`0x022b690b-0xb40dfa63`	Circom	N/A

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 1 - May 12, 2023	Manual Auditing, Automated Tools, & Formal Verification	4	8 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	1	1
Low-Severity Issues	0	0
Warning-Severity Issues	2	2
Informational-Severity Issues	1	1
TOTAL	4	4

Table 2.4: Verification Summary.

Type	Number
Functional Correctness	6

Table 2.5: Category Breakdown.

Name	Number
Maintainability	2
Frontrunning	1
Data Validation	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Privacy and Scaling Exploration's Rate Limiting Nullifier smart contracts. In our audit, we sought to answer the following questions:

- Can registered users submit a message ID outside of the allowed range?
- Can malicious users abuse the ability to slash themselves?
- Are any of the circuits under-constrained?
- How do algebraic attacks against Poseidon modify the security of the Shamir secret sharing scheme?
- Can any user's identity be revealed when following the protocol correctly?
- Do any of the slashing actions provide additional spamming opportunities?
- Are all necessary checks on proof input/output properly documented?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts, automated program analysis, and formal verification. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom circuit-analysis tool ZK-Solid. This tool is designed to find instances of common circuit vulnerabilities, such as under-constrained bugs or missing range checks. Rate Limiting Nullifier had no vulnerabilities identified by ZK-Solid.
- *Fuzzing/Property-based Testing.* We also leveraged fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we used our fuzzing framework zkOrCa to identify under-constrained bugs, in which a false witness is able to pass the constraints. We fuzzed the protocol for 24 hours, and found no under-constrained circuits.
- *Formal Verification.* To prove the correctness of the ZK circuits we used Coda, our formal verification project based on the Coq interactive theorem prover. To do this, we formalized the intended behavior of the Circom templates and then proved the correctness of the implementation with respect to the formalized specifications.

Scope. The scope of this audit is limited to the `circuits/` folder of the source code provided by the Privacy and Scaling Exploration developer. In particular, the following files were audited:

- `rln-diff.circom`
- `rln-same.circom`
- `utils.circom`
- `withdraw.circom`

Methodology. Veridise auditors inspected the provided tests and read the Rate Limiting Nullifier documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Privacy and Scaling Exploration developers to ask questions about the code.

During the manual audit, Veridise auditors prepared specifications for formal verification and discussed them with the Privacy and Scaling Exploration team to ensure they were happy with the specifications. Following the manual audit, Veridise auditors formally verified these specifications.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
RLN-001	Spammers may slash themselves	Medium	Acknowledged
RLN-002	Unregistered users can be slashed	Warning	Intended Behavior
RLN-003	Undocumented lack of range checks in IsInInterval	Warning	Fixed
RLN-004	identitySecret Naming Inconsistency	Info	Intended Behavior

4.1 Detailed Description of Issues

4.1.1 RLN-001: Spammers may slash themselves

Severity	Medium	**Commit**	022b690
Type	Frontrunning	**Status**	Acknowledged
File(s)	`withdraw.circom`		
Location(s)	Withdraw		

The RLN documentation describes the purpose of `withdraw.circom`

> withdraw.circom is a template that's used for the withdrawal/slashing and is needed to prevent front run while withdrawing the stake from the smart-contract/registry.

The `Withdraw` template consists of a proof of knowledge of an `identityCommitment`'s pre-image, i.e. the `identity_secret_hash` (referred to as `identitySecret` in the implementation).

```
template Withdraw() {
    signal input identitySecret;
    signal input addressHash;

    signal output identityCommitment <== Poseidon(1)([identitySecret]);
}
```

Snippet 4.1: The `Withdraw` template.

While this does prevent frontrunners from simply replaying a transaction with an address they own as the beneficiary, it does not prevent front-running from the user targeted by the slashing.

For instance, suppose Alice is intending to spam a permissionless chat application which is using RLN for spam filtering via an economic stake.

1. Alice deposits 1 coin to register.
2. Alice sends as many messages as possible until she sees someone submit a slash request on her `identityCommitment`.
3. Alice front-runs the request, slashing herself (since she knows her own `identitySecret`) and recovering her 1 coin.

Note that Alice is not required to front-run. She could instead preemptively slash herself after sending some number of pre-determined messages.

Impact Applications using RLN and economic stake to implement spam resistance may suffer from adversaries with large collateral.

Spammers who are able to slash themselves before others (or successfully front-run others) will be able to recover their economic stake.

Recommendation One simple fix is to add documentation describing the potential issue, and recommend that applications only give a portion of the staked amount to the slasher. For instance, half of the stake could be given to the slasher, and the remaining half split amongst all non-malicious protocol participants.

A second approach is to disallow self-slashing by requiring `Withdraws` to provide evidence of two `identity_secret_hashes`, one identity which is the slasher (along with a proof of tree membership) and one identity to be slashed. This doubles the stake required to self-slash.

Developer Response We plan to take a fee as part of money laundering prevention. The gas fees are also part of the cost, which are not always cheap due to snark verification.

Since some applications will be a relayer, gas costs may not always prevent this attack.

Updated Recommendation The developers asked us about how to allocate the portion of a slashing reward not reserved for the slasher. We considered three main approaches: burning the reward, providing the reward to a random slasher, or providing the reward to a random registrant.

Burning the reward provides the surest guarantee, and from a security perspective is most likely to not be abused. However, if this is not feasible, we recommend providing the reward to a random registrant—weighted by stake. By making the expected value proportional to stake, a malicious user must be willing to stake a large amount into the protocol in order to not lose money at an exponential rate when self-slashing. Note, however, that this solution is still context-dependent and should be heavily analyzed on a case-by-case basis to determine appropriate pricing.

We do not recommend providing the reward to a random slasher. For instance, providing the fee to a random user chosen from the most recent N slashers has an expected cost of 0 for self-slashers.

Updated Developer Response We will add documentation in an upcoming RFC to recommend a reward system which takes a fee from the slashing reward.

For slashing, transfer happens immediately and the fee is taken. Since the fee is taken, it is not economically efficient to slash yourself. For withdrawing, fees are not taken, but we check if this user can be slashed too. We first freeze the stake and wait for possible slashing. If there's no slasher after n blocks, then the user can take the frozen money.

We have decided to burn fees, but not directly. We intend to make Fee-Receiver a contract with a function to swap an ERC20 to ETH, and then burn this ETH (gas expenses for this function call should be compensated).

4.1.2 RLN-002: Unregistered users can be slashed

Severity	Warning	**Commit**	022b690
Type	Data Validation	**Status**	Intended Behavior
File(s)	`withdraw.circom`		
Location(s)	Withdraw		

The `Withdraw` template is used for slashing. It records the `identitySecret` to be hashed and the hash of the slasher's address.

```
template Withdraw() {
    signal input identitySecret;
    signal input addressHash;

    signal output identityCommitment <== Poseidon(1)([identitySecret]);
}
```

Snippet 4.2: The `Withdraw` template.

There is no check performed that the slashed identity is registered with the protocol.

Impact A malicious user may produce a large number of slash requests in an attempt to deny service to the protocol by either taking up network bandwidth or storage space.

Recommendation Add documentation to the verification procedure for slashing requiring slashed users to be registered.

For `rln-diff`, this may require publishing user message limits for each `identityCommitment`.

Developer Response The check is in the application layer for `rln-same`. A map from `idCommitment` to user message limit is stored for `rln-diff`.

4.1.3 RLN-003: Correctness of IsInInterval is based on implicit assumptions

Severity	Warning	**Commit**	022b690
Type	Maintainability	**Status**	Fixed
File(s)	`circuits/utils.circom`		
Location(s)	IsInInterval		

The `IsInInterval` template (see attached snippet) is correct for its current instantiations based on the following two assumptions:

- Adversarial users can only control `in[1]`. This is true for both `rln-same` and `rln-diff` in the current version of the protocol, since `in[0]` is hardcoded to `1` and altering `in[2]` would prevent users from using the protocol.
- It uses two instances of `LessEqThan`, where `in[1]` is the second argument of the first instance and the first argument of the second instance.

Our team has formally proven that, based on these assumptions, the `IsInterval` template is correct.

```
template IsInInterval(LIMIT_BIT_SIZE) {
    signal input in[3];

    signal output out;

    signal firstCmp <== LessEqThan(LIMIT_BIT_SIZE)([in[0], in[1]]);
    signal secondCmp <== LessEqThan(LIMIT_BIT_SIZE)([in[1], in[2]]);

    out <== firstCmp * secondCmp;
}
```

Snippet 4.3: The `IsInInterval` template

If either of the above two assumptions is violated, attackers can prove false statements by tricking the `LessEqThan` template. This is because the `LessEqThan` template works as expected only if its inputs fit within `LIMIT_BIT_SIZE` bits. However, the current implementation does not enforce this constraint.

Impact Although the current version of the circuit is not exploitable, future modifications could open the door to attackers. For instance, if someone attempts to simplify the circuit by removing the first `LessEqThan` and proving that `in[1]` belongs to the interval `[0, n)`, then attackers could prove statements for values of `in[1]` that are greater than `n`.

Recommendation As the reasons for correctness of `IsInInterval` are fairly complex, we recommend properly documenting the above assumptions in the code to prevent developers from introducing bugs in the future. Alternatively, the template can be simplified as mentioned above. This involves using a single instance of `LessEqThan` to prove that the message is in the interval `[0,n)`, and performing appropriate range checks on the inputs of `LessEqThan`.

Developer Response We have implemented the recommended fix of using a range check and restricting the message to be in the half-open interval `[0,n)`.

4.1.4 RLN-004: identitySecret Naming Inconsistency

Severity	Info	**Commit**	022b690
Type	Maintainability	**Status**	Intended Behavior
File(s)	`rln-same.circom`,`rln-diff.circom`		
Location(s)	RLN		

In the RFC-V1, the following terms are defined for user identification:

```
identity_secret = [identity_nullifier, identity_trapdoor]
identity_secret_hash = poseidonHash(identity_secret)
identity_commitment = poseidonHash([identity_secret_hash])
```

Snippet 4.4: Excerpt from RFC

In the `RLN` templates defined in `rln-same.circom` and `rln-diff.circom`, the term `identitySecret` is used to refer to the `identity_secret_hash`.

```
signal identityCommitment <== Poseidon(1)([identitySecret]);
```

Snippet 4.5: Excerpt from `rln-same.circom`.

Impact Protocol users may be confused about which value to supply as `identitySecret`.

Recommendation Rename `identitySecret` to `identitySecretHash`.

Developer Response RFC for V1 is out-of-date relative to V2. The identitySecret will come from Semaphore. This is just something that needs to be updated in the RFC.

5 Formal Verification

5.1 Formal Verification Procedure

Our team verified several properties of the code at commit `0x10437bc2`. See Table 5.1 for a complete list. See Section 5.3 for a detailed description of each property.

We formally verified the code using Veridise's tool Coda. Coda is an open-source library that can be used to prove the functional correctness of circom circuits by leveraging the Coq proof assistant. These properties were verified by first rewriting the circuits in ML, then proving the properties using Coq.

The only property which must be assumed in the proof is that the `messageLimit` for users is within bounds. This value is public during registration, and checking that it is in bounds is included in the documentation for running Rate Limiting Nullifier.

5.2 Properties Verified

A complete list of the properties verified is shown in Table 5.1. Each row displays a natural language description of the property proved, and its current status (i.e. verified, not verified). Section 5.3 provides a detailed description of each property, along with formal circuit definitions and property specifications.

Table 5.1: Formally verified properties.

ID	Property	Status
V-RLN-SPEC-001	Surpassing the voting limit in an epoch reveals identity.	Verified
V-RLN-SPEC-002	The internal nullifier links messages with the same ID.	Verified
V-RLN-SPEC-003	The computed Merkle root is correct, and uses the correct leaf.	Verified
V-RLN-SPEC-004	messageID is in the range [0, limit).	Verified
V-RLN-SPEC-005	RangeCheck returns true exactly when the signal is in [0, limit).	Verified
V-RLN-SPEC-006	The circuit is not underconstrained.	Verified

Note that V-RLN-SPEC-006 is a consequence of the functional correctness proofs in the prior 5 specifications.

5.3 Detailed Description of Formal Verification Results

In the following section, we outline each formally verified property in detail. Note that due to the size and complexity of the proofs, we will not include them in the official report, the circuit definitions* and proofs† can be found at the provided URLs.

* `https://github.com/Veridise/Coda/tree/39ebd33767924fab127cc0ade561043217e685a4/dsl/circuits/circom_rln`

† `https://github.com/Veridise/Coda/blob/39ebd33767924fab127cc0ade561043217e685a4/BigInt/src/Benchmarks/CircomRLN/Proof.v`

In the header of each specification, RLN refers to both the implementations in `rln-same.circom` and `rln-diff.circom`.

5.3.1 V-RLN-SPEC-001: Surpassing the Voting Limit in an Epoch Reveals Identity

Commit	`0x10437bc2`	**Status**	Verified
Files	`rln-same.circom`, `rln-diff.circom`		
Circuits	RLN		

Description Once someone reaches the message limit, they have revealed two points on the same line, with their secret identity at the y-intercept.

Formal Definition Listing 5.1 shows the formal definition for the `RLN` template from `rln-same.circom`. See Listing 5.2 for the formal definition of the `RLN` template from `rln-diff.circom`.

Listing 5.1: RLN template from `rln-same.circom`

```
let rln =
  Circuit
    { name= "RLN_same"
    ; inputs=
        [ ("DEPTH", tnat)
        ; ("LIMIT_BIT_SIZE", attaches [lift (nu <. zn 253)] tnat)
        ; ("identitySecret", tf)
        ; ("messageId", tf)
        ; ("pathElements", tarr_t_k tf (v "DEPTH"))
        ; ("identityPathIndex", tarr_t_k tf (v "DEPTH"))
        ; ("x", tf)
        ; ("externalNullifier", tf)
        ; ("messageLimit", tf) ]
    ; outputs=
        [ ("y", t_y identity_secret message_id x external_nullifier)
        ; ("root", t_root identity_secret path_elements identity_path_index)
        ; ( "nullifier"
          , t_nullifier identity_secret message_id external_nullifier ) ]
    ; dep= None
    ; body=
        elet "identityCommitment"
          (call "Poseidon" [z1; const_array tf [identity_secret]])
          (elet "root"
             (call "MerkleTreeInclusionProof"
                [ v "DEPTH"
                ; v "identityCommitment"
                ; v "identityPathIndex"
                ; v "pathElements" ] )
             (elet "rangeCheck"
                (call "RangeCheck"
                   [v "LIMIT_BIT_SIZE"; v "messageId"; v "messageLimit"] )
                (elet "a1"
                   (call "Poseidon"
```

```
                    [ z3
                    ; const_array tf
                        [ v "identitySecret"
                        ; v "externalNullifier"
                        ; v "messageId" ] ] )
                (elet "y"
                    (fadd (v "identitySecret") (fmul (v "a1") (v "x")))
                    (elet "nullifier"
                        (call "Poseidon" [z1; const_array tf [v "a1"]])
                        (make [v "y"; v "root"; v "nullifier"]) ) ) ) ) ) }
```

Formal Specification The following shows the formal specification for the desired property for `rln-same.circom`. The specification for `rln-diff.circom` is similar, and included at the listed URL.

```
let t_y identity_secret message_id x external_nullifier =
  tfq
    (qeq nu
       (fadds
          [ identity_secret
          ; fmul
              (u_poseidon z3
                 (const_array tf
                    [identity_secret; external_nullifier; message_id] ) )
              x ] ) )
```


5.3.2 V-RLN-SPEC-002: Internal Nullifier Links Messages with the Same ID

Commit	`0x10437bc2`	**Status**	Verified
Files	`rln-same.circom`, `rln-diff.circom`		
Circuits	RLN		

Description The internal nullifier is the same for two different messages from the same user using the same message ID.

Formal Definition Listings 5.1 and 5.2 show the formal definition for the `RLN` template from `rln-same.circom` and `rln-diff.circom`, respectively.

Listing 5.2: RLN template from `rln-diff.circom`

```
let rln =
  Circuit
    { name= "RLN"
    ; inputs=
        [ ("DEPTH", tnat)
        ; ("LIMIT_BIT_SIZE", attaches [lift (nu <. zn 253)] tnat)
        ; ("identitySecret", tf)
        ; ("userMessageLimit", tf)
        ; ("messageId", tf)
        ; ("pathElements", tarr_t_k tf (v "DEPTH"))
        ; ("identityPathIndex", tarr_t_k tf (v "DEPTH"))
        ; ("x", tf)
        ; ("externalNullifier", tf) ]
    ; outputs=
        [ ("y", t_y identity_secret message_id x external_nullifier)
        ; ( "root"
          , t_root identity_secret user_message_limit path_elements
              identity_path_index )
        ; ( "nullifier"
          , t_nullifier identity_secret message_id external_nullifier ) ]
    ; dep= None
    ; body=
        elet "identityCommitment"
          (call "Poseidon" [z1; const_array tf [identity_secret]])
          (elet "rateCommitment"
             (call "Poseidon"
                [ z2
                ; const_array tf [v "identityCommitment"; v "userMessageLimit"]
                ] )
             (elet "root"
                (call "MerkleTreeInclusionProof"
                   [ v "DEPTH"
                   ; v "rateCommitment"
                   ; v "identityPathIndex"
                   ; v "pathElements" ] )
                (elet "rangeCheck"
                   (call "RangeCheck"
```

```
                    [v "LIMIT_BIT_SIZE"; v "messageId"; v "userMessageLimit"] )
                 (elet "a1"
                    (call "Poseidon"
                       [ z3
                       ; const_array tf
                           [ v "identitySecret"
                           ; v "externalNullifier"
                           ; v "messageId" ] ] )
                    (elet "y"
                       (fadd (v "identitySecret") (fmul (v "a1") (v "x")))
                       (elet "nullifier"
                          (call "Poseidon" [z1; const_array tf [v "a1"]])
                          (make [v "y"; v "root"; v "nullifier"]) ) ) ) ) ) )
  }
```

Formal Specification The following shows the formal specification for the desired property for `rln-same.circom`. The specification for `rln-diff.circom` is similar, and included at the listed URL.

```
let t_nullifier identity_secret message_id external_nullifier =
  tfq
    (qeq nu
       (u_poseidon z1
          (const_array tf
             [ u_poseidon z3
                 (const_array tf
                    [identity_secret; external_nullifier; message_id] ) ] ) ) )
```


5.3.3 V-RLN-SPEC-003: Computed Merkle Root Functional Correctness

Commit	`0x10437bc2`	**Status**	Verified
Files	`utils.circom`		
Circuits	MerkleTreeInclusionProof		

Description The Merkle root is computed correctly.

Formal Definition The following shows the formal definition for the `MerkleTreeInclusionProof` template:

```
let lam_mtip z =
  lama "_i" tint
    (lama "x" tf
       (elet "u0"
          (* path_index[i] binary *)
          (assert_eq (fmul (z_i_0 z) (fsub f1 (z_i_0 z))) f0)
          (elet "c"
             (const_array (tarr_tf z2)
                [const_array tf [x; z_i_1 z]; const_array tf [z_i_1 z; x]] )
             (elet "m"
                (call "MultiMux1" [z2; c; z_i_0 z])
                (call "Poseidon" [z2; m]) ) ) ) )

let hasher z len init =
  iter z0 len (lam_mtip z) ~init ~inv:(fun i ->
      tfq (qeq nu (u_hasher (u_take i z) init)) )

let mrkl_tree_incl_pf =
  Circuit
    { name= "MerkleTreeInclusionProof"
    ; inputs=
        [ ("DEPTH", tnat)
        ; ("leaf", tf)
        ; ("pathIndex", tarr_tf depth)
        ; ("pathElements", tarr_tf depth) ]
    ; outputs= [("root", t_r)]
    ; dep= None
    ; body= elet "z" (zip path_index path_elements) (hasher z depth leaf) }
```

Formal Specification The following shows the formal specification for the desired property.

```
let t_r = tfq (qeq nu (u_hasher (u_zip path_index path_elements) leaf))
```

5.3.4 V-RLN-SPEC-004: messageID is in [0, limit)

Commit	`0x10437bc2`	**Status**	Verified
Files	`rln-same.circom`, `rln-diff.circom`		
Circuits	RLN		

Description The used messageID is guaranteed to be in the range [0, limit).

Formal Definition See Listings 5.1 and 5.2 for the definitions of `RLN`.

Formal Specification This property is proven via the usage of `RangeCheck` in the definition of `RLN`. and the property proved in 5.3.5.

5.3.5 V-RLN-SPEC-005: RangeCheck Functional Correctness

Commit	`0x10437bc2`	**Status**	Verified
Files	`utils.circom`		
Circuits	RangeCheck		

Description `RangeCheck` outputs 1 exactly when the messageID is in the $[0, \text{limit})$ range, under the assumption that limit is representable within `LIMIT_BIT_SIZE` bits. This assumption is necessary. However, Privacy and Scaling Exploration indicated that they will document the necessity of this check in an upcoming to RFC.

Formal Definition The following shows the formal definition for the `RangeCheck` template.

```
let range_check =
  Circuit
    { name= "RangeCheck"
    ; inputs=
        [ ("LIMIT_BIT_SIZE", attaches [lift (nu <. zn 253)] tnat)
        ; ("messageId", tf)
        ; ("limit", tf) ]
    ; outputs= [("rangeCheck", t_lt messageId limit)]
    ; dep= None
    ; body=
        elet "bitCheck"
          (call "Num2Bits" [limit_bit_size; messageId])
          (call "LessThan" [limit_bit_size; messageId; limit]) }
```

Formal Specification The following shows the formal specification for the desired property.

```
let t_lt a b = tfq (ind_dec nu (toUZ a <. toUZ b))
```

Glossary

Circom a programming language used to express both a witness computation and constraints for ZK-circuit generation. To learn more, visit https://docs.circom.io. 1

Coq A system/programming language for formal proofs. Read more at https://coq.inria.fr . 15

ML A Haskell-like programming language. See https://en.wikipedia.org/wiki/ML_(programming_language) for more details . 15

Shamir's Secret Sharing A method which breaks a piece of secret information into several parts, requiring some minimum number of parts to recover the secret. For more info, see https://en.wikipedia.org/wiki/Shamir%27s_secret_sharing. 1

Zero Knowledge Circuit An encoding of a computation which allows verification that third-parties have performed a certain computation, without revealing anything else about the inputs to that computation. 1, 23

ZK-circuits Zero Knowledge Circuit. 1