Formal Verification Report: Worldcoin Semaphore MTB

2024-02-08

Marcin Kostrzewa
Reilabs

Giuseppe Afentoulis
Reilabs

1. Abstract

This report is a summary of the formal verification efforts undertaken by Reilabs for the Worldcoin Semaphore MTB (Merkle Tree Batcher) project. It cointans a summary of the work done, a description of the methodology and our interpretation of the results. This effort has been financed by the Tools for Humanity corporation, the leading contributor to the Worldcoin protocol.

The codebase for this project can be found in the Semaphore MTB repository, with the main results described here contained in the `Main.lean` file.

Contents

2. Introduction

The Worldcoin protocol is building a decentralized digital identity and finance network. As part of its operations it maintains a set of verified users on the Ethereum mainnet. The protocol uses a variety of novel zero knowledge proof techniques in order to reduce the costs of maintaing such a large—over three million at the time of writing—set of identities.

Despite their cost effectiveness, these techniques make it difficult to reason about the behavior of the system. The behavior is described using the arithmetic circuit model, which gives rise to a system of polynomial equations, rather than a more traditional step-by-step algorithm. Since such equation systems are not generally guaranteed to have a unique solution, this inherently introduces non-determinism to the system. Moreover, arithmetic operations are performed modulo a large prime number, making them somewhat counterintuitive for developers and auditors alike.

Reilabs is a research and development company, specializing in cryptography, developer tools and formal methods. We co-authored the Semaphore MTB protocol, and built the tooling necessary for formal verification of arithmetic circuits created with the Gnark framework. This tooling allowed us to undertake a formal verification effort to ensure the correctness of the Semaphore MTB protocol.

2.1. Semaphore MTB

2.1.1. Merkle Tree Transformations

Semaphore MTB (Merkle Tree Batcher) is a set of ZK protocols used for maintaining a merkle tree in an ethereum smart contract while minimizing the cost of updates. It consists of two arithmetic circuit templates, parametrized by B - the batch size and D - the depth of the tree.

2.1.1.1. Insertion Circuit

The goal of this circuit is ensure a correct merkle tree root transition by setting a contiguous segment of empty tree leaves to the given values. The relevant inputs are:

- `PreRoot`, the root of the merkle tree before update;
- `StartIndex`, the index of the first leaf to be updated;
- `IdentityCommitments`, a list $[\mathrm{id}_0, \mathrm{id}_1, ..., \mathrm{id}_{B-1}]$ of B values that will become the new leaf values;
- `PostRoot`, the merkle tree root after the update is performed.

The intended operation of this circuit, is that it has a satisfying assignment if and only if:

1. `PreRoot` is the root of a known merkle tree T;
2. The values in T at indices $[\mathrm{StartIndex}, \mathrm{StartIndex} + 1, ..., \mathrm{StartIndex} + B - 1]$ are all equal to 0;
3. `PostRoot` is the root of T after modifying it such that the value at $\mathrm{StartIndex} + i$ is equal to id_i for $i = 0, 1, ..., B - 1$ and all other leaf values are unchanged.

2.1.1.2. Deletion Circuit

The goal of this circuit is to ensure a correct merkle root transition by changing the value of selected B leaves to 0, and thereby remove those leaves from the tree. The relevant inputs are:

- `PreRoot`, the root of the merkle tree before update;
- `DeletionIndices`, a list $[i_0, i_1, ..., i_{B-1}]$ of B indices of leaves to be set to 0;
- `PostRoot`, the merkle tree root after the update is performed.

The intended operation of this circuit, is that it has a satisfying assignment if and only if:

1. `PreRoot` is the root of a known merkle tree T;
2. All deletion indices are less than 2^{D+1}. Indices i such that $2^D \leq i < 2^{D+1}$ are specifically designated as no-op, allowing for "batch padding" - generating proofs of removal of fewer than B leaves;
3. `PostRoot` is the root of T after modifying it such that the value at i_j is 0 for $j = 0, 1, ..., B-1$ and all other leaf values are unchanged.

2.1.2. Public Input Hashing

The ZK proof system used for Semaphore MTB is based on the Groth16 scheme. This incurs a verifier cost of an additional elliptic curve multiplication and addition for *each public input*. Since the verification is performed on-chain where these operations are expensive, this directly translates to a significant monetary cost for the party performing the proof verification.

In order to reduce this cost, the Semaphore MTB system reduces the problem to *a single public input*. This is done through a process that concatenates all of the inputs and then hashes that array using Keccak256. The result is then reduced modulo the group order to make sure it fits into a single circuit input.

2.2. Formal Verification Methodology

The circuit templates are implemented using the Gnark framework. Reilabs has created an automated toolchain to enable the extraction of Gnark circuits into equivalent definitions in the Lean 4 theorem prover. This allows us to model the semantics of the extracted circuits in a formal setting.

It is important to note that the circuits are being analyzed using a model of the Gnark circuit DSL represented in Lean by the ProvenZK library. No analysis has been performed on the *correctness of the translation* from the DSL to the relevant constraint system, or any of the other cryptographic operations used by the system.

Due to the nature of the extraction process, we perform the analysis on circuits, *not* circuit templates. That is, all circuit size-and-shape parameters must be chosen before extraction happens. Therefore, the D and B parameters must be specified at the time of extraction. We have chosen to use the values of $D = 30$ and $B = 4$. The choice of tree depth corresponds exactly to the one used by Worldcoin in their production deployment of Semaphore MTB. The batch size is chosen such that it is large enough to be representative of the real-world use case, but small enough to allow for reasonable compilation times of the Lean codebase. This choice follows the scaling properties of the circuits. Tree indices are serialized into 31-bit values, so there is a natural limit to the depth of the tree that can be used in practice and we must ensure that the tree depth chosen fits within this limit. The batch size, on the other hand, has no such limits (other than resource use) and values as high as 1200 have been previously used in practice.

The correctness of the properties proven is continuously checked using Semaphore MTB's CI service, ensuring their ongoing validity during the sys-

tem's evolution. While this document refers to the repository state at commit #7f6099c3843bff97871aa55f4bb40c3f3c36f513, the formal verification results are continuously updated and can be checked in any newer commit of interest.

3. Verified Properties

This section describes the properties that have been formally proven for the Semaphore MTB system. They are laid out such that they present a coherent argument for overall correctness of the system. The argument will proceed in three stages:

1. First, we establish the correctness of the hash functions implementations used throughout the system. This is a fundamental building block for further analysis: both the public input hashing approach and the merkle tree implementations rely on the correctness of these functions.
2. Then we focus on proving the input hashing algorithm correct. The input hash is the only public input to the Groth16 verifier and therefore we must ensure that it is deterministic (that is, for any choice of other inputs, there is exactly one valid input hash) and collision resistant (that is, for any given input hash, there is only one *known or efficiently computable* choice of inputs that produces it).
3. Finally we proceed to verify the correctness of merkle tree transformations implemented in the circuits. We will show that both circuits ensure valid evolution of the tree and that they will not spuriously get "stuck" in an unexpected state.

This structure should be sufficiently convincing that the operation of the system adheres to the specification outlined informally in the previous sections.

3.1. Conventions and Assumptions

Throughout this document, we will use the following conventions:

1. D is the depth of the merkle tree and is equal to 30;
2. B is the batch size and is equal to 4;
3. F denotes the field $\mathrm{GF}(p)$, where p is the order of the $\mathbb{G}_1$ group of the $\mathrm{BN254}$ curve, i.e. $p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$;

In the following sections, we will only present theorem statements. The proofs are available in the linked source files, but their contents are not relevant for this analysis: Lean is a proof assistant, which means that all proofs are automatically checked for correctness. We run this verification on each run of the Continuous Integration pipeline.

Furthermore, we freely assume the collision resistance of the Poseidon hash function and functions in the Keccak256 family. We model it through injectivity: that is for a given hash function H, the equality $H(a) = H(b)$ yields $a = b$. While this assumption is demonstrably untrue (e.g. by a simple counting argument), it follows both the typical usage and the industry standard for reasoning about hash functions.

Circuit Calling Semantics

Reilabs' `gnark-lean-extractor` compiles the circuit into a Lean function returning a proposition. This proposition is true if and only if all gates can be assigned values such that the circuit is satisfied. Therefore, any occurrence of a circuit call in a proposition should be interpreted as "there is an assignment of values to gates that satisfies the circuit".

3.2. Extracted Circuits and Gadgets

The automatically-generated definitions can be inspected in the `FormalVerification.lean` file. While it is commited to the repository for reading convenience, its contents are being regenerated on every CI run to ensure its relevance with respect to the actual (Gnark) circuit definitions. The below signatures of the top-level circuits specify the parameter names, types and argument orderings that correspond to the main circuit templates.

```
def DeletionMbuCircuit_4_4_30_4_4_30
  (InputHash: F)
  (DeletionIndices: Vector F 4)
  (PreRoot: F)
  (PostRoot: F)
  (IdComms: Vector F 4)
  (MerkleProofs: Vector (Vector F 30) 4): Prop
```

and

```
def InsertionMbuCircuit_4_30_4_4_30
  (InputHash: F)
  (StartIndex: F)
  (PreRoot: F)
  (PostRoot: F)
  (IdComms: Vector F 4)
  (MerkleProofs: Vector (Vector F 30) 4): Prop
```

All the other definitions in that file correspond to gadgets (reusable circuit fragments) that are being directly or indirectly used by the main circuits.

3.3. Hash Functions

There are two families of hash functions used in the Semaphore MTB system:

1. **Poseidon:** Used for computation of the nodes in the merkle tree, the Poseidon hash function [1] is chosen due to Worldcoin's reliance on the Semaphore protocol [2].
2. **Keccak256:** Used for hashing the public input, this is the only hash function natively supported on the EVM [3]. We represent this by two sub-circuits—one taking 640 bits of input for the deletion circuit, and one taking 1568 bits of input for the insertion circuit—as we work with known-size inputs.

For each of these subcircuits, we establish two basic properties:

1. **Determinism**: That the output gates are uniquely assigned based on the values of the input gates;
2. **Correct Output on Test Inputs**: That the assignment is verified to be correct against another implementation on some test inputs.

Property 1 alone mitigates one of the largest sources of exploits in ZK protocols: gate value assignments that were not anticipated by the protocol designers. Determinism ensures no such assignments exist - that is gates designated as outputs have a unique assignment based on the inputs.

Property 2 is more akin to a unit test. We claim that it is *overwhelmingly improbable*, within the collision resistance bounds of the hash function, for an arbitrary function and a given hash function to produce the same output for a randomly-selected input unless **they are the same function**. We check multiple such instances as part of Property 2.

Therefore, this combination of properties is sufficiently convincing to show equivalence to the intended implementation of the hash functions.

We demonstrate determinism by using the `UniqueAssignment` structure. The existence of such a structure for a given gadget means that the gadget is equivalent to a function, i.e. there is exactly one valid output value corresponding to a given input.

Extracted Circuit Structure and `UniqueAssignment`

To properly model non-determinism, the ProvenZK model of a circuit uses existential quantification and CPS (continuation-passing style). A sub-circuit (also referred to as a "gadget") that performs a number of assertions and produces a value of type α is encoded in the type $(\alpha \to \text{Prop}) \to \text{Prop}$. The `UniqueAssignment` structure is defined as follows:

```
structure UniqueAssignment (f : (β → Prop) → Prop) (emb : α → β) where
  val: α
  equiv: ∀k, f k = k (emb val)
```

The structure consists of:

- The constant `val` that we claim the gadget call can be replaced with;
- The proof that for any continuation proposition k, the gadget call can be replaced with a direct application of k to the embbeded value of `val`, without changing the proposition's truth value.

Therefore, the `UniqueAssignment` structure serves both as a computational counterpart to a propositional gadget and as a proof of its equivalence in any context.

The `emb` parameter is used to improve composability, by allowing the space of possible assignments to be mapped from a more natural representation, e.g. by restricting α to boolean values, even if they are only consumed after embedding into F.

3.3.1. Poseidon

Poseidon's determinism is demonstrated in `FormalVerification/Poseidon.lean:22`, by deriving a term of the following type:

```
def poseidon_3_uniqueAssignment (inp : Vector F 3):
    UniqueAssignment (SemaphoreMTB.poseidon_3 inp) id
```

That is, for any input `inp`, the `SemaphoreMTB.poseidon_3` gadget with inputs set to `inp` can be replaced with the provided constant using the trivial (identity) embedding.

Poseidon's correctness on a test input is demonstrated in `FormalVerification/Poseidon.lean:31`, by verifying that:

```
theorem poseidon_3_testVector:
  (poseidon_3_uniqueAssignment vec![0,1,2]).val =
    vec![0x115cc0f5e7d690413df64c6b9662e9cf2a3617f2743245519e19607a4417189a,
         0x0fca49b798923ab0239de1c9e7a4a9a2210312b6a2f616d18b5a87f9b628ae29,
         0x0e7ae82e40091e63cbd4f16a6d16310b3729d4b6e138fcf54110e2867045a30c
        ]
```

These values are taken from the test vectors provided in the reference implementation.

3.3.2. Keccak256

Keccak's determinism is demonstrated for both instantiations in `FormalVerification/Keccak.lean:129-189`, by deriving terms of the following types:

```
def KeccakGadget_640_64_24_640_256_24_1088_1_uniqueAssignment
  (input : Vector Bool 640)
  (rc : Vector (Vector Bool 64) 24):
  UniqueAssignment
    (SemaphoreMTB.KeccakGadget_640_64_24_640_256_24_1088_1
      (input.map Bool.toZMod)
      (rc.map (Vector.map Bool.toZMod)))
    (Vector.map Bool.toZMod)
```

and

```
def KeccakGadget_1568_64_24_1568_256_24_1088_1_uniqueAssignment
  (input : Vector Bool 1568)
  (rc : Vector (Vector Bool 64) 24):
  UniqueAssignment
    (SemaphoreMTB.KeccakGadget_1568_64_24_1568_256_24_1088_1
      (input.map Bool.toZMod)
      (rc.map (Vector.map Bool.toZMod)))
    (Vector.map Bool.toZMod)
```

These show that for any input that is a result of naturally embedding a boolean vector into F (that is mapping `false` to 0 and `true` to 1), the gadgets have unique assignments that are themselves embeddings of boolean vectors. While this does not state anything about the gadget's behavior outside of the boolean vector domain, that behavior is not relevant - all uses of the gadgets are proven to only ever be assigned within this domain.

Correctness on test inputs is combined with the reduction modulo field order and established in `Main.lean:144` and `Main.lean:336`. Reference values are obtained by running the following Solidity snippet:

```
string memory data = "<TEST STRING OF CORRECT LENGTH>";
uint256 result;
assembly {
  result := mod(
    keccak256(add(data, 0x20), mload(data)),
    0x30644e72e131a029b85045b68181585d2833e84879b9709143e1f593f0000001
  )
}
```

in Chisel v.0.2.0. The reference strings are:

```
"This string is exactly 80 bytes long, which is unbelievably lucky for this test."
```

and

```
"This is string is exactly 196 bytes long, which happens to be exactly the length we need to test the 1568-bit keccak hash implementation, that can be found in the SemaphoreMTB Insertion Circuit..."
```

for the 640-bit and 1568-bit versions respectively.

3.4. Input Hashing

As described in the specification section, the public inputs to Semaphore MTB circuits are hashed using Keccak256 and reduced modulo the group order. In this section we establish the correctness of this operation. For each of the circuits, we will demonstrate the following properties:

1. **Input Hash Determinism**: That given two satisfying assignments to the circuit, if they agree on the public inputs, they must also agree on the input hash;
2. **Input Hash Collision Resistance**: That given two satisfying assignments to the circuit, if they agree on the input hash, they must also agree on the public inputs.

Property 1 may seem trivial, but due to the nature of operations involved (i.e. binary decompositions and bitwise operations), it is not immediately obvious that it holds. In fact, an earlier version of the insertion circuit violated this property, resulting in the need to perform additional checks in the on-chain verifier contracts.

Property 2 only holds under the assumption of the injectivity of the Keccak 256 hash family. More strictly: the knowledge of two assignments agreeing on the input hash, but not the public inputs, would immediately yield knowledge of a pair of conflicting Keccak 256 values modulo the field order.

3.4.1. Insertion Circuit

Determinism is stated in `Main.lean:307` as:

```
theorem inputHash_deterministic:
    SemaphoreMTB.InsertionMbuCircuit_4_30_4_4_30
      InputHash₁ StartIndex PreRoot PostRoot IdComms MerkleProofs₁ ∧
    SemaphoreMTB.InsertionMbuCircuit_4_30_4_4_30
      InputHash₂ StartIndex PreRoot PostRoot IdComms MerkleProofs₂ →
    InputHash₁ = InputHash₂
```

That is: if two assignments satisfy the circuit and reuse `StartIndex`, `PreRoot`, `PostRoot` and `IdComms`, they must necessarily use the same value for `InputHash`.

Collision resistance is stated in `Main.lean:352` as:

```
theorem inputHash_injective:
  SemaphoreMTB.InsertionMbuCircuit_4_30_4_4_30
    InputHash StartIndex₁ PreRoot₁ PostRoot₁ IdComms₁ MerkleProofs₁ ∧
  SemaphoreMTB.InsertionMbuCircuit_4_30_4_4_30
    InputHash StartIndex₂ PreRoot₂ PostRoot₂ IdComms₂ MerkleProofs₂ →
  StartIndex₁ = StartIndex₂ ∧
  PreRoot₁    = PreRoot₂    ∧
  PostRoot₁   = PostRoot₂   ∧
  IdComms₁    = IdComms₂
```

That is: if two assignments satisfy the circuit and use the same `InputHash`, they must necessarily agree on `StartIndex`, `PreRoot`, `PostRoot` and `IdComms`. This theorem is only proven assuming the additional axiom

```
axiom reducedKeccak1568_collision_resistant :
  ∀x y, reducedKeccak1568 x = reducedKeccak1568 y → x = y
```

which expresses the lack of conflicts in the keccak hash function. Note that `reducedKeccak1568` is the extracted function that we've reasoned about in the hash functions section.

3.4.2. Deletion Circuit

Determinism is stated in `Main.lean:112` as:

```
theorem inputHash_deterministic:
    SemaphoreMTB.DeletionMbuCircuit_4_4_30_4_4_30
      InputHash₁ DeletionIndices PreRoot PostRoot IdComms₁ MerkleProofs₁ ∧
    SemaphoreMTB.DeletionMbuCircuit_4_4_30_4_4_30
      InputHash₂ DeletionIndices PreRoot PostRoot IdComms₂ MerkleProofs₂ →
    InputHash₁ = InputHash₂
```

That is: if two assignments satisfy the circuit and reuse `DeletionIndices`, `PreRoot` and `PostRoot`, they must necessarily use the same value for `InputHash`.

Collision resistance is stated in `Main.lean:161` as:

```
theorem inputHash_injective:
    SemaphoreMTB.DeletionMbuCircuit_4_4_30_4_4_30
      InputHash DeletionIndices₁ PreRoot₁ PostRoot₁ IdComms₁ MerkleProofs₁ ∧
    SemaphoreMTB.DeletionMbuCircuit_4_4_30_4_4_30
      InputHash DeletionIndices₂ PreRoot₂ PostRoot₂ IdComms₂ MerkleProofs₂ →
    DeletionIndices₁ = DeletionIndices₂ ∧
    PreRoot₁         = PreRoot₂         ∧
    PostRoot₁        = PostRoot₂
```

That is: if two assignments satisfy the circuit and use the same `InputHash`, they must necessarily agree on `DeletionIndices`, `PreRoot` and `PostRoot`. This theorem is only proven assuming the additional axiom

```
axiom reducedKeccak640_collision_resistant :
  ∀x y, reducedKeccak640 x = reducedKeccak640 y → x = y
```

which expresses the lack of conflicts in the keccak hash function. Note that `reducedKeccak640` is the extracted function that we've reasoned about in the hash functions section.

3.5. Merkle Tree Transformations

While these circuits are expected to behave according to the rules outlined in the specification section, their actual implementation operates on very different notions. Where the specification is phrased in terms of merkle trees and elements stored in their leaves, the implementation operates solely based on a series of merkle proof verifications, using merkle proofs passed as private inputs.

It is therefore of utmost importance that we bridge this gap, by providing formal proofs of the equivalence between the specification and implementation. To that end, we use a model of merkle trees defined in the ProvenZK library.

Most theorems in this section operate under the additional assumption that the poseidon hash function is collision resistant, which is expressed by said theorems depending on a parameter denoted as `[Fact (CollisionResistant poseidon₂)]`.

Lean Indexing Syntax

The theorems in the following sections will use special Lean notations for indexing into vectors and merkle trees. Below is an explanation of each of them:

- `collection[i]` accesses the `i`-th element of `collection`, as long as `i` is proven to be in the valid range (e.g. `i < Vector.length v` for vectors or `i < 2 ^ D` for a depth-D merkle tree).
- `collection[i]!` accesses the `i`-th element of `collection`, if `i` is in the valid range, and returns the default element (0 in F) otherwise.
- `collection[i]?` returns an `Option` type, with `none` if `i` is not in the valid range and `some` of the `i`-th element otherwise.

It is particularly important to note, due to the visual similarity to other programming languages, that `v[i]! = 0` in Lean is interpreted as `(v[i]!) = 0` and **not** `(v[i]) != 0`.

3.5.1. Insertion Circuit

This section describes theorems relating to the semantics of the insertion circuit.

3.5.1.1. Previous Value Validation

It is a requirement that this circuit only serves for *inserting* values, without the ability to overwrite an existing value. Therefore, the circuit checks that the values at all overwritten leaves are zero in the initial tree. This behavior is captured by the following theorem (Main.lean:176):

```
theorem before_insertion_all_items_zero
  [Fact (CollisionResistant poseidon₂)]
  {tree: MerkleTree F poseidon₂ D}
  {startIndex : F}:
    SemaphoreMTB.InsertionMbuCircuit_4_30_4_4_30
      InputHash startIndex tree.root PostRoot IdComms MerkleProofs →
    ∀ i ∈ [startIndex.val:startIndex.val + B], tree[i]! = 0
```

3.5.1.2. Value Update

The stated semantics of the circuit are that the leaves in the range from $\mathrm{StartIndex}$ to $\mathrm{StartIndex} + B - 1$ are set to the respective values from $\mathrm{IdentityCommitments}$, while all other leves remain unchanged. This is captured by the following theorem (Main.lean:208):

```
theorem root_transformation_correct
  [Fact (CollisionResistant poseidon₂)]
  {Tree : MerkleTree F poseidon₂ D}:
  SemaphoreMTB.InsertionMbuCircuit_4_30_4_4_30
    InputHash StartIndex Tree.root PostRoot IdComms MerkleProofs →
  ∃(postTree : MerkleTree F poseidon₂ D),
    postTree.root = PostRoot ∧
    (∀ i ∈ [StartIndex.val:StartIndex.val + B],
      postTree[i]! = IdComms[i-StartIndex.val]!) ∧
```

```
    (∀ i, i ∉ [StartIndex.val:StartIndex.val + B] →
      postTree[i]! = Tree[i]!)
```

3.5.1.3. Ensured Progress and Lack of Censorship

So far we've only been exploring the space of behaviors the circuit exhibits when it is satisfied. However, it is equally important that we establish when the circuit can actually be used – otherwise system progress may become impossible at some point.

Under such circumstances it would become impossible to insert new identities. Similarly, it could become possible to deny insertion of certain identities at the level of the circuit.

In this section we establish the final result for the insertion circuit: that as long as there is a contiguous segment of empty leaves of length B, for any input list of identities, we can construct a satisfying assignment. This is captured by the following theorem (Main.lean:263):

```
theorem assignment_exists
  [Fact (CollisionResistant poseidon₂)]
  {tree : MerkleTree F poseidon₂ D}:
    startIndex + B < 2 ^ D ∧
    (∀i ∈ [startIndex : startIndex + B], tree[i]! = 0) →
    ∃ proofs postRoot inputHash,
      SemaphoreMTB.InsertionMbuCircuit_4_30_4_4_30
        inputHash startIndex tree.root postRoot idComms proofs
```

3.5.2. Deletion Circuit

This section describes theorems relating to the semantics of the deletion circuit.

3.5.2.1. Value Update

The stated semantics of the circuit are that the leaves at $\mathrm{DeletionIndices}[0], \mathrm{DeletionIndices}[1], ..., \mathrm{DeletionIndices}[\mathrm{B\text{-}1}]$ are all set to zero, while all other leaves remain unchanged. Moreover, indices that are greater than or equal to 2^D are designated as no-op. The following theorem captures this behavior (Main.lean:39):

```
theorem root_transformation_correct
  [Fact (CollisionResistant poseidon₂)]
  {tree : MerkleTree F poseidon₂ D}:
    SemaphoreMTB.DeletionMbuCircuit_4_4_30_4_4_30
      inputHash deletionIndices tree.root postRoot identities merkleProofs →
    ∃(postTree : MerkleTree F poseidon₂ D),
    postTree.root = postRoot ∧
    (∀ i ∈ deletionIndices, postTree[i.val]! = 0) ∧
    (∀ i, i ∉ deletionIndices → postTree[i.val]! = tree[i.val]!)
```

Note that while this theorem does not explicitly mention the no-op behavior for indices out of range, it is still a corollary. Indeed, the theorem gives us a complete description of all leaves of $\mathrm{postTree}$, where valid indices are zero when present in $\mathrm{deletionIndices}$ and equal to the corresponding leaves of tree otherwise. Therefore, any invalid indices can not have any bearing on the root value.

3.5.2.2. Ensured Progress and Lack of Privileged Values

As in the case of the insertion circuit, we need to ensure that the system can always make progress: that is it is always possible to remove members from the tree, regardless of their value. More formally: for any choice of tree and a vector of indices, we can choose the re-

maining parameters such the circuit is satisfied. This is captured by the following theorem (Main.lean:71)"

```
theorem assignment_exists
  [Fact (CollisionResistant poseidon₂)]
  {tree : MerkleTree F poseidon₂ D}
  {indices : Vector F B}:
    (∀i ∈ indices, i.val < 2^(D+1)) →
    ∃ inputHash identities proofs postRoot,
      SemaphoreMTB.DeletionMbuCircuit_4_4_30_4_4_30
        inputHash indices tree.root postRoot identities proofs
```

4. Risks and Limitations

It must be noted that this analysis, even though it provides *strong evidence* for the correctness of the Semaphore MTB protocol, is inherently incapable of proving the **absence** of attack vectors. The following is a (non-exhaustive) list of risks that are not covered by this analysis and must be considered:

- The `gnark-lean-extractor` tool has not been audited or formally verified. Errors in this tool may lead to the extracted circuits not being equivalent to the source circuits.
- The `ProvenZK` Lean library, providing models of circuit gates and other cryptographic primitives, has not been audited. Errors in this library may lead to the modelled behavior of arithmetic gates not corresponding to their actual implementation in the Gnark framework.
- This analysis takes certain liberties when it comes to the treatment of collision resistance of certain hash functions. While the authors believe these assumptions reasonable and have taken care to not use them beyond their typical use in cryptograhic analysis, they are provably false and could be used to render the Lean prover inconsistent. Despite that this could be manually proven false, this is not something we do and so the analysis contained herein is still sound.
- The gate representation is an early representation of the circuit, that undergoes many further transformations, including translation to R1CS and further cryptographic operations. Therefore, due to errors that may occur in the Gnark toolchain, the actual behavior of the protocol may differ from its circuit-level specification.

5. Conclusion

Our analysis of the design of the Semaphore MTB system, accompanied by the results that have been stated and explained with aid of formal verification above, provide strong evidence for the correctness of the arithmetic circuits used by said system. The statement of this specification in a formal setting, in combination with the automatically-extracted model of the system's implementation, has allowed us to prove that the existing implementation satisfies its specification.

Our work here is key to establishing confidence in the smooth operation of Semaphore MTB, the foundation of the WorldID system.

Bibliography

[1] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems", Jul. 2023. [Online]. Available: https://eprint.iacr.org/2019/458

[2] K. Gurkan, K. Wei Jie, and B. Whitehat, "Community Proposal: Semaphore: Zero-Knowledge Signaling on Ethereum". Mar. 31, 2020. Accessed: Feb. 02, 2024. [Online]. Available: https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-semaphore.pdf

[3] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", Aug. 2015. Accessed: Feb. 02, 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf