Formal Verification Report: Light Protocol Merkle Tree Operations

2024-08-08

Ara Adkins
Reilabs

Marcin Kostrzewa
Reilabs

1. Summary

This report summarizes the formal verification work undertaken by Reilabs to audit the Light Protocol subsystems for manipulating Merkle Trees. This effort makes use of Reilabs' framework for the extraction of circuit definitions from Gnark in conjunction with the Lean theorem proving environment where properties are proved about the subsystem in order to demonstrate its correctness.

The sections below contain:

- An introduction to the project and its goals.
- A high-level overview of the domain, providing the background necessary to understand the formal verification work presented in this report.
- A description of the formal verification methodology, the properties that have been verified, and the verification effort itself.
- A summary of the risks and limitations of such a formal verification effort.

The result is that the Merkle Tree operations as part of the Light Protocol have been shown to operate correctly, and that these properties are verified continuously as part of the protocol's test suite.

1.1. Conventions

Throughout this document we use callout blocks to highlight important pieces of information about the content being discussed. Conventionally we colors them as follows:

- **Blue** callouts are used to provide *additional information* about the topic at hand.
- **Purple** callouts are used to provide information about a *related* topic, or to answer an obvious question.
- **Orange** callouts are used to provide a note on something that may be a *project risk*.
- **Red** callouts are used to provide *warnings*.

Contents

2. Introduction

Light Protocol Labs is a company building tooling to enable compression of on-chain state for the Solana L1. They have introduced the “ZK Compression” paradigm, which reduces the cost of computation on that state by orders of magnitude [1]. While doing so, they are able to preserve the security, performance and composability of Solana [2], as well as perform custom ZK computation atop the compressed data.

Reilabs is a software consultancy that specializes in solving difficult problems. We focus on cryptography, tooling, and—most importantly for this project—*Formal Verification*. In order to give Light Protocol Labs and their clients confidence in the Merkle Tree operations, we have employed our custom tooling to extract and prove correctness properties about these operations as they exist in the production version of the Light Protocol.

2.1. Goals

This project focuses on the three primary circuits in the light-prover package: inclusion, non-inclusion, and the combination of the two. We use formal verification to ensure the correctness of these specific tree operations through a demonstration of their correct operation with regards to the specification of their behavior.

3. Domain Overview

In order to understand the analysis in this document, we need to provide some amount of background for the concepts it discusses. The following sections are intended to be *extremely high level* overviews, and do not provide comprehensive detail about the topics discussed.

3.1. Merkle Trees

A **Merkle Tree**—or hash tree—is a binary tree where the leaf nodes contain the cryptographic hash of some data, and every non-leaf node contains the cryptographic hash of the concatenation of its child nodes [3]. In doing so it provides for efficient and secure verification of large amounts of data, as it is effectively impossible for the root node of the tree not to change if the tree is updated with new data.

This means that the root value of the tree can stand as an effective cryptographic commitment to the data contained within the tree. This can be thought of as compression, as one small field element can represent the entire contents of the tree while still ensuring that said data is uniquely identified. You can see this operation in Figure 1 below.

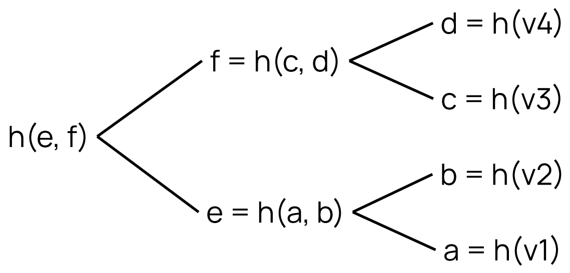

Figure 1: A diagram demonstrating the a Merkle Tree of depth 2 for some hash function h.

Merkle trees provide the ability to prove that a revealed value v is an element in the tree by also providing a list of *auxiliary values*—and the index at which v occurs—such that those values can be hashed together with the known value to yield the also-known root value of the tree. This list of values is known as a **Merkle Proof**.

By way of example, take d to be our value v in the tree above. The merkle proof is then the elements c and e, and the index 3.

Security of Cryptographic Hashes

Merkle Trees make use of *cryptographic hash operations* in computing the values for the nodes in the tree.

Cryptographic hashes are able to map an arbitrary-length input to a fixed-length output, and thereby are vulnerable to **collisions**. In the context of cryptography, we consider the security level of the hash (described as a number of bits) to be the probability of finding **two different inputs** that hash to the same value: a collision.

In the analysis that follows, we assume that the hashes in question cannot exhibit collisions, as is done for the overwhelming majority of cryptanalysis of such systems. It should nevertheless be noted, that the choice of hash for the Merkle Tree will affect the chance of collision and hence the security level of the system relying on such a tree.

3.2. Range Tree

A **Range Tree**, or Indexed Merkle Tree, is a Merkle Tree that allows performing efficient proofs of non-membership (non-inclusion) while still storing arbitrary field elements [4].

These were developed as a means to solve the problem of append-only nullifier trees. Nullifier trees provide a private mechanism by which state can be "destroyed"—as naïvely updating values can leak information—and are typically represented as a sparse Merkle Tree (one in which not all of the leaves are populated). When wanting to alter the private state tree, one must create a nullifier and prove that said nullifier does not already exist in the nullifier tree [4].

Range trees provide a *more efficient* but *semantically equivalent* means to create a nullifier tree that can contain all values in the field without having to allocate a leaf for every field value. It does this by extending each leaf node to additionally contain pointers to the leaf with the nearest inserted higher value. This conceptually turns the Merkle Tree into a Linked List.

$$\text{leaf} = \{v, i_{\text{next}}, v_{\text{next}}\}$$

Insertion into the tree is defined such that **no values exist in the tree** between v and v_{next}. A formal definition of this process can be found in the Aztec documentation [4], though we omit it here for the sake of brevity.

This brings a *huge* performance improvement, as it becomes possible to select an arbitrary depth for the tree, rather than the depth being defined by the field size as is the case for a classic nullifier tree. Not only that, but it makes very efficient checks for *non-membership* possible; to prove that a value v_b is not in the tree, one only has to reveal the leaf $v_a < v_b$ that has $v_{\text{next}} > v_b$. This can be optimized even further by inserting entire subtrees at once, though care must be taken to do this correctly in-circuit.

3.3. Light Protocol

The Light Protocol is built atop the Solana L1 blockchain, bringing with it the concept of **ZK Compression**. ZK Compression is a cryptographic facility that aims to significantly reduce the cost of operating upon on-chain state for users running ZK computations [1]. The protocol enables this functionality using two main innovations:

- **Compression:** Only the roots of the state—which store fingerprints of all the compressed accounts—are stored on chain. All of the other data is stored on the Solana Ledger, which has significantly cheaper storage costs. These fingerprints are stored in such a way that if the underlying data changes, the fingerprint must also change.
- **Zero-Knowledge:** Zero-Knowledge proofs are cryptographic attestations that allow one party (here the protocol itself) to prove a statement to another party (the users of the protocol) without revealing data or forcing the protocol users to perform the check themselves. Light Protocol uses ZK proofs as validity checks to ensure the integrity of the compressed state and assure users of that integrity as cheaply as possible.

The compression step may sound quite familiar, and indeed, the Light Protocol makes use of Merkle Trees to compress the state [5]. This ensures that they have a unique root that allows integrity checking for *all* the data stored in the tree.

Each root of a compressed tree is then stored as a leaf node in the state tree. It is this state tree's root that then uniquely represents the compressed state at a moment in time. This allows efficient verification of the validity of any leaf without the need to perform a costly read of the underlying compressed data [5]. By doing this, the protocol enables computation over state much more cheaply than if the state had to be accessed directly. Furthermore, such computation can be done in Zero-Knowledge, allowing much more flexible computational designs to be run on top of Solana [1].

As these trees are so crucial to the security of the Light Protocol, it is extremely important that they operate as expected, and do not admit adversarial behavior by a potential attacker.

4. Tree Operations Specification

This section describes the intended behavior of the Merkle Tree operations used as part of the Light Protocol. These behaviors are encoded as ZK circuits that are then proved. These proofs should only be able to be verified successfully if the operations in question have been performed correctly. It is these ZK circuits—that are claimed to implement the specified behavior—that have been verified by this formal verification effort.

Algebraic Hashing and Poseidon

Traditional cryptographic hash functions, such as Keccak, are extremely expensive to encode inside the arithmetic circuits used by Zero-Knowledge cryptographic systems. To that end, there are hash functions—such as Poseidon [6]—that are *arithmetic*, meaning that they avoid operating directly on bits. These are much cheaper to encode in such circuits.

In all cases we are discussing a **binary Merkle Tree** which uses **Poseidon** as its hash function. In the case of the non-inclusion circuit, the Merkle Tree additionally has the Range Tree structure imposed upon it.

Public and Private Inputs

ZK circuits are traditionally expressed in a form where they have two kinds of inputs:

- **Public Inputs:** These are inputs that are part of the *public commitment* accompanying the proof. They are *revealed* alongside the proof and are required to successfully verify the proof.
- **Private Inputs:** These are inputs that are provided to the circuit as part of the proving process but are *not revealed* to any entity beyond the prover.

The circuits used for the Merkle Tree operations as part of the Light Protocol make use of both kinds of input.

4.1. Inclusion

The inclusion circuit proves that the specified leaves have been included in the tree with the corresponding root.

The **public** inputs to this circuit are as follows:

- `roots[]`, an array containing the root values of the `len(roots)` Merkle Trees in which inclusion should be checked.
- `leaves[]`, an array containing the leaf to check for inclusion in the corresponding tree. Each `leaves[i]` is checked for inclusion in the tree described by `roots[i]`, meaning that only one leaf can be checked per tree in a batch.

The **private** inputs to this circuit are as follows:

- `InPathIndices[]`, an array containing the index at which the corresponding leaf can be found in the corresponding tree. In other words `InPathIndices[i]` is the index in the tree given by `roots[i]` at which `leaves[i]` can be found. This is necessary to know whether the

leaf is the left or right argument to the node hash function when computing with a Merkle Proof.
- `InPathElements[][]`, an array that provides the Merkle Proof for the value `leaf[i]` and root `roots[i]` at the index `InPathElements[i]`.

In the intended operation of this circuit, a satisfying assignment should only be found if and only if the following property holds: for all `i`, if `roots[i]` is the root of a known Merkle Tree, then `leaves[i]` is a leaf in that tree. We do not specify the behavior of the circuit if `roots[i]` is not a root of a Merkle Tree.

4.2. Non-Inclusion

The non-inclusion circuit proves that the specified Range Trees do not contain the corresponding leaf values.

The **public** inputs to this circuit are as follows:

- `roots[]`, an array containing the root values of the `len(roots)` Range Trees in which the provided values will be checked for non-inclusion.
- `values[]`, an array containing values that should be shown to not be included in the corresponding tree. Each `values[i]` is checked for non-inclusion in the tree described by `roots[i]`.

The **private** inputs to this circuit are as follows:

- `InPathIndices[]`, an array containing the index at which the corresponding low-range value can be found in the corresponding tree. In other words, `InPathIndices[i]` is the index in the tree given by `roots[i]` at which `values[i]` can be found.
- `LeafLowerRangeValues[]`, an array containing the values at each `InPathIndices[i]`. This value is the $v_a < v_b$ (where v_b = `values[i]`) such that $v_a.v_{\text{next}} > v_b$.
- `LeafHigherRangeValues[]`, an array containing the values that are the next adjacent value greater than the corresponding `LeafLowerRangeValues[i]`. In other words, $v_a.v_{\text{next}}$ = `LeafHigherRangeValues[i]`.
- `NextIndices[]`, an array containing the index at which the higher range value is located in the corresponding tree. In other words, `NextIndices[i]` is the index at which `LeafHigherRangeValues[i]` occurs in the Range Tree given by `roots[i]`.
- `InPathElements[][]`, an array that provides the Merkle Proof for the value `values[i]` and root `roots[i]` at the index `InPathElements[i]`.

Conceptually, each range can be thought of as containing a lower bound v, an upper bound v_{next}, and a salt i_{next}. While i_{next} is operationally important in other parts of the protocol, for the circuits discussed here its value is irrelevant and is only provided to make knowledge proofs possible. The ranges are exclusive of both bounds, making it an open interval.

In the intended operation of this circuit, a satisfying assignment should only be found if the following property holds: for all `i`, if `roots[i]` is the root of a known range tree, then `values[i]` is included in some range that is included in the tree. While the data structure has inclusion semantics, inclusion within it is *used* to confer non-inclusion semantics on another tree. We do not specify the behavior of the circuit if `roots[i]` is not the root of a Range Tree.

4.3. Inclusion and Non-Inclusion

This circuit combines the ability to prove inclusion and non-inclusion in a single circuit (and hence a single proof). At the implementation level it is a combination of the inclusion and non-inclusion circuits, and hence takes the combination of their public and private inputs (modulo some renaming).

In the intended operation of this circuit, a satisfying assignment should only be found if all of the following properties hold:

- The inclusion circuit is satisfied.
- The non-inclusion circuit is satisfied.

No other set of public and private inputs should result in a satisfying assignment for the circuit.

5. Formal Verification Overview

As the circuits whose behaviors are described above are implemented in the Gnark Zero-Knowledge proof system, Reilabs was able to use our automated toolchain to extract the Gnark circuits into equivalent definitions in the Lean theorem prover and programming language [7].

This allowed us to model the semantics of those circuits in a formal setting, and these models form the basis of Reilabs' formal verification of the required properties of these circuit operations.

5.1. Methodology

By necessity, the extraction process operates on *instantiated circuits* rather than any generic function that templates circuits over certain parameters. To that end, we are forced to select concrete values for both the number of roots operated on in a batch (equivalent to `len(roots)`), and the depth of the tree itself. For the purposes of our analysis here, we have selected the depth of the tree to be 26, and set our batch size to be 8, as these values correspond to the configuration of the system as deployed in production.

Extraction from the Gnark DSL

The analysis of the aforementioned circuits is being performed in the context of a model of the Gnark Circuit DSL embedded in Lean using Reilabs' ProvenZK library. This models the intended semantics of the Gnark DSL directly.

As a result, no analysis has been performed on the correctness of the translation from the Gnark DSL's description of the circuit to the corresponding constraint system within Gnark. Similarly, no analysis has been performed on any of the other cryptographic operations used by the system.

The correctness of the specified behaviors is continuously checked using the CI Service for the Light Protocol, ensuring the ongoing validity of the proofs as the system evolves. While this document refers to the state of the Light Protocol repository at commit `62916e5` the formal verification results are continuously updated and can be checked against any newer commit of interest.

5.2. Conventions and Assumptions

Reilabs' Gnark extraction library compiles the circuit into a Lean function returning a proposition (`Prop`). This proposition is true *if and only if* all of the gates in the circuit 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".

In addition, we freely assume the collision resistance of the Poseidon hash function in accordance with the cryptanalysis given in the technical report defining it [6]. We model this collision resistance through an assumption of injectivity: for a given hash function H, we have $H(a) = H(b) \Rightarrow a = b$. While this assumption is demonstrably untrue (by a simple counting argument from the input to output spaces), it follows both the typical usage and industry standard for reasoning about such hash functions.

5.3. Verified Properties

This section describes the properties that have been formally proven for the Merkle Tree operations inside the Light Protocol. They are laid out such that they present a coherent argument for the overall correctness of the system, which will proceed in two stages:

1. First, we establish the correctness of the Poseidon hash function implementation used throughout the system, as this is a fundamental building block used for the hashing operations in the Merkle Trees.
2. We then proceed to verify the correctness of the inclusion, non-inclusion and combination circuits in the light of the hash function's correctness. In doing so, we show that proofs of the circuits will only verify correctly if the circuit behavior is correct and fully-constrained.

This structure aims to sufficiently convince the reader that the implementations adhere to the specification of the tree behavior given in the previous section.

The automatically-extracted circuit definitions can be found in the `Circuit.lean` file which is committed to the repository for ease of inspection. Please note that the committed version is **not used on CI**, as it is instead regenerated dynamically from the circuit definition in Gnark to ensure that the continuous proving operates on the true 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.

The inclusion circuit is given as follows:

```
def InclusionCircuit_8_8_8_26_8_8_26
  (Roots: Vector F 8)
  (Leaves: Vector F 8)
  (InPathIndices: Vector F 8)
  (InPathElements: Vector (Vector F 26) 8): Prop
```

The non-inclusion circuit is given as follows:

```
def NonInclusionCircuit_8_8_8_8_8_8_26_8_8_26
  (Roots: Vector F 8)
  (Values: Vector F 8)
  (LeafLowerRangeValues: Vector F 8)
  (LeafHigherRangeValues: Vector F 8)
  (NextIndices: Vector F 8)
  (InPathIndices: Vector F 8)
  (InPathElements: Vector (Vector F 26) 8): Prop
```

Finally, the combined circuit is given as follows:

```
def CombinedCircuit_8_8_8_26_8_8_8_8_8_8_8_26_8
  (Inclusion_Roots: Vector F 8)
  (Inclusion_Leaves: Vector F 8)
  (Inclusion_InPathIndices: Vector F 8)
  (Inclusion_InPathElements: Vector (Vector F 26) 8)
  (NonInclusion_Roots: Vector F 8)
  (NonInclusion_Values: Vector F 8)
  (NonInclusion_LeafLowerRangeValues: Vector F 8)
  (NonInclusion_LeafHigherRangeValues: Vector F 8)
  (NonInclusion_NextIndices: Vector F 8)
```

```
    (NonInclusion_InPathIndices: Vector F 8)
    (NonInclusion_InPathElements: Vector (Vector F 26) 8): Prop
```

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

5.3.1. Poseidon's Correctness

Used for the computation of the nodes in the tree, the Poseidon hash function was selected here for its ability to be efficiently implemented inside arithmetic circuits. For the Poseidon sub-circuit, we establish two basic properties:

1. **Determinism:** That the output gates of the sub-circuit 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 of the same hash function on some test inputs.

Property 1 alone is sufficient to mitigate one of the most common sources of exploits in ZK protocols: gate value assignments that were not anticipated or were insufficiently controlled by the designers of the protocol. Determinism ensures that no such arguments exist —gates designated as outputs have a **unique assignment** based solely on the inputs.

Extracted Circuit Structure and `UniqueAssignment`

In order to properly model non-deterministic behavior, the ProvenZK model of a circuit uses a combination of existential quantification (∃) and Continuation-Passing Style (CPS). A sub-circuit (or "gadget") that performs a number of assertions and produces a value of type α is encoded in the type $(\alpha \to \mathrm{Prop}) \to \mathrm{Prop}$.

The `UniaueAssignment` structure is defined as follows:

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

This 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 embedded value of `val`, without changing the truth value of the proposition.

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

The `emb` parameter is used to improve composition 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.

Property 2 is more akin to a unit test. We claim that it is *overwhelmingly improbable*, within the bounds of the hash function's collision resistance, for an arbitrary function F and a hash function H to produce the same output for a *randomly selected input* unless they are **the same function**.

Therefore, this combination of properties is a sufficiently convincing demonstration of the equivalence of the extracted circuit to the intended implementation of the hash function.

We demonstrate the **determinism** property by using the `UniqueAssignment` structure. The existence of such a structure for a given gadget means that said gadget is equivalent to a function; that there is only one valid output value that corresponds to a given input.

Poseidon's determinism is demonstrated in `Poseidon.lean` by deriving terms that return the correct instantiation of `UniqueAssignment`. As Poseidon is used over inputs of both three and four field elements in length, we derive two such terms.

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

This says that for any input `inp`, the `LightProver.poseidon_3` gadget with inputs set to `inp` can be replaced with the provided constant using the trivial (identity) embedding.

```
def poseidon_4_uniqueAssignment (inp : Vector F 4):
  UniqueAssignment (LightProver.poseidon_4 inp) id
```

Similarly, this says that for any input `inp`, the `LightProver.poseidon_4` gadget with inputs set to `inp` can be replaced with the provided constant using the trivial (identity) embedding.

These terms are then used to define the corresponding hash functions, which hash their inputs by prepending a `0` to the input vector and then taking the first element of the resulting vector:

```
def poseidon₂ : Hash F 2 := fun a =>
  (poseidon_3_uniqueAssignment vec![0, a.get 0, a.get 1]).val.get 0

def poseidon₃ : Hash F 3 := fun a =>
  (poseidon_4_uniqueAssignment vec![0, a.get 0, a.get 1, a.get 2]).val.get 0
```

We also demonstrate the correctness of both `poseidon₂` and `poseidon₃` by evaluating it on test vectors. These test vectors have been derived from the Poseidon implementation in Circom v2.1.9 and the corresponding version of circomlib.

Using these test vectors, we are able to derive terms of the following types, and thereby demonstrate the output correctness property for the Light Protocol's Poseidon implementation.

```
theorem poseidon₂_testVector :
  poseidon₂ vec![1, 2] =
    7853200120776062878684798364095072458815029376092732009249414926327459813530
      := rfl

theorem poseidon₃_testVector :
  poseidon₃ vec![1, 2, 3] =
    6542985608222806190361240322586112750744169038454362455181422643027100751666
      := rfl
```

5.3.2. Verifying Inclusion

We demonstrate the correctness of the inclusion circuit—as predicated on the correctness of the Poseidon sub-circuit—in a theorem that states all of the required properties of the operation. If this theorem passes type-checking in Lean, then it has been verified to be correct.

```
theorem InclusionCircuit.correct
  [Fact (CollisionResistant poseidon₂)]
  {trees : Vector (MerkleTree F poseidon₂ 26) 8}
  {leaves : Vector F 8}:
    (∃p₁ p₂, LightProver.InclusionCircuit_8_8_8_26_8_8_26
      (trees.map (·.root)) leaves p₁ p₂)
    ↔ ∀i (_: i∈[0:8]), leaves[i] ∈ trees[i]
```

5.3.3. Verifying Non-Inclusion

We demonstrate the correctness of the non-inclusion circuit—as predicated on the correctness of the Poseidon sub-circuit—in a theorem that states all of the required properties of this operation. If this theorem passes type-checking in Lean, then it has been verified to be correct.

```
theorem NonInclusionCircuit.correct
  [Fact (CollisionResistant poseidon₃)]
  [Fact (CollisionResistant poseidon₂)]
  {trees : Vector (RangeTree 26) 8}
  {leaves : Vector F 8}:
    (∃p₁ p₂ p₃ p₄ p₅,
      LightProver.NonInclusionCircuit_8_8_8_8_8_8_26_8_8_26
        (trees.map (·.val.root)) leaves p₁ p₂ p₃ p₄ p₅)
    ↔ ∀i (_: i∈[0:8]), leaves[i] ∈ trees[i]
```

5.3.3.1. The `RangeTree` Type

Note that the theorem above uses the `RangeTree` type. It is defined as follows:

```
structure Range : Type where
  lo : Fin (2^248)
  hi : Fin (2^248)
  index : F

def Range.hash : Range → F := fun r => poseidon₃ vec![r.lo, r.index, r.hi]

def RangeTree (d : ℕ) : Type :=
  { t: MerkleTree F poseidon₂ d //
    ∀ (i : Fin (2^d)), ∃ range, t.itemAtFin i = Range.hash range }
```

`Range` is a structure containing the lower and upper bounds of the range. The `Fin (2^248)` type guarantees that the bounds are less than `2^248`, as is specified by Light Protocol. A `RangeTree` is a Merkle Tree, with the additional property that each of its leaves is a hash of some `Range`.

Moreover, we define the membership relation for `RangeTree` as follows:

```
def rangeTreeMem {d} : Range → RangeTree d → Prop := fun r t => r.hash ∈ t.val

instance : Membership F Range where
  mem x r := r.lo.val < x.val ∧ x.val < r.hi.val

instance {d} : Membership F (RangeTree d) where
  mem x t := ∃(r:Range), rangeTreeMem r t ∧ x ∈ r
```

This says that a value is a member of a `Range` if and only if it is strictly between the lower and upper bounds of the range. Furthermore, it says that a value is a member of a `RangeTree` if and only if there exists a `Range` such that the value is a member of that `Range` and the `Range` is a member of the `RangeTree`.

5.3.3.2. Correctness of the Comparison Gadget

The comparison operator implemented in the circuit requires some additional care. The `AssertIsLess` sub-circuit as implemented in the Light Protocol is only correct when the lower value is bounded by some constant. This is demonstrated by the following example:

```
example : LightProver.AssertIsLess_248
  (Order - 20) 10 ∧ (Order - 20 : F).val > 10
```

What this example says—given that it compiles and passes type-checking in Lean—is that the field prime $p - 20$ is less than 10 according to the range-checking sub-circuit, while *also* being greater than 10 by Lean's native ordering semantics. This is quite a problem given that some values are not explicitly checked for bounds in the circuit.

However, the nature of the circuit has it operate *correctly* in *certain circumstances*: namely, when the value of A in the range-check is less than 2^{249}. Moreover, in such a case, the gadget also imposes an upper bound on the larger value. We also demonstrate this fact with a theorem. Under this condition, `LightProver.AssertIsLess_248` behaves properly with respect to correct ordering semantics.

```
theorem AssertIsLess_bounds
  {A B : F}
  (A_range : A.val ≤ 2 ^ 249):
    LightProver.AssertIsLess_248 A B → A.val < B.val ∧ B.val ≤ A.val + 2^248
```

The fact that the semantics operate as expected within this range is only useful if we can then show that the real use-case in the circuit has this property. Thankfully we were able to do so as, by construction, the tree has its values restricted to 2^{248}.

```
theorem AssertIsLess_range
  {hi lo val : F}
  (lo_range : lo.val < 2^248) :
  LightProver.AssertIsLess_248 lo val ∧
    LightProver.AssertIsLess_248 val hi → lo.val < val.val ∧
    val.val < hi.val
```

This theorem, found in `Rangecheck.lean` is used in the theorem for the non-inclusion circuit, and therefore the circuit still exhibits the expected behavior in this usage even under the anomalous definition of the range-checking sub-circuit.

5.3.4. Verifying Both

We demonstrate the correctness of the combined circuit—as predicated on the correctness of the Poseidon sub-circuit—in a theorem that states all of the required properties of this operation. If this theorem passes type-checking in Lean, then it has been verified to be correct.

```
theorem CombinedCircuit.correct
  [Fact (CollisionResistant poseidon₃)]
  [Fact (CollisionResistant poseidon₂)]
  {inclusionTrees : Vector (MerkleTree F poseidon₂ 26) 8}
  {nonInclusionTrees : Vector (RangeTree 26) 8}
  {inclusionLeaves nonInclusionLeaves : Vector F 8}:
    (∃p₁ p₂ p₃ p₄ p₅ p₆ p₇,
      LightProver.CombinedCircuit_8_8_8_26_8_8_8_8_8_8_8_26_8
        (inclusionTrees.map (·.root)) inclusionLeaves p₁ p₂
        (nonInclusionTrees.map (·.val.root)) nonInclusionLeaves p₃ p₄ p₅ p₆ p₇)
    ↔ ∀i (_: i∈[0:8]), inclusionLeaves[i] ∈ inclusionTrees[i]
                      ∧ nonInclusionLeaves[i] ∈ nonInclusionTrees[i]
```

6. Risks and Limitations

The analysis and formal verification presented above provides *strong evidence* for the correctness of the Merkle Tree operations. Nevertheless, it is inherently incapable of proving the **absence** of attack vectors. The following is a non-exhaustive list of risk factors that are not accounted for by this analysis.

- Reilabs' Gnark extractor tool has **not** been audited or formally verified. While all care has been taken to ensure its correctness, errors in this tool may lead to the extracted circuit definitions in Lean not being equivalent to the source circuits as they are described using the Gnark DSL.
- The ProvenZK library is used to provide models of circuit gates and other cryptographic primitives in this verification effort. Errors in this library may result in the modeled behavior of the gates being different from their *actual* behavior in the Gnark framework.
- This analysis is deliberately imprecise—as discussed above—when it comes to its treatment of the collision resistance of cryptographic hash functions. While we believe that out assumption is reasonable—and have taken care not to use said assumption beyond its standard usage in cryptanalysis—it is provably false and could be used to render the Lean Prover inconsistent. Despite that such a thing *could* be manually proven to be false, this is not something that we do in our work, and hence the analysis contained herein is sound within those bounds.
- The gate representation—from which the Lean circuit definitions are extracted—is a representation of the circuit at a very early stage of the Gnark pipeline. This representation undergoes many further transformations, including translation into a polynomial arithmetization and further cryptographic operations. As a result, any errors occurring in the Gnark toolchain after the point of circuit extraction may render the real-world behavior of the protocol different from its circuit-level specification.

7. Conclusion

This analysis of the design of the inclusion, non-inclusion, and combined inclusion-non-inclusion circuits for the Light Protocol, accompanied by the results that have been stated and explained with the aid of formal verification, provides **strong evidence for the correctness of the circuits** as used in this system.

The statement of this specification in a formal setting, combined with the automatically-extracted model of the system's implementation, has allowed us to prove that the existing implementation satisfies its specification. Furthermore, the continuous checking of these properties on CI as part of the Light Protocol repository ensures that, if any change to the codebase would invalidate these properties, the team will be alerted before they can introduce a correctness flaw or potential attack vector.

Our work here is key to establishing confidence in the smooth operation of the Merkle Trees that underlie the ZK compression facility of the Light Protocol, and hence in the smooth operation of the entire protocol.

Bibliography

[1] Light Protocol Labs, "Light Protocol: Introduction." Accessed: Jul. 17, 2024. [Online]. Available: https://docs.lightprotocol.com/

[2] Light Protocol Labs, "Light Protocol." Accessed: Jul. 17, 2024. [Online]. Available: https://lightprotocol.com/

[3] G. Becker, "Merkle Signature Schemes, Merkle Trees and their Cryptanalysis," Jul. 2008. [Online]. Available: https://web.archive.org/web/20141222120036/http://www.emsec.rub.de/media/crypto/attachments/files/2011/04/becker_1.pdf

[4] Aztec, "Indexed Merkle Tree." Accessed: Jul. 17, 2024. [Online]. Available: https://docs.aztec.network/aztec/concepts/storage/trees/indexed_merkle_tree

[5] Light Protocol Labs, "Light Protocol: State Trees." Accessed: Jul. 17, 2024. [Online]. Available: https://docs.lightprotocol.com/learn/core-concepts/state-trees

[6] 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

[7] Reilabs, "reilabs/gnark-lean-extractor v2.2.0." Accessed: Jul. 17, 2024. [Online]. Available: https://github.com/reilabs/gnark-lean-extractor/commit/37dfe6a3135dfecbb095f4df563e666f6e3d95d9